UNIVERSITÄT
MANNHEIM

Mobile Phone Forensics: Analysis of the Android Filesystem (YAFFS2)

Diploma Thesis by **Christian Zimmermann**

April 30, 2011

First Examiner:	Prof. Dr. Ing. Felix C. Freiling
Second Examiner:	Prof. Dr. Ing. Wolfgang Effelsberg
Advisors:	Michael Spreitzenbarth
	Sven Schmitt

University of Mannheim
Laboratory of Dependable Distributed Systems
D - 68131 Mannheim
`http://pi1.informatik.uni-mannheim.de/`

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Mannheim, den 30. April 2011 Christian Zimmermann

Abstract

Modern smartphones are not only used for communication via phone calls but also for a variety of other purposes such as emailing or storing personal data. To enable these services, most smartphones possess a large memory capacity to save files and application data. As smartphones are not only used for legal purposes but also for criminal actions, this data can contain valuable evidence for forensic investigators. In order to be able to analyze this data, it is very important to understand the way smartphones store and delete data. Therefore, it is necessary to understand the file system that is used to handle the smartphone's internal flash memory.

The major focus within this diploma thesis is the analysis of the flash file system YAFFS2 which is used by the popular Android smartphones. To that purpose, YAFFS2 is theoretically and practically analyzed in a forensic perspective in order to determine possibilities to recover modified or deleted files from a smartphone's flash memory. Additionally, YAFFS2 is compared to the common file system NTFS within this diploma thesis.

This diploma thesis also includes a chapter discussing ways to safely delete files from a YAFFS2 flash memory.

Zusammenfassung

Moderene Smartphones werden nicht nur zum Kommunizieren mittels Telefonaten genutzt, sondern auch für eine Vielzahl anderer Zwecke, wie beispielsweise zum E-Mai-Versand oder zum Speichern persönlicher Daten. Um diese Dienste zu ermöglichen, verfügen die meisten Smartphones über große Speicherkapazitäten, um Dateien und Anwendungsdaten zu speichern. Da Smartphones nicht nur zu legalen Zwecken, sondern auch für kriminelle Handlungen verwendet werden, können die auf ihnen gespeicherten Daten wertvolle Beweise für Computerforensiker darstellen. Um diese Daten analysieren zu können, ist es wichtig, zu verstehen, wie Smartphones Daten speichern und löschen. Dazu ist ist es notwendig, das Dateisystem zu verstehen, das von einem Smartphone verwendet wird, um seinen internen Flash-Speicher zu verwalten.

Das Hauptaugenmerk dieser Diplomarbeit liegt auf der Untersuchung des Dateisystems YAFFS2, das in den weit verbreiteten Android Smartphones Verwendung findet. Zu diesem Zweck wird YAFFS2 sowohl theoretisch wie auch praktisch unter forensischen Gesichtspunkten analysiert, um Möglichkeiten zur Wiederherstellung gelöschter oder veränderter Dateien zu bestimmen. Zusätzlich wird YAFFS2 im Rahmen dieser Diplomarbeit mit dem weit verbreiteten Dateisystem NTFS verglichen.

Des Weiteren wird in dieser Diplomarbeit das sichere Löschen von Dateien von YAFFS2 Flash-Speichern diskutiert.

Contents

List of Figures

List of Tables

Listings

1. Introduction

1.1. Motivation

Nowadays, mobile phones have become one of the most important communication devices. Beside traditional mobile phones, especially smartphones are gaining more importance in business as well as in many people's everyday life, due to their tremendous range of functions. This reflects in rising smartphone sales, which have increased to 19 percent of total mobile communication device sales in 2010 [3]. As a consequence of the increasing popularity of smartphones, a rapid growth of malware designed to attack smartphones can be observed. As depicted in Figure 1.1, according to McAfee Labs [1], *the number of new mobile malware in 2010 increased by 46% compared with 2009.* Additionally to malware cases, there is also a large number of other criminal acts whose solution depends on forensic analyses of a smartphone's contents.

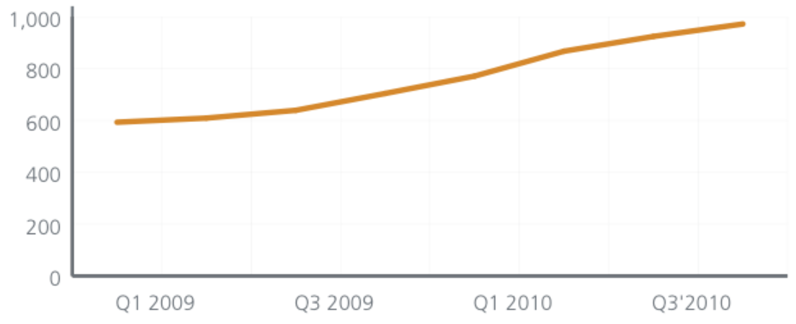


Figure 1.1.: Mobile malware growth in recent years [1]

In contrast to traditional mobile phones, smartphones can not only be used to communicate via phone calls or text messages but also for a variety of other purposes such as browsing the internet, listening to music or chatting. To enable these services, modern smartphones possess a large memory capacity to store files and application data like mp3 files, stored passwords, SMS messages, call logs or pictures. In order to be able to analyze these contents, it is very important to understand the way modern smartphones store and delete those data. Therefore, it is important to understand the file system that is used to handle the smartphone's internal storage device.

Due to its low power consumption, small size and shock resistance, many smartphones use *NAND* flash memory as mass storage device. However, the physical structure of

NAND flash memory necessitates use of special file systems in order to ensure the flash memory's durability. Although several file systems suitable for NAND flash memory exist, a significant amount of smartphones are using the file system *YAFFS2* (*Yet Another Flash File System 2*) [6]. That is because, up to now, YAFFS2 is the default file system for Google's mobile operating system *Android OS* [7], which is one of the most popular mobile operating systems on the market [3]. As can be seen in Table 1.1, 67,224,500 smartphones running Android OS were sold in 2010, accounting for 22.7 percent of all smartphones sold in 2010 and making Android OS the second most sold smartphone operating system in 2010.

Company	**Units in 2010**	**2010 market share (%)**
Symbian	111,576.7	37.6
Android	67,224.5	22.7
Research in Motion	47,451.6	16.0
iOS	46,598.3	15.7
Microsoft	12,378.2	4.2
Other OSs	11417.4	3.8
Total	**296,646.6**	**100.0**

Table 1.1.: Worldwide smartphone sales to end users by operating system in 2010 (Thousands of units) [3]

Considering this large number and still growing popularity of smartphones using the Android operating system, there is a clear need for a detailed forensic understanding of the file system YAFFS2 in order to perform forensic analyses of Android OS smartphones.

1.2. Task

Both YAFFS2 and the Android operating system are completely open source software and well documented. However, from a forensic point of view, especially YAFFS2 is still insufficiently studied. The task of this diploma thesis is an examination of the file system YAFFS2's behavior in a forensic perspective. This includes a comparison of YAFFS2's specification and its actual behavior.

As YAFFS2 is explicitly designed to be used on NAND flash memory, it makes use of wear leveling techniques in order to prevent reduction of flash memory chips' lifespan. These wear leveling techniques as well as YAFFS2's techniques regarding garbage collection are suspected to have a significant influence on the amount of evidence forensic investigators are able to recover from a Android smartphone. Therefore, the analysis of YAFFS2's wear leveling and garbage collection mechanisms is a major focus within this diploma thesis.

Apart from a theoretical analysis of the YAFFS2 source code, this diploma thesis includes a practical evaluation of YAFFS2's behavior when performing commonly occurring file system operations such as file modifications or deletions in different scenarios.

Based on this evaluation as well as on the theoretical analysis of the YAFFS2 source code, ways to recover deleted files and previous versions of modified files from a YAFFS2 NAND flash memory device are analyzed within this diploma thesis. Additionally, the results of these recovery attempts are compared with the results of similar analyses of a NTFS device.

Finally, ways to securely delete data from a YAFFS2 storage device are discussed in the last part of this diploma thesis.

1.3. Outline

This diploma thesis is outlined as follows: In Chapter 2, we provide background information on flash memory and an introduction on how to use YAFFS2 in a Linux environment. Chapter 3 provides a theoretical analysis of YAFFS2's source code and an introduction to YAFFS2's general functionality. This includes a comparison of YAFFS2's specification and its actual behavior as well as an in-depth analysis of YAFFS2's garbage collection techniques. Based on the results of the analyses conducted in Chapter 3, we analyze YAFFS2 in a forensic perspective in Chapter 4. In Chapter 5, we provide a practical analysis of YAFFS2 and discuss ways to recover modified or deleted files from a YAFFS2 NAND flash memory device. We also discuss ways to safely delete files from a YAFFS2 device in Chapter 6. Finally, in Chapter 7 we conclude this thesis with a short summary and propose opportunities for future research.

1.4. Acknowledgements

First of all, I want to express my deepest gratitude to Prof. Dr. Felix C. Freiling for giving me the opportunity to work on such an interesting topic for my diploma thesis. It was especially very exciting to work on the cutting edge of mobile phone forensics. Furthermore, I wish to thank Prof. Dr. Wolfgang Effelsberg for being my second examiner. I also want to thank my advisors Michael Spreitzenbarth and Sven Schmitt for their incessant support and very valuable feedback. Another important concern to me is to thank Michael Spreitzenbarth for supplying me with a mobile phone that helped a lot to deepen my knowledge of YAFFS2. Many thanks go to my friends and especially my cohabitants for their support, the interesting discussions and great time we had. Last but not least, I want to thank my family for giving me constant backup and support throughout my thesis and studies.

2. Prerequisites

In order to understand the basic design principles of flash memory file systems like YAFFS2 and the issues they have to address, it is important to understand the underlying technical characteristics of flash memory chips. Therefore, we provide an introduction to flash memory in Section 2.1 of this chapter. As most analyses of YAFFS2 within this diploma thesis were performed on a simulated NAND flash memory device in a Linux environment, we provide an introduction on how to integrate YAFFS2 into a Linux environment in Section 2.2 of this chapter. Additionally, we describe how to simulate a NAND flash memory device in a Linux environment using RAM in the last part of this chapter.

2.1. Flash memory

Flash memory is a type of *EEPROM* (*Electrically Erasable Read Only Memory*) consisting of memory cells made from floating-gate transistors [8] and is often used as storage medium in mobile devices. This is mostly because flash memory provides shock resistance due to being a solid state storage, making it much better suited for mobile devices than magnetic disks or other storage devices featuring moving parts. Additionally, flash memory provides high density at low production costs [9].

Flash memory is organized in blocks [10] of a device-dependent fixed size. On an empty flash memory, typically all bits are set to a logical one and can be changed to a logical zero by a write operation [8, 10]. However, changing a bit's value from a logical zero to a logical one can only be achieved by erasing the whole block containing this bit, that is, by setting all bits of the block to a logical one [10, 11]. Thus, except from rewrite operations that include only the writing of logical zeros, all rewrite operations require the erasure of the block the bits to be rewritten are located in. As each block can only withstand a limited number of erase and rewrite operations [8], techniques to evenly distribute erasures among all blocks have to be provided by a flash memory file system. These techniques are referred to as *wear leveling*. Effects of wear leveling on forensic analyses of flash memory are discussed in Chapter 4.

Flash memory is available in two major types, *NOR* and *NAND* flash memory. Although sharing the aforementioned limitations, NOR and NAND flash memory have completely different characteristics and differ greatly in their suitability regarding their use as non-volatile data storage in mobile devices. The reason for that are architectural differences between NOR and NAND flash memory. While NOR flash memory features parallel connection of memory cells, thus acting like a NOR gate, NAND flash memory consists of serially connected memory cells, hence acting like a NAND gate [10].

Another major difference between NOR and NAND flash memory concerns their physical interfaces. Featuring enough address pins to map its entire media, NOR flash memory offers direct access to every single one of its bytes to a processor. Thus, NOR flash memory is accessible like other random access memory devices and can be used for code execution [9].

In contrast to NOR flash memory, NAND flash memory does not offer direct accessibility and can only be accessed sequentially through an I/O interface with typically 8 or 16 pins [12]. Therefore, code can not be executed directly on the NAND flash memory but has to be loaded into RAM to be executed from there. Additionally, memory cells of NAND flash memory are not written or read individually like NOR flash memory cells, but in *pages*. These pages are a further subunit of the NAND flash memory's blocks and constitute the unit of read and write operations of a NAND flash memory [12]. That means, that once a page has been written, its bytes can only be rewritten after the block containing the page has been erased. Every page features an *OOB area* (*Out-Of-Band area*) to store additional data for error correction code, bad block information or other data associated with the page's content [12]. Bad block handling is necessary, as almost every NAND flash memory device has some bad blocks, i.e. unusable blocks, due to manufacturing errors and wear marks.

Characteristic	**NAND** (2048 bytes/page, 64 bytes OOB)	**NOR**
Random access read	25μs (first byte) 0.025μs each for remaining 2111 bytes	0.075μs
Sustained read speed (sector basis)	26 MB/s (x8) 41 MB/s (x16)	31 MB/s (x8) 62 MB/s (x16)
Random write speed	$\sim 220\mu$s/2112 bytes	128μs/32 bytes
Sustained write speed (sector basis)	7.5 MB/s	0.250 MB/s
Erase block size	128 KB	128 KB
Erase time per block	500μs	1 sec

Table 2.1.: Comparison of a Micron MT29F2G08A NAND flash memory and a Micron TE28F128J3 NOR flash memory [4]

As depicted in Table 2.1, the aforementioned architectural differences between NOR and NAND flash memory result in different read and write performances. In application scenarios where fast read operations of small amounts of data or code execution on the flash memory are essential and write and erase operations do not occur very often, NOR flash memory is clearly superior to NAND flash memory. Because of that, NOR flash memory is well suited for use as code storage as, for instance, in BIOS chips. NOR flash memory can also be used as mass storage but, due to its parallel architecture, needs large physical dimensions to realize storage capacities sufficient for modern requirements [9]. NAND flash memory, on the other hand, is not capable of code execution but offers significantly higher write and erase performance than NOR flash memory and features a considerably higher memory density. Thus, NAND flash memory is far better suited for use as mass storage than NOR flash memory and is used in almost all modern pen drives, MP3 players and memory cards. NAND flash memory is also widely used as mass storage for smartphones, where it is sometimes accompanied by a NOR flash memory that is used for code execution such as booting the operating system.

Because of the aforementioned characteristics and limitations of flash memory, file systems originally designed for hard disk drives are not very well suited for use on flash memory, as these file systems are not designed to deal with wear leveling and the special ways, data has to be rewritten on flash memory. Because of that, several file systems specially designed for flash memory exist, one of them being YAFFS2. In the following section, we describe the technical setup for the analyses of YAFFS2 as used in the following chapters and provide an introduction on how to use YAFFS2 in a Linux environment.

2.2. YAFFS2 in a Linux environment

As YAFFS2 is an active open source project, its source code is subject to constant development. Within this diploma thesis, we analyzed version 0bc94484426d0aa0db445360d6e 5845696936229 [13] of the YAFFS2 source code.[1] A Debian Linux workstation running kernel version 2.6.36 was used to perform all practical analyses.

In order to use YAFFS2 on a Linux workstation, two problems have to be solved first. Firstly, YAFFS2 is not integrated into the Linux kernel by default. Secondly, as a flash memory device is not a block device, Linux is not able to access flash memory by default. Therefore, a custom kernel has to be built to analyze YAFFS2 in a Linux environment. This requires patching the YAFFS2 source code into the kernel source tree and compiling the kernel with a configuration that supports NAND flash memory. To patch the YAFFS2 source code into the kernel tree, the following steps have to be executed:

1. All of YAFFS2's source code and header files have to be copied to subdirectory `/fs/yaffs2` of the kernel source tree.
2. The files `Kconfig` and `Makefile.kernel` from the YAFFS2 source code package have to be copied to the subdirectory `/fs/yaffs2` of the kernel source tree. The file `Makefile.kernel` then has to be renamed to `Makefile`.
3. The file `Kconfig` in subdirectory `fs` of the kernel source tree has to be modified to include the following line within the file's `MISC_FILESYSTEMS` section:

   ```
   source "fs/yaffs2/Kconfig"
   ```

4. Finally, the file `Makefile` in subdirectory `fs` of the kernel source tree has to be modified to include the following line:

   ```
   obj-$(CONFIG_YAFFS_FS) += yaffs2/
   ```

As a next step, the kernel has to be configured to support YAFFS2 and NAND flash

[1]On the attached DVD, see also: /Source code/yaffs2/

memory. Most of our analyses were not performed on an actual NAND flash memory device but on a software simulation of a NAND flash memory device. To simulate a NAND flash memory device, we used the `nandsim` module of the linux kernel. To compile a kernel supporting YAFFS2 and NAND flash memory devices as well as the NAND flash memory simulator, the following kernel configuration options have to be enabled:

- `CONFIG_YAFFS_FS`
- `CONFIG_YAFFS_YAFFS2`
- `CONFIG_BLK_DEV_LOOP`
- `CONFIG_MTD_CFI`
- `CONFIG_MTD`
- `CONFIG_MTD_CHAR`
- `CONFIG_MTD_BLOCK`
- `CONFIG_MTD_NAND`
- `CONFIG_MTD_NAND_NANDSIM`
- `CONFIG_MTD_NAND_ECC`

On a Linux workstation running a kernel compiled with the aforementioned configuration, a NAND flash memory with a size of 64 MiB and a page size of 2048 bytes can be simulated by using the following command: '`modprobe nandsim first_id_byte=0x20 second_id_byte=0xa2 third_id_byte=0x00 fourth_id_byte=0x15`'. Information on how to simulate a NAND flash memory device with a different size can be found in Appendix A.2.

To be able to access the simulated NAND flash memory device, a device file has to be created. This can be achieved by using the command '`mknod /dev/mtd0 c 90 0`'. The so created device can be used to write an image to the simulated NAND flash memory device or dump its contents using the `nandwrite` and `nanddump` utilities from the `mtd-utils` [14] Debian package. However, to be able to mount the simulated NAND flash memory device and use common Linux commands like `ls` or `nano`, it is necessary to link a block device to the NAND flash memory device, which can be done by using the command '`mknod /dev/mtdblock0 b 31 0`'. The `mtdblock` driver does not support wear leveling and bad block management but as YAFFS2 is designed to take care of these tasks itself, the `mtdblock` driver is an appropriate tool to access the simulated NAND flash memory in order to modify or delete files on the simulated NAND flash memory device. The `mtdblock` driver is also used by Android OS to access a smartphone's NAND flash memory device.

2.3. Summary

As depicted at the beginning of this chapter, NAND flash memory's restrictions and characteristics make conventional file systems unsuitable for NAND flash memory devices. Instead, specially designed flash file systems like YAFFS2 have to be used in order to ensure a NAND flash memory's durability and to make use of the advantages NAND flash memory offers. The techniques used by YAFFS2 to handle NAND flash memory's characteristics discussed above are introduced in the next chapter. In the last section of this chapter, we introduced how to use YAFFS2 in a linux environment and how to simulate a NAND flash memory device in this environment.

3. Analysis of YAFFS2

In the following, we introduce and analyze YAFFS2 and its basic design principles. Major focus of this chapter is a theoretical analysis of YAFFS2's source code. The main goal of this chapter is to give an in-depth understanding of YAFFS2 and to reveal discrepancies between YAFFS2's specification and its actual behavior. We analyze and practically evaluate this chapter's findings' implications on forensic analyses of YAFFS2 storage devices in Chapter 4 and Chapter 5.

This chapter is outlined as follows: In Section 3.1, we introduce YAFFS2 and its basic design. In Section 3.2, we analyze YAFFS2's techniques to manage data. This also includes an analysis of YAFFS2's actual behavior regarding data organization and a comparison of this behavior to the behavior described in YAFFS2's specification. As YAFFS2's garbage collection techniques are likely to have a major impact on the amount of data that can be restored from a storage device with YAFFS2 as file system, we provide an analysis of YAFFS2's garbage collection techniques in Section 3.3. Additionally, in Section 3.4, we analyze YAFFS2's techniques to minimize wear of NAND flash memory devices.

3.1. YAFFS2 basics

YAFFS2 is a file system design mainly by Charles Manning specially for use with NAND flash memory devices. It is the successor to YAFFS1 and shares some basic characteristic with its predecessor. However, nowadays, the use of YAFFS1 is deprecated, because it violates the specification of some NAND flash memory chips by writing deletion markers into the OOB areas of pages containing deleted objects' contents, thus rewriting pages' OOB areas without erasing their respective blocks beforehand. Additionally, YAFFS1 can only handle NAND flash memory pages with a maximum size of 512 bytes, while YAFFS2 supports page sizes of 2048 bytes and more [2].

To correspond to the layout of NAND flash memory, YAFFS2 structures data in so-called *chunks* and blocks, with a chunk being the unit of allocation and writing and a block being the unit of erasure. To write data to a NAND flash memory device, YAFFS2 first allocates a block and then writes chunk-wise within this block. Typically, the size of a chunk equals the size of a page on the NAND flash memory, but, if necessary, a chunk can also be mapped to several pages, for example to write to several NAND flash memories in parallel [2]. In the following, unless stated otherwise, a chunk has the size of a page and the terms chunk and page are used synonymously. For simplification, the terms *NAND flash memory device* and *NAND* are also used synonymously in the following.

To YAFFS2, anything that can be stored in the file system is, first of all, an *object*. The way YAFFS2 manages a specific object depends on the type of the object. An object can either be a data file, a directory, a hard link, a soft link or a special object, such as a device file or a pipe. During mounting of a YAFFS2 device, information about all objects on the device, as well as information about the device itself are loaded into RAM. This includes building a directory structure in RAM to be able to find objects by name, as well as building a hash table to be able to find objects by their unique object number. To speed up mounting of a device, YAFFS2 stores parts of this RAM structure in specially reserved blocks on the device. This so called *checkpoint* contains information about the device, the states of the device's blocks and information about objects and their chunks on the device including the directory structure. The number of blocks reserved for this checkpoint and the number of these blocks in use are relevant to the intensity with which YAFFS2 tries to perform garbage collection. More information about YAFFS2's garbage collector is provided in Section 3.3.

To be able to allocate blocks, identify bad blocks and select blocks for delete operations, YAFFS2 distinguishes between ten states a block can be in. These states are defined in lines 229 to 273 of file `yaffs_guts.h`. During runtime, YAFFS2 keeps state information of all blocks in RAM. As can be seen in Figure 3.1, blocks can only transition from one state into another in a certain order.

On a device that has never been mounted using YAFFS2 or if YAFFS2 cannot recover the device's blocks' states from a checkpoint, all blocks are in initial state `UNKNOWN`. In that case, YAFFS2 needs to scan the device to determine all blocks' states. The scan is performed as defined in function `yaffs2_ScanBackwards` in lines 911 to 1539 of file `yaffs_yaffs2.c`. If a block turns out to be unusable due to production errors or wear out, its state transitions from `UNKOWN` into `DEAD`. If a block is not unusable it needs scanning to determine its state and transitions from state `UNKNOWN` into state `NEEDS SCANNING` until it has actually been scanned. While being scanned, a block is in state `SCANNING`. During runtime, a block can either be in state `FULL`, `EMPTY`, `ALLOCATING`, `DIRTY` or `COLLECTING`. The block in state `ALLOCATING` is the block currently selected for write operations. A block stays in state `ALLOCATING` until it is completely filled. A block is in state `FULL` if all of its chunks have previously been allocated and at least one chunk contains current data. A block that has not been fully allocated and is not currently selected for write operations is also in state `FULL`. This can happen, if a device is disconnected from its power source without proper unmounting. In this case, the block's state is set to `FULL` on scanning after the device is mounted again. A block in state `EMPTY` has been erased and does not contain any data. An empty block can be allocated for regular write operations or to store checkpoint data. Full blocks are examined by the garbage collector to check whether they contain current chunks. If the garbage collector becomes active and copies current chunks from a full block to the block in state `ALLOCATING`, the block that is being copied off is in state `COLLECTING`. A block that contains only obsolete chunks is in state `DIRTY` and can transition into state `EMPTY` by being erased. YAFFS2 always reserves some blocks on a device to store checkpoint data. A block containing checkpoint data is in state `CHECKPOINT`.

When allocating blocks and chunks, YAFFS2 follows two main principles to match

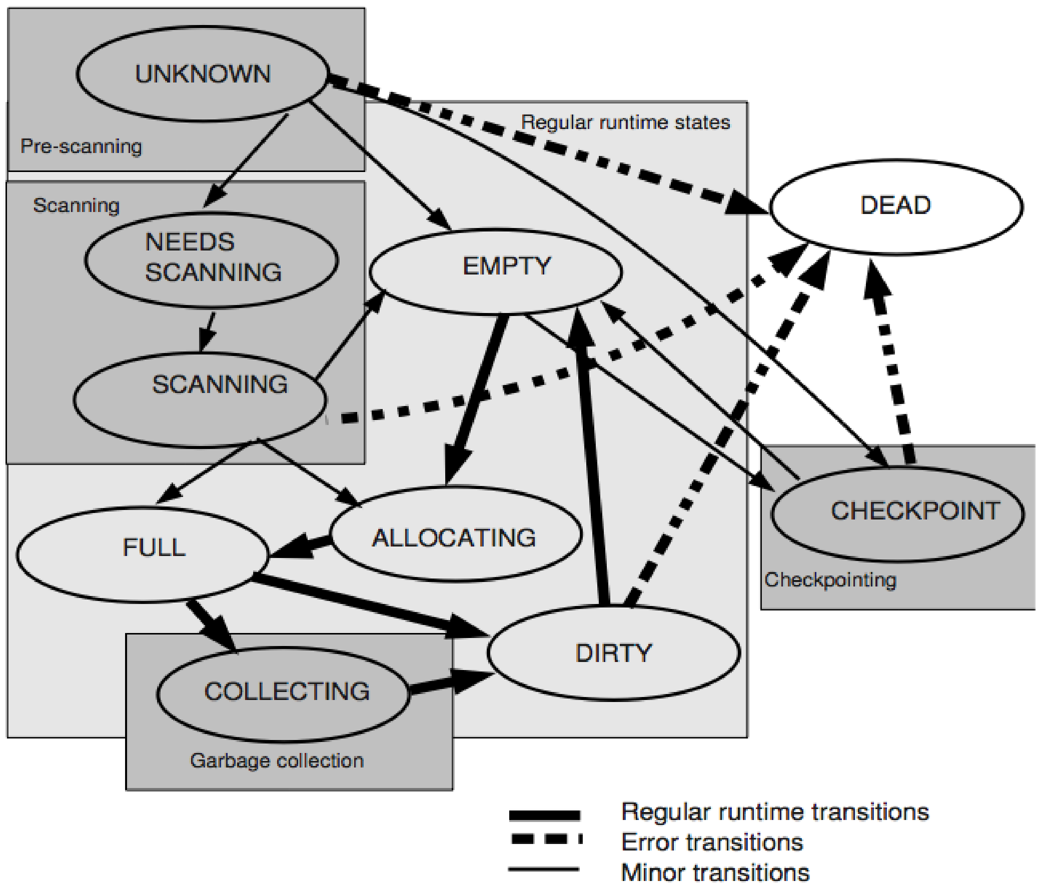


Figure 3.1.: YAFFS2's block states [2]

modern NAND flash memories' specifications. These are a *zero overwrite* policy and sequential writing of chunks within a block. Although it is technically possible to change bits of an already written page from a logical one to a logical zero without having to erase the whole block containing the page, many modern NAND flash memories' specifications deprecate such rewrites to improve flash memories' reliability and lifespan. YAFFS1 violates these specifications by overwriting a byte in the OOB area of a page to mark the page's contents as deleted. YAFFS2 refrains from writing these deletion markers and never overwrites already written chunks without erasing their blocks beforehand, hence following a zero overwrite policy. Typically, modern NAND flash memories' specifications also require sequential writing of pages within a block. As depicted in Listing 3.1, YAFFS2 complies to this requirement by allocating new chunks in sequential order within a block.

```c
static int yaffs_AllocateChunk(yaffs_Device *dev, int useReserve,
                yaffs_BlockInfo **blockUsedPtr)
{
        int retVal;
        yaffs_BlockInfo *bi;

        if (dev->allocationBlock < 0) {
        /* Get next block to allocate off */
        dev->allocationBlock = yaffs_FindBlockForAllocation(dev);
        dev->allocationPage = 0;
        }

        [...]


        /* Next page please.... */
        if (dev->allocationBlock >= 0) {
                bi = yaffs_GetBlockInfo(dev, dev->allocationBlock);

                retVal = (dev->allocationBlock * dev->param.nChunksPerBlock) +
                        dev->allocationPage;
                bi->pagesInUse++;
                yaffs_SetChunkBit(dev, dev->allocationBlock,
                                dev->allocationPage);

                dev->allocationPage++;

                dev->nFreeChunks--;

                /* If the block is full set the state to full */
                if (dev->allocationPage >= dev->param.nChunksPerBlock) {
                        bi->blockState = YAFFS_BLOCK_STATE_FULL;
                        dev->allocationBlock = -1;
                }

                if (blockUsedPtr)
                        *blockUsedPtr = bi;

                return retVal;
        }

        T(YAFFS_TRACE_ERROR,
                (TSTR("!!!!!!!!! Allocator out !!!!!!!!!!!!!!!!!" TENDSTR)));

        return -1;
}
```

Listing 3.1: Function `yaffs_AllocateChunk` (Excerpt from yaffs_guts.c)

Strictly sequential writing of chunks within a block does not only comply to NAND flash memory specifications but also enables YAFFS2 to keep a chronological log of file modifications. Additionally, as can be seen in Listing 3.2, as long as empty blocks can be found on a NAND, YAFFS2 also tries to allocate blocks for writing in sequential order. These methods of allocating chunks and blocks make YAFFS2 a truly log-structured file system. YAFFS2's techniques to organize its log are introduced and analyzed in Section 3.2. As YAFFS2 is a log-structured file system and follows a zero overwrite policy, it is inevitable that a certain amount of obsolete data is almost always stored somewhere on the NAND. To identify and delete this obsolete data, a garbage collector is needed. We provide an analysis of YAFFS2's garbage collector in Section 3.3. In the following, we analyze YAFFS2's techniques to manage data and organize its log.

3.2. Data organization and OOB area tags

When data has to be written to or read from a NAND, YAFFS2 distinguishes between data chunks and header chunks. While data chunks contain actual content of an object, header chunks are used to identify the object's type and to store meta data about the object, such as the object's name, size or its timestamps. Data chunks are only used for files whereas directories and links consist only of header chunks. In the following, we introduce and analyze YAFFS2's techniques to manage chunks and organize its log. In Section 3.2.1, we introduce YAFFS2's OOB area tags. YAFFS2's way to perform modifications and deletions of objects are then introduced in Section 3.2.3.

3.2.1. OOB area tags

In order to be able to associate chunks with an object and store information such as a chunk's position within an object, the OOB areas of NAND flash memory's pages are used. The OOB areas are also used to organize YAFFS2's log chronologically. For that purpose, a sequence number is used. As can be seen in lines 1982 and 1983 of Listing 3.2, every time a block on a device is allocated for writing, the device's block sequence number is incremented. Every chunk that is written to the newly allocated block is marked with the current block sequence number within the chunks OOB area. That way, the block that has been written to most recently has always the highest block sequence number. Thus, a chronological order of blocks can be derived from their sequence numbers, regardless of the blocks' physical position on the device. As chunks are allocated strictly sequentially within a block, starting from the lowest address of the block, all chunks within a block are in chronological order by default.

```
1954 static int yaffs_FindBlockForAllocation(yaffs_Device *dev)
1955 {
1956         int i;
1957         yaffs_BlockInfo *bi;
1958 
1959         [...]

1969         /* Find an empty block. */
1970 
1971         for (i = dev->internalStartBlock; i <= dev->internalEndBlock; i++) {
1972                 dev->allocationBlockFinder++;
```

```
                        if (dev->allocationBlockFinder < dev->internalStartBlock
                            || dev->allocationBlockFinder > dev->internalEndBlock) {
                                dev->allocationBlockFinder = dev->internalStartBlock;
                        }

                        bi = yaffs_GetBlockInfo(dev, dev->allocationBlockFinder);

                        if (bi->blockState == YAFFS_BLOCK_STATE_EMPTY) {
                                bi->blockState = YAFFS_BLOCK_STATE_ALLOCATING;
                                dev->sequenceNumber++;
                                bi->sequenceNumber = dev->sequenceNumber;
                                dev->nErasedBlocks--;
                                T(YAFFS_TRACE_ALLOCATE,
                                  (TSTR("Allocated block %d, seq  %d, %d left" TENDSTR),
                                   dev->allocationBlockFinder, dev->sequenceNumber,
                                   dev->nErasedBlocks));
                                return dev->allocationBlockFinder;
                        }
                }

                T(YAFFS_TRACE_ALWAYS,
                  (TSTR
                   ("yaffs tragedy: no more erased blocks, but there should have been %d"
                    TENDSTR), dev->nErasedBlocks));

                return -1;
}
```

Listing 3.2: Function `yaffs_FindBlockForAllocation` (Excerpt from yaffs_guts.c)

Along with the block sequence number, several other important values are written to a page's OOB area. All OOB area tags used by YAFFS2 can be seen in Table 3.1. Other meta data such as timestamps are not written to a header chunk's OOB area, but into the data area of the chunk.

Field	Size for 1024 bytes chunks	Size for 2048 bytes chunks
`blockState`	1 byte	1 byte
`chunkID`	4 bytes	4 bytes
`objectID`	4 bytes	4 bytes
`nBytes`	2 bytes	2 bytes
`blockSequence`	4 bytes	4 bytes
`tagsEcc`	3 bytes	3 bytes
`ecc` [1]	12 bytes	24 bytes

[1] The size of this tag depends on page size. `ecc` has 3 bytes of size per 256 bytes of page size

Table 3.1.: YAFFS2 OOB area tags according to YAFFS2's official specification [5]

During our analysis of YAFFS2's actual behavior, it became obvious that, although all tags shown in Table 3.1 were actually written to the OOB areas, the actual size of the `nByte` tag did not necessarily always match the size stated in Table 3.1. The reasons for this discrepancy and the actual meanings of YAFFS2's OOB tags are provided below. It is important to know, that the actual order in which all aforementioned tags are written to the OOB areas of a NAND is not controlled by YAFFS2 itself but by the NAND flash memory driver used by the operating system to access the flash memory chip [15]. Because of that, it is difficult to perform a fully automated analysis of a NAND dump.

blockState

The `blockState` tag is used to mark bad blocks by writing a value different than `0xff` to the respective byte within a bad block's pages' OOB areas.

objectID

The `objectID` tag contains the unique object number of the object a chunk is associated with. As depicted in Listing 3.3, certain values are reserved for special *pseudo* objects, such as the root directory of a device or the `lost+found` folder. The `lost+found` directory is not written to a NAND. Instead, it is created in RAM on mounting of the NAND.

```
[...]
/* Some special object ids for pseudo objects */
#define YAFFS_OBJECTID_ROOT             1
#define YAFFS_OBJECTID_LOSTNFOUND       2
#define YAFFS_OBJECTID_UNLINKED         3
#define YAFFS_OBJECTID_DELETED          4
[...]
```

Listing 3.3: Possible values for the `objectID` tag (Excerpt from yaffs_guts.h)

The `unlinked` and `deleted` objects are used by YAFFS2's to perform object deletions. Their meaning is provided in Section 3.2.3. All non-pseudo objects have an `objectID` tag value of at least 257. In Listing 3.4, the procedure to assign an object number to a newly created object is depicted, showing that no object numbers below 257 can be assigned to objects other than the aforementioned pseudo objects. This is because the variable `bucketFinder` is of type `unsigned integer` (see line 693 of `yaffs_guts.h`) and thus has a minimal value of zero. As the variable `bucketFinder` is incremented at least once inside of function `yaffs_FindNiceObjectBucket`, the variable `bucket` in line 1365 of Listing 3.4 has an initial value of at least one. Thus, as `YAFFS_NOBJECT_BUCKETS` has a value of 256 (see line 61 of `yaffs_guts.h`), the lowest object number available to a non-pseudo object is 257.

```
1353 static int yaffs_FindNiceObjectBucket(yaffs_Device *dev)
1354 {
1355         int i;
1356         int l = 999;
1357         int lowest = 999999;
1358 
1359 
1360         /* Search for the shortest list or one that
1361          * isn't too long.
1362          */
1363 
1364         for (i = 0; i < 10 && lowest > 4; i++) {
1365                 dev->bucketFinder++;
1366                 dev->bucketFinder %= YAFFS_NOBJECT_BUCKETS;
1367                 if (dev->objectBucket[dev->bucketFinder].count < lowest) {
1368                         lowest = dev->objectBucket[dev->bucketFinder].count;
1369                         l = dev->bucketFinder;
1370                 }
1371 
```

```c
        }

        return 1;
}

static int yaffs_CreateNewObjectNumber(yaffs_Device *dev)
{
        int bucket = yaffs_FindNiceObjectBucket(dev);

        /* Now find an object value that has not already been taken
         * by scanning the list.
         */

        int found = 0;
        struct ylist_head *i;

        __u32 n = (__u32) bucket;

        /* yaffs_CheckObjectHashSanity();   */

        while (!found) {
                found = 1;
                n += YAFFS_NOBJECT_BUCKETS;
                if (1 || dev->objectBucket[bucket].count > 0) {
                        ylist_for_each(i, &dev->objectBucket[bucket].list) {
                                /* If there is already one in the list */
                                if (i && ylist_entry(i, yaffs_Object,
                                                hashLink)->objectId == n) {
                                        found = 0;
                                }
                        }
                }
        }

        return n;
}
```

Listing 3.4: Assignment of object numbers (Excerpt from yaffs_guts.c)

In a data chunk, the `objectID` tag contains just the object number of the object the data chunk is associated with. However, in a header chunk, the `objectID` is not only used to associate the chunk with an object, but also to identify the type of the object. As can be seen in line 83 and 84 of Listing 3.5, the highest byte of the `objectID` field is used for that purpose. In YAFFS2's documentation, the extended use of the `objectID` tag is mentioned briefly, but no indication of YAFFS2 using extended tags by default is given. During our analyses we detected the values depicted in Table 3.2 being used in the `objectID` field's highest byte. For example, a file with object number 300 has `0x1000012c` as value for its header chunk's `objectID` tag, whereas a directory with object number 300 would have `0x3000012c` as value for its header chunk's `objectID` tag. The object type is also written to Byte 0 of the header chunk's data area. That way, YAFFS2 is able to identify the type of an object, regardless of extended tags being used or not.

```c
28 /* Extra flags applied to chunkId */
29
30 #define EXTRA_HEADER_INFO_FLAG  0x80000000
31 #define EXTRA_SHRINK_FLAG       0x40000000
32 #define EXTRA_SHADOWS_FLAG      0x20000000
33 #define EXTRA_SPARE_FLAGS       0x10000000
34
35 #define ALL_EXTRA_FLAGS         0xF0000000
36
```

```c
/* Also, the top 4 bits of the object Id are set to the object type. */
#define EXTRA_OBJECT_TYPE_SHIFT (28)
#define EXTRA_OBJECT_TYPE_MASK  ((0x0F) << EXTRA_OBJECT_TYPE_SHIFT)
[...]

void yaffs_PackTags2TagsPart(yaffs_PackedTags2TagsPart *ptt,
                const yaffs_ExtendedTags *t)
{
        ptt->chunkId = t->chunkId;
        ptt->sequenceNumber = t->sequenceNumber;
        ptt->byteCount = t->byteCount;
        ptt->objectId = t->objectId;

        if (t->chunkId == 0 && t->extraHeaderInfoAvailable) {
                /* Store the extra header info instead */
                /* We save the parent object in the chunkId */
                ptt->chunkId = EXTRA_HEADER_INFO_FLAG
                        | t->extraParentObjectId;
                if (t->extraIsShrinkHeader)
                        ptt->chunkId |= EXTRA_SHRINK_FLAG;
                if (t->extraShadows)
                        ptt->chunkId |= EXTRA_SHADOWS_FLAG;

                ptt->objectId &= ~EXTRA_OBJECT_TYPE_MASK;
                ptt->objectId |=
                    (t->extraObjectType << EXTRA_OBJECT_TYPE_SHIFT);

                if (t->extraObjectType == YAFFS_OBJECT_TYPE_HARDLINK)
                        ptt->byteCount = t->extraEquivalentObjectId;
                else if (t->extraObjectType == YAFFS_OBJECT_TYPE_FILE)
                        ptt->byteCount = t->extraFileLength;
                else
                        ptt->byteCount = 0;
        }

        yaffs_DumpPackedTags2TagsPart(ptt);
        yaffs_DumpTags2(t);
}
```

Listing 3.5: Extension of header chunks' tags (Excerpt from yaffs_packedtags2.c)

Object type	Highest byte of the `objectID` tag value
File	0x10
Soft link	0x20
Directory	0x30
Hard link	0x40
Special object (e.g. a pipe)	0x50

Table 3.2.: Possible values of the `objectID` tag in header chunks for different object types

chunkID

Basically, the `chunkID` tag defines a chunk's position within an object. In a data chunk, the value of the `chunkID` tag defines the chunk's offset from the beginning of an object, e.g. the data chunk with a `chunkID` tag value of four is the fourth data chunk of its object and the data chunk with a `chunkID` tag value of one is the first data chunk of its object. Regarding header chunks, YAFFS2's documentation claims that *a chunkId of zero signifies that this chunk contains an objectHeader* [2]. However, during our analyses, no chunks with a `chunkID` tag value of zero could be found. Instead, we observed, that the value of a header chunk's `chunkID` tag is constituted of two parts, namely an additional flag in the top byte of the `chunkID` tag and the object number of the object's parent directory in the other three bytes of the `chunkID` tag. As can be seen in Listing 3.5 and Listing 3.6, the reason for that is, that YAFFS2 uses its extended tags functionality. In doing so, YAFFS2 writes at least `0x80` to the top byte of a header chunk's `chunkID` tag and the the object number of the the header chunk's object's parent directory to the remaining three bytes of the `chunkID` tag. The object number of an object's parent directory is also stored in Bytes 4 to 7 of the object's header chunk's data area to enable YAFFS2 to identify the object's parent directory regardless of extended tags being used or not.

If a file's header chunk is marked with a *shrink header marker* to indicate a hole inside the file (see Section 3.2.1), the `chunkID` tag's top byte is set to `0xC0` by an OR-operation of `0x80` and `0x40`. YAFFS2 uses its extended tag functionality to speed up scanning of a device by writing more information into tags than absolutely necessary. Although writing the value zero into the `chunkID` tag would be enough to identify a header chunk, writing `0x80` into the highest byte of the tag and the object's parent directory to the rest of the tag speeds up identifying the directory structure of a device.

```
typedef struct {

        unsigned validMarker0;
        unsigned chunkUsed;     /*  Status of the chunk: used or unused */
        unsigned objectId;      /* If 0 then this is not part of an object (unused)
            */
        unsigned chunkId;       /* If 0 then this is a header, else a data chunk */
        unsigned byteCount;     /* Only valid for data chunks */

        /* The following stuff only has meaning when we read */
        yaffs_ECCResult eccResult;
        unsigned blockBad;

        /* YAFFS 1 stuff */
        unsigned chunkDeleted;  /* The chunk is marked deleted */
        unsigned serialNumber;  /* Yaffs1 2-bit serial number */

        /* YAFFS2 stuff */
        unsigned sequenceNumber;        /* The sequence number of this block */

        /* Extra info if this is an object header (YAFFS2 only) */

        unsigned extraHeaderInfoAvailable;      /* There is extra info available if
            this is not zero */
        unsigned extraParentObjectId;   /* The parent object */
        unsigned extraIsShrinkHeader;   /* Is it a shrink header? */
        unsigned extraShadows;          /* Does this shadow another object? */

        yaffs_ObjectType extraObjectType;       /* What object type? */

        unsigned extraFileLength;               /* Length if it is a file */
        unsigned extraEquivalentObjectId;       /* Equivalent object Id if it is a
            hard link */

        unsigned validMarker1;

} yaffs_ExtendedTags;
```

Listing 3.6: Extended tags structure (Excerpt from yaffs_guts.h)

blockSequence

The `blockSequence` tag is used to determine the chronological order in which chunks have been written to the NAND. Every time a new block is allocated for writing, the block sequence number is incremented and every chunk written to the block is tagged with this block sequence number. Thus, obsolete chunks can easily be detected and discarded. If two chunks with the same object numbers and chunk numbers can be found in different blocks, the chunk with the higher value in its `blockSequence` tag field is the most recent version. As can be seen in Listing 3.7, the value of the `blockSequence` tag can range from 4096 to 4 026 531 584. As depicted in Listing 3.2, a device's sequence number is incremented before a new block is allocated. Thus, the first block allocated by YAFFS2 on a NAND has the sequence number 4097.

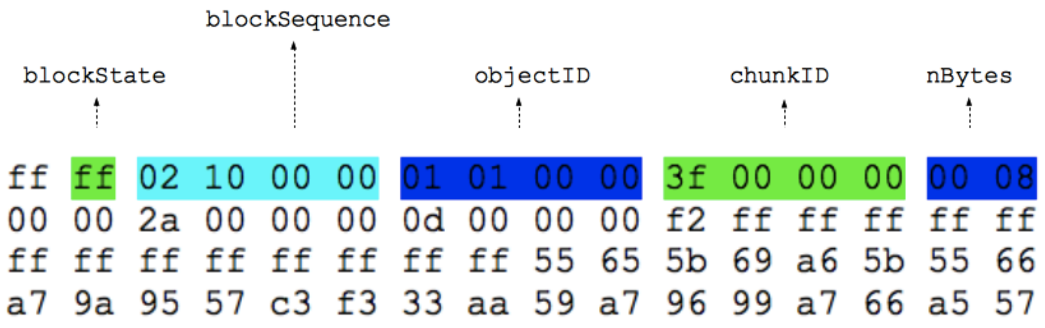
```
#define YAFFS_LOWEST_SEQUENCE_NUMBER     0x00001000
#define YAFFS_HIGHEST_SEQUENCE_NUMBER    0xEFFFFF00
```

Listing 3.7: Possible values for the `blockSequence` tag (Excerpt from yaffs_guts.h)

nBytes

According to YAFFS2's specification [5], the two byte `nBytes` tag tells the number of data bytes of a chunk. However, during the analysis of YAFFS2, we discovered, that the `nBytes` tag can have several different meanings and sizes. This is the result of YAFFS2's use of its extended tags.

In the OOB area of a data chunk, the value of the `nBytes` tag defines how many bytes of data are contained within the chunk. In Figure 3.2, the OOB area of a 2048 bytes NAND page containing a data chunk is depicted. The `nBytes` value of `0x0800` shows, that this chunk is completely in use and holds 2048 bytes of data content. As the size of a chunk typically matches the size of NAND flash memories' pages, two bytes are sufficient for the `nBytes` tag of a data chunk.

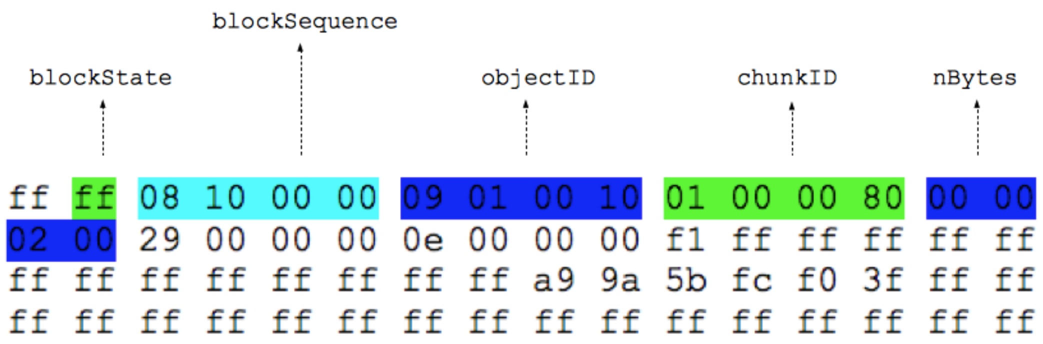


```
ff ff 02 10 00 00 01 01 00 00 3f 00 00 00 00 08
00 00 2a 00 00 00 0d 00 00 00 f2 ff ff ff ff ff
ff ff ff ff ff ff ff ff 55 65 5b 69 a6 5b 55 66
a7 9a 95 57 c3 f3 33 aa 59 a7 96 99 a7 66 a5 57
```

Figure 3.2.: OOB area of a data chunk

In the OOB area of a header chunk, the meaning of the `nBytes` tag depends on the type of the object associated with the header chunk. As depicted in Listing 3.5 and

Listing 3.6, in the OOB area of a file header chunk, the `nBytes` tag stores the size of the file associated with the header chunk. As the size of a file can easily exceed the maximum value of 64 KB, two bytes are not sufficient to store a file's size. Therefore, as can be seen in Figure 3.3, four bytes are used for the `nBytes` tag of a file header chunk. The size of the file associated with the file header chunk depicted in Figure 3.3 amounts to 128 KB. Therefore, the value `0x00020000` is stored in the chunk's `nBytes` tag field.



```
ff ff 08 10 00 00 09 01 00 10 01 00 00 80 00 00
02 00 29 00 00 00 0e 00 00 00 f1 ff ff ff ff ff
ff ff ff ff ff ff ff ff a9 9a 5b fc f0 3f ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
```

Figure 3.3.: OOB area of a file header chunk

In the header chunk of a hard link, the `nBytes` tag serves a completely different purpose. As a hard link just links another name to an existing file and does not have any data chunks, it does not need a `nByte` tag itself. Therefore the OOB area bytes that would normally contain the `nByte` flag, can be used for other information. As can be seen in line 87 and 88 of Listing 3.5, these bytes are used to store the object number of the object the hard link links to.

A soft link does also not feature any data chunks and consists only of one header chunk. In this header chunk, as can be seen in line 92 of Listing 3.5, the `nBytes` tag is not used and always contains the value zero. This is because a soft link uses the data area of the header chunk and not the chunk's OOB area for linking information. In figure 3.4, a chunk storing a soft link named *softlink2* is depicted. The soft link *softlink2* links to a file named *testFile1* that is stored in a directory named *aFolder*. As can be seen, the data area of a soft link's header chunk contains the absolute path of the object the soft link links to. The object type (`0x20`, marked blue) can be seen in Byte 0 of the chunk's data area as well as in the `objectID` tag in the chunk's OOB area. As can be seen in Bytes 4 to 7 (marked green) of the data area as well as in the `chunkID` tag in the OOB area, this soft link is stored in a directory with object number 1. As this object number is reserved for the root directory of a device, the soft link *softlink2* must be stored in the root directory.

tagsEcc and ecc

As NAND flash memory is prone to errors like bit flipping, error correction is necessary. Information regarding error correction is stored in the `tagsEcc` and `ecc` OOB area tags, with `tagsEcc` containing error correction information regarding a chunk's OOB area tags and `ecc` containing error correction information regarding a chunk's data contents.

```
02 00 00 00 01 00 00 00 ff ff 73 6f 66 74 6c 69  |..........softli|
6e 6b 32 00 00 00 00 00 00 00 00 00 00 00 00 00  |nk2.............|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 ff ff ff a1 00 00  |................|
00 00 00 00 00 00 00 00 6c 73 6e 4d 6c 73 6e 4d  |........lsnMlsnM|
6c 73 6e 4d ff ff ff ff ff ff ff ff 61 46 6f 6c  |lsnM........aFol|
64 65 72 2f 74 65 73 74 46 69 6c 65 31 00 00 00  |der/testFile1...|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  |................|
ff ff ff ff ff ff ff ff 00 00 00 00 ff ff ff ff  |................|
ff ff ff ff ff ff ff ff 00 00 00 00 00 00 00 00  |................|
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  |................|
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  |................|
                          ...
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  |................|
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff  |................|

ff ff 01 01 00 00 06 01 00 20 01 00 00 80 00 00
00 00 33 00 00 00 00 00 00 00 00 00 00 00 ff ff
ff ff ff ff ff ff ff ff fc c0 f3 f0 03 0f ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
```

Figure 3.4.: Data area and OOB area of a soft link header chunk

Shrink header markers

Shrink header markers are YAFFS2's way to handle files with a hole. Such files occur when a file is truncated to a size smaller than its original size and a subsequent write operation on that file starts writing at an offset beyond the truncated size. As shrink header markers have an important influence on garbage collection, YAFFS2's way to handle files with holes are introduced in the following.

According to the POSIX standard, a hole inside a file should always read back as bytes of value zero [16]. As YAFFS2 does not rewrite already written chunks, YAFFS2 needs another way to create a hole inside a file and fill it with zeros. Additionally, data chunks that are located inside the hole after a truncation and a write operation have to be marked as obsolete. Depending on the size of a hole, YAFFS2 chooses one of two ways to handle files with a hole.

In case the hole is a hole smaller than four chunks (see lines 32 and 807 to 890 of `yaffs_yaffs2.c`) YAFFS2 actually writes zeros to the NAND to indicate the hole inside a file. This can be seen in Table 3.3.[1] The file *file.Hole* depicted in this NAND dump has been put through the following:

1. Writing of 15 000 'a' to the file, leading to a file size of 15 000 bytes
2. Truncation of the file to 1000 bytes
3. Writing of 3000 'b' at position 9191 of the file, leading to a hole of 8191 bytes and a new file size of 12 191 bytes

To represent the truncation of a file to a smaller size, YAFFS2 writes the data chunk that contains the new end of the file after the truncation to the NAND, followed by a file header chunk with the truncated size in its `nByte` OOB area tag. To represent the hole in the file, YAFFS2 then writes a respective number of chunks filled with zeros to the NAND, followed by chunks containing the file's content beyond the hole.

In case the hole inside a file is bigger than four chunks, YAFFS2 does not write chunks completely filled with zeros to the NAND but makes use of its so-called shrink header markers to indicate a hole inside a file. As can be seen in Table 3.4, this saves a lot of space on the NAND.[2] The file *file.Hole2* depicted in this NAND dump has been put through the following:

1. Writing of 15 000 'a' to the file, leading to a file size of 15 000 bytes
2. Truncation of the file to 1000 bytes
3. Writing of 3000 'b' at position 9192 of the file, leading to a hole of 8192 bytes (equivalent to four chunks of 2048 bytes) and a new file size of 12 192 bytes.

The header chunk marked with a shrink header marker, recognizable by its `chunkID` OOB area tag's highest byte's value of `0xC0`, marks the end of the hole inside the file and the data chunk written due to the truncation of the file (see chunk no. 16 in Table

[1] On the attached DvD, see also: /Chapter3.2/nanddump.smallHole.pretty
[2] On the attached Dvd, see also: /Chapter3.2/nanddump.bigHole.pretty

Chunk no.	Content	chunkID	objectID	nBytes
...	...	...	...	...
8	file.Hole: Data (2048 'a')	1	258	2048
9	file.Hole: Data(2048 'a')	2	258	2048
10	file.Hole: Data (2048 'a')	3	258	2048
11	file.Hole: Data (2048 'a')	4	258	2048
12	file.Hole: Data (2048 'a')	5	258	2048
13	file.Hole: Data (2048 'a')	6	258	2048
14	file.Hole: Data (2048 'a')	7	258	2048
15	file.Hole: Data (664 'a', 1384 '0')	8	258	664
16	file.Hole: Data (1000 'a', 1048 '0')	1	258	1000
17	Header file.Hole	0x80000001	0x10000102	1000
18	Header file.Hole	0x80000001	0x10000102	1000
19	file.Hole: Data (1000 'a', 1048 '0')	1	258	2048
20	file.Hole: Data (2048 '0')	2	258	2048
21	file.Hole: Data (2048 '0')	3	258	2048
22	file.Hole: Data (2048 '0')	4	258	2048
23	file.Hole: Data (999 '0', 1049 'b')	5	258	2048
24	file.Hole: Data (1951 'b', 97 '0')	6	258	1951
25	file.Hole: Header	0x80000001	0x10000102	12191
...	...	...	...	...

Table 3.3.: Representation of a file with a hole smaller than four chunks on a NAND

3.4) the beginning of the hole. Therefore all of the file's chunks written before this chunk are obsolete. The size of the hole can be derived from the number of zeros written to the end respectively the beginning of the chunks enframing the hole and the gap between the `chunkID` tag values of the chunks enframing the hole.

Chunk no.	Content	chunkID	objectID	nBytes
...	...	...	...	...
8	file.Hole2: Data (2048 'a')	1	258	2048
9	file.Hole2: Data (2048 'a')	2	258	2048
10	file.Hole2: Data (2048 'a')	3	258	2048
11	file.Hole2: Data (2048 'a')	4	258	2048
12	file.Hole2: Data (2048 'a')	5	258	2048
13	file.Hole2: Data (2048 'a')	6	258	2048
14	file.Hole2: Data(2048 'a')	7	258	2048
15	file.Hole2: Data (664 'a', 1384 '0')	8	258	664
16	file.Hole2: Data(1000 'a', 1048 '0')	1	258	1000
17	file.Hole2: Header	0x80000001	0x10000102	1000
18	file.Hole2: Header	0x80000001	0x10000102	1000
19	file.Hole2: Header	0xC0000001	0x10000102	1000
20	file.Hole2: Data (1000 '0', 1048 'b')	5	258	2048
21	file.Hole2: Data (1952 'b', 96 '0')	6	258	1952
22	file.Hole2: Header	0x80000001	0x10000102	12192
...	...	...	...	...

Table 3.4.: Representation of a file with a hole of four chunks on a NAND

3.2.2. Meta data

During a forensic analysis of a storage device, the device's files' meta data can provide valuable information. YAFFS2 uses an object's header chunk to store meta data such as time stamps, permissions and ownership information. This meta data of an object is not stored within the object header chunk's OOB area, but in the chunk's data area. In the following, we introduce the way YAFFS2 stores meta data.

Time stamps

Time stamps can provide relevant information about a file's history, that is, information about when a file has been accessed or modified. Typically, such information can be obtained from a file's `mtime`, `atime` and `ctime` time stamps. In a Linux environment, `mtime` stores the time a file's content has been modified, `atime` stores the time a file has been accessed and `ctime` the time a file's inode has been modified. Modifications of a file's inode include modifications of a file's meta data as well as modifications of a file's content.

YAFFS2 stores meta data such as time stamps inside a file header chunk's data area. However, regarding time stamps, YAFFS2 does not follow the default Unix or Linux way of keeping track of file modification and access times. During our analysis, we observed that YAFFS2 does store three time stamps inside an object's header chunk but these time stamps do not completely match classic Unix `mtime`, `atime` and `ctime` time stamps. Although YAFFS2 does store `ctime` and `mtime`, it does not store `atime` on a NAND. Instead of `atime` it stores a time stamp containing the time of the object's creation. On a simulated NAND as well as on a *HTC Magic* smartphone running Android OS 2.2, `mtime` could be found in Bytes 284 to 287 of an object's header chunk's data area, `ctime` could be found in Bytes 288 to 291 and the object's creation time in Bytes 280 to 283 of an object's header chunk's data area.

The reason for YAFFS2 waiver of writing `atime` to a NAND lies in wear considerations. Keeping track of an object's access times would require writing of a new object header chunk every time the object has been accessed. This would lead to an enormous amount of object header chunks being written to the NAND which would lead to a drastic increase of block erasures and thus increased wear out of NAND flash memory. Because of that, by default, YAFFS2 does not write `atime` to a NAND [17].

Permissions and owner

In a Linux environment, every object, such as a file or a directory, has a user and a group that own the object. Read, write and execution permissions are assigned separately to the object's owner, the object's group and to all users that are neither the object's owner nor part of the group owning the object. Hence, for every object, the object's owner and group have to be stored along with the permissions granted to the owner, group and all other users.

On a simulated NAND as well as on a *HTC Magic* smartphone running Android OS 2.2, an object's permissions could be found in Bytes 268 to 271 of the object's header chunk's data area. The object's owner's UID (*User ID*) could be found in Bytes 272 to 275 of the object's header chunk's data area and the object's group's GID (*Group ID*) in Bytes 276 to 279.

3.2.3. Object modifications and deletions

As above-mentioned, YAFFS2 does not overwrite already written chunks. Thus, modification of an object's content or deletion of an object can not be performed by direct modification or deletion of the object's chunks on a NAND. Instead, existing chunks have to be marked as obsolete without overwriting these chunks on the NAND. In the following, we introduce YAFFS2's techniques to create, modify and delete objects. A practical evaluation of these techniques is provided in Chapter 5 of this diploma thesis.

Object creation

When creating an object, YAFFS2 first creates the object in RAM and then writes the object's chunks to the NAND. When writing the object's chunks to the NAND, YAFFS2 uses different writing patterns depending on the object's type. Hard links, soft links and directories do not feature any data chunks, but only consist of an object header chunk. When creating such an object on a NAND, YAFFS2 thus only writes the respective object header chunk. Additionally, a header chunk for the directory in which the newly created object is located in is written to the NAND to store the directory's changed meta data.

When creating a file, YAFFS2 not only has to write a file header chunk but also at least one data chunk to a NAND. To create the file on a NAND, YAFFS2 first writes a file header chunk with a `nByte` OOB area tag value of zero to indicate creation of an empty file. Subsequently, the file's data chunks are written to the NAND, followed by a file header chunk containing the actual file size. This way to create files on a NAND thus always leads to writing of a file header chunk that gets obsolete as soon as all the file's data chunks and the second file header chunk are written to the NAND. Finally, as with all other object types, a header chunk for the directory in which the newly created file is located in is written to the NAND to store the directory's changed meta data.

Object modifications

As above-mentioned, YAFFS2 keeps information about all objects stored in a RAM structure. This RAM structure does not only contain general information about an object, such as the object's type, name or object number, but also the object's *tnode tree*. Regarding modifications of objects, this tnode tree plays an important role. An object's tnode tree is used to map file positions to chunks on a NAND and thus is needed to determine which chunks need to be replaced in the course of an object's modification. When executing a modification of an object, YAFFS2 first modifies the object's meta information and tnode tree in RAM and then writes the respective modifications to the NAND.

YAFFS2 objects can be modified in several ways. Basically, there are two main categories of object modifications. Firstly, an object can be modified by modification of its meta data, such as permissions or time stamps. Secondly, objects can be modified by modification of their content. At that, the object's size can either stay the same, increase or decrease. These different cases of object modifications are handled differently by YAFFS2.

In case a modification only modifies an object's meta data, for example by use of `chmod`, only the object's header chunk is affected as no actual contents within the object's data chunks are modified. Thus, YAFFS2 only has to modify the object's header chunk. As YAFFS2 cannot overwrite the header chunk, YAFFS2 instead writes a new object header chunk containing the new meta data information. The new object header chunk contains the new meta data information and the same OOB area tags the old and now obsolete object header chunk features. As chunks and blocks are allocated for writing sequentially, YAFFS2, when scanning the device, discovers the new object header chunk before discovering the obsolete object header chunks. Thus, YAFFS2 recognizes which object header chunk has to be the current object header chunk and ignores the obsolete ones.

In case a modification changes an object's content, chunks containing the old content have to be marked as obsolete and new chunks containing the current content have to be written to the NAND. Additionally, a new object header chunk has to be written, as a modification of an object's content always leads to modification of an object's meta data. As only files contain data chunks, in the following, we only take modifications of files into consideration. In case modification of a file does not change the file's size, but only replaces parts of its contents, YAFFS2 needs to replace the affected data chunks. For this purpose, YAFFS2 determines which chunks have to be replaced by use of the file's tnode tree. The affected data chunks are then written to the NAND, containing the new content and the same OOB area tags the old and now obsolete data chunks feature. Additionally, a new file header chunk is written to store the file's modified meta data information and to indicate the modification. When scanning the NAND, YAFFS2 discovers the new file header chunk and new data chunks before discovering the obsolete file header and data chunks and thus considers the first discovered chunks current.

In case a modification changes a file's content in a way that changes the object's size, YAFFS2 additionally has to check whether the modification puts a hole into the file. Additionally, the `nByte` OOB area tag of the file header chunks has to be updated and, if necessary, also the `nByte` and `chunkID` OOB area tags of some data chunks. In case a modification puts a hole into a file, YAFFS2 needs to represent the hole on the NAND. YAFFS2's techniques to do so are described above in Section 3.2.1. In case a modification changes a file's size without resulting in a file with a hole, YAFFS2 does not have to write shrink header markers but has to consider whether the modification changes the file's data chunk count. Additionally, in case a modification that changes a file's size does not occur at the end of a file, the file positions of the data chunks located behind the point of modification have to be updated. Thus, the `chunkID` OOB area tags of these chunks have to be updated too.

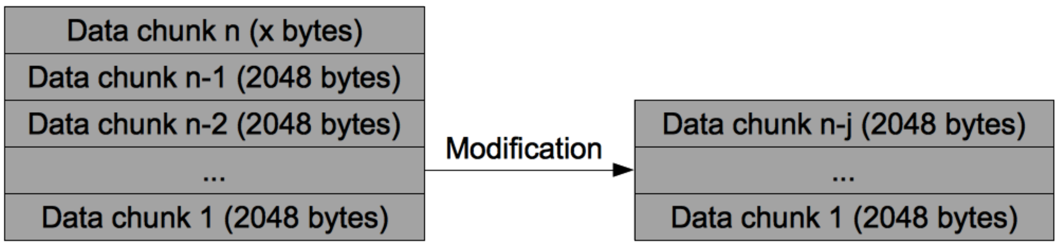

Data chunk n (x bytes)
Data chunk n-1 (2048 bytes)
Data chunk n-2 (2048 bytes)
...
Data chunk 1 (2048 bytes)
Modification
Data chunk n-j (2048 bytes)
...
Data chunk 1 (2048 bytes)

Figure 3.5.: Modification at the end of a file that decreases the file size to a multiple of a chunk's size

In case a modification of a file occurs at the end of a file, either only the file's last data chunk or several data chunks at the end of the file are affected. If the modification is small enough only to change the last data chunk without changing the file's chunk count, YAFFS2 only has to rewrite the last data chunk. Thus, a new data chunk containing the new content is written to the NAND, featuring the same OOB area tags as the old last data chunk of the file with the only difference that the new data chunk's `nByte` OOB area tag has to contain the new number of bytes within the data chunk. Additionally, a new file header chunk featuring the new file size in its `nByte` OOB area tag has to be written. If a modification at the end of a file changes the file's data chunk count, YAFFS2 has to differentiate between modifications that increase the file's size and modifications that decrease the file's size. A modification that decreases a file's size and its data chunk count can either lead to a decrease that leads to a new end of file that corresponds to the end of one of the file's data chunks or to a new end of file that is located within one of the file's data chunks. An end of file that corresponds to the end of one of the file's data chunks is given, if the new file size matches the multiple of a chunk's size. For example, as depicted in Figure 3.5, on a NAND with 2048 bytes per page, the new end of file after a file's modification corresponds to the end of one of the file's data chunks if the new file size is a multiple of 2048 bytes. Such a modification does not affect the content of the file's new last data chunk. Thus, as such a modification only truncates the file and does not affect the file's new last data chunk's contents, YAFFS2 only needs to write a new file header chunk containing the new file size in its `nByte` OOB tag. When scanning the device, YAFFS2 then considers all data chunks that lie behind the file's new last data chunk obsolete as, based on the file size given in the file's header chunk, they cannot be part of the file.

In case a modification at the end of a file leads to a decrease in file size that does not lead to the new end of file corresponding to the end of one of the file's data chunks, the contents of the file's new last data chunk is affected by the modification. Thus, in this case, YAFFS2 needs to rewrite the file's new last data chunk with new content and updated OOB area tags. Therefore, a new data chunk is written to the NAND, containing the new content and the same OOB area tags as the affected data chunk, except for the `nByte` tag that now features a smaller value. Additionally, a new file header chunk featuring the new file size in its `nByte` OOB area tag is written to the NAND.

Modifications at the end of a file that increase the file's size and the file's chunk count by definition append content to the end of the file. Thus, such a modification requires at least writing of one new chunk containing the new content and, if necessary, rewriting of some of the file's old chunks. Simply appending content to a file only requirers rewriting of the file's old last chunk if this chunk was not completely filled before the modification took place and thus parts of the content appended to the file are to be located inside this chunk. In this case, YAFFS2 writes a data chunk containing the file's old last chunk's content plus as much of the appended content to completely fill the chunk on the NAND. This chunk's OOB area tag values match the file's old last data chunk's tag values except for the `nByte` tag that now features a value representing a completely filled chunk. The remaining content to be appended to the file is written to additional new data chunks whose `chunkID` OOB area tags contain values placing them at the end of the file. If the modification does not only append content to the file but also replaces content located in several consecutive data chunks at the end of the file, this data chunks have to be rewritten too. Additionally, a new file header chunk containing the new file size in its `nByte` OOB area tag is written to the NAND.

Obviously, modifications of a file's content do not only occur at the end of a file, but also at other positions of a file. Such modifications require more of the file's data chunks to be rewritten as not only the data chunks whose content actually changes get modified but also the positions of all data chunks behind the point of the modification. This requires updating the `chunkID` OOB area tag of these chunks which again requires rewriting these chunks. If such a modification leads to hole in the file, the modification is handled by YAFFS2 as described in Section 3.2.1. In case a modification of a file's content at a position other than the end of the file does not change the file's size and causes no hole in the file, only the data chunks affected by the modification have to be rewritten. YAFFS2 thus writes new data chunks containing the new content and featuring the same OOB area tags as the replaced chunks to the NAND. Additionally, YAFFS2 writes a new file header chunk. If the modification also changes the file's size, also all data chunks located behind the point of the modification are rewritten. This is because at least some of them feature new contents after the modification and all of them require updating of their `chunkID` OOB area tag.

Practical evaluation of YAFFS2's behavior when performing file modifications is provided in Chapter 5 of this diploma thesis.

Object deletions

As mentioned in Section 3.2.1 of this diploma thesis, YAFFS2 uses the pseudo objects `unlinked` and `deleted` to perform deletion of objects. These pseudo objects represent virtual directories that are not actually written to a NAND. When a link to an object is deleted, the link is virtually moved to the `deleted` directory. If an object is deleted and no links to the object exist anymore on the device, the object is virtually moved to the `unlinked` and `deleted` directory. The reason for that way of marking objects as deleted is YAFFS2's zero overwrite policy that prevents YAFFS2 from overwriting already written chunks without erasing the chunks' respective blocks beforehand. In the following, we introduce YAFFS2's way to mark objects as deleted. A practical evaluation of YAFFS2's behavior when performing delete operations is provided in Chapter 5 of this diploma thesis.

As above-mentioned, the `unlinked` and `deleted` directories do not actually exist on a NAND. Therefore, no object header chunks for these objects are written to the NAND. Hence, to move an object to the `deleted` or `unlinked` directory, a special `unlinked` or `deleted` object header chunk for the deleted or unlinked object is written to the NAND. After writing of a `deleted` or an `unlinked` header chunk, a directory header chunk for the directory containing the deleted object is also written to the NAND to update the directory's time stamps. Although the `unlinked` and `deleted` header chunks resemble regular object header chunks in many ways, there are some differences. In Table 3.5 the way YAFFS2 uses a `deleted` header chunk's OOB area tags is depicted. All OOB area tags not listed in Table 3.5 are used as they are in the deleted object's header chunk.

As shown in Section 3.2.1 and depicted in Listing 3.3, object number 4 is reserved for the `deleted` directory. Thus, as can be seen in Table 3.5, a `deleted` header chunk resembles the object header chunk of an object located inside the `deleted` directory. However, some differences are discernible. The `deleted` header chunk's `chunkID` OOB area tag is also marked with a shrink header marker. Although shrink header markers are

OOB area tag	**content**
`objectID`	value of the deleted object's `objectID` OOB area tag
`chunkID`	`0xc0000004`
`nBytes`	"0" or the deleted object's object number

Table 3.5.: Usage of OOB area tags in `deleted` header chunks

typically used to mark files with a hole in them, they are also used to indicate deletion of an object. Thus, deletion of an object can also be seen as a resizing of the object to a size of zero bytes. The `nByte` OOB area tag of a `deleted` header chunk does either contain the value zero or an object number. The `nByte` OOB area tag value is always set to zero, if an object gets deleted and no links to the object are present on the device anymore. If a hard link gets deleted and the object it links to is still present on the device, the hard link's `deleted` header chunk's `nByte` OOB area tag features the object number of the object the hard link linked to.

Usually, an object header chunk features the object's name in the object header chunk's data area. Although a `deleted` header chunk features the same value in its `objectID` OOB area tag as the deleted object, the name stored in the `deleted` header chunk is always "deleted".

When all links to an object have been deleted, the object is not only virtually moved to the `deleted` directory but also to the `unlinked` directory to indicate the object's complete deletion. For example, if a file "*xFile*" and a hard link to this file exist on a device, deletion of the hard link leads to a `deleted` header chunk for the hard link being written to the NAND. If subsequently the file "*xFile*" gets deleted both an `unlinked` and a `deleted` header chunk for the file are written to the NAND. Further analysis of YAFFS2's behavior in handling deletion of hard links and the objects they link to is provided below.

OOB area tag	**content**
`objectID`	value of the unlinked object's `objectID` OOB area tag
`chunkID`	`0x80000003`
`nBytes`	0

Table 3.6.: Usage of OOB area tags in `unlinked` header chunks

As can be seen in Table 3.6, an `unlinked` header chunk's OOB area tag values strongly resemble a `deleted` header chunk's OOB area tag values. The main difference between `unlinked` and `deleted` header chunks lies in their different `chunkID` OOB area tag values. Additionally, an `unlinked` header chunk features the name "unlinked" instead of "deleted" in its data area. As shown in Section 3.2.1 and depicted in Listing 3.3, the virtual `unlinked` directory has the special object number 3. Thus, writing an `unlinked` header chunk for an object and thus virtually moving the object to the virtual `unlinked` directory requires the object number "3" being written to the `unlinked` header chunk's `chunkID` OOB area tag as the object number of the deleted object's parent directory.

As an `unlinked` header chunk is alway written to NAND flash memory together with a `deleted` header chunk, an `unlinked` header chunk does not need a shrink header marker.

Technically, a file name represents a link to a file object. If no link to this object other than the file name exists, deletion of the object is performed by writing `unlinked` and `deleted` header chunks for the object. But in case hard links to the file object exist on a device, deletion of the file does only delete the file name and not the underlying object as links to the object still exist on the device. The way YAFFS2 handles this problem is depicted in Table 3.7.[3]

Chunk no.	**Content**	**chunkID**	**objectID**	**nBytes**
0	testFile: Header	0x80000001	0x10000101	0
1	testFile: Data	1	257	2048
2	testFile: Header	0x80000001	0x10000101	2048
...	...	...	...	...
8	hardLink1: Header	0x80000001	0x40000104	257
...	...	...	...	...
10	hardLink2: Header	0x80000001	0x40000105	257
...	...	...	...	...
15	deleted: Header	0xC0000004	0x40000105	257
...	...	...	...	...
17	hardLink1: Header	0x80000001	0x10000101	2048
18	deleted: Header	0xC0000004	0x40000104	257
...	...	...	...	...
20	unlinked: Header	0x80000003	0x10000101	0
21	deleted: Header	0xC0000004	0x10000101	0
...	...	...	...	...

Table 3.7.: Deletion of a file with hard links on a YAFFS2 NAND

In Table 3.7, a dump of a NAND on which the following actions have been performed is depicted:

1. Creation of file *"testFile"* (object number 257)
2. Creation of hard link *"hardLink1"* to *"testFile"* (object number 260)
3. Creation of hard link *"hardLink2"* to *"testFile"* (object number 261)
4. Deletion of *"hardLink2"*
5. Deletion of *"testFile"*
6. Deletion of *"hardLink1"*

As can be seen in Table 3.7, deletion of *"hardLink2"* is performed as described above by writing a `deleted` header chunk for *"hardLink2"* to the NAND (see chunk no. 15 in Table 3.7). However, deletion of *"testFile"* does not result in writing of an `unlinked` or

[3]On the attached Dvd, see also: /Chapter3.2/nanddump.deletion.pretty

deleted header chunk for the file object but to writing of a new file header chunk for the file object with object number 257 and deletion of *"hardLink1"*. The reason for that is, that both the file name *"testFile"* and the hard link *"hardLink1"* link to the same file object and thus deletion of *"testFile"* can not result in actual deletion of the file object. Instead, only the link to the file object represented by the file name *"testFile"* is deleted. Thus, YAFFS2 writes a new file header chunk for the file object with object number 257 to the NAND, thereby renaming the file object to *"hardLink1"* (see chunk no. 17 in Table 3.7). As the name *"hardLink1"* is now the only remaining link to the file object, the hard link object with object number 260 has to be deleted by writing a deleted header chunk for this object (see chunk no. 18 in Table 3.7). Only after the file name *"hardLink1"* is deleted, the file object with object number 257 is finally actually deleted. As no links to the object exist anymore, it is deleted by writing an unlinked and a deleted header chunk for the object to the NAND (see chunk no. 20 and 21 in Table 3.7).

3.3. Garbage collection

To comply to NAND flash memory's demands, YAFFS2 never overwrites an already written chunk without deleting the chunk's block beforehand. Hence, chunks whose content becomes obsolete through modification or deletion of their respective objects remain stored on the NAND until the blocks containing these chunks are deleted. Obsolete chunks need to be deleted at some point to free up space on the device. While it is not a problem to delete a block that is completely filled with obsolete chunks, simply deleting a block filled with a mixture of obsolete and current chunks would lead to data loss. Thus, YAFFS2 needs a way to ensure that all current chunks within a block that is to be deleted are copied to another block before performing the deletion. Saving these current chunks is task of YAFFS2's garbage collector. From a forensic perspective, as obsolete chunks contain potential evidence, it is important to understand the way, YAFFS2's garbage collector performs its task. Therefore, in the following, we provide an in-depth analysis of the YAFFS2 garbage collector.

YAFFS2 distinguishes between two different modes of garbage collection, *aggressive* garbage collection and *passive* garbage collection [2]. Which of these modes is used is decided in the first of three steps of YAFFS2's garbage collection. YAFFS2's garbage collector's functionality is defined in files `yaffs_guts.c` and `yaffs_yaffs2.c` and features four main functions representing three major steps of garbage collection. These functions are:

- `yaffs_CheckGarbageCollection` (lines 2544 to 2632 of `yaffs_guts.c`, Listing 3.8)
- `yaffs_FindBlockForGarbageCollection` (lines 2379 to 2533 of `yaffs_guts.c`)
- `yaffs2_FindRefreshBlock` (lines 142 to 192 of `yaffs_yaffs2.c`),
- `yaffs_GarbageCollectBlock` (lines 2102 to 2372 of `yaffs_guts.c`)

Every garbage collection starts with a call of `yaffs_CheckGarbageCollection` as a first step in which necessity of garbage collection is determined. Execution of `yaffs_CheckGarbageCollection` results in either abortion of the current attempt of garbage collection or in a call of `yaffs_FindBlockForGarbageCollection` or `yaffs2_FindRefreshBlock` to find a block for garbage collection. After a suitable block for garbage collection has been found, actual garbage collection of this block is performed as third step of garbage collection by use of `yaffs_GarbageCollectBlock`.

Function `yaffs_CheckGarbageCollection` is called to check whether garbage collection is necessary every time one of the following functions is called:

- `yaffs_BackgroundGarbageCollect` (lines 2639 to 2647 of `yaffs_guts.c`)
- `yaffs_WriteChunkDataToObject` (lines 2936 to 2994 of `yaffs_guts.c`)
- `yaffs_updateObjectHeader` (lines 2999 to 2171 of `yaffs_guts.c`)
- `yaffs_ResizeFile` (lines 3792 to 3847 of `yaffs_guts.c`)

At least one of these functions is called every time a file system operation requires writing of data to the NAND. Thus, YAFFS2 checks necessity of garbage collection every time data has to be written to the NAND. That does not only include creation of objects, but also deletion or modification of existing objects. Additionally, YAFFS2 checks for necessity of garbage collection at fixed time intervals by use of function `yaffs_BackgroundGarbageCollect`. On our test system, we observed checks for necessity of garbage collection by `yaffs_BackgroundGarbageCollect` every two seconds. In case of a NAND that was never mounted before, the first check for necessity of garbage collection by `yaffs_BackgroundGarbageCollect` was performed directly after mounting of the NAND. In case the NAND had been mounted before, the first check for necessity of garbage collection by `yaffs_BackgroundGarbageCollect` was performed after the first write operation to the NAND. Whether garbage collection of a block is actually initiated after a check for necessity for garbage collection and, if it is, in which mode it is performed, depends on several factors. First of all, YAFFS2 checks, whether garbage collection is permanently or temporarily deactivated for a device (see line 2553 to 2560 of Listing 3.8). Garbage collection is temporarily deactivated during garbage collection of a block to prevent recursive garbage collection. As can be seen in Listing 3.8, steps two of garbage collection is always performed if aggressive garbage collection proves necessary. This is the case, if not enough free blocks are available to store a checkpoint. As defined in function `yaffs2_CalcCheckpointBlocksRequired` (lines 213 to 247 of `yaffs_yaffs2.c`) and line 2571 of `yaffs_guts.c`, at least n free blocks must be available for checkpoint data in order not to trigger aggressive garbage collection, with n having the following value:

$$\begin{aligned} n = {} & \text{number of reserved blocks} \\ & + \text{number of complete blocks actually necessary to store current checkpoint data} \\ & - \text{number of blocks currently used for checkpoint data} \\ & + 4 \end{aligned}$$

Detailed information on how to calculate the number of blocks actually necessary to store checkpoint data is provided in Section 4.3.

```
static int yaffs_CheckGarbageCollection(yaffs_Device *dev, int background)
{
        int aggressive = 0;
        int gcOk = YAFFS_OK;
        int maxTries = 0;
        int minErased;
        int erasedChunks;
        int checkpointBlockAdjust;

        if(dev->param.gcControl &&
                (dev->param.gcControl(dev) & 1) == 0)
                return YAFFS_OK;

        if (dev->gcDisable) {
                /* Bail out so we don't get recursive gc */
                return YAFFS_OK;
        }

        /* This loop should pass the first time.
         * We'll only see looping here if the collection does not increase space.
         */

        do {
                maxTries++;

                checkpointBlockAdjust = yaffs2_CalcCheckpointBlocksRequired(dev);

                minErased  = dev->param.nReservedBlocks + checkpointBlockAdjust + 1;
                erasedChunks = dev->nErasedBlocks * dev->param.nChunksPerBlock;

                /* If we need a block soon then do aggressive gc.*/
                if (dev->nErasedBlocks < minErased)
                        aggressive = 1;
                else {
                        if(!background && erasedChunks > (dev->nFreeChunks / 4))
                                break;

                        if(dev->gcSkip > 20)
                                dev->gcSkip = 20;
                        if(erasedChunks < dev->nFreeChunks/2 ||
                                dev->gcSkip < 1 ||
                                background)
                                aggressive = 0;
                        else {
                                dev->gcSkip--;
                                break;
                        }
                }

                dev->gcSkip = 5;

                /* If we don't already have a block being gc'd then see if we should
                     start another */

                if (dev->gcBlock < 1 && !aggressive) {
                        dev->gcBlock = yaffs2_FindRefreshBlock(dev);
                        dev->gcChunk = 0;
                        dev->nCleanups=0;
                }
                if (dev->gcBlock < 1) {
                        dev->gcBlock = yaffs_FindBlockForGarbageCollection(dev,
                            aggressive, background);
                        dev->gcChunk = 0;
                        dev->nCleanups=0;
                }

                if (dev->gcBlock > 0) {
                        dev->allGCs++;
                        if (!aggressive)
                                dev->passiveGCs++;

                        T(YAFFS_TRACE_GC,
                          (TSTR
                           ("yaffs: GC erasedBlocks %d aggressive %d" TENDSTR),
                           dev->nErasedBlocks, aggressive));

                        gcOk = yaffs_GarbageCollectBlock(dev, dev->gcBlock, aggressive);
                }

                if (dev->nErasedBlocks < (dev->param.nReservedBlocks) && dev->gcBlock > 0) {
                        T(YAFFS_TRACE_GC,
                          (TSTR
                           ("yaffs: GC !!!no reclaim!!! erasedBlocks %d after try %d block %d"
                            TENDSTR), dev->nErasedBlocks, maxTries, dev->gcBlock));
                }
        } while ((dev->nErasedBlocks < dev->param.nReservedBlocks) &&
                 (dev->gcBlock > 0) &&
                 (maxTries < 2));

        return aggressive ? gcOk : YAFFS_OK;
}

}
```

Listing 3.8: Function `yaffs_CheckGarbageCollection` (Excerpt from yaffs_guts.c)

If at least n free blocks are available to store current checkpoint data and for that reason no aggressive garbage collection has to be performed, YAFFS2 checks, whether passive garbage collection is necessary. Execution of passive garbage collection depends on two factors. These are:

1. the way the garbage collection check has been invoked
2. the number of free chunks within free blocks in relation to the total number of free chunks on the device

Regarding passive garbage collection, YAFFS2 distinguishes between garbage collection checks that have been caused by background threads and garbage collection checks that have been caused by foreground threads. In the following, garbage collection that is caused by a background thread is also referred to as *background garbage collection*. Typically, YAFFS2's periodical check for necessity of garbage collection is the main trigger for background garbage collection. As can be seen in Listing 3.8, once aggressive garbage collection proved unnecessary, passive garbage collection is performed unless the `if`-statement in line 2578 returns `true` or the `if`-statement in line 2583 returns `false`. If a check for necessity of garbage collection is caused by a background thread, variable `background` is set to value 1. This always leads to the `if`-statement in line 2578 returning `false` and the `if`-statement in line 2583 returning `true` and thus, given that aggressive garbage collection is not necessary, to execution of step two of passive garbage collection. If a check for necessity of garbage collection is caused by a foreground thread, variable `background` is set to value 0. This means, that the `if`-statement in line 2578 can only return `false` if the number of free chunks within free blocks does not exceed one quarter of the total number of free chunks on the device. This automatically leads to the `if`-statement in line 2583 returning `true`, as, if the number of free chunks within free blocks does not exceed one quarter of the total number of free chunks on the device, it cannot exceed half the total number of free chunks on the device. Hence, if a check for necessity of garbage collection is caused by a foreground thread and aggressive garbage collection is not necessary, execution of step two of passive garbage collection depends only on the ratio of free chunks within free blocks to the total number of free chunks on the device.

Thus, a check for necessity of garbage collection leads to further steps of garbage collection if garbage collection is not deactivated and at least one of the following conditions is met:

- shortage of free blocks would prevent storing of checkpoint data
- garbage collection is initiated from a background thread
- garbage collection is initiated from a foreground thread and the number of free chunks within free blocks does not exceed one quarter of the total number of free chunks on the device

As can be seen in Listing 3.8, YAFFS2 checks for necessity of garbage collection up to two times. However, the second check is only performed if the first check leads to garbage collection and garbage collection does not free up enough space. If, after a first garbage collection, the number of free blocks is smaller than the number of blocks reserved for checkpoint data and a block for garbage collection is still available, a second check is performed. This second check always leads to aggressive garbage collection, as the number of blocks reserved for checkpoint data is always smaller than the value of variable `minErased` (see line 2571 of Listing 3.8), because function `yaffs2_CalcCheckPointBlocksRequired` of `yaffs_yaffs2.c` never returns a value smaller than 0. A second check can only occur if the number of free blocks is smaller than the number of blocks reserved for checkpoint data. Aggressive garbage collection is performed if the number of free blocks is smaller than the value of variable `minErased` which is always bigger than the number of blocks reserved for checkpoint data. Therefore, the `if`-statement in line 2575 of Listing 3.8 always returns `true` during a second check, thus causing aggressive garbage collection.

After checking necessity of garbage collection, selection of a block to actually garbage collect is performed as second step of every garbage collection. This step is skipped if a block for garbage collection has already been selected in an earlier garbage collection cycle and has not yet been completely garbage collected. If no block for garbage collection has been selected, either function `yaffs_FindBlockForGarbageCollection` or function `yaffs2_FindRefreshBlock` is called to select a block for garbage collection. If passive garbage collection is performed, first function `yaffs2_FindRefreshBlock` of `yaffs_yaffs2.c` is called. This function's purpose is to enable *block refreshing*, a wear leveling technique. Function `yaffs2_FindRefreshBlock` returns the oldest block in state `FULL` with the oldest block being the block with the lowest sequence number and thus the block that has not been written to for longer than any other block. However, `yaffs2_FindRefreshBlock` only returns this block if a fixed number of blocks have been selected for garbage collection beforehand by function `yaffs_FindBlockForGarbage Collection`. By default, `yaffs2_FindRefreshBlock` only returns the oldest block every 500 executions of `yaffs_FindBlockForGarbageCollection` that actually lead to a block being selected for garbage collection. Otherwise, `yaffs2_FindRefreshBlock` returns 0 and therefore does not select a block for garbage collection. If `yaffs2_FindRefreshBlock` returns the value 0 or aggressive garbage collection has been chosen, function `yaffs_FindBlockForGarbageCollection` is used to find a block for garbage collection.

In function `yaffs_FindBlockForGarbageCollection`, most of the main differences between aggressive and passive garbage collection are defined. When trying to select a block for garbage collection, passive and aggressive garbage collection differ in three points, which are:

- the consideration of prioritized blocks.
- the intensity with which a block to garbage collect is searched for
- the number of obsolete chunks inside a block that are necessary to make the block a candidate for garbage collection

YAFFS2 marks a block as prioritized for garbage collection if the block shows abnormal behavior, such as errors on read or write operations or a failed ECC check. As such errors can indicate a forthcoming failure of a block, its prioritization for garbage collection can prevent data loss through copying the block's contents to the block currently selected for allocation. However, when choosing a block to garbage collect, YAFFS2 only takes prioritizations into consideration when garbage collection is performed passively. When trying to find a block for passive garbage collection, YAFFS2, as can be seen in lines 2392 to 2408 of Listing 3.9, always selects the first prioritized block of a device for garbage collection, unless this block is not in state `FULL` or is disqualified for garbage collection. A block is disqualified for garbage collection if a file header chunk with a shrink header marker can be found within the block and the block's sequence number is higher than another block's sequence number and this block is in state `FULL` and features at least on obsolete chunk. Hence, a block featuring a header chunk marked with a shrink header marker is disqualified for garbage collection until it becomes the oldest of all blocks in state `FULL` that contain at least one obsolete chunk. If all prioritized blocks are disqualified for garbage collection, YAFFS2 tries to select the oldest block with obsolete chunks for passive garbage collection.

In case garbage collection is performed aggressively or passive garbage collection failed to select a block during its check for prioritized blocks in lines 2392 to 2423 of Listing 3.9, YAFFS2 starts a more extensive search for a block to garbage collect. The intensity of this search depends solely on whether garbage collection is performed aggressively or passively. However, how many obsolete chunks a block has to feature before becoming a valid candidate for garbage collection, does not only depend on whether garbage collection is performed aggressively or passively but, in case garbage collection is performed passively, also on whether garbage collection is triggered by a background thread or not. As garbage collection is performed aggressively only if not enough free blocks are available to store checkpoint data, aggressive garbage collection's goal is to free up space as quickly as possible. Therefore, when garbage collection is performed aggressively, YAFFS2 searches much harder for a block to garbage collect than is does when garbage collection is performed passively. As can be see in line 2436 and lines 2460 to 2481 of Listing 3.9, aggressive garbage collection, if necessary, checks all of a device's blocks for their suitability to be garbage collected. Passive garbage collection does not need to free up space as quickly as aggressive garbage collection and thus does not search for a block to garbage collect as intensely as aggressive garbage collection. Passive garbage collection checks at least one-sixteenth of all a device's blocks plus one block but maximal 100 blocks. Both aggressive and passive garbage collection check a device's blocks sequentially starting from the block the last search for a block to garbage collect has stopped at. As can be seen in lines 2474 to 2480 of Listing 3.9, both aggressive and passive garbage collection first select the best block for garbage collection from those blocks getting checked, which in case of aggressive garbage collection are all blocks. A block is suitable for garbage collection if it is in state `FULL`, has obsolete chunks and is not disqualified for garbage collection. From all blocks checked that meet these requirements the block featuring the most obsolete chunks is selected as the best candidate for being garbage collected.

If this best candidate is finally actually selected to be garbage collected, again, depends on the mode garbage collection is performed in. As can be seen in line 2483 of Listing 3.9, before a block is selected to be garbage collected, one last check is performed. Because of this check, the best candidate for garbage is only selected to be garbage collected if the number of its chunks containing valid data does not exceed a a certain *threshold*. The actual value of this threshold depends on the mode garbage collection is performed in. In case garbage collection is performed aggressively, the threshold value equals the number of chunks per block so that the best candidate is always selected to be garbage collected, even if it features only one single obsolete chunk. As passive garbage collection does not need to free up space as quickly as aggressive garbage collection, it has no need to garbage collect blocks that feature only a very small amount of obsolete chunks. Thus, the threshold is set to a smaller value when garbage collection is performed passively. As can be seen in lines 2438 to 2458 of Listing 3.9, the value used as threshold also depends on whether passive garbage collection is a background garbage collection or not. In case passive garbage collection is a background garbage collection, the threshold is set to at least double the number of skipped garbage collections increased by 2 and maximal one half of the number of chunks per block. Thus, background garbage collection accepts a higher number of valid chunks inside a block to be garbage collected after every skipped garbage collection but never garbage collects a block that has more than half of its chunks in use. If the value chosen for the threshold is smaller than the value of `YAFFS_GC_PASSIVE_THRESHOLD`, which by default is four, `YAFFS_GC_PASSIVE_THRESHOLD` is used as threshold. A garbage collection is skipped, if none of the blocks checked meets all requirements to be garbage collected. As seen above, this can only happen in case garbage collection is performed passively. In case passive garbage collection is not a background garbage collection the threshold is set to at least the value of `YAFFS_GC_PASSIVE_THRESHOLD` and maximal one-eighth of the number of chunks per block. The combination of passive garbage collection's limited intensity of search for a block to garbage collect and the threshold requirement is the reason that passive garbage collection does not always find a block to garbage collect and thus is skipped. As passive garbage collection only checks a subset of all blocks for their suitability to be garbage collected it is likely that no suitable block is found. Additionally, if a suitable block is found there is still no guarantee that this block is actually selected to be garbage collected because of the threshold requirement. To mitigate this problem, YAFFS2 keeps track of the number of consecutively skipped garbage collections. If passive garbage collection has been consecutively skipped often enough, YAFFS2 tries to select the device's oldest block in state `FULL` that features at least one obsolete chunk to be garbage collected. As can be seen in lines 2492 to 2503 of listing 3.9, for this to happen, passive garbage collection must have been skipped ten consecutive times in case of background garbage collection and twenty consecutive times in case garbage collection was triggered by a foreground thread. In the following, this way to select a block for garbage collection is also referred to as *oldest dirty garbage collection*.

```
2379 static unsigned yaffs_FindBlockForGarbageCollection(yaffs_Device *dev,
2380                                         int aggressive,
2381                                         int background)
2382 {
2383         int i;
2384         int iterations;
2385         unsigned selected = 0;
2386         int prioritised = 0;
2387         int prioritisedExists = 0;
2388         yaffs_BlockInfo *bi;
2389         int threshold;
2390 
2391         /* First let's see if we need to grab a prioritised block */
2392         if (dev->hasPendingPrioritisedGCs && !aggressive) {
2393                 dev->gcDirtiest = 0;
2394                 bi = dev->blockInfo;
2395                 for (i = dev->internalStartBlock;
2396                         i <= dev->internalEndBlock && !selected;
2397                         i++) {
2398 
2399                         if (bi->gcPrioritise) {
2400                                 prioritisedExists = 1;
2401                                 if (bi->blockState == YAFFS_BLOCK_STATE_FULL &&
2402                                    yaffs2_BlockNotDisqualifiedFromGC(dev, bi)) {
2403                                         selected = i;
2404                                         prioritised = 1;
2405                                 }
2406                         }
2407                         bi++;
2408                 }
2409 
2410                 /*
2411                  * If there is a prioritised block and none was selected then
2412                  * this happened because there is at least one old dirty block
                         gumming
2413                  * up the works. Let's gc the oldest dirty block.
2414                  */
2415 
2416                 if(prioritisedExists &&
2417                         !selected &&
2418                         dev->oldestDirtyBlock > 0)
2419                         selected = dev->oldestDirtyBlock;
2420 
2421                 if (!prioritisedExists) /* None found, so we can clear this */
2422                         dev->hasPendingPrioritisedGCs = 0;
2423         }
2424 
2425         /* If we're doing aggressive GC then we are happy to take a less-dirty block
                 , and
2426          * search harder.
2427          * else (we're doing a leasurely gc), then we only bother to do this if the
2428          * block has only a few pages in use.
2429          */
2430 
2431         if (!selected){
2432                 int pagesUsed;
2433                 int nBlocks = dev->internalEndBlock - dev->internalStartBlock + 1;
2434                 if (aggressive){
2435                         threshold = dev->param.nChunksPerBlock;
2436                         iterations = nBlocks;
2437                 } else {
2438                         int maxThreshold;
```

```

                        if(background)
                                maxThreshold = dev->param.nChunksPerBlock/2;
                        else
                                maxThreshold = dev->param.nChunksPerBlock/8;

                        if(maxThreshold <  YAFFS_GC_PASSIVE_THRESHOLD)
                                maxThreshold = YAFFS_GC_PASSIVE_THRESHOLD;

                        threshold = background ?
                                (dev->gcNotDone + 2) * 2 : 0;
                        if(threshold <YAFFS_GC_PASSIVE_THRESHOLD)
                                threshold = YAFFS_GC_PASSIVE_THRESHOLD;
                        if(threshold > maxThreshold)
                                threshold = maxThreshold;

                        iterations = nBlocks / 16 + 1;
                        if (iterations > 100)
                                iterations = 100;
                }

                for (i = 0;
                        i < iterations &&
                        (dev->gcDirtiest < 1 ||
                                dev->gcPagesInUse > YAFFS_GC_GOOD_ENOUGH);
                        i++) {
                        dev->gcBlockFinder++;
                        if (dev->gcBlockFinder < dev->internalStartBlock ||
                                dev->gcBlockFinder > dev->internalEndBlock)
                                dev->gcBlockFinder = dev->internalStartBlock;

                        bi = yaffs_GetBlockInfo(dev, dev->gcBlockFinder);

                        pagesUsed = bi->pagesInUse - bi->softDeletions;

                        if (bi->blockState == YAFFS_BLOCK_STATE_FULL &&
                                pagesUsed < dev->param.nChunksPerBlock &&
                                (dev->gcDirtiest < 1 || pagesUsed < dev->
                                    gcPagesInUse) &&
                                yaffs2_BlockNotDisqualifiedFromGC(dev, bi)) {
                                dev->gcDirtiest = dev->gcBlockFinder;
                                dev->gcPagesInUse = pagesUsed;
                        }
                }

                if(dev->gcDirtiest > 0 && dev->gcPagesInUse <= threshold)
                        selected = dev->gcDirtiest;
        }

        /*
         * If nothing has been selected for a while, try selecting the oldest dirty
         * because that's gumming up the works.
         */

        if(!selected && dev->param.isYaffs2 &&
                dev->gcNotDone >= ( background ? 10 : 20)){
                yaffs2_FindOldestDirtySequence(dev);
                if(dev->oldestDirtyBlock > 0) {
                        selected = dev->oldestDirtyBlock;
                        dev->gcDirtiest = selected;
                        dev->oldestDirtyGCs++;
                        bi = yaffs_GetBlockInfo(dev, selected);
                        dev->gcPagesInUse =  bi->pagesInUse - bi->softDeletions;
                } else
                        dev->gcNotDone = 0;
        }

        if(selected){
                T(YAFFS_TRACE_GC,
                  (TSTR("GC Selected block %d with %d free, prioritised:%d" TENDSTR)
                      ,
                  selected,
                  dev->param.nChunksPerBlock - dev->gcPagesInUse,
                  prioritised));

                dev->nGCBlocks++;
                if(background)
                        dev->backgroundGCs++;

                dev->gcDirtiest = 0;
                dev->gcPagesInUse = 0;
                dev->gcNotDone = 0;
                if(dev->refreshSkip > 0)
                        dev->refreshSkip--;
        } else{
                dev->gcNotDone++;
                T(YAFFS_TRACE_GC,
                  (TSTR("GC none: finder %d skip %d threshold %d dirtiest %d using %
                      d oldest %d%s" TENDSTR),
                  dev->gcBlockFinder, dev->gcNotDone,
                  threshold,
                  dev->gcDirtiest, dev->gcPagesInUse,
                  dev->oldestDirtyBlock,
                  background ? " bg" : ""));
        }

        return selected;
}
```

Listing 3.9: Function `yaffs_FindBlockForGarbageCollection` (Excerpt from yaffs_guts.c)

As seen above, YAFFS2 puts a lot of effort in selecting a block for garbage collection and knows a variety of reasons why not to garbage collect a specific block. Nonetheless, a basic goal in selecting a block to garbage collect is recognizable. This is the goal to select the dirtiest and oldest block possible for garbage collection.

After checking for necessity of garbage collection and selecting a block to garbage collect, the third and last step of every garbage collection cycle consists of actually copying valid chunks from the selected block to the block currently allocated for write operations. This task is performed by function `yaffs_GarbageCollectBlock` (lines 2102 to 2372 of `yaffs_guts.c`). This function shows the last important difference between aggressive and passive garbage collection. While aggressive garbage collection collects the whole block at one go, passive garbage collection only collects five valid chunks per garbage collection cycle. Because of that, passive garbage collection can need several garbage collection cycles to collect a block. Once a block has been completely collected, its state is set to `DIRTY` which leads to its immediate deletion and transition into state `EMPTY`. Additionally, if the block selected for garbage collection is a block containing checkpoint data, the block's state is also set to `DIRTY` immediately and no chunks are copied off.

3.4. Wear leveling

One of the issues that a NAND flash filesystem has to deal with, is flash memory's limited endurance. Flash memory's erase blocks can only endure a limited number of erase and rewrite cycles, typically ranging from 10^4 to 10^6 cycles [18]. In order to prevent single blocks to wear out faster than the device's other blocks, write and erase operations have to be evenly distributed among all of a flash memory's blocks. Techniques to achieve such a distribution of write and erase cycles are referred to as wear leveling and are an important feature of flash memory file systems.

Basically, a flash memory file system has two options to perform wear leveling. These can be described as explicit and implicit wear leveling techniques. A flash file system performing explicit wear leveling features special functionality exclusively dedicated to perform wear leveling, whereas a flash file system performing implicit wear leveling does not feature any special functionality regarding wear leveling. Implicit wear leveling relies on the basic design of the flash file system to evenly distribute erase and write load among the flash memory's blocks.

YAFFS2 performs wear leveling mainly implicitly but also features one explicit wear leveling technique, block refreshing. YAFFS2's design implicitly supports wear leveling in several ways. First of all, YAFFS2 does not use any central structures, such as a file allocation table. Therefore, YAFFS2 has no need to write such data to fixed addresses on the flash memory, hence preventing that blocks at these addresses wear out much faster than other blocks of the flash memory. Additionally to not writing specific blocks excessively often, YAFFS2 distributes its write operations evenly by means of its block allocation policy. As already depicted in Listing 3.2, as long as free blocks exist on a flash memory, YAFFS2, being a truly log-structured file system, tries to allocate them in sequential order. Thus, YAFFS2 achieves a certain level of wear leveling solely implicitly. Additionally, YAFFS2 can use its block refreshing technique to distribute block usage evenly among all blocks of a NAND. Block refreshing, in short, tries to spread wear by moving the oldest full block's content to other blocks and erasing the oldest block. In doing so, block refreshing ensures that every block is erased at some point, even if no obsolete chunks can be found on that block and thus the block is not a candidate for regular garbage collection. Although block refreshing can be disabled at compile time, it is enabled by default. As already mentioned in Section 3.3, block refreshing is a task performed by YAFFS2's garbage collector. As can be seen in line 2597 of Listing 3.8, block refreshing can only occur, if a check for necessity of garbage collection proves passive garbage collection necessary and no block has already been selected to be garbage collected.

```c
__u32 yaffs2_FindRefreshBlock(yaffs_Device *dev)
{
        __u32 b ;

        __u32 oldest = 0;
        __u32 oldestSequence = 0;

        yaffs_BlockInfo *bi;

        if(!dev->param.isYaffs2)
                return oldest;

        /*
         * If refresh period < 10 then refreshing is disabled.
         */
        if(dev->param.refreshPeriod < 10)
                return oldest;

        /*
         * Fix broken values.
         */
        if(dev->refreshSkip > dev->param.refreshPeriod)
                dev->refreshSkip = dev->param.refreshPeriod;

        if(dev->refreshSkip > 0)
                return oldest;

        /*
         * Refresh skip is now zero.
         * We'll do a refresh this time around....
         * Update the refresh skip and find the oldest block.
         */
        dev->refreshSkip = dev->param.refreshPeriod;
        dev->refreshCount++;
        bi = dev->blockInfo;
        for (b = dev->internalStartBlock; b <=dev->internalEndBlock; b++){

                if (bi->blockState == YAFFS_BLOCK_STATE_FULL){

                        if(oldest < 1 ||
                                bi->sequenceNumber < oldestSequence){
                                oldest = b;
                                oldestSequence = bi->sequenceNumber;
                        }
                }
                bi++;
        }

        if (oldest > 0) {
                T(YAFFS_TRACE_GC,
                  (TSTR("GC refresh count %d selected block %d with sequenceNumber %
                      d" TENDSTR),
                   dev->refreshCount, oldest, oldestSequence));
        }

        return oldest;
}
```

Listing 3.10: Function `yaffs_FindRefreshBlock` (Excerpt from yaffs_yaffs2.c)

As can be seen in Listing 3.10, `yaffs2_FindRefreshBlock` only selects the oldest block for refreshing, if variable `refreshSkip` has value zero. By default, this variable is set to value zero on mount of a YAFFS2 device [19] and to a value of 500 after the first block refreshing. The value of `refreshSkip` is only decremented in function `yaffs_FindBlockForGarbageCollection` in case a block for garbage collection is selected (see Listing 3.9). Thus, block refreshing can only occur as the first garbage collection after mounting of a YAFFS2 device and subsequently every 500 executions of `yaffs_FindBlockForGarbageCollection` that lead to selection of a block for garbage collection.

3.5. Summary

In this chapter we introduced YAFFS2's basic characteristics and behavior. In the first part of this chapter we showed that YAFFS2 is a truly log-structured file system which makes it highly likely that parts of deleted or modified files can be found on a YAFFS2 device within a forensic investigation. In this chapter, we also introduced YAFFS2 way to use NAND flash memory's pages' OOB areas to organize data and analyzed discrepancies between YAFFS2's documentation and its actual behavior. As can be seen in Section

3.2 of this chapter, YAFFS2's documentation proved to be generally accurate but did not provide detailed information on some of YAFFS2's behavior's aspects, especially YAFFS2's garbage collection techniques.

During the analysis of YAFFS2's garbage collection techniques it became obvious that garbage collection has substantial impact on the amount of obsolete chunks containing potential evidence that can be recovered from a YAFFS2 device within a forensic analysis. As can be seen in Section 3.3 of this chapter, YAFFS2 uses sophisticated methods to decide whether garbage collection has to be performed and, if yes, which block is best suited to be garbage collected. Despite the complexity of YAFFS2's garbage collector's techniques to select a block for garbage collection, a basic goal of these techniques became clear, namely to select that block for garbage collection that features the largest number of obsolete chunks. Additionally, preferably older blocks are garbage collected. We discovered that YAFFS2 uses aggressive garbage collection only in case a NAND does not have enough free blocks available to store checkpoint data. The answer to the question whether garbage collection has to be performed at a specific time depends mostly on the ratio of free chunks within free blocks to the total amount of free chunks and on the kind of thread that was used to invoke a check for garbage collection. As can be seen in Section 3.3 of this chapter, a check for necessity of garbage collection only leads to further steps of garbage collection if storage space is so scarce that aggressive garbage collection is necessary, the check was invoked by a background check or at max one quarter of all a device's free chunks reside within free blocks. A further analysis and practical evaluation of YAFFS2's garbage collection techniques and their impact on forensic analyses of YAFFS2 devices is provided in Chapter 4 and Chapter 5.

In the last part of this chapter, we analyzed YAFFS2's wear leveling techniques. As can be seen in Section 3.4, YAFFS2 performs wear leveling mostly implicitly and uses its garbage collector for explicit wear leveling also called block refreshing. Block refreshing is at least performed during the first execution of garbage collection after a YAFFS2 device has been mounted. Additionally, block refreshing is performed on a regular basis, by default every 500 times a block is selected to be garbage collected. Block refreshing is just a variant of regular garbage collection with the crucial difference that block refreshing always selects the oldest block for garbage collection, regardless of the number of obsolete chunks within this block.

4. YAFFS2 in a forensic view

In Chapter 3, we introduced and analyzed YAFFS2's behavior and functionalities. Based on the findings acquired in the last chapter, we discuss the effects of YAFFS2's behavior on forensic analyses of YAFFS2 NAND flash memory devices in this chapter. In Section 4.1 and Section 4.2, we discuss the influences of YAFFS2's wear leveling techniques and shrink header markers on the amount of data that can be recovered from a YAFFS2 NAND. In Section 4.3, we discuss and analyze YAFFS2's garbage collector in a forensic perspective. This includes a discussion of best case and worst case scenarios regarding recovery of obsolete chunks from a YAFFS2 NAND. We practically analyze these scenarios in Chapter 5.

4.1. Wear leveling

YAFFS2's only explicit wear leveling technique is block refreshing. As analyzed in Chapter 3, block refreshing is only performed during the first execution of garbage collection after mounting of a YAFFS2 NAND flash memory device and every 500 times a block is selected for garbage collection by function `yaffs_FindBlockForGarbageCollection`. Basically, block refreshing is a variant of garbage collection that does not pay attention to the number of obsolete chunks within the block that is to be garbage collected. Instead, block refreshing's goal is to move a block's contents to another location on the NAND in order to distribute erase operations evenly. Block refreshing always selects the device's oldest block for garbage collection, regardless of the number of obsolete chunks within this block. Thus, if the oldest block does not contain any obsolete chunks, block refreshing does not lead to deletion of data, as all the oldest block's chunks are copied to the current allocation block.

In comparison to regular garbage collection, block refreshing does occur relatively seldom and does not necessarily lead to deletion of obsolete chunks and thus loss of possible evidence. Therefore, influence of block refreshing on the amount of recoverable data can be considered relatively insignificant.

4.2. Shrink header markers

As analyzed in Section 3.2 and Section 3.3 of this diploma thesis, shrink header markers can delay garbage collection of a block. As analyzed in Section 3.3, a block that features an object header chunk marked with a shrink header marker is disqualified for garbage collection unless the block is the device's *oldest dirty block*. A block is defined as dirty if it is in state `FULL` and contains at least one obsolete chunk. Thus, as defined in function `yaffs2_CalcOldestDirtySequence` (see lines 43 to 71 of `yaffs_yaffs2.c`), the oldest dirty block is that dirty block that features the lowest block sequence number. A dirty block is not to be confused with a block in state `DIRTY`. A block that is in state `DIRTY` does not feature any valid chunks at all.

From a forensic point of view, shrink header markers can play an important role. As a block containing an object header chunk marked with a shrink header marker is disqualified for garbage collection, its contents can remain stored on a device for a comparatively long time without being deleted by YAFFS2's garbage collector. Hence, even a block that features a large number of obsolete chunks and thus constitutes a preferred candidate for garbage collection can stay undeleted for a long time. As blocks with a high number of obsolete chunks also have the potential to contain a high amount of recoverable data relevant to a forensic inquiry, shrink header markers can be a decisive factor in a forensic analysis. We practically evaluate the effects of shrink header markers on the recoverability of obsolete chunks in Chapter 5

4.3. Garbage Collection

Among all of YAFFS2's characteristics and functionalities, YAFFS2's garbage collector obviously has the most significant impact on the amount of deleted or modified data that can be recovered from a YAFFS2 NAND. As analyzed in Chapter 3 of this diploma thesis, YAFFS2 never overwrites chunks or deletes single chunks without completely erasing their respective blocks beforehand. Thus, all data ever written to a block is recoverable until the respective block gets erased during garbage collection. In the following, we introduce and discuss best case and worst scenarios regarding garbage collection.

4.3.1. Best and worst case scenarios

As shown in Chapter 3, execution of garbage collection does not primarily depend on a device's storage occupancy. Background garbage collection even occurs completely independent of a device's storage occupancy. As described in Chapter 3, passive garbage collection is executed directly after a write operation only in case at max one quarter of all free chunks are located in free blocks. As long as this is not the case, passive garbage collection is only executed as background garbage collection, respectively oldest dirty garbage collection. A device's storage occupancy is mainly relevant regarding the question whether aggressive garbage collection is necessary and how soon after a modification or deletion garbage collection is executed, not regarding the question how often garbage collection is performed.

Every garbage collection potentially destroys evidence. Thus, from a forensic point of view, a best case scenario regarding garbage collection of a YAFFS2 NAND to be analyzed includes only a minimum of executed garbage collection cycles, whereas a worst case scenario includes a high number of executed garbage collection cycles, especially aggressive garbage collection cycles. In the following, we introduce occupancy patterns representing these best case and worst case scenarios. Although existence of these scenarios is theoretically possible, actually being able to create such a scenario in practice is not necessarily possible. We further discuss this in Section 4.3.2.

Best case scenarios

From a forensic point of view, a best case scenario regarding recovery of deleted or modified data is a scenario that enables a maximum of obsolete data to be recovered from a device. As discussed above, the amount of obsolete data that can be recovered from a YAFFS2 NAND depends heavily on the way obsolete and current chunks are distributed among the device's blocks. Additionally, the ratio of obsolete chunks to valid chunks within a block is crucial to the amount of obsolete chunks that can be recovered from a YAFFS2 NAND.

As analyzed in Chapter 3, aggressive garbage collection occurs if a device does not feature enough free blocks to store checkpoint data. As aggressive garbage collection potentially deletes a higher number of obsolete chunks per garbage collection cycle than passive garbage collection, aggressive garbage collection should never occur in a best case scenario. Thus, in a best case scenario, a device features at least n free blocks during its whole time of usage with n having the following value:

$$\begin{aligned} n = {} & \text{number of reserved blocks} \\ & + \text{number of complete blocks actually necessary to store current checkpoint data} \\ & - \text{number of blocks currently used for checkpoint data} \\ & + 4 \end{aligned}$$

By default, YAFFS2 reserves five blocks for checkpoint data (see line 2891 of `yaffs_fs.c`). However, the number of blocks actually necessary to store checkpoint data depends on the number of objects stored on a device and is calculated in function `yaffs2_CalcCheckpointBlocksRequired` of `yaffs_yaffs2.c`.

```
int yaffs2_CalcCheckpointBlocksRequired(yaffs_Device *dev)
{
        int retval;

        if(!dev->param.isYaffs2)
                return 0;

        if (!dev->nCheckpointBlocksRequired &&
                yaffs2_CheckpointRequired(dev)){
                /* Not a valid value so recalculate */
                int nBytes = 0;
                int nBlocks;
                int devBlocks = (dev->param.endBlock - dev->param.startBlock + 1);

                nBytes += sizeof(yaffs_CheckpointValidity);
                nBytes += sizeof(yaffs_CheckpointDevice);
                nBytes += devBlocks * sizeof(yaffs_BlockInfo);
                nBytes += devBlocks * dev->chunkBitmapStride;
                nBytes += (sizeof(yaffs_CheckpointObject) + sizeof(__u32)) * (dev->
                    nObjects);
                nBytes += (dev->tnodeSize + sizeof(__u32)) * (dev->nTnodes);
                nBytes += sizeof(yaffs_CheckpointValidity);
                nBytes += sizeof(__u32); /* checksum*/

                /* Round up and add 2 blocks to allow for some bad blocks, so add 3
                    */

                nBlocks = (nBytes/(dev->nDataBytesPerChunk * dev->param.
                    nChunksPerBlock)) + 3;

                dev->nCheckpointBlocksRequired = nBlocks;
        }

        retval = dev->nCheckpointBlocksRequired - dev->blocksInCheckpoint;
        if(retval < 0)
                retval = 0;
        return retval;
}
```

Listing 4.1: Function `yaffs_CalcCheckpointBlocksRequired` (Excerpt from yaffs_yaffs2.c)

As can be seen in Listing 4.1, the number of blocks needed to store a checkpoint consists of a fixed number of bytes used for checksums and general information regarding the device and a variable number of bytes depending on the number of objects stored on the device and the device's size. For the following calculations the sizes of all data types are the sizes they feature in a Linux environment.

```
typedef struct {

        int softDeletions:10;
        int pagesInUse:10;
        unsigned blockState:4;
        __u32 needsRetiring:1;

        __u32 skipErasedCheck:1;
        __u32 gcPrioritise:1;

        __u32 chunkErrorStrikes:3;

#ifdef CONFIG_YAFFS_YAFFS2
        __u32 hasShrinkHeader:1;
        __u32 sequenceNumber;
#endif

} yaffs_BlockInfo;
[...]

typedef struct {
        int structType;
        __u32 objectId;
        __u32 parentId;
        int hdrChunk;
        yaffs_ObjectType variantType:3;
        __u8 deleted:1;
        __u8 softDeleted:1;
        __u8 unlinked:1;
        __u8 fake:1;
        __u8 renameAllowed:1;
        __u8 unlinkAllowed:1;
        __u8 serial;

        int nDataChunks;
        __u32 fileSizeOrEquivalentObjectId;
} yaffs_CheckpointObject;
```

```c
[...]

typedef struct {
        int structType;
        int nErasedBlocks;
        int allocationBlock;
        __u32 allocationPage;
        int nFreeChunks;

        int nDeletedFiles;
        int nUnlinkedFiles;
        int nBackgroundDeletions;

        /* yaffs2 runtime stuff */
        unsigned sequenceNumber;

} yaffs_CheckpointDevice;


typedef struct {
        int structType;
        __u32 magic;
        __u32 version;
        __u32 head;
} yaffs_CheckpointValidity;
```

Listing 4.2: Structs used to calculate the number of blocks needed to store checkpoint data (Excerpt from yaffs_guts.h)

The data types `__u32`, `__u16` and `__u8` are defined in lines 32 to 34 of `devextras.h` and have the following sizes:

- `__u32`: four bytes
- `__u16`: two bytes
- `__u8`: one byte

As mentioned above, a checkpoint consists of a part of fixed size and a part of variable size. The part of fixed size consists of the *structs* `yaffs_CheckpointValidity`, `yaffs_CheckpointDevice` and a checksum of data type `__u32`. As `yaffs_Checkpoint Validity` is used to mark the beginning and the end of checkpoint data, it is used twice within every checkpoint. Thus, as can be seen in Listing 4.2, every checkpoint needs 72 bytes for its part of fixed size. The variable part of a checkpoint consists of information on blocks, objects and *tnodes*. YAFFS2 keeps a so-called tnode tree in RAM for every object. This tree is used to provide mapping of object positions to actual chunk positions on a NAND flash memory device. This tree's nodes are called tnodes. To store information on blocks, objects and tnodes in a checkpoint, 12 bytes per block, 32 bytes per object and 8 bytes per tnode are needed.

Passive garbage collection only collects blocks with a certain number of obsolete chunks. Background garbage collection only garbage collects blocks with at least half of their chunks being obsolete chunks. Passive garbage collection that was triggered by a foreground thread such as a write operation only garbage collects blocks with at least seven-eighths of their chunks being obsolete chunks. Hence, as long as every block of a device has at least half of its chunks filled with valid data, the only way a block can be garbage collected is through oldest dirty garbage collection or block refreshing.

Oldest dirty garbage collection occurs every time garbage collection has been skipped ten, respectively twenty, consecutive times (see Section 3.3). Oldest dirty garbage collection selects the oldest block that features at least one obsolete chunk and cannot be prevented.

Thus, a best case scenario requires that, during the whole time of its usage, a device features enough free blocks to store checkpoint data and a distribution of obsolete and valid chunks that leads to every block having just more than half of its chunks being valid. Additionally, enough free blocks must be available to ensure that more than one quarter of all free chunks is located within empty blocks. That way, all blocks are garbage collected as seldom as possible and still feature a high number of obsolete chunks that potentially contain evidence. As in such a scenario all blocks can only be selected for garbage collection by oldest dirty garbage collection, shrink header markers do not play a crucial role.

Worst case scenarios

From a forensic point of view, a worst case scenario regarding recovery of deleted or modified data is a scenario that enables a minimum of obsolete data to be recovered from a device. As discussed above, only garbage collection can lead to deletion of obsolete chunks. Thus, a worst case scenario is a scenario in which every check for necessity of garbage collection of a block containing obsolete chunks leads to actual garbage collection and as many garbage collection cycles as possible are performed aggressively. Obviously, the unlikely case of a NAND that does not contain any obsolete chunks at all also constitutes a worst case scenario. As in this case no recoverable obsolete data exists on the device in the first place, we do not further consider this scenario. Hence, in the following, a worst case scenario is a scenario where a certain amount of obsolete chunks exists on a device but their contribution on the device and the degree of the device's storage occupancy lead to a maximum of deletions of obsolete chunks.

As analyzed in Section 3.3 of this diploma thesis, garbage collection is performed aggressively if a device does not feature enough free blocks to store checkpoint data. Thus, the amount of obsolete data deleted due to aggressive garbage collection is at its max in case a device's storage occupancy does not leave enough free blocks for checkpoint data. In order to garbage collect a block, at least one free block must exist on a device to store valid chunks that have to be copied off from the block to be garbage collected. Thus, in a worst case scenario, a device's storage capacity is used to a degree that prevents checkpoint data to be stored but leaves at least one block free for allocation. A small number of empty blocks also increases the chance, that than one quarter or less of the device's free chunks is located in free blocks. Thus, a small number of empty blocks can increase the chance of passive garbage collection being executed due to write operations.

As mentioned before, execution of garbage collection depends heavily on the ratio of valid chunks to obsolete chunks within a block. While aggressive garbage collection even collects blocks that contain only one obsolete chunk, passive garbage collection accepts a higher number of obsolete chunks within a block before garbage collecting the block. This behavior enables another kind of worst case scenario that does not include aggressive garbage collection. If a device features enough free block to store checkpoint data and thus no aggressive garbage collection takes place, there is still the possibility that passive garbage collection leads to a maximum of deletions of obsolete chunks. Background garbage collection does not depend on a device's storage occupancy and therefore is able to delete a high amount of obsolete data even if a device's storage capacity is hardly used. In case each of a device's used blocks features only four valid chunks, every check for necessity of background garbage collection of such a block leads to collection of the block. As can be seen in Listing 4.3, when performing background garbage collection, YAFFS2 accepts a block for garbage collection, if the block features `(dev->gcNotDone + 2) * 2` or less valid chunks. As `dev->gcNotDone` has a minimum value of zero, the lowest threshold for execution of background garbage collection amounts to four valid chunks per block.

```
        if (!selected){
                int pagesUsed;
                int nBlocks = dev->internalEndBlock - dev->internalStartBlock + 1;
                if (aggressive){
                        threshold = dev->param.nChunksPerBlock;
                        iterations = nBlocks;
                } else {
                        int maxThreshold;

                        if(background)
                                maxThreshold = dev->param.nChunksPerBlock/2;
                        else
                                maxThreshold = dev->param.nChunksPerBlock/8;

                        if(maxThreshold <  YAFFS_GC_PASSIVE_THRESHOLD)
                                maxThreshold = YAFFS_GC_PASSIVE_THRESHOLD;

                        threshold = background ?
                                (dev->gcNotDone + 2) * 2 : 0;
                        if(threshold <YAFFS_GC_PASSIVE_THRESHOLD)
                                threshold = YAFFS_GC_PASSIVE_THRESHOLD;
                        if(threshold > maxThreshold)
                                threshold = maxThreshold;

                        iterations = nBlocks / 16 + 1;
                        if (iterations > 100)
                                iterations = 100;
                }

                for (i = 0;
                        i < iterations &&
                        (dev->gcDirtiest < 1 ||
                                dev->gcPagesInUse > YAFFS_GC_GOOD_ENOUGH);
                        i++) {
                        dev->gcBlockFinder++;
                        if (dev->gcBlockFinder < dev->internalStartBlock ||
                                dev->gcBlockFinder > dev->internalEndBlock)
                                dev->gcBlockFinder = dev->internalStartBlock;

                        bi = yaffs_GetBlockInfo(dev, dev->gcBlockFinder);

                        pagesUsed = bi->pagesInUse - bi->softDeletions;

                        if (bi->blockState == YAFFS_BLOCK_STATE_FULL &&
                                pagesUsed < dev->param.nChunksPerBlock &&
                                (dev->gcDirtiest < 1 || pagesUsed < dev->
                                    gcPagesInUse) &&
                                yaffs2_BlockNotDisqualifiedFromGC(dev, bi)) {
                                dev->gcDirtiest = dev->gcBlockFinder;
                                dev->gcPagesInUse = pagesUsed;
                        }
                }

                if(dev->gcDirtiest > 0 && dev->gcPagesInUse <= threshold)
                        selected = dev->gcDirtiest;
        }
```

Listing 4.3: Calculation of garbage collection threshold (Excerpt from yaffs_guts.c)

In this scenario, a high number of obsolete chunks can be found on a YAFFS2 NAND but their distribution leads to very quick deletion of these chunks. Thus, although obsolete chunks exist on the device, the chance to recover these chunks is relatively small, as garbage collection is likely to delete them before they can be recovered. However, a block can be protected from garbage collection for a certain time if the block features header chunks marked with shrink header markers.

4.3.2. Evaluation of best and worst case scenarios

As mentioned before, all data written to a YAFFS2 NAND remains stored on the device until the blocks containing the data are erased during execution of garbage collection. Therefore, recovery of modified or deleted files is always a race against YAFFS2's garbage collector. In the following, the above-described best and worst case scenarios are further analyzed for their practical relevance.

As above-mentioned, the critical factor of a forensic analysis of a YAFFS2 NAND is time. As garbage collection of the device cannot be prevented completely, sooner or later all obsolete chunks present on the device are deleted and thus no previous versions of modified files or deleted files are recoverable at this point. The reason for that are YAFFS2's unpreventable oldest dirty garbage collection and block refreshing techniques. Even if a YAFFS2 NAND features a chunk occupancy pattern that does not allow aggressive and regular passive garbage collection (background garbage collection or passive garbage collection directly after write operations), oldest dirty garbage collection is executed regularly. This is because YAFFS2 checks for necessity of background garbage collection regularly. If background collection proves unnecessary every time it is checked for and thus is skipped, oldest dirty garbage collection is performed regularly. Additionally, every 500 oldest dirty garbage collections, garbage collection is performed as block refreshing. As described in Section 3.3, by default YAFFS2 tries to execute background garbage collection every two seconds. In case background garbage collection cannot be executed ten consecutive times, oldest dirty garbage collection is performed. Oldest dirty garbage collection is also performed in case garbage collection triggered by write operations is skipped twenty consecutive time.

As shown in Section 3.3, passive garbage collection including oldest dirty garbage collection only collects five valid chunks per execution of passive garbage collection. Thus, not every execution of passive garbage collection necessarily leads to deletion of a block. In case a block consisting of 64 pages respectively chunks features only one obsolete chunk, thirteen executions of passive garbage collection are necessary before the block gets erased. As described in Section 3.3, once a block has been selected for garbage collection, YAFFS2 does not need to select another block to garbage collect until the current garbage collection block is completely collected. Hence, as soon as a block has been chosen for oldest dirty garbage collection, every subsequent attempt of background garbage collection leads to collection of this block. Thus, even in a best case scenario, even a block that features only one obsolete chunk gets erased 24 seconds at most after it was selected for oldest dirty garbage collection. Obviously, in a best case scenario where no garbage collection other than oldest dirty garbage collection and block refreshing takes place, a block that becomes the oldest dirty block is not selected for garbage collection immediately after becoming the oldest dirty block. Necessity of background garbage collection is only checked for every two seconds. Thus, up to 22 seconds can pass before oldest dirty garbage collection selects the block for garbage collection

To evaluate this hypothesis, we created one file of size 124 928 bytes on an otherwise empty NAND. Due to writing of one obsolete file header chunk on creation of a file and writing of a directory header chunk for the root directory of the device, this lead to a completely filled block that featured exactly one obsolete chunk. As no write operations were performed after creation of the file, passive garbage collection triggered by a foreground thread could not be performed. Additionally, aggressive garbage collection was ruled out due to only one block of the device being occupied. As the block only featured one obsolete chunk, regular background garbage collection was also unable to select the block for garbage collection. Thus, only after ten consecutive tries to background garbage collect a block, the block was selected for oldest dirty garbage collection. Subsequently, the block was garbage collected every two seconds due to background garbage collection.

As can be seen in Listing 4.4, the block containing the file is selected for garbage collection six seconds after the last chunk of the block has been written. This is because of background garbage collection attempts before creation of the file making oldest dirty garbage collection necessary. As can be seen in Listing 4.5, after selection of the block for garbage collection and subsequent execution of garbage collection, garbage collection is executed another twelve times in order to copy all valid chunks to another block. The reason for `backgroundGCs` and `oldestDirtyGCs` having the value one is, that only selection of a block for garbage collection increments these values. Execution of background garbage collection increments `passiveGCs`. As expected, the block is deleted 24 seconds after being selected for garbage collection. The complete log file can be found on the attached DVD.[1]

```
11:22:50 debian-DA4 kernel: [ 1799.181232] yaffs_create
11:22:50 debian-DA4 kernel: [ 1799.181249] yaffs_mknod: parent object 1 type 3
11:22:50 debian-DA4 kernel: [ 1799.181265] yaffs_mknod: making oject for testFile,
         mode 81a4 dev 0
11:22:50 debian-DA4 kernel: [ 1799.181281] yaffs locking eec87710
11:22:50 debian-DA4 kernel: [ 1799.181294] yaffs locked eec87710
11:22:50 debian-DA4 kernel: [ 1799.181305] yaffs_mknod: making file
11:22:50 debian-DA4 kernel: [ 1799.181321] yaffs: Tnodes added
11:22:50 debian-DA4 kernel: [ 1799.181341] yaffs_MarkSuperBlockDirty() sb = f56ac400
11:22:50 debian-DA4 kernel: [ 1799.181357] Allocated block 1, seq  4097, 511 left
[...]
11:22:50 debian-DA4 kernel: [ 1799.297144] Writing chunk 63 tags 257 0
[...]
11:22:52 debian-DA4 kernel: [ 1801.199970] Background gc 0
11:22:52 debian-DA4 kernel: [ 1801.199988] GC none: finder 297 skip 9 threshold 20
         dirtiest 0 using 0 oldest 0 bg
[...]
```

[1]On the attached DvD, see /Chapter4.3/kern.log_smallBestCase.rtf

```
11:22:54 debian-DA4 kernel: [ 1803.200717] Background gc 0
11:22:54 debian-DA4 kernel: [ 1803.200735] GC none: finder 330 skip 10 threshold 22
         dirtiest 0 using 0 oldest 0 bg
[...]
11:22:56 debian-DA4 kernel: [ 1805.217170] Background gc 0
11:22:56 debian-DA4 kernel: [ 1805.217187] GC Selected block 1 with 1 free,
         prioritised:0
11:22:56 debian-DA4 kernel: [ 1805.217202] yaffs: GC erasedBlocks 511 aggressive 0
11:22:56 debian-DA4 kernel: [ 1805.217218] Collecting block 1, in use 63, shrink 0,
         wholeBlock 0
[...]
11:23:20 debian-DA4 kernel: [ 1829.378242] Erased block 1
```

Listing 4.4: Excerpt from `kern.log` showing garbage collection of a block with one obsolete chunk

```
[...]
nBlockErasures..... 1
nGCCopies.......... 63
allGCs............. 13
passiveGCs......... 13
oldestDirtyGCs..... 1
nGCBlocks.......... 1
backgroundGCs...... 1
[...]
```

Listing 4.5: Excerpt from YAFFS2's statistics showing garbage collection of a block with one obsolete chunk

Even in a best case scenario, garbage collection cannot be prevented completely and thus, over time, all obsolete chunks are erased. Hence, the number of previous versions of modified files and the number of deleted files that can be recovered from a YAFFS2 NAND also depends on the time span between the execution of the file deletion or modification and a forensic analysis of the device respectively the disconnection of the device from its power source. Due to block refreshing and oldest dirty garbage collection, chunks on a YAFFS2 NAND are in constant movement. As shown above, the speed of this movement depends to a part on the occupancy of the device's storage capacity. However, the number and distribution of obsolete chunks on the device and the device's size have a much greater influence on the speed of this movement. Passive garbage collection only checks 100 blocks at most during selection of a block to garbage collect. Therefore, it can take longer for a specific block to be selected for garbage collection on a large NAND featuring a high number of blocks than it would on a smaller NAND. The movement of chunks on a YAFFS2 NAND never stops and, as long as obsolete chunks exist on a device, always leads to deletion of obsolete chunks. Thus, a YAFFS2 NAND can stay in a state that can be considered a best case or worst case scenario regarding recovery of obsolete chunks only for a very brief span of time. Although these states constituting either a best case or a worst case scenario theoretically exist, they can hardly be found in practice. However, when trying to create such a state on purpose, taking into account the criteria described in Section 4.3.1 leads to states close to the best case and worst case scenarios described above.

As shown in Chapter 3, the degree to which a YAFFS2 NAND flash memory device's capacity is used, is mostly relevant regarding the question whether garbage collection has to be performed aggressively, not the question, whether garbage collection has to be preformed at all. However, the device's occupancy can still have a great influence on the amount of obsolete data that can be recovered from the device. In case a YAFFS2 NAND flash memory device's storage capacity is used to a high extend and features a distribution of obsolete chunks matching the criteria for a best case scenario, garbage collection of the most currently written blocks takes place only after all other blocks have been garbage collected. This, of course, only applies if the most currently written blocks also feature enough valid blocks to prevent being garbage collected immediately. Thus, previous versions of modified files and deleted files can be recovered for a longer time from a YAFFS2 NAND flash memory device that features a high number of used blocks. Hence, in scenario close to a best case scenario, a YAFFS2 NAND flash memory device's capacity is used to a high extend but not so much as to make aggressive garbage collection necessary and to enable passive garbage collection caused by write operations.

4.4. Summary

In this chapter, we discussed the effects of YAFFS2's behavior and general functionality on the recoverability of obsolete data from a YAFFS2 NAND. As described in Section 4.1, we showed that YAFFS2's block refreshing does not have a significant impact on the amount of recoverable obsolete chunks. However, shrink header markers can have great influence on the time span in which obsolete chunks can be recovered from a specific block. We provide practical evaluation of this in the next chapter.

As described in Section 4.3, we discovered that all obsolete chunks are deleted from a YAFFS2 NAND after a certain time span. The length of this time span depends on the distribution of obsolete and valid chunks on the NAND as well as on the storage capacity of the NAND and its occupancy. We presented best case and worst case scenarios regarding recovery of obsolete chunks and showed that these scenarios can only exist temporarily on a YAFFS2 NAND due to constant movement of the chunks stored on the device.

5. Recovery of files

As described in Chapter 4 of this diploma thesis, the storage capacity and occupancy of a YAFFS2 NAND flash memory device as well as the distribution of obsolete and valid chunks are the major influences on how long obsolete data can be recovered from the device. In this chapter, we provide a practical evaluation of the possibilities to recover previous versions of a modified file or a deleted file. In Section 5.1, we introduce the tools we used for the evaluation. In Section 5.2 of this chapter, we analyze and evaluate recovery of previous versions of a modified file. Analysis and evaluation of possibilities to recover a deleted file from a YAFFS2 NAND are provided in Section 5.3. In Section 5.4, we compare the possibilities to recover modified or deleted files from YAFFS2 and NTFS devices.

5.1. Used tools and side effects

Theoretically, YAFFS2 performs file modifications as described in Section 3.2.3. However, practically, the amount of chunks that are written to a NAND by YAFFS2 in the course of a file modification depends heavily on the program used to modify the file. This is because every program, such as a text editor, uses different techniques of handling file modifications. For example, some text editors create swapfiles while performing modification of a file. This leads to writing of chunks that contain unmodified and still current content. This writing of chunks containing unmodified data can influence the amount of modified data that can be recovered from a YAFFS2 NAND. The more chunks are written to a NAND in the course of a file modification, the faster the respective block gets completely filled and transitions into state `FULL`. As only blocks in state `FULL` are candidates for garbage collection and thus possible deletion of relevant evidence, writing more chunks than necessary in the course of a file modification can reduce the amount of obsolete chunks that are recoverable from a device.

In order to analyze YAFFS2's behavior regarding file modifications as accurate as possible, influences of the program used to perform file modifications have to be kept to a minimum. To find a program that is best-suited for an analysis of YAFFS2, we tested several text editors for their influences on YAFFS2's behavior regarding file modifications. Exemplarily, dumps of a YAFFS2 NAND on which a single file was modified by use of the common Linux text editors `nano` and `vim` are depicted in Table 5.1 and Table 5.2. In the course of this tests, we created the file *"testFile"* on the NAND. The file had a size of 3000 bytes, thus occupying three of the NAND's pages. Two of these pages were used for data chunks and one for the file header chunk. We then modified *"testFile"* by overwriting the file's last five bytes. As described in Section 3.2.3, this should have resulted in the file's second data chunk along with a file header chunk being newly written to the NAND. As can be seen in Table 5.1, use of `nano` lead to rewriting of all the file's data chunks and two additional file header chunks.[1] That suggest the assumption that `nano` rewrites the whole file in the course of a file's modification. Additionally, the file's size was increased by one byte, probably by appending of a line break. Thus, `nano` is not well suited for an analysis of YAFFS2 that minimizes external influences.

Chunk no.	Content	chunkID	objectID	nBytes
0	testFile: Header	0x80000001	0x10000101	0
1	testFile: Data	1	257	2048
2	testFile: Data	2	257	952
3	testFile: Header	0x80000001	0x10000101	3000
4	root directory: Header	0x80000000	0x30000001	0
5	testFile: Header	0x80000001	0x10000101	0
6	testFile: Header	0x80000001	0x10000101	0
7	testFile: Data	1	257	2048
8	testFile: Data	2	257	953
9	testFile: Header	0x80000001	0x10000101	3001

Table 5.1.: Result of a file modification using `nano`

The common Linux text editor `vim` proved even more unsuitable for an analysis of YAFFS2 that minimizes external influences by the program used to modify files. As can be seen in Table 5.2, `vim` wrote and deleted several swapfiles during modification of a file thereby writing a large amount of chunks to the NAND.

As can be seen in Table 5.2, when performing modification of a file, `vim` did not only change the file's size and write several swapfiles, but also deleted the original file object and created a new file object with the same name as the original file object.[2] Thus, `vim` is badly suited to be used as a tool to perform file modifications during an analysis of YAFFS2's behavior. As all other text editors we tested also either wrote swapfiles or lead to other side effects, another way to perform file modification had to be used during our analyses of YAFFS2. Therefore, unless stated otherwise, we performed all file creations and modifications analyzed within this diploma thesis by use of some basic custom `C`-programs. These programs use default `C` file handling techniques in order to create and modify files with a minimum of side effects. The source code of these tools (`fileWriter`, `replacer` and `truncater`) is provided in Appendix A.1 and on the DVD delivered with this diploma thesis.[3] The tool `fileWriter` was used to create a file and write a specific number of bytes to the file. To truncate an existing file to a certain length, `truncater` was used. To overwrite a specific number of bytes at a specific point within a file, `replacer` was used. As can be seen in table 5.3, in the same scenario as described above, use of `replacer` instead of `vim` or `nano`, lead to YAFFS2 performing the file modification as described in Section 3.2.3.[4]

[1]On the attached DVD, see also: /Chapter5.1/nanddump.tools.nano.pretty
[2]On the attached DVD, see also: /Chapter5.1/nanddump.tools.vim.pretty
[3]On the attached DVD, see: /Source Code/Tools/
[4]On the attached DVD, see also: /Chapter5.1/nanddump.tools.replacer.pretty

Chunk no.	Content	chunkID	objectID	nBytes
0	testFile: Header	0x80000001	0x10000101	0
1	testFile: Data	1	257	2048
2	testFile: Data	2	257	952
3	testFile: Header	0x80000001	0x10000101	3000
4	root directory: Header	0x80000000	0x30000001	0
5	testFile.swp: Header	0x80000001	0x10000102	0
6	testFile.swpx: Header	0x80000001	0x10000103	0
7	unlinked: Header	0x80000003	0x10000103	0
8	deleted: Header	0xc0000004	0x10000103	0
9	unlinked: Header	0x80000003	0x10000102	0
10	deleted: Header	0xc0000004	0x10000102	0
11	testFile.swp: Header	0x80000001	0x10000104	0
12	testFile.swp: Data	1	260	2048
13	testFile.swp: Data	2	260	2048
14	testFile.swp: Header	0x80000001	0x10000104	4096
15	root directory: Header	0x80000000	0x30000001	0
16	testFile.swp: Data	1	260	2048
17	testFile.swp: Data	2	260	2048
18	testFile.swp: Header	0x80000001	0x10000104	4096
19	4913: Header	0x80000001	0x10000105	0
20	4913: Header	0x80000001	0x10000105	0
21	unlinked: Header	0x80000003	0x10000105	0
22	deleted: Header	0xc0000004	0x10000105	0
23	testFile : Header	0x80000001	0x10000101	3000
24	testFile: Header	0x80000001	0x10000106	0
25	testFile: Data	1	262	2048
26	testFile: Data	2	262	953
27	testFile: Header	0x80000001	0x10000106	3001
28	testFile: Header	0x80000001	0x10000106	3001
29	testFile.swp: Data	1	260	2048
30	testFile.swp: Data	2	260	2048
31	unlinked: Header	0x80000003	0x10000101	0
32	deleted: Header	0xc0000004	0x10000101	0
33	testFile.swp: Header	0x80000001	0x10000104	4096
34	unlinked: Header	0x80000003	0x10000104	0
35	deleted: Header	0xc0000004	0x10000104	0
36	root directory: Header	0x80000000	0x30000001	0

Table 5.2.: Result of a file modification using `vim`

Chunk no.	Content	chunkID	objectID	nBytes
0	testFile: Header	0x80000001	0x10000101	0
1	testFile: Data	1	257	2048
2	testFile: Data	2	257	952
3	testFile: Header	0x80000001	0x10000101	3000
4	root directory: Header	0x80000000	0x30000001	0
5	testFile: Data	2	257	952
6	testFile: Header	0x80000001	0x10000101	3000

Table 5.3.: Result of a file modification using `replacer`

All practical evaluations of YAFFS2 discussed within this chapter were performed on a simulated NAND flash memory device. The device was simulated in RAM of a Debian Linux system running kernel version 2.6.36 as described in Chapter 2. The simulated NAND featured 512 blocks and each block consisted of 64 pages with a size of 2048 bytes. Thus, the device had a storage capacity of 64 MiB. As customary, YAFFS2 reserved five of the device's blocks for checkpoint data and used a chunk size matching the device's page size. Hence, a chunk featured a size of 2048 bytes. As described in Chapter 2, for all analyses, we created images of the simulated NAND by use of `nanddump` from the `mtd-utils`.

All analyses of NTFS took place on a 62,5 MiB[5] NTFS (version 3.1) partition of a flash drive. The partition featured a cluster size of 2048 bytes and a sector size of 512 bytes. All files were created and modified the same way and in the same Linux environment as they were on the simulated YAFFS2 NAND flash memory device. For the analyses of the flash drive's NTFS partition, we created images of the partition by use of `dd`. We then analyzed these images by use of common forensic software *SleuthKit* [20] and *Autopsy* [21].

In the following, we focus on the analysis of best case and worst case scenarios regarding recovery of modified or deleted files. Obviously, a multitude of scenarios between a best case and a worst case scenario exist. However, a typical average case scenario regarding recovery of files from a YAFFS2 NAND can not be universally defined. The reason for that is, that the amount of obsolete data that can be recovered from a YAFFS2 NAND depends heavily on the way the device has been used and the time span that has passed between deletion or modification of a file and the attempt to recover the file. However, all scenarios between worst case and best case scenarios can be deduced from the worst and best case scenarios we analyze in the following.

5.2. Recovery of a modified file

In this section, we analyze and discuss recovery of obsolete chunks belonging to previous versions of a modified file from a YAFFS2 NAND flash memory device. As described

[5]65 533 952 bytes

in Chapter 4, the amount of a file's obsolete chunks recoverable from a YAFFS2 NAND depends heavily on the time span between the modification and the analysis of the device. The shorter this time span, the higher the probability that previous versions of a modified file can be recovered. Thus, an attempt to recover a file from a YAFFS2 NAND provides best possible results, if the recovery takes place right after the modification of the file. Hence, in an ideal situation, the device should be disconnected from its power source right after the file has been modified.

A smartphone's integrated NAND flash memory storage device can be disconnected from its power source by simply removing the smartphone's battery. Subsequently, an image of the device can be created by removing the device from the smartphone and connecting it to a power source without mounting the device. As, in most cases, this requires destruction of the smartphone and use of special hardware to connect the device to a forensic workstation, this was not an option for the analyses we performed for this diploma thesis. However, disconnecting a NAND flash memory device simulated in RAM from its power source is also not possible without data loss. Thus, to analyze the simulated NAND flash memory device's content, another way to create an image of the device had to be found. Creating an image of a device while the device is still mounted does not provide valuable results as YAFFS2's garbage collection modifies the device's content during creation of the image. Unmounting the device before creating an image alters the device's content but prevents garbage collection during creation of the image. However, unmounting a YAFFS2 NAND only alters the device's content to a small degree. Additionally, these alternations are predictable and do not affect the amount of recoverable obsolete chunks. On unmount of a YAFFS2 NAND flash memory device, YAFFS2 writes checkpoint data to the device, thus allocating a small number of blocks. However, YAFFS2 always keeps a sufficient number of blocks empty in order to store checkpoint data so that no data has to be deleted in order to store checkpoint data. Hence, for all analyses performed for this diploma thesis, we unmounted the simulated NAND flash memory device right after the last modification of a file on the device in order to obtain best possible results. In case a file was modified several times, no unmount and mount operations were performed between the individual modifications, as mounting a YAFFS2 NAND flash memory device alters the device's content to a much higher degree than unmounting. This, as described in Chapter 4, is because the first passive garbage collection cycle after mounting is always performed as block refreshing. Therefore, in case several modifications of a file were performed consecutively, we unmounted the device after every modification to create an image and subsequently completely erased the device. Thus, in order to perform the next modifications, all previous modifications had to be performed again. The created images could not be written back to the device as subsequently mounting the device would have changed the device's content. In the following, we discuss and analyze worst case scenarios regarding recovery of a modified file's previous versions.

5.2.1. Worst case

A worst case scenario regarding recovery of a specific file is given if no previous versions of the file can be recovered after a modification of the file. This is the case if all chunks that become obsolete after the modification are located on blocks that are erased by garbage collection very quickly after the modification was performed.

In Chapter 4 of this diploma thesis, we introduced worst case scenarios regarding recovery of files from a YAFFS2 NAND. These scenarios focus on the overall recoverability of obsolete chunks from a device. However, a worst case scenario regarding recovery of one specific file does not necessarily have to constitute an overall worst case scenario regarding recovery of obsolete chunks from a YAFFS2 NAND. Additionally, an overall best case scenario can still include a worst case scenario regarding recovery of one specific file. In the following, we analyze worst case scenarios regarding recovery of a specific file in overall best case scenarios as well as in overall worst case scenarios.

The most basic worst case scenario regarding recovery of a specific file in an overall best case scenario consists of one file filling up all chunks but one of a block on an otherwise empty device. The chunk not used by the file is used for the root directory's header chunk. Modifying the file in a way that makes up to one half of the file's chunks obsolete constitutes an overall best case scenario as described in Chapter 4. This is because the only way the block containing the obsolete chunks can be erased is by oldest dirty garbage collection. Aggressive garbage collection is not necessary as the device's storage capacity is hardly used. Additionally, garbage collection is not triggered by write operations as most free chunks are located within empty blocks. Background garbage collection can also not be executed as no block features enough obsolete blocks to be a candidate for background garbage collection. Thus, theoretically, one previous version of the modified file can be recovered for a very short timespan after the modification. However, as described in Chapter 4, the block containing the obsolete chunk should be erased 46 seconds at most after the file's modification.

To evaluate this scenario, we created a file named *"fileA"* with a size of 126 976 bytes on an otherwise empty device with a block size of 131 072 bytes as described in section 5.1. Thus, the file needed 62 of the block's chunks for its content and one for its file header chunk. In Table 5.4, the initial state of the YAFFS2 NAND before the modification of *"fileA"* was performed is depicted. At this point, one oldest dirty garbage collection cycle had already been performed due to the obsolete file header chunk written to the device's first block during creation of the file (see Section 3.2.3). Therefore, initially all of the file's chunks are located in the device's second block.[6]

Subsequently, we overwrote the file's last 30 data chunks. As *"fileA"* consisted of 62 data chunks and one header chunk, overwriting 30 of the file's data chunks resulted in 31 chunks of the block becoming obsolete. As a block featured 64 chunks, 30 data chunks was the maximum amount of the file's data chunks that could be overwritten without making the device's second block a valid candidate for background garbage collection. Hence, in this case, one previous version of *"fileA"* should have been recoverable directly after we performed the modification but not for longer than 46 seconds at most. In Table 5.5, a dump[7] of the device created directly after the last 30 data chunks of *"fileA"* had been overwritten and the device had been unmounted is depicted.

As can be seen in Table 5.5, a complete previous version of *"fileA"* was recoverable

[6]On the attached DVD, see also: /Chapter5.1/nanddump.smallWorstInOverallBest.initial.pretty
[7]On the attached DVD, see also: /Chapter5.2/nanddump.smallWorstInOverallBest.mod.pretty

Chunk no.	Content	chunkID	objectID	nBytes
0	-	-	-	-
...	...	...	...	...
63	-	-	-	-
64	root directory: Header	0x80000000	0x30000001	0
65	fileA: Data	1	257	2048
66	fileA: Data	2	257	2048
...	...	...	...	...
126	fileA: Data	62	257	2048
127	fileA: Header	0x80000001	0x10000101	126976
128	-	-	-	-
...	...	...	...	...

Table 5.4.: Initial state before modification of *"fileA"*

Chunk no.	Content	chunkID	objectID	nBytes
0	Checkpoint data	1	2	2048
1	Checkpoint data	2	2	2048
2	Checkpoint data	3	2	2048
3	Checkpoint data	4	2	2048
4	Checkpoint data	5	2	2048
5	-	-	-	-
...	...	...	...	...
63	-	-	-	-
64	root directory: Header	0x80000000	0x30000001	0
65	fileA: Data	1	257	2048
66	fileA: Data	2	257	2048
...	...	...	...	...
126	fileA: Data	62	257	2048
127	fileA: Header	0x80000001	0x10000101	126976
128	fileA: Data	33	257	2048
129	fileA: Data	34	257	2048
...	...	...	...	...
157	fileA: Data	62	257	2048
158	fileA: Header	0x80000001	0x10000101	126976
169	-	-	-	-
...	...	...	...	...

Table 5.5.: State directly after overwriting the last 30 chunks of *"fileA"* and unmounting the NAND in an overall best case scenario

from the device directly after the modification was performed. However, as depicted in Listing 5.1, only 18 seconds after the modification was fully performed, no previous version of *“fileA”* could be recovered anymore because the device’s second block was deleted due to oldest dirty garbage collection. As can bee seen in Listing 5.1, eight attempts to perform background garbage collections had already failed before *“fileA”* was modified. Thus, only two more failed attempts of background garbage collection sufficed to trigger oldest dirty garbage collection. The complete log file can be found on the attached DVD.[8]

```
21:00:11 debian-DA4 kernel: [38480.876652] yaffs_file_write about to write writing
         4096(1000) bytesto object 257 at 65536(10000)
21:00:11 debian-DA4 kernel: [38480.876673] yaffs_MarkSuperBlockDirty() sb = f5219600
21:00:11 debian-DA4 kernel: [38480.876688] Allocated block 3, seq  4099, 510 left
[...]
21:00:11 debian-DA4 kernel: [38480.946432] Writing chunk 158 tags 257 0
[...]
21:00:13 debian-DA4 kernel: [38482.731067] Background gc 0
21:00:13 debian-DA4 kernel: [38482.731086] GC none: finder 478 skip 9 threshold 20
         dirtiest 0 using 0 oldest 0 bg
[...]
21:00:15 debian-DA4 kernel: [38484.748789] Background gc 0
21:00:15 debian-DA4 kernel: [38484.748807] GC none: finder 511 skip 10 threshold 22
         dirtiest 0 using 0 oldest 0 bg
[...]
21:00:17 debian-DA4 kernel: [38486.762048] Background gc 0
21:00:17 debian-DA4 kernel: [38486.762071] GC Selected block 2 with 31 free,
         prioritised:0
[...]
21:00:29 debian-DA4 kernel: [38498.848095] Erased block 2
```

Listing 5.1: Deletion of the device’s second block 18 seconds after modification of *”fileA”* in an overall best case scenario

As can be seen, even in an overall best case scenario, previous versions of a modified file can only be recovered within a certain time span. In the following, we discuss and analyze worst case scenarios regarding recovery of a specific file from a YAFFS2 NAND flash memory device in an overall worst case scenario.

As discussed in Chapter 4, an overall worst case scenario is given if every check of a dirty block for necessity of garbage collection leads to execution of garbage collection and every execution of garbage collection leads to a maximum of deleted obsolete chunks. Hence, the scenario we analyzed above can easily be transformed into an overall worst case scenario by modifying *“fileA”* in a way that makes more than half of the device’s second block’s chunks obsolete. As discussed above, this leads to the device’s second block being selected for garbage collected two seconds at most after the modification is performed. However, as passive garbage collection only collects five chunks per garbage collection cycle, the block is not erased immediately.

Therefore, a more interesting overall worst case scenario is an overall worst case scenario that includes aggressive garbage collection. Aggressive garbage collection always collects the whole block selected for garbage collection and thus erases a block much faster than passive garbage collection. Such an overall worst case scenario including a worst case scenario regarding recovery of a specific file can be created by filling up a

[8]On the attached DVD, see /Chapter5.2/kern.log.smallWorstInOverallBest.rtf

YAFFS2 NAND to an extend that makes aggressive garbage collection necessary once the file in question is modified. In case all of the modified file's obsolete chunks are located in the block being garbage collected aggressively, no previous version of the file should be recoverable after only a very short timespan. In order to create such a scenario, we performed the following steps:

1. Creation of a large file with a file size of 65 667 072 bytes
2. Creation of a smaller file (*"fileB"*) with a file size of 192 512 bytes
3. Overwriting of data chunks 23 to 53 of *"fileB"*

As for each file a file header chunk was written to the device as well as an additional object header chunk for the device's root directory, 502 of the device's blocks were completely filled with valid chunks after creation of the files and garbage collection of obsolete file header chunks that had been written during the files' creation. Additionally, one further block featured 33 valid chunks. As can be seen in Table 5.6, these 33 chunks were data chunks 23 to 55 of *"fileB"* stored in the device's Block 512.[9]

Chunk no.	Content	chunkID	objectID	nBytes
...	...	...	...	...
32704	fileB: Data	23	258	2048
32705	fileB: Data	24	258	2048
...	...	...	...	...
32736	fileB: Data	55	258	2048
32737	-	-	-	-
...	...	...	...	...
32767	-	-	-	-

Table 5.6.: Initial content of Block 512 before modification of *"fileB"*

As described in Chapter 3, aggressive garbage collection is necessary if a device does not feature enough empty blocks to store checkpoint data. This is the case if less than n empty blocks exist on the device with n being defined as follows:

$$\begin{aligned} n = & \text{ number of reserved blocks} \\ & + \text{number of complete blocks actually necessary to store current checkpoint data} \\ & - \text{number of blocks currently used for checkpoint data} \\ & + 4 \end{aligned}$$

In this scenario, no checkpoint data was stored on the device and storing checkpoint data for the two files on the device would have needed less than a complete block. Thus, as YAFFS2 reserves five blocks for checkpoint data by default, nine blocks had to be kept free at all time to store checkpoint data. As above-mentioned, after creation of the two files, 502 blocks were completely filled and one block featured 33 valid chunks. Therefore, nine completely empty blocks existed on the device before *"fileB"* was modified.

[9]On the attached DVD, see also: /Chapter5.1/nanddump.worstInOverallWorst.initial.pretty

Overwriting chunks 23 to 53 of *"fileB"* lead to writing of 31 data chunks and a new file header chunk for the file. Thus, performing the modification completely filled Block 512 and occupied one chunk of a block reserved for checkpoint data, in this case the device's first block. Hence, aggressive garbage collection had to be performed immediately after *"fileB"* was modified. As can be seen in Listing 5.2, Block 512 containing all of the file's obsolete chunks was erased only two seconds[10] after the file's header chunk had been written to the device's first block. Thus, only two seconds after the modification was performed, no obsolete chunks of *"fileB"* could be recovered anymore. In case aggressive garbage collection had not been necessary, oldest dirty garbage collection would have been necessary to erase the block, as more than half of the block's chunks were valid. This shows, that aggressive garbage collection leads to much faster deletion of potentially recoverable data than passive garbage collection does.

```
[...]
14:04:25 debian-DA4 kernel: [17389.830753] Allocated block 1, seq  4609, 8 left
14:04:25 debian-DA4 kernel: [17389.830771] Writing chunk 0 tags 258 0
[...]
14:04:27 debian-DA4 kernel: [17391.620181] GC Selected block 512 with 31 free, prioritised:0
14:04:27 debian-DA4 kernel: [17391.620197] yaffs: GC erasedBlocks 8 aggressive 1
[...]
14:04:27 debian-DA4 kernel: [17391.660251] Erased block 512
```

Listing 5.2: Deletion of Block 512 two seconds after modification of *"fileB"*

In the following, we analyze best case scenarios regarding recovery of obsolete chunks after file modifications.

5.2.2. Best case

Repeated modification of a file can lead to several different versions of the file being stored on a YAFFS2 NAND. These obsolete versions of the file stay on the device until the respective chunks are deleted by garbage collection. A best case scenario regarding recovery of a specific file's previous versions is given if all previous versions of a modified file are recoverable from a YAFFS2 NAND for the longest time possible. As discussed in Chapter 4, overall best case scenarios can only temporarily exist on a device, as garbage collection constantly changes the device's state. Best case scenarios regarding recovery of a specific file's previous versions can also only exist temporarily as garbage collection erases blocks containing obsolete chunks sooner or later. In the following, we analyze best case scenarios regarding recovery of previous versions of a specific file.

As described in Chapter 4, an overall best case scenario regarding recovery of modified files features a distribution of obsolete and valid chunks on a device that enables no garbage collection but oldest dirty garbage collection. In that case, obsolete chunks stay on the device for the longest time possible. Thus, recovery of a maximum of a file's obsolete chunks is possible if all of these chunks are located in the device's most recently written blocks. Hence, a best case scenario regarding recovery of a specific file after modifications of the file includes an overall best case scenario. Additionally, the file's obsolete chunks have to be located in the device's most currently written blocks in order

[10]On the attached DvD, see also: /Chapter5.2/kern.log.worstInOverallWorst.rtf

to be recoverable for as long as possible. As an overall best case scenario only features oldest dirty garbage collection, a block is only erased after all blocks featuring lower sequence numbers have been erased. Therefore, a modified file's obsolete chunks can be recovered for a longer time if the device's storage capacity is used to a large extent but not as much as to make garbage collection other than oldest dirty garbage collection necessary. Additionally, in this case, the time span in which recovery of a file's obsolete chunks is possible can be longer on a larger NAND that features a higher number of blocks.

As depicted in Listing 5.1, modification of a file in an overall best case scenario on an otherwise empty YAFFS2 NAND leads to quick deletion of the file's obsolete chunks. As described above, a modified file's obsolete chunks should be recoverable for much longer if the device's storage capacity is used to a larger extend. To verify this hypothesis, we created an overall best case scenario on a YAFFS2 NAND with ten blocks in use. Although the device's storage capacity was still only used to a small extend in this scenario, this scenario is sufficient to show significant increase of the timespan in which recovery of a modified file's obsolete chunks was possible. To create a stable initial state on the YAFFS2 NAND, we created ten files on the device as follows:

1. Creation of *"file1"* (126 976 bytes, respectively 62 data chunks)
2. Creation of *"file2"* (129 024 bytes, respectively 63 data chunks)
3. Creation of *"file3"* (129 024 bytes, respectively 63 data chunks)
4. Creation of *"file4"* (129 024 bytes, respectively 63 data chunks)
5. Creation of *"file5"* (129 024 bytes, respectively 63 data chunks)
6. Creation of *"file6"* (129 024 bytes, respectively 63 data chunks)
7. Creation of *"file7"* (129 024 bytes, respectively 63 data chunks)
8. Creation of *"file8"* (129 024 bytes, respectively 63 data chunks)
9. Creation of *"file9"* (129 024 bytes, respectively 63 data chunks)
10. Creation of *"file10"* (129 024 bytes, respectively 63 data chunks)

After each file creation we waited for garbage collection to be performed before we created the next file. Thus, after all obsolete header chunks had been erased, ten of the device's blocks were completely filled with valid chunks. Of these chunks, 629 were data chunks, one chunk was the root directory's header chunk and ten chunks were used as file header chunks. At this point, no garbage collection was performed anymore, as no obsolete chunks existed on the device. Creation of the above-mentioned files and garbage collection lead to the block states[11] depicted in Table 5.7.

Subsequently, we created a file named *"fileC"* with a file size of 26 624 bytes, respectively thirteen data chunks. Including the file header chunks and the root directory's header chunk that were written to the NAND during creation of *"fileC"*, this occupied sixteen chunks of Block 30. Additionally, we modified all other files on the device in a way that overwrote one chunk of every file. In this, we made sure that each overwritten

[11]On the attached DVD, see also: /Chapter5.1/nandump.bestInOverallBest.initial.pretty

Block no.	**Block state**
1	`EMPTY`
2	`EMPTY`
3	`EMPTY`
4	`FULL`
5	`EMPTY`
6	`EMPTY`
7	`FULL`
8	`EMPTY`
9	`EMPTY`
10	`FULL`
11	`EMPTY`
12	`EMPTY`
13	`FULL`
14	`EMPTY`
15	`EMPTY`
16	`FULL`
17	`EMPTY`
18	`EMPTY`
19	`FULL`
20	`EMPTY`
21	`EMPTY`
22	`FULL`
23	`EMPTY`
24	`EMPTY`
25	`FULL`
26	`EMPTY`
27	`EMPTY`
28	`FULL`
29	`FULL`
...	`EMPTY`

Table 5.7.: Initial state before creation of *"fileC"*

chunk was located in another block so that each block featured at least one obsolete chunk after the modifications. Immediately after these modifications, *"fileC"* was modified, so that no garbage collection was executed between the modifications. To create a high number of different versions of *"fileC"*, the file's first chunk was overwritten 29 consecutive times. Thus, as can be seen in Table 5.8 and Table 5.9, before garbage collection was performed, 31 obsolete chunks of *"fileC"* could be found in Block 30 and 28 obsolete chunks of *"fileC"* in Block 31. Therefore, 29 different previous versions of *"fileC"* were completely recoverable directly after the file's last modification. As each of the device's blocks featured obsolete chunks and no block featured enough obsolete chunks to make garbage collection other than oldest dirty garbage collection necessary, Blocks 30 and 31 were not erased until all other blocks had been erased. As can be seen in Listing 5.3, Block 30 was erased eight minutes and three seconds[12] after the last modification of *"fileC"* had been performed. Until then, all previous version of *"fileC"* were recoverable. After deletion of Block 30, still 28 obsolete chunks of *"fileC"* could be found on Block 31. Thus, fourteen previous versions of *"fileC"* could still be recovered until Block 31 was finally erased eight minutes and 39 seconds after *"fileC"* had been modified for the last time.

Chunk no.	**Content**	**chunkID**	**objectID**	**nBytes**
..	..	..	..	
1920	fileC: Data	1	267	2048
1921	fileC: Header	0x80000001	0x1000010b	26624
1922	fileC: Data	1	267	2048
1923	fileC: Header	0x80000001	0x1000010b	26624
1924	fileC: Data	1	267	2048
1925	fileC: Header	0x80000001	0x1000010b	26624
1926	fileC: Data	1	267	2048
1927	fileC: Header	0x80000001	0x1000010b	26624
...	...	...	...	...
1948	fileC: Data	1	267	2048
1949	fileC: Header	0x80000001	0x1000010b	26624
1950	-	-	-	-
...	...	...	...	...
1983	-	-	-	-
...	...	...	...	...

Table 5.8.: Block 31 after modification of all files on the device

As can be seen, given ideal conditions, all of a file's previous versions and obsolete data can be recovered from a YAFFS2 NAND flash memory device for a relatively long time. In the above-described scenario, only ten of the device's blocks were in use before the file to be recovered was created and modified. Given a larger device and more blocks in use that have to be garbage collected before the blocks containing obsolete chunks of the file in question, previous versions of the file can be recovered up to several days after

[12]On the attached DVD, see also: /Chapter5.1/kern.log.bestInOverallBest.rtf

Chunk no.	**Content**	**chunkID**	**objectID**	**nBytes**
..	..	..	..	
1856	fileC: Header	0x80000001	0x1000010b	0
1857	root directory: Header	0x80000000	0x30000001	0
1858	fileC: Data	1	267	2048
..	..	..	..	
1870	fileC: Data	13	267	2048
1871	fileC: Header	0x80000001	0x1000010b	26624
1872	file1: Data	62	257	2048
1873	file1: Header	0x80000001	0x10000101	126976
1874	file2: Data	61	258	2048
1875	file2: Header	0x80000001	0x10000102	129024
..	..	..	..	
1889	file9: Header	0x80000001	0x10000109	129024
1890	file10: Data	57	266	2048
1891	file10: Header	0x80000001	0x1000010a	129024
1892	fileC: Data	1	267	2048
1893	fileC: Header	0x80000001	0x1000010b	26624
1894	fileC: Data	1	267	2048
1895	fileC: Header	0x80000001	0x1000010b	26624
1896	fileC: Data	1	267	2048
1897	fileC: Header	0x80000001	0x1000010b	26624
1898	fileC: Data	1	267	2048
1899	fileC: Header	0x80000001	0x1000010b	26624
1900	fileC: Data	1	267	2048
1901	fileC: Header	0x80000001	0x1000010b	26624
1902	fileC: Data	1	267	2048
1903	fileC: Header	0x80000001	0x1000010b	26624
1904	fileC: Data	1	267	2048
1905	fileC: Header	0x80000001	0x1000010b	26624
1906	fileC: Data	1	267	2048
1907	fileC: Header	0x80000001	0x1000010b	26624
1908	fileC: Data	1	267	2048
1909	fileC: Header	0x80000001	0x1000010b	26624
1910	fileC: Data	1	267	2048
1911	fileC: Header	0x80000001	0x1000010b	26624
1912	fileC: Data	1	267	2048
1913	fileC: Header	0x80000001	0x1000010b	26624
1914	fileC: Data	1	267	2048
1915	fileC: Header	0x80000001	0x1000010b	26624
1916	fileC: Data	1	267	2048
1917	fileC: Header	0x80000001	0x1000010b	26624
1918	fileC: Data	1	267	2048
1919	fileC: Header	0x80000001	0x1000010b	26624
..	..	..	..	

Table 5.9.: Block 30 after modification of all files on the device

```
15:25:56 debian-DA4 kernel: [19744.687465] yaffs_mknod: making oject for fileC,
         mode 81a4 dev 0
[...]
15:26:07 debian-DA4 kernel: [19754.946618] Allocated block 30, seq  4126, 501 left
[...]
15:26:11 debian-DA4 kernel: [19759.672286] Allocated block 31, seq  4127, 500 left
[...]
15:26:12 debian-DA4 kernel: [19760.274180] Writing chunk 1949 tags 267 0
15:26:12 debian-DA4 kernel: [19760.274196] nandmtd2_WriteChunkWithTagsToNAND chunk 1949
         data e70e5800 tags f2dcfeb0
15:26:12 debian-DA4 kernel: [19760.274545] packed tags obj 268435723 chunk -2147483647
          byte 26624 seq 4127
15:26:12 debian-DA4 kernel: [19760.274577] ext.tags eccres 0 blkbad 0 chused 1 obj 267
         chunk0 byte 0 del 0 ser 31 seq 4127
[...]
15:26:21 debian-DA4 kernel: [19769.056367] GC Selected block 4 with 2 free,prioritised:0
[...]
15:26:45 debian-DA4 kernel: [19793.194822] Erased block 4
[...]
15:27:07 debian-DA4 kernel: [19815.321721] GC Selected block 7 with 3 free,prioritised:0
[...]
15:27:31 debian-DA4 kernel: [19839.485312] Erased block 7
[...]
15:27:53 debian-DA4 kernel: [19861.604232] GC Selected block 10 with 2 free,prioritised:0
[...]
15:28:17 debian-DA4 kernel: [19885.786414] Erased block 10
[...]
15:28:39 debian-DA4 kernel: [19907.900224] GC Selected block 13 with 2 free,prioritised:0
[...]
15:29:04 debian-DA4 kernel: [19932.053038] Erased block 13
[...]
15:29:26 debian-DA4 kernel: [19954.149845] GC Selected block 16 with 2 free,prioritised:0
[...]
15:29:50 debian-DA4 kernel: [19978.282276] Erased block 16
[...]
15:30:12 debian-DA4 kernel: [20000.304321] GC Selected block 19 with 2 free,prioritised:0
[...]
15:30:36 debian-DA4 kernel: [20024.474886] Erased block 19
[...]
15:30:58 debian-DA4 kernel: [20046.540725] GC Selected block 22 with 2 free,prioritised:0
[...]
15:31:22 debian-DA4 kernel: [20070.612892] Erased block 22
[...]
15:31:44 debian-DA4 kernel: [20092.704417] GC Selected block 25 with 2 free,prioritised:0
[...]
15:32:08 debian-DA4 kernel: [20116.768829] Erased block 25
[...]
15:32:30 debian-DA4 kernel: [20138.893062] GC Selected block 28 with 2 free,prioritised:0
[...]
15:32:55 debian-DA4 kernel: [20162.960199] Erased block 28
[...]
15:33:17 debian-DA4 kernel: [20185.088337] GC Selected block 29 with 2 free,prioritised:0
[...]
15:33:41 debian-DA4 kernel: [20209.254728] Erased block 29
[...]
15:34:03 debian-DA4 kernel: [20231.304609] GC Selected block 30 with 31 free,prioritised:0
[...]
15:34:15 debian-DA4 kernel: [20243.339236] Erased block 30
[...]
15:34:37 debian-DA4 kernel: [20265.408944] GC Selected block 31 with 28 free,prioritised:0
[...]
15:34:51 debian-DA4 kernel: [20279.446441] Erased block 31
[...]
```

Listing 5.3: Oldest dirty garbage collection of blocks 30 and 31

the modification has been performed. However, as long as the device is mounted, a file's previous versions are erased from the device at some point in any case, even if no further write operations are performed on the device by the user or the operating system.

5.3. Recovery of a deleted file

In a forensic analysis, recovery of deleted files can reveal important data that was once stored on a device. In this section, we discuss and analyze possibilities to recover deleted files from a YAFFS2 NAND flash memory device.

As described in Chapter 3, YAFFS2 uses `deleted` and `unlinked` header chunks to mark an object as deleted. Hence, an object is recoverable from a YAFFS2 NAND until garbage collection deletes all of the object's chunks. Although recovery of a specific deleted file does not differ fundamentally from recovery of a specific modified file, one important difference exists. As show in Chapter 3 and Chapter 4, YAFFS2's `deleted` header chunk is always marked with a shrink header marker. A block containing a chunk marked with a shrink header marked is disqualified for garbage collection until the block gets the oldest dirty block. Thus, obsolete chunks on such a block can potentially be recovered for a longer time than obsolete chunks on blocks that feature no chunks marked with shrink header markers. Considering this, several variants of best case and worst case scenarios regarding recovery of a deleted file exist. However, only a few of these scenarios depend on shrink header markers. Those scenarios not depending on shrink header markers do not differ from the scenarios we described in Section 5.2. In the following, we introduce best case and worst case scenarios regarding recovery of a specific deleted file and practically analyze those scenarios that differ from scenarios already described in Section 5.2.

5.3.1. Worst case

A worst case scenario regarding recovery of a specific deleted file is a scenario in which none of the file's obsolete chunks can be recovered. This is the case if the blocks containing the deleted file's obsolete chunks get erased by garbage collection soon after the file was deleted. Basically, two variants of such a worst case scenario exist. Either all of the file's chunks are located in the same block as the file's `deleted` header chunk or the file's `deleted` header chunk is located in a block containing none of the deleted file's chunks.

In case the deleted file's `deleted` header chunk is located in a block containing none of the deleted file's chunks, garbage collection of the blocks containing the deleted file's chunks is independent of the `deleted` header chunk's shrink header marker. In this case, recovery of a deleted file does not differ from recovery of previous versions of a modified file as described in Section 5.2.

If all of a deleted file's chunks are stored in the same block as the file's `deleted` header chunk, a worst case scenario regarding recovery of this file always requires the file's block to be the device's oldest block. Only then, the block is not disqualified for garbage collection and can be erased directly after the file has been deleted. Thus, such a worst case scenario is equivalent to a worst case scenario regarding recovery of a modified file in an overall best case scenario as we presented in Section 5.2.1.

5.3.2. Best case

A best case scenario regarding recovery of a delete file is a scenario in which the deleted file is completely recoverable for the longest time possible. In contrast to the worst case scenarios discussed above, some best case scenarios regarding recovery of a deleted file differ greatly from best case scenarios regarding recovery of modified files. In an absolute best case scenario regarding recovery of a deleted file, the file's `deleted` header chunk has to be stored in the same block as all of the file's chunks. If the `deleted` header chunk is stored in a block that contains none of the deleted file's chunks, this block has been more currently written then all blocks containing chunks of the deleted file. As a block containing a `deleted` header chunk can only be erased if it is the oldest dirty block, it cannot be erased before the blocks containing the deleted file's chunks. Thus, this scenario cannot constitute a best case as the block containing the deleted file's obsolete chunks are not erased at the latest possible moment. However, in this case, the deleted file can still be recovered for a relatively long time if the distribution of obsolete and valid chunks constitute a best case scenario as described in Section 5.2.

Hence, an absolute best case scenario regarding recovery of a specific deleted file requires all of the deleted file's chunks to be stored in the same block as the file's `deleted` header chunk. However, such a best case scenario does not necessarily need a complete overall best case scenario. As shown before, an overall best case scenario only features oldest dirty garbage collection. As a block containing a `deleted` header chunk can only be garbage collected if it is the device's oldest block, it does not need to feature a minimum amount of valid chunks to be disqualified for garbage collection. Thus, a best case scenario regarding recovery of a deleted file is a scenario where all block except for the block containing the deleted file's chunks and its `deleted` header must comply to the criteria of an overall best case scenario. Although the block containing the deleted file's chunks and its `deleted` header chunk can comply to these criteria, it does not necessarily have to, as it is protected from garbage collection by the `deleted` header chunk's shrink header marker.

To verify this hypothesis, we performed an analysis of a variant of the overall best case scenario analyzed in Section 5.2. Similar to the approach we presented in Section 5.2, we created ten files to fill exactly ten of the device's blocks with valid chunks. This lead to the same block states[13] as depicted in Table 5.7. After creation of this stable initial state, we performed the following steps on the device:

1. Creation of *"fileD"* (77 824 bytes, respectively 38 data chunks)
2. Modification of all files on the device except for *"fileD"*
3. Deletion of *"fileD"*

[13]On the attached DVD, see also: /Chapter5.3/nanddump.bestCaseDeletion.initial.pretty

To modify the ten initial files we overwrote one data chunk of each file in a way that lead to one obsolete data chunk in each of the ten initially filled blocks. Hence, featuring only a very small number of obsolete chunks, these blocks complied to the criteria of an overall best case scenario. However, the block containing the chunks written due to execution of the above-mentioned steps, did not comply to the criteria of an overall best case scenario. As depicted in Table 5.10, Block 30 of the device was allocated to store *"fileD"* and the chunks written due to the initial files' modification. Due to deletion of *"fileD"*, Block 30 contained 43 obsolete chunks and thus was a valid candidate for regular background garbage collection. However, as can be seen in Listing 5.4, Block 30 was still only collected after all other blocks containing obsolete chunks had been collected by oldest dirty garbage collection. This shows, that the `deleted` header chunk's shrink header marker protected Block 30 from being garbage collected until the block became the oldest block containing obsolete chunks.

Chunk no.	**Content**	**chunkID**	**objectID**	**nBytes**
..	..	..	..	
1856	fileD: Header	0x80000001	0x1000010b	0
1857	root directory: Header	0x80000000	0x30000001	0
1858	fileD: Data	1	267	2048
1859	fileD: Data	2	267	2048
...	...	...	...	...
1895	fileD: Data	38	267	2048
1896	fileD: Header	0x80000001	0x1000010b	77824
1897	file1: Data	62	257	2048
1898	file1: Header	0x80000001	0x10000101	126976
1899	file2: Data	61	258	2048
1900	file2: Header	0x80000001	0x10000102	129024
...	...	...	...	...
1915	file10: Data	57	266	2048
1916	file10: Header	0x80000001	0x1000010a	129024
1917	unlinked: Header	0x80000003	0x1000010b	0
1918	root directory: Header	0x80000000	0x30000001	0
1919	deleted: Header	0xC0000004	0x1000010b	0
..	..	..	..	

Table 5.10.: Block 30 after deletion of *"fileD"*

As can be seen in Listing 5.4, Block 30 was erased seven minutes and 53 seconds after *"fileD"* was deleted. The block was selected for garbage collection after background garbage collection was skipped ten consecutive times. However, the reason for that was not, that Block 30 was disqualified for regular background garbage collection. As depicted in Listing 5.4, all attempts of background garbage collection were skipped because Block 30 was not checked for necessity of garbage collection during these attempts. Thus, Block 30 was not selected for regular background garbage collection immediately after it became the only dirty block, although that would have been possible.[14] This shows, that obsolete chunks can potentially be recovered for a longer time from a larger NAND than from a small NAND as passive garbage collection only checks a subset of all blocks when trying to select a block to garbage collect.

```
17:33:00 debian-DA4 kernel: [23441.835541] nandmtd2_WriteChunkWithTagsToNAND
         chunk 1919 e5b8a800 tags e5917e7c
17:33:00 debian-DA4 kernel: [23441.835560] packed tags obj 268435723 chunk -1073741820
         byte 0 seq 4126
17:33:00 debian-DA4 kernel: [23441.835580] ext.tags eccres 0 blkbad 0 chused 1 obj 267
         chunk0 byte 0 del 0 ser 4 seq 4126
[...]
17:33:06 debian-DA4 kernel: [23449.874065] GC none: finder 467 skip 10 threshold 22
         dirtiest 0 using 0 oldest 0 bg
[...]
17:33:08 debian-DA4 kernel: [23451.889709] GC Selected block 4 with 2 free,prioritised:0
[...]
17:33:33 debian-DA4 kernel: [23476.023857] Erased block 4
[...]
17:33:55 debian-DA4 kernel: [23498.125647] GC Selected block 7 with 3 free,prioritised:0
[...]
17:34:19 debian-DA4 kernel: [23522.191842] Erased block 7
[...]
17:34:41 debian-DA4 kernel: [23544.229238] GC Selected block 10 with 2 free,prioritised:0
[...]
17:35:05 debian-DA4 kernel: [23568.307565] Erased block 10
[...]
17:35:27 debian-DA4 kernel: [23590.364772] GC Selected block 13 with 2 free,prioritised:0
[...]
17:35:51 debian-DA4 kernel: [23614.419809] Erased block 13
[...]
17:36:13 debian-DA4 kernel: [23636.489919] GC Selected block 16 with 2 free,prioritised:0
[...]
17:36:37 debian-DA4 kernel: [23660.639091] Erased block 16
[...]
17:36:59 debian-DA4 kernel: [23682.741816] GC Selected block 19 with 2 free,prioritised:0
[...]
17:37:23 debian-DA4 kernel: [23706.875099] Erased block 19
[...]
17:37:46 debian-DA4 kernel: [23728.994192] GC Selected block 22 with 2 free,prioritised:0
[...]
17:38:10 debian-DA4 kernel: [23753.157933] Erased block 22
[...]
17:38:32 debian-DA4 kernel: [23775.307320] GC Selected block 25 with 2 free,prioritised:0
[...]
17:38:56 debian-DA4 kernel: [23799.487782] Erased block 25
[...]
17:39:18 debian-DA4 kernel: [23821.605934] GC Selected block 28 with 2 free,prioritised:0
[...]
17:39:42 debian-DA4 kernel: [23845.755454] Erased block 28
[...]
17:40:04 debian-DA4 kernel: [23867.857559] GC Selected block 29 with 2 free,prioritised:0
[...]
17:40:29 debian-DA4 kernel: [23892.007087] Erased block 29
[...]
17:40:31 debian-DA4 kernel: [23894.015366] GC none: finder 216 skip 1 threshold 4
         dirtiest 0 using 0 oldest 0 bg
[...]
17:40:49 debian-DA4 kernel: [23912.124845] GC none: finder 1 skip 10 threshold 22
         dirtiest 0 using 0 oldest 0 bg
[...]
17:40:51 debian-DA4 kernel: [23914.140462] GC Selected block 30 with 43 free,prioritised:0
[...]
17:40:59 debian-DA4 kernel: [23922.194880] Erased block 30
[...]
```

Listing 5.4: Oldest dirty garbage collection of block 30

[14]On the attached DVD, see also: /Chapter5.3/kern.bestCaseDeletion.rtf

As this analysis shows, shrink header markers protect a block from being garbage collected until it becomes the oldest dirty block. Thus, if all of a deleted file's chunks are stored in the same block as the file's `deleted` header chunk, in an otherwise overall best case scenario, the block does not have to comply to the criteria of an overall best case scenario in order to be selected for garbage collection at the latest moment possible.

5.4. YAFFS2 in comparison to NTFS

In this section we compare the file systems YAFFS2 and NTFS regarding recovery of modified or deleted files. In Section 5.4.1 we introduce the basic functionality of NTFS. In Section 5.4.2 and Section 5.4.3 we analyze possibilities to recover modified or deleted files from a NTFS device in scenarios similar to those we presented in Section 5.2 and Section 5.3.

5.4.1. Introduction to NTFS

NTFS (*New Technology File System*) is a file system developed by Microsoft and is widely used on computers running the Windows operating system. Although NTFS is closed source software, it has been widely analyzed during the last years and its functionality is well understood. In the following, only a brief introduction to the basic functionality of NTFS is provided. Among others, detailed information about NFTS and the forensic analysis of this file system can be found in Brian Carrier's *File System Forensic Analysis* [22].

NTFS uses its so-called MFT (*Master File Table*) as a central structure in which all information about all files on a NTFS device are stored [23, 22]. NTFS treats all objects on a device as files, even the basic file system administrative data such as the MFT itself. Therefore, information about all objects on a NTFS device can be found within the device's MFT. To store information about a file in the MFT, a *MFT record* for the file containing several attributes is stored in the MFT. By default, NTFS uses the attributes depicted in Table 5.11 [22]. For our analysis, especially the `$DATA` attribute is of relevance. The `$DATA` attribute is used to store a file's content. In case a file is very small, the file's content is stored inside the `$DATA` attribute of the file's MTF record. During our analyses, files containing up to 648 bytes were stored that way. Larger files' content is stored in external clusters outside the MFT. In that case, the `$DATA` attribute contains information on where on the device the files' content can be found. For that purpose, so-called *cluster runs* are stored in the `$DATA` attribute within a file's MFT record. Each of these cluster runs contains a pointer to a cluster along with information on how many clusters starting from the cluster pointed to have to be read to find a file's content. In case a file's content is stored outside the MFT, the affected clusters are not deleted on deletion of the file. Instead, these clusters are unlocked for allocation. Thus, deleted files' content can be recovered until the respective clusters are newly allocated and written to. When a file's content is modified, the new content is either written to the `$DATA` attribute inside the MFT or to clusters outside the MFT. To which clusters the new content is written depends on the program used to perform the modification. The new content is either written to same clusters the old content was stored in or to other clusters. Thus, the possibility to recover previous versions of a modified file depends heavily on the program used to perform the modification. We further discuss this in Section 5.4.2.

Name	Description
`$STANDARD_INFORMATION`	General information, such as time stamps and owner
`$ATTRIBUTE_LIST`	List where other attributes for the file can be found
`$FILE_NAME`	File name and time stamps
`$VOLUME_VERSION`	Volume information (only in version 1.2)
`$OBJECT_ID`	Unique identifier for the file (only in versions 3.0+)
`$SECURITY_DESCRIPTOR`	Acces control and security properties of the file
`$VOLUME_NAME`	The volume's name
`$VOLUME_INFORMATION`	File system version and other flags
`$DATA`	File contents
`$INDEX_ROOT`	Root node of an index tree
`$INDEX_ALLOCATION`	Nodes of an index tree rooted in `$INDEX_ROOT` attribute
`$BITMAP`	A bitmap for the $MFT file and for indexes
`$SYMBOLIC_LINK`	Soft link information (only in version 1.2)
`$REPARSE_POINT`	Information about a reparse point (only versions 3.0+)
`$EA_INFORMATION`	For backwards compatibility with OS/2 applications
`$EA`	For backwards compatibility with OS/2 applications
`$LOGGED_UTILITY_STREAM`	Information on encrypted attributes (only versions 3.0+)

Table 5.11.: Default MFT Record attributes

5.4.2. Recovery of a modified file

As above-mentioned, the program that is used to modify a file has great influence on the amount of recoverable obsolete data. For our analyses of YAFFS2 we used the tool `replacer` to overwrite specific bytes of a file with new values. We used this tool as its use lead to a minimum of undesired side effects such as complete rewriting of a modified file and thus allowed undistorted analysis of YAFFS2's behavior. However, we observed that, on a NTFS device, use of `replacer` always lead to loss of all previous versions of a modified file. The reason for that was, that modifications performed with `replacer` were always written to the same clusters the original file was stored in. Hence, every modification directly destroyed the file's previous version and left no obsolete data to be recovered. Therefore, as long as we performed modification by use of `replacer`, YAFFS2 was clearly superior to NTFS regarding possibilities to restore previous versions of modified files.

In order to be able to recover previous versions of a modified file from a NTFS device and compare the results with our analyses of YAFFS2, we also analyzed recovery of files modified by use of the text editor `nano`. In the following, we provide analyses of the scenarios described in Section 5.2. To that purpose, we recreated the scenarios described in Section 5.2 on a NTFS partition of a flash drive as described in Section 5.1.

Worst Case

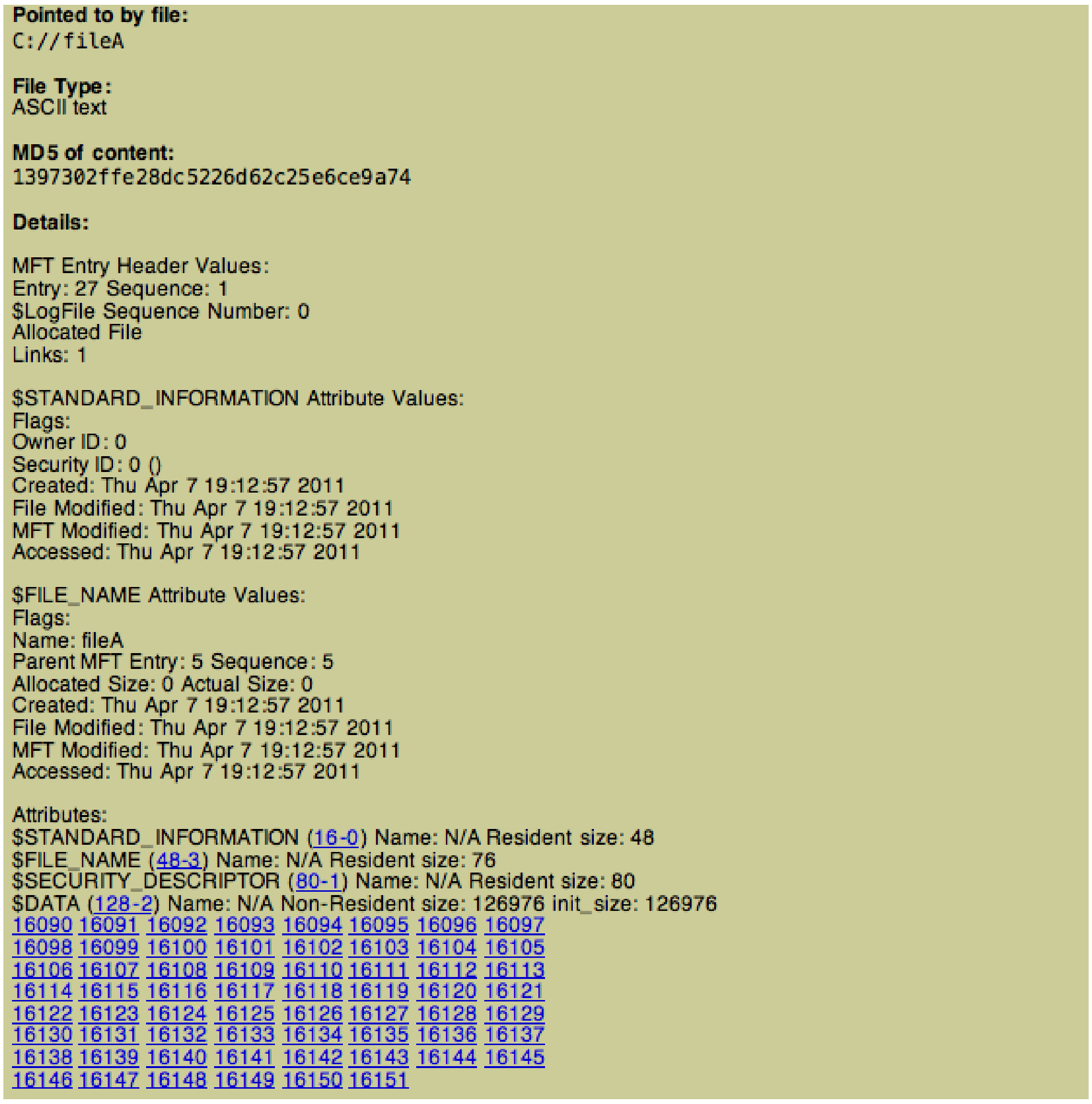
In Section 5.2.1, we present two worst case scenarios regarding recovery of a modified file from a YAFFS2 NAND. In the first scenario, a small file of 126 976 bytes was stored on an otherwise empty device. The file was a raw text file and consisted of 7936 lines of 15 "A" and one line break each. The file was modified by overwriting its last 30 data chunks. To recreate this scenario on a NTFS device, we created a file (*"fileA"*) with a file size of 126 976 bytes on our NTFS partition and modified the file's last 61 440 bytes, which corresponds to the number of bytes of 30 data chunks. As depicted in Figure 5.1, after creation of *"fileA"*, the file's content was located in Clusters 16 090 to 16 151 of the partition.[15]

After modifying the file by overwriting the file's last 61 440 bytes by use of `nano`, the file was rewritten completely and thereby stored in Clusters 20 184 to 20 245. As depicted in Figure 5.2, the file's original content was still stored in Clusters 16 090 to 16 151.[16] Modifying *"fileA"* again, lead to writing of the new content to Clusters 24 280 to 24 341. Thus, at this point, two obsolete versions of *"fileA"* were recoverable from the NTFS partition. Performing a third modification of *"fileA"* lead to writing of the new content to Cluster 28 376 to 28 437.[17] However, a fourth modification[18] lead to writing of the file's new content to Clusters 16 090 to 16 151 again, thus destroying the file's original version. Every subsequent modification lead to overwriting of the file's oldest version, so that a maximum of three previous versions of *"fileA"* were completely recoverable from the NTFS partition. As shown in Section 5.2, in the same scenario, only one previous version of the *"fileA"* was recoverable for just 18 seconds.

The second worst case scenario presented in Section 5.2.1 featured a YAFFS2 NAND flash memory device almost used to capacity. To occupy most of the devices storage capacity we created a large file with a file size of 65 667 072 bytes. The file that was subject to modification (*"fileB"*) had a file size of 192 512 bytes. This file was modified by overwriting the file's Data Chunks 23 to 53, corresponding to overwriting Byte 45 056 to Byte 108 543 of *"fileB"*. Recreating this scenario on our NTFS device needed adjustment of the used files' sizes. This was necessary, as the NTFS device featured a slightly smaller storage capacity than the YAFFS2 NAND and, by default, 2,5 MB were already occupied on the freshly formatted NTFS device. We thus resized the files, so that they occupied the same percentage of the NTFS device's capacity as they did on the YAFFS2 NAND. Therefore, we preformed the following steps on the NTFS device:

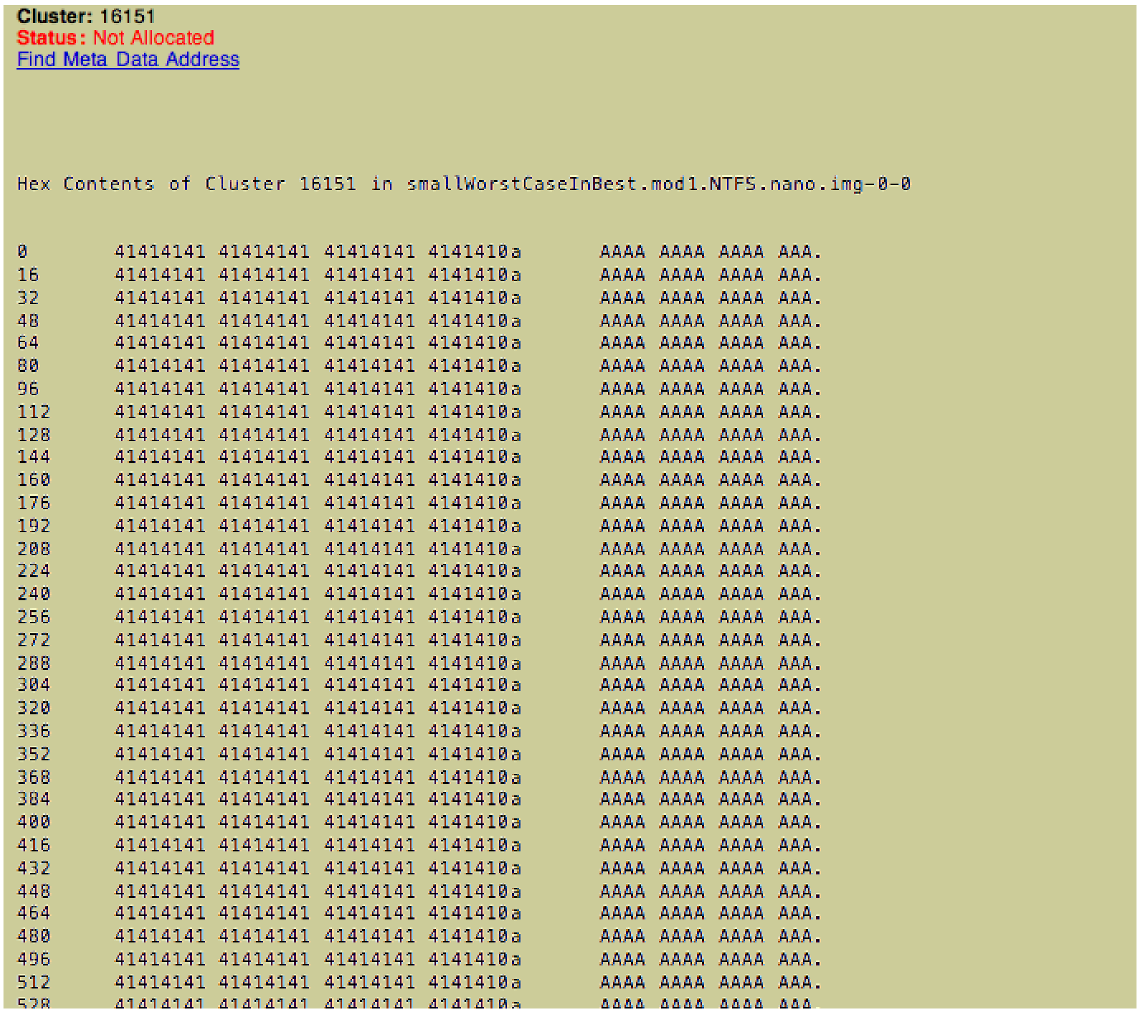
1. Creation of a large file with a file size of 61 747 338 bytes
2. Creation of a smaller file (*"fileB"*) with a file size of 180 741 bytes
3. Overwriting of Bytes 42 302 to 101 908 of *"fileB"*

[15]On the attached DVD, see also: /Chapter5.4.2/smallWorstCaseInBest.initial.NTFS.nano.img
[16]On the attached DVD, see also: /Chapter5.4.2/smallWorstCaseInBest.mod1.NTFS.nano.img
[17]On the attached DVD, see also: /Chapter5.4.2/smallWorstCaseInBest.mod3.NTFS.nano.img
[18]On the attached DVD, see also: /Chapter5.4.2/smallWorstCaseInBest.mod4.NTFS.nano.img



```
Pointed to by file:
C://fileA

File Type:
ASCII text

MD5 of content:
1397302ffe28dc5226d62c25e6ce9a74

Details:

MFT Entry Header Values:
Entry: 27 Sequence: 1
$LogFile Sequence Number: 0
Allocated File
Links: 1

$STANDARD_INFORMATION Attribute Values:
Flags:
Owner ID: 0
Security ID: 0 ()
Created: Thu Apr 7 19:12:57 2011
File Modified: Thu Apr 7 19:12:57 2011
MFT Modified: Thu Apr 7 19:12:57 2011
Accessed: Thu Apr 7 19:12:57 2011

$FILE_NAME Attribute Values:
Flags:
Name: fileA
Parent MFT Entry: 5 Sequence: 5
Allocated Size: 0 Actual Size: 0
Created: Thu Apr 7 19:12:57 2011
File Modified: Thu Apr 7 19:12:57 2011
MFT Modified: Thu Apr 7 19:12:57 2011
Accessed: Thu Apr 7 19:12:57 2011

Attributes:
$STANDARD_INFORMATION (16-0) Name: N/A Resident size: 48
$FILE_NAME (48-3) Name: N/A Resident size: 76
$SECURITY_DESCRIPTOR (80-1) Name: N/A Resident size: 80
$DATA (128-2) Name: N/A Non-Resident size: 126976 init_size: 126976
16090 16091 16092 16093 16094 16095 16096 16097
16098 16099 16100 16101 16102 16103 16104 16105
16106 16107 16108 16109 16110 16111 16112 16113
16114 16115 16116 16117 16118 16119 16120 16121
16122 16123 16124 16125 16126 16127 16128 16129
16130 16131 16132 16133 16134 16135 16136 16137
16138 16139 16140 16141 16142 16143 16144 16145
16146 16147 16148 16149 16150 16151
```

Figure 5.1.: Initial location of *"fileA"* on the NTFS partition

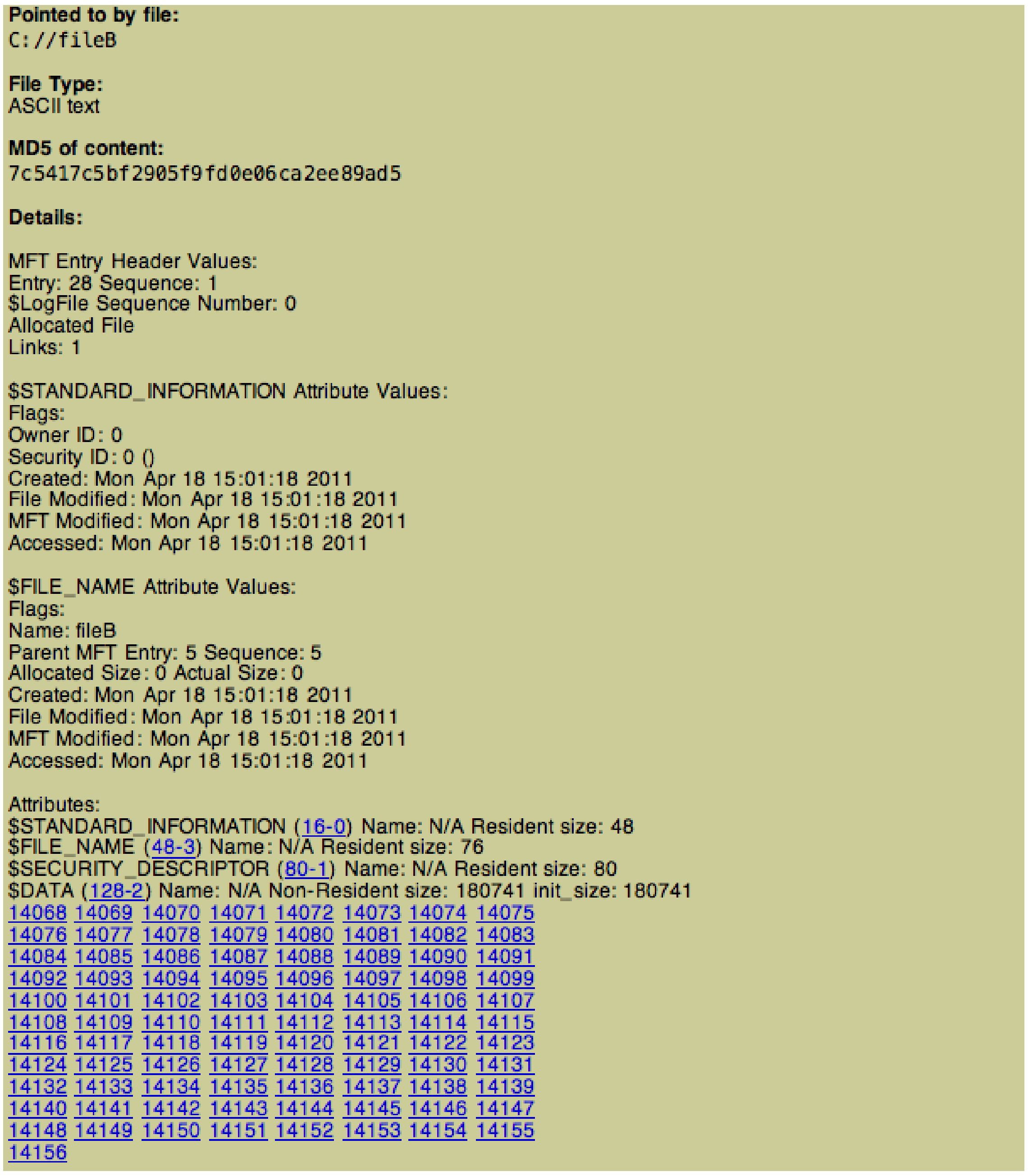


Cluster: 16151
Status: Not Allocated
Find Meta Data Address

```
Hex Contents of Cluster 16151 in smallWorstCaseInBest.mod1.NTFS.nano.img-0-0


0       41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
16      41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
32      41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
48      41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
64      41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
80      41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
96      41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
112     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
128     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
144     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
160     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
176     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
192     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
208     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
224     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
240     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
256     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
272     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
288     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
304     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
320     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
336     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
352     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
368     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
384     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
400     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
416     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
432     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
448     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
464     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
480     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
496     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
512     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
528     41414141 41414141 41414141 4141410a      AAAA AAAA AAAA AAA.
```

Figure 5.2.: Content of cluster 16 151 after first modification of *"fileA"*

Initially, as depicted in Figure 5.3, *"fileB"* was stored in Clusters 14 068 to 14 156 of the NTFS partition.[19] As can be seen in Figure 5.4, modification of *"fileB"* by use of nano lead to immediate loss of the file's original version, as the the new content was written to the same clusters[20] the original content was stored in. In this scenario, one previous version of *"fileB"* was recoverable from a YAFFS2 NAND, albeit only for two seconds.

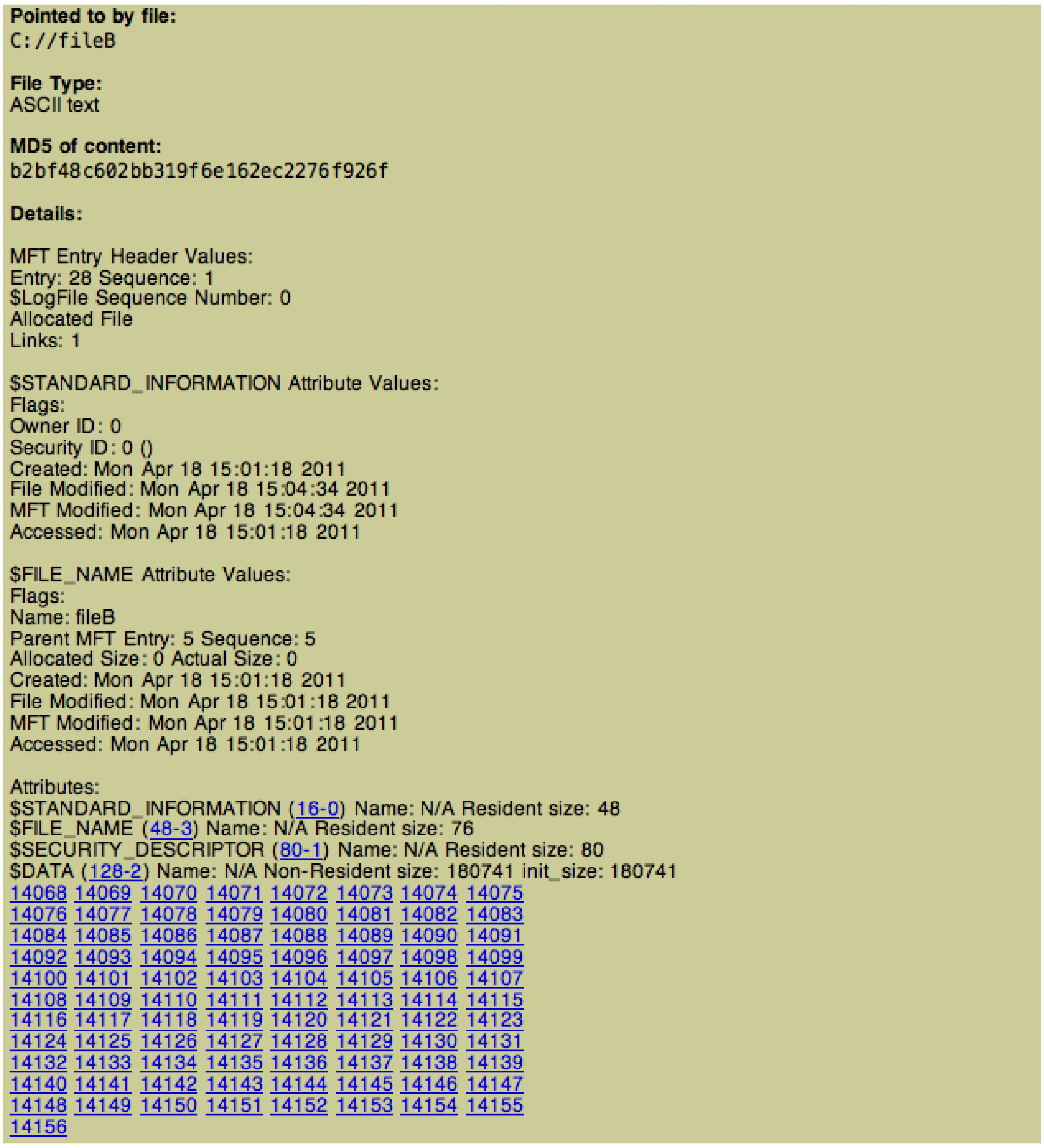


```
Pointed to by file:
C://fileB

File Type:
ASCII text

MD5 of content:
7c5417c5bf2905f9fd0e06ca2ee89ad5

Details:

MFT Entry Header Values:
Entry: 28 Sequence: 1
$LogFile Sequence Number: 0
Allocated File
Links: 1

$STANDARD_INFORMATION Attribute Values:
Flags:
Owner ID: 0
Security ID: 0 ()
Created: Mon Apr 18 15:01:18 2011
File Modified: Mon Apr 18 15:01:18 2011
MFT Modified: Mon Apr 18 15:01:18 2011
Accessed: Mon Apr 18 15:01:18 2011

$FILE_NAME Attribute Values:
Flags:
Name: fileB
Parent MFT Entry: 5 Sequence: 5
Allocated Size: 0 Actual Size: 0
Created: Mon Apr 18 15:01:18 2011
File Modified: Mon Apr 18 15:01:18 2011
MFT Modified: Mon Apr 18 15:01:18 2011
Accessed: Mon Apr 18 15:01:18 2011

Attributes:
$STANDARD_INFORMATION (16-0) Name: N/A Resident size: 48
$FILE_NAME (48-3) Name: N/A Resident size: 76
$SECURITY_DESCRIPTOR (80-1) Name: N/A Resident size: 80
$DATA (128-2) Name: N/A Non-Resident size: 180741 init_size: 180741
14068 14069 14070 14071 14072 14073 14074 14075
14076 14077 14078 14079 14080 14081 14082 14083
14084 14085 14086 14087 14088 14089 14090 14091
14092 14093 14094 14095 14096 14097 14098 14099
14100 14101 14102 14103 14104 14105 14106 14107
14108 14109 14110 14111 14112 14113 14114 14115
14116 14117 14118 14119 14120 14121 14122 14123
14124 14125 14126 14127 14128 14129 14130 14131
14132 14133 14134 14135 14136 14137 14138 14139
14140 14141 14142 14143 14144 14145 14146 14147
14148 14149 14150 14151 14152 14153 14154 14155
14156
```

Figure 5.3.: Initial location of *"fileB"* on the NTFS partition

In the following, we analyze NTFS in a scenario that constitutes a best case scenario

[19]On the attached DvD, see also: Chapter5.4.2/worstCaseInWorst.inital.NTFS.nano.img
[20]On the attached DvD, see also: Chapter5.4.2/worstCaseInWorst.mod1.NTFS.nano.img



```
Pointed to by file:
C://fileB

File Type:
ASCII text

MD5 of content:
b2bf48c602bb319f6e162ec2276f926f

Details:

MFT Entry Header Values:
Entry: 28 Sequence: 1
$LogFile Sequence Number: 0
Allocated File
Links: 1

$STANDARD_INFORMATION Attribute Values:
Flags:
Owner ID: 0
Security ID: 0 ()
Created: Mon Apr 18 15:01:18 2011
File Modified: Mon Apr 18 15:04:34 2011
MFT Modified: Mon Apr 18 15:04:34 2011
Accessed: Mon Apr 18 15:01:18 2011

$FILE_NAME Attribute Values:
Flags:
Name: fileB
Parent MFT Entry: 5 Sequence: 5
Allocated Size: 0 Actual Size: 0
Created: Mon Apr 18 15:01:18 2011
File Modified: Mon Apr 18 15:01:18 2011
MFT Modified: Mon Apr 18 15:01:18 2011
Accessed: Mon Apr 18 15:01:18 2011

Attributes:
$STANDARD_INFORMATION (16-0) Name: N/A Resident size: 48
$FILE_NAME (48-3) Name: N/A Resident size: 76
$SECURITY_DESCRIPTOR (80-1) Name: N/A Resident size: 80
$DATA (128-2) Name: N/A Non-Resident size: 180741 init_size: 180741
14068 14069 14070 14071 14072 14073 14074 14075
14076 14077 14078 14079 14080 14081 14082 14083
14084 14085 14086 14087 14088 14089 14090 14091
14092 14093 14094 14095 14096 14097 14098 14099
14100 14101 14102 14103 14104 14105 14106 14107
14108 14109 14110 14111 14112 14113 14114 14115
14116 14117 14118 14119 14120 14121 14122 14123
14124 14125 14126 14127 14128 14129 14130 14131
14132 14133 14134 14135 14136 14137 14138 14139
14140 14141 14142 14143 14144 14145 14146 14147
14148 14149 14150 14151 14152 14153 14154 14155
14156
```

Figure 5.4.: Location of *"fileB"* after modification

regarding recovery of a specific file from a YAFFS2 NAND flash memory device.

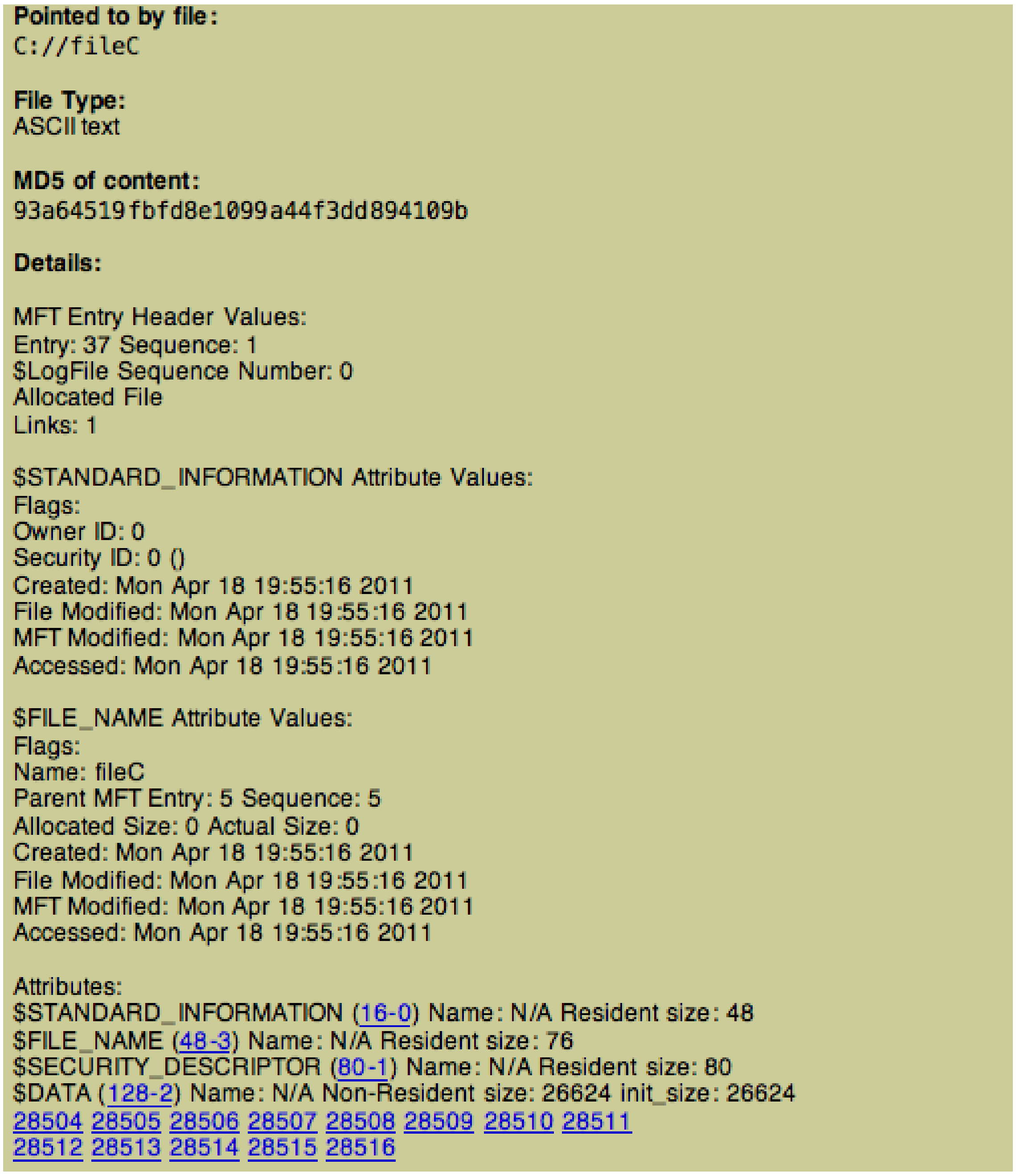
Best Case

For the following analysis, we recreated the scenario described in Section 5.2.2 on a NTFS partition. For that purpose we created ten files on the device as follows:

1. Creation of *"file1"* (126 976 bytes)
2. Creation of *"file2"* (129 024 bytes)
3. Creation of *"file3"* (129 024 bytes)
4. Creation of *"file4"* (129 024 bytes)
5. Creation of *"file5"* (129 024 bytes)
6. Creation of *"file6"* (129 024 bytes)
7. Creation of *"file7"* (129 024 bytes)
8. Creation of *"file8"* (129 024 bytes)
9. Creation of *"file9"* (129 024 bytes)
10. Creation of *"file10"* (129 024 bytes)

Additionally, we created a file named *"fileC"* with a file size of 26 624 bytes on the device. As can be seen in Section 5.2.2, during our analysis of YAFFS2, we modified *"file1"* to *"file10"* in a way that made garbage collection of every used block on the YAFFS2 NAND necessary. Although NTFS does not feature a garbage collector that reacts to such modifications, we still performed the same modifications on these files in order to recreate the scenario as exactly as possible. Thus, each of the ten files was modified by overwriting 2048 of its bytes with new values. As in the analysis we presented in Section 5.2.2, the bytes we overwrote were at the following positions within the files:

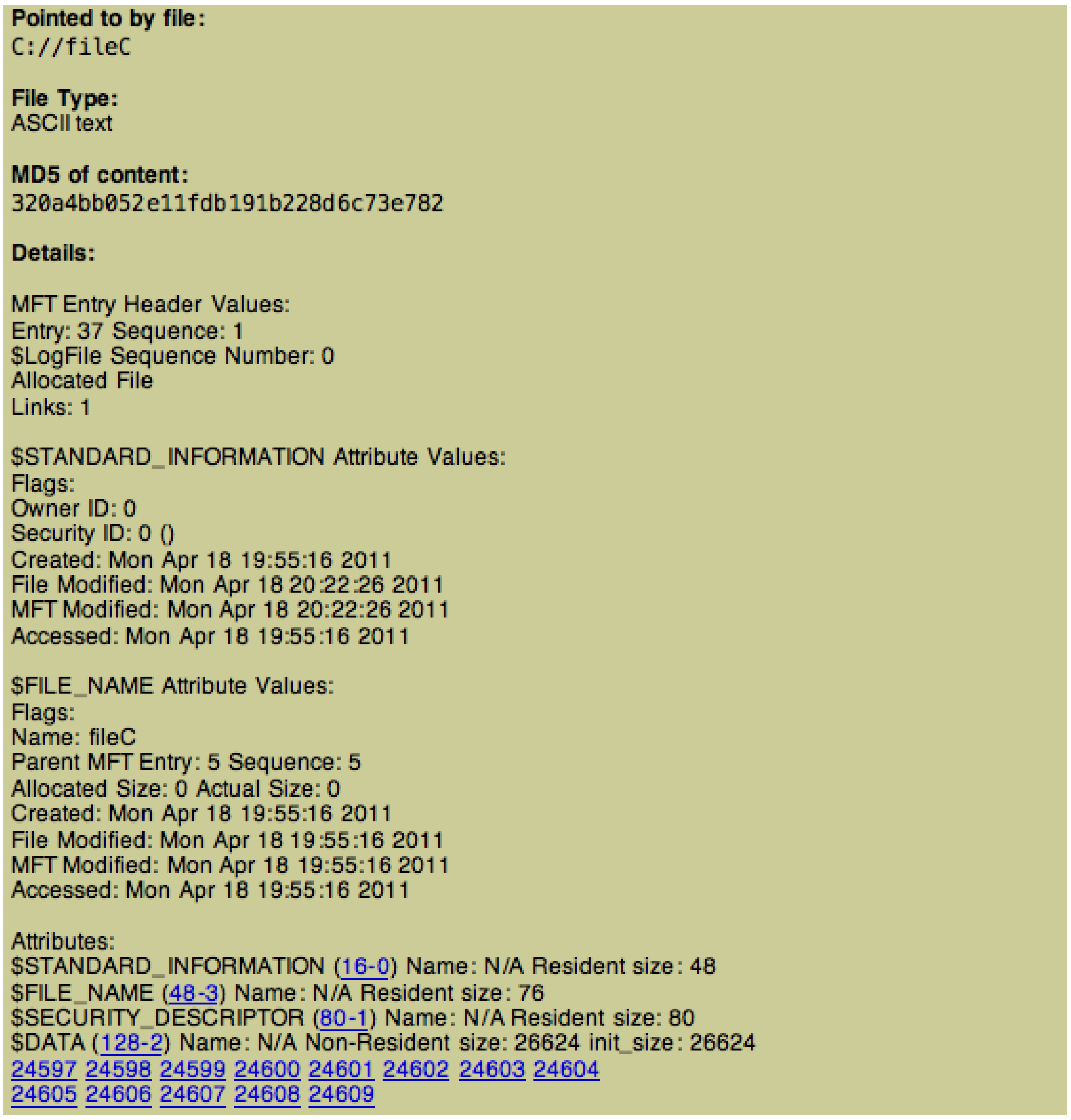
- *"file1"*: Byte 124 928 to Byte 126 975
- *"file2"*: Byte 122 880 to Byte 124 927
- *"file3"*: Byte 122 880 to Byte 124 927
- *"file4"*: Byte 122 880 to Byte 124 927
- *"file5"*: Byte 122 880 to Byte 124 927
- *"file6"*: Byte 122 880 to Byte 124 927
- *"file7"*: Byte 122 880 to Byte 124 927
- *"file8"*: Byte 122 880 to Byte 124 927
- *"file9"*: Byte 122 880 to Byte 124 927
- *"file10"*: Byte 114 688 to Byte 116 735



```
Pointed to by file:
C://fileC

File Type:
ASCII text

MD5 of content:
93a64519fbfd8e1099a44f3dd894109b

Details:

MFT Entry Header Values:
Entry: 37 Sequence: 1
$LogFile Sequence Number: 0
Allocated File
Links: 1

$STANDARD_INFORMATION Attribute Values:
Flags:
Owner ID: 0
Security ID: 0 ()
Created: Mon Apr 18 19:55:16 2011
File Modified: Mon Apr 18 19:55:16 2011
MFT Modified: Mon Apr 18 19:55:16 2011
Accessed: Mon Apr 18 19:55:16 2011

$FILE_NAME Attribute Values:
Flags:
Name: fileC
Parent MFT Entry: 5 Sequence: 5
Allocated Size: 0 Actual Size: 0
Created: Mon Apr 18 19:55:16 2011
File Modified: Mon Apr 18 19:55:16 2011
MFT Modified: Mon Apr 18 19:55:16 2011
Accessed: Mon Apr 18 19:55:16 2011

Attributes:
$STANDARD_INFORMATION (16-0) Name: N/A Resident size: 48
$FILE_NAME (48-3) Name: N/A Resident size: 76
$SECURITY_DESCRIPTOR (80-1) Name: N/A Resident size: 80
$DATA (128-2) Name: N/A Non-Resident size: 26624 init_size: 26624
28504 28505 28506 28507 28508 28509 28510 28511
28512 28513 28514 28515 28516
```

Figure 5.5.: Initial location of *"fileC"* on the NTFS partition

As can be seen in Figure 5.5, after these modifications, *“fileC”* was stored in Clusters 28 504 to 28 516. These were the same clusters the file had been stored before the modification of all other files. All other files were completely rewritten during their modifications and thus were stored in other clusters than they were originally.[21] Starting from this initial state of the NTFS partition we modified *“fileC”* repeatedly by overwriting the file’s first 2048 bytes with new values. As can be seen in Figure 5.6, the first modification of *“fileC”* lead to rewriting of *“fileC”* to Clusters 24 597 to 24 609. As the file’s original version was still stored in Cluster 28 504 to 28 516, at this point, one complete previous version of *“fileC”* was recoverable from the NTFS partition.[22]



Pointed to by file:
C://fileC

File Type:
ASCII text

MD5 of content:
320a4bb052e11fdb191b228d6c73e782

Details:

MFT Entry Header Values:
Entry: 37 Sequence: 1
$LogFile Sequence Number: 0
Allocated File
Links: 1

$STANDARD_INFORMATION Attribute Values:
Flags:
Owner ID: 0
Security ID: 0 ()
Created: Mon Apr 18 19:55:16 2011
File Modified: Mon Apr 18 20:22:26 2011
MFT Modified: Mon Apr 18 20:22:26 2011
Accessed: Mon Apr 18 19:55:16 2011

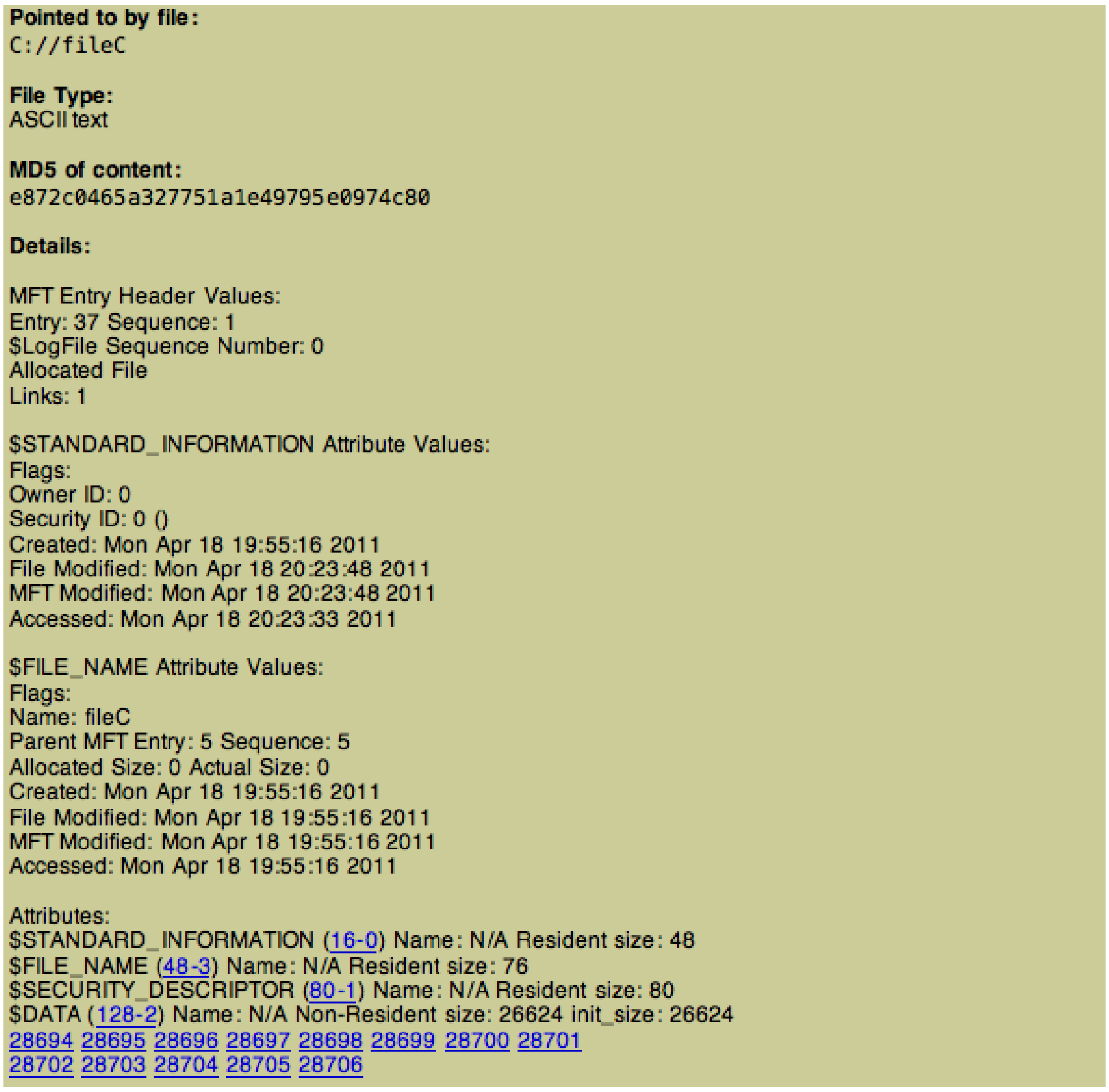
$FILE_NAME Attribute Values:
Flags:
Name: fileC
Parent MFT Entry: 5 Sequence: 5
Allocated Size: 0 Actual Size: 0
Created: Mon Apr 18 19:55:16 2011
File Modified: Mon Apr 18 19:55:16 2011
MFT Modified: Mon Apr 18 19:55:16 2011
Accessed: Mon Apr 18 19:55:16 2011

Attributes:
$STANDARD_INFORMATION (16-0) Name: N/A Resident size: 48
$FILE_NAME (48-3) Name: N/A Resident size: 76
$SECURITY_DESCRIPTOR (80-1) Name: N/A Resident size: 80
$DATA (128-2) Name: N/A Non-Resident size: 26624 init_size: 26624
24597 24598 24599 24600 24601 24602 24603 24604
24605 24606 24607 24608 24609

Figure 5.6.: Location of *“fileC”* on the NTFS partition after one modification

[21]On the attached DvD, see also: /Chapter5.4.2/bestInOverallBest.mod.files1-10.NTFS.nano.img
[22]On the attached DvD, see also: /Chapter5.4.2/bestCaseInOverallBest.mod.fileC-1.NTFS.nano.img

As depicted in Figure 5.7, a second modification of *"fileC"* stored the file's most current version in Clusters 28 694 to 28 706. Hence, at this point, all previous versions of *"fileC"* were recoverable.[23]



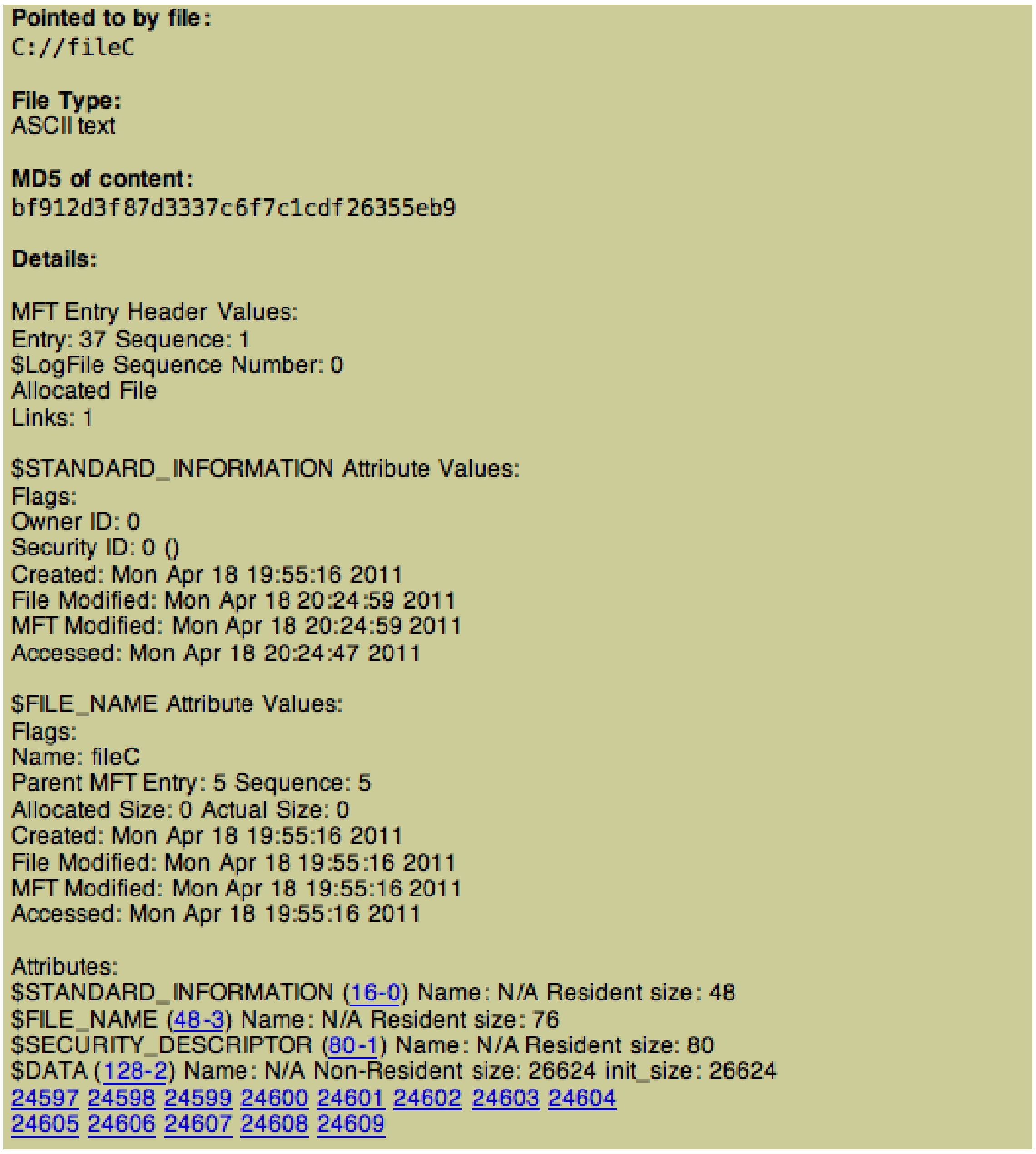
```
Pointed to by file:
C://fileC

File Type:
ASCII text

MD5 of content:
e872c0465a327751a1e49795e0974c80

Details:

MFT Entry Header Values:
Entry: 37 Sequence: 1
$LogFile Sequence Number: 0
Allocated File
Links: 1

$STANDARD_INFORMATION Attribute Values:
Flags:
Owner ID: 0
Security ID: 0 ()
Created: Mon Apr 18 19:55:16 2011
File Modified: Mon Apr 18 20:23:48 2011
MFT Modified: Mon Apr 18 20:23:48 2011
Accessed: Mon Apr 18 20:23:33 2011

$FILE_NAME Attribute Values:
Flags:
Name: fileC
Parent MFT Entry: 5 Sequence: 5
Allocated Size: 0 Actual Size: 0
Created: Mon Apr 18 19:55:16 2011
File Modified: Mon Apr 18 19:55:16 2011
MFT Modified: Mon Apr 18 19:55:16 2011
Accessed: Mon Apr 18 19:55:16 2011

Attributes:
$STANDARD_INFORMATION (16-0) Name: N/A Resident size: 48
$FILE_NAME (48-3) Name: N/A Resident size: 76
$SECURITY_DESCRIPTOR (80-1) Name: N/A Resident size: 80
$DATA (128-2) Name: N/A Non-Resident size: 26624 init_size: 26624
28694 28695 28696 28697 28698 28699 28700 28701
28702 28703 28704 28705 28706
```

Figure 5.7.: Location of *"fileC"* on the NTFS partition after two modifications

However, beginning with the file's third modification, every subsequent modification lead to loss of one obsolete version. As depicted in Figure 5.8, modifying *"fileC"* for the third time stored the file's new version in Clusters 24 597 to 24 609 again.[24] All subsequent modifications were alternately written to Clusters 28 694 to 28 706 and Clusters 24 597 to 24 609. Thus, at every point, the file's original version was recoverable from the NTFS partition along with the file's most current previous version. As shown in Section 5.2.2, twenty-nine version of *"fileC"* were recoverable from a YAFFS2 NAND flash memory device. However, these obsolete versions were only recoverable for a eight min-

[23]On the attached DvD, see also: /Chapter5.4.2/bestCaseInOverallBest.mod.fileC-2.NTFS.nano.img
[24]On the attached DvD, see also: /Chapter5.4.2/bestCaseInOverallBest.mod.fileC-3.NTFS.nano.img

utes and three seconds, whereas the file's previous versions could be recovered from the NTFS partition until the respective clusters were allocated again by a write operation.



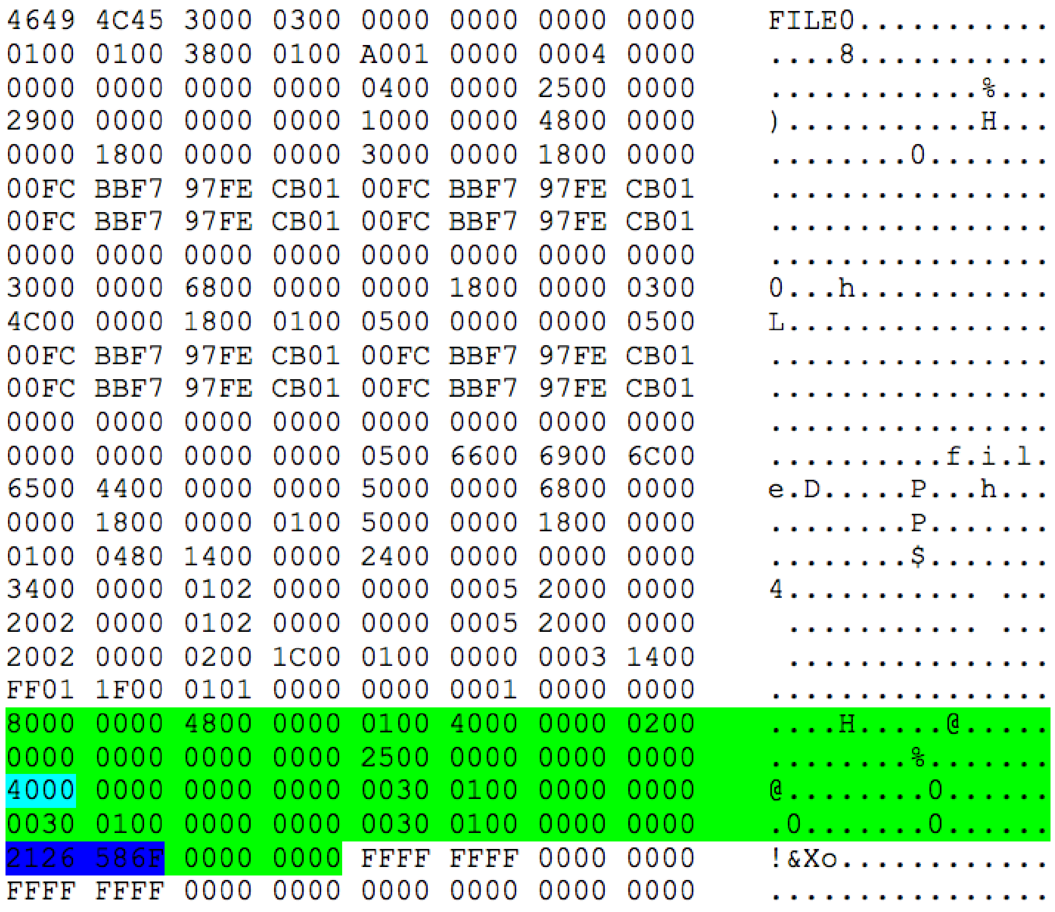
```
Pointed to by file:
C://fileC

File Type:
ASCII text

MD5 of content:
bf912d3f87d3337c6f7c1cdf26355eb9

Details:

MFT Entry Header Values:
Entry: 37 Sequence: 1
$LogFile Sequence Number: 0
Allocated File
Links: 1

$STANDARD_INFORMATION Attribute Values:
Flags:
Owner ID: 0
Security ID: 0 ()
Created: Mon Apr 18 19:55:16 2011
File Modified: Mon Apr 18 20:24:59 2011
MFT Modified: Mon Apr 18 20:24:59 2011
Accessed: Mon Apr 18 20:24:47 2011

$FILE_NAME Attribute Values:
Flags:
Name: fileC
Parent MFT Entry: 5 Sequence: 5
Allocated Size: 0 Actual Size: 0
Created: Mon Apr 18 19:55:16 2011
File Modified: Mon Apr 18 19:55:16 2011
MFT Modified: Mon Apr 18 19:55:16 2011
Accessed: Mon Apr 18 19:55:16 2011

Attributes:
$STANDARD_INFORMATION (16-0) Name: N/A Resident size: 48
$FILE_NAME (48-3) Name: N/A Resident size: 76
$SECURITY_DESCRIPTOR (80-1) Name: N/A Resident size: 80
$DATA (128-2) Name: N/A Non-Resident size: 26624 init_size: 26624
24597 24598 24599 24600 24601 24602 24603 24604
24605 24606 24607 24608 24609
```

Figure 5.8.: Location of *"fileC"* on the NTFS partition after three modifications

5.4.3. Recovery of a deleted file

In the following, we analyze possibilities to recover deleted files from a NTFS device in comparison to possibilities to restore deleted files from a YAFFS2 NAND flash memory device. To that purpose we recreated the scenario described in Section 5.3 on a NTFS partition.

Similar to file deletions on YAFFS2 devices, deletion of a file on a NTFS device does not directly lead to actual deletion of the file's content. In NTFS, a file is deleted in three steps. In the first step, the file's entry in its parent directory's `$INDEX_ROOT` is removed. In the course of that, the parent directory's time stamps are updated. In the second step, the deleted file's MFT record is unallocated by cleaning its *in-use* flag. Hence, this record is free for reallocation. In the final step, the clusters containing non-resident attributes of the file's MFT record are set to an unallocated state. Thus, these clusters are free to be reallocated and overwritten. However, at this point, the file's content is still recoverable from the NTFS device, as its MFT record is still stored in the MFT, its non-resident attributes are still stored in their respective clusters and the links between the MFT record and these clusters still exist. In contrast to YAFFS2, NTFS does not feature a garbage collector that deletes all obsolete data over time, so that the deleted file's content stays on the NTFS device until the respective clusters are actually allocated again. In the following, we analyze how much of a deleted file can be recovered from a NTFS partition in a scenario as described in Section 5.3.2.



```
4649 4C45 3000 0300 0000 0000 0000 0000    FILE0...........
0100 0100 3800 0100 A001 0000 0004 0000    ....8...........
0000 0000 0000 0000 0400 0000 2500 0000    ............%...
2900 0000 0000 0000 1000 0000 4800 0000    )...........H...
0000 1800 0000 0000 3000 0000 1800 0000    ........0.......
00FC BBF7 97FE CB01 00FC BBF7 97FE CB01    ................
00FC BBF7 97FE CB01 00FC BBF7 97FE CB01    ................
0000 0000 0000 0000 0000 0000 0000 0000    ................
3000 0000 6800 0000 0000 1800 0000 0300    0...h...........
4C00 0000 1800 0100 0500 0000 0000 0500    L...............
00FC BBF7 97FE CB01 00FC BBF7 97FE CB01    ................
00FC BBF7 97FE CB01 00FC BBF7 97FE CB01    ................
0000 0000 0000 0000 0000 0000 0000 0000    ................
0000 0000 0000 0000 0500 6600 6900 6C00    ..........f.i.l.
6500 4400 0000 0000 5000 0000 6800 0000    e.D.....P...h...
0000 1800 0000 0100 5000 0000 1800 0000    ........P.......
0100 0480 1400 0000 2400 0000 0000 0000    ........$.......
3400 0000 0102 0000 0000 0005 2000 0000    4........... ...
2002 0000 0102 0000 0000 0005 2000 0000     ........... ...
2002 0000 0200 1C00 0100 0000 0003 1400     ...............
FF01 1F00 0101 0000 0000 0001 0000 0000    ................
8000 0000 4800 0000 0100 4000 0000 0200    ....H.....@.....
0000 0000 0000 0000 2500 0000 0000 0000    ........%.......
4000 0000 0000 0000 0030 0100 0000 0000    @........0......
0030 0100 0000 0000 0030 0100 0000 0000    .0.......0......
2126 586F 0000 0000 FFFF FFFF 0000 0000    !&Xo............
FFFF FFFF 0000 0000 0000 0000 0000 0000    ................
```

Figure 5.9.: MFT record of *"fileD"* before the file's deletion

To recreate the scenario we presented in Section 5.3.2 on a NTFS partition, we created ten files on the partition as described above. Subsequently, we created a file named *"fileD"* with a file size of 77 824 bytes on the partition. After modifying all files except for *"fileD"* as described in Section 5.4.2, we deleted *"fileD"*. In Figure 5.9, the file's

MFT record is depicted. At this point, all other files had already been modified and *"fileD"* had not been deleted yet. The clusters containing the file's content are defined in the file's MFT record's `$DATA` attribute (marked green). At offset 32 of the `$DATA` attribute, the value `0x40` (marked cyan) told us, that the first cluster run within this `$DATA` attribute could be found at offset 64 of the attribute. Every cluster run (marked blue) consists of three parts:

1. Cluster run information, here: `0x21`
2. Length of the cluster run, here: `0x26`
3. First cluster, here: `0x6F58`

The cluster run information defined that the last two bytes of the cluster run contained the file's first cluster and that the first byte of the run contained the number of cluster allocated for the file. Hence, the content of *"fileD"* was stored in 38 (`0x26`) clusters starting from Cluster 28 504 (`0x6F58`). Thus, the file's content was stored in Clusters 28 504 to 28 541 of the NTFS partition.[25]

```
4649 4C45 3000 0300 0000 0000 0000 0000    FILE0...........
0200 0000 3800 0000 3801 0000 0004 0000    ....8...8.......
0000 0000 0000 0000 0400 0000 2500 0000    ............%...
2A00 0000 0000 0000 1000 0000 4800 0000    *...........H...
0000 1800 0000 0000 3000 0000 1800 0000    ........0.......
00FC BBF7 97FE CB01 00FC BBF7 97FE CB01    ................
00FC BBF7 97FE CB01 00FC BBF7 97FE CB01    ................
0000 0000 0000 0000 0000 0000 0000 0000    ................
5000 0000 6800 0000 0000 1800 0000 0100    P...h...........
5000 0000 1800 0000 0100 0480 1400 0000    P...............
2400 0000 0000 0000 3400 0000 0102 0000    $.......4.......
0000 0005 2000 0000 2002 0000 0102 0000    .... ... .......
0000 0005 2000 0000 2002 0000 0200 1C00    .... ... .......
0100 0000 0003 1400 FF01 1F00 0101 0000    ................
0000 0001 0000 0000 8000 0000 4800 0000    ............H...
0100 4000 0000 0200 0000 0000 0000 0000    ..@.............
2500 0000 0000 0000 4000 0000 0000 0000    %.......@.......
0030 0100 0000 0000 0030 0100 0000 0000    .0.......0......
0030 0100 0000 0000 2126 586F 0000 0000    .0......!&Xo....
FFFF FFFF 0000 0000 0100 0000 0003 1400    ................
FF01 1F00 0101 0000 0000 0001 0000 0000    ................
8000 0000 4800 0000 0100 4000 0000 0200    ....H.....@.....
0000 0000 0000 0000 2500 0000 0000 0000    ........%.......
4000 0000 0000 0000 0030 0100 0000 0000    @........0......
0030 0100 0000 0000 0030 0100 0000 0000    .0.......0......
2126 586F 0000 0000 FFFF FFFF 0000 0000    !&Xo............
FFFF FFFF 0000 0000 0000 0000 0000 0000    ................
```

Figure 5.10.: MFT record of *"fileD"* after the file's deletion

Deleting *"fileD"* updated the MFT as above-mentioned but left the file completely

[25] On the attached DVD, see also: /Chapter5.4.3/deletion.mod.file1-10.NTFS.img

recoverable. In Figure 5.10, the file's MFT record after the file's deletion is depicted. As can be seen, the file's name was not stored in the MFT anymore. However, the cluster run linking to the file's content was still stored in the MFT record's `$DATA` attribute. Additionally, the clusters still actually contained the file's content. Therefore, except for the file's name, *"fileD"* was completely recoverable[26] from the NTFS partition. In the same scenario, *"fileD"* was also recoverable from a YAFFS2 NAND. However, the file was only recoverable for seven minutes and 53 seconds from the YAFFS2 NAND, whereas the file was recoverable from the NTFS partition until the file's respective clusters were allocated again by a write operation.

5.5. Summary

In this chapter, we presented best and worst case scenarios regarding recovery of obsolete chunks from a YAFFS2 NAND and analyzed how these scenarios influenced the amount of obsolete data that was recoverable after a file's modification or deletion. We also showed that the tools used to perform file modifications have great influence on the amount of recoverable obsolete chunks.

We observed that YAFFS2's design allows easy recovery of obsolete data as all data written to a YAFFS2 NAND can only be deleted by garbage collection. However, as long as a YAFFS2 NAND is in use, the amount of recoverable data decreases over time, even if no further write operations are performed on the NAND. After a certain time span, no obsolete data can be recovered from a YAFFS2 NAND anymore. In this chapter, we showed that the length of this time span depends on the NAND's storage capacity and its occupancy as well as on the distribution of obsolete and valid chunks within the NAND's blocks

In the last section, we compared NTFS and YAFFS2 regarding possibilities to recover obsolete data after modification or deletion of a file. For that purpose we analyzed NTFS's behavior in scenarios that constitute best and worst case scenarios regarding recovery of obsolete data from a YAFFS2 NAND. We observed that YAFFS2 allows a much higher amount of obsolete data to be recovered after modifications of a file than NTFS does. However, while obsolete parts of a modified file were deleted by YAFFS2's garbage collector after a certain time span, obsolete parts could be recovered from the NTFS partition until they were overwritten by a subsequent write operation. In the scenarios we analyzed, both YAFFS2 as well as NTFS allowed complete recovery of a deleted file's content. However, as was the case regarding file modifications, the deleted file was only recoverable from the YAFFS2 NAND for a certain time span while it could be recovered from the NTFS partition until the cluster containing its content were overwritten by a subsequent write operation.

[26]On the attached DVD, see also: /Chapter5.4.3/deletion.removed.fileD.NTFS.img

6. Safe deletion on YAFFS2 devices

Safe deletion of a file means deletion of the file in a way that leaves no traces of the file on a device. Safe deletion of a device means deletion of all data on the device in a way, that no data that was stored on the device can be recovered from the device at all. There are many reasons that make complete and safe deletion of a specific file or a complete NAND flash memory device desirable. Fore example, safe deletion of a specific file or a whole device can be necessary to destroy sensitive data for privacy reasons before reselling a smartphone.

As shown in the previous chapters of this diploma thesis, simply deleting files from a YAFFS2 NAND flash memory device does not necessarily instantly completely and safely delete the files from the device. As long as the device's blocks containing the deleted files' chunks are not erased by garbage collection, the files can still be recovered from the device. Thus, regular deletion by use of tools like `rm` is not necessarily sufficient to safely delete a file from a YAFFS2 NAND flash memory. In this chapter, we discuss ways to safely delete files from a YAFFS2 NAND flash memory device. Additionally, we introduce ways to completely erase a YAFFS2 NAND flash memory device.

6.1. Safe deletion of files

Deleting a file from a YAFFS2 NAND flash memory can be performed in two ways. Either the file can be deleted using YAFFS2 functionality or the file can be deleted by bypassing the file system and deleting the relevant blocks directly on the NAND flash memory device. In the following, both ways to safely delete files are discussed.

6.1.1. Safe deletion of files using YAFFS2

To delete a specific file from a YAFFS2 NAND flash memory device in a way that leaves no traces of the file on the device requires complete deletion of all the file's chunks. This includes all the file's data chunks as well as the file's header chunks on the device. These chunks are not deleted from a YAFFS2 NAND until the blocks containing the chunks are completely garbage collected and subsequently erased. As shown in Chapter 4, even in a best case regarding recovery of files, garbage collection is executed regularly as long as obsolete chunks can be found on the device. Thus, at some point in time, a deleted file is completely and safely removed from a YAFFS2 NAND. However, depending on the number of blocks on the device and the distribution of obsolete and valid chunks within these blocks, it can take a very long time until no obsolete chunks can be found on the NAND anymore. As shown in Chapter 4, in our analysis environment, in case each of a device's blocks contains one obsolete chunk and the device features enough free blocks as to make aggressive garbage collection unnecessary, oldest dirty garbage collection can delete a block every 46 seconds at most. Thus, even on a relatively small device with 500 blocks in use as described and enough free blocks available, complete deletion of a file can take longer than six hours.

As can be seen, safely deleting a file is theoretically viable by deleting the file by use of default commands like `rm` and subsequently waiting long enough until garbage collection has deleted all the file's chunks. However, practically, this way is not useful. First, without creating a dump of the device and lengthy analysis of the dump, the user has no way to calculate the time garbage collection actually needs to completely delete all the deleted file's chunks. Second, some situations may require quick and safe deletion of a file so that waiting for an unspecified and possibly very long time is not an option.

A hypothetical tool for safe deletion of a file from a YAFFS2 NAND has to ensure that all blocks containing the deleted file's chunks get garbage collected as soon as possible. Additionally the tool has to trace these blocks' garbage collection to determine at which point the blocks are actually erased. The only way for such a tool to speed up garbage collection of the blocks containing the deleted file's chunks is to ensure that these blocks are not disabled for garbage collection and feature maximal four valid chunks or are prioritized for garbage collection. That way, these blocks are garbage collected relatively soon after deletion of the file by background garbage collection. However, the block containing the deleted file's `deleted` header chunk is still disabled for garbage collection until it has become the oldest dirty block on the device, as a `deleted` header chunk features a shrink header marker. Making all other blocks containing chunks of the deleted file candidates for quick garbage collection includes ensuring that these blocks are in state `FULL` and modifying other objects in a way that makes their chunks on the blocks obsolete. This also possibly includes creation of new objects or increasing the size of existing objects in order to fill blocks. However, if header chunks marked with shrink header markers exist on one or more of the blocks containing chunks of the deleted file, these blocks are still disabled for garbage collection until each of them has become the oldest dirty block at some point. Prioritizing a block for garbage collection by manipulating its block information requires manipulation of YAFFS2's RAM structure and also ensuring that the block is not disqualified for garbage collection. As can be seen, a tool to safely delete files from a YAFFS2 NAND has to perform actually unnecessary modifications of objects other than the files to be safely deleted and still cannot ensure instant or at least quick deletion of the files to be safely deleted.

Thus, there is no way to safely delete a file from a YAFFS2 NAND flash memory device without having to wait for garbage collection to erase all blocks containing the deleted file's chunks or bypassing the YAFFS2 file system. Speeding up garbage collection in order to completely delete a file more quickly is theoretically possible but requires creation, modification or deletion of other objects on the device. Additionally, garbage collection can only be sped up to a certain degree as at least the block containing the deleted file's `deleted` header chunk can only be deleted once it has become the oldest dirty block.

6.1.2. Safe deletion of files directly on NAND flash memory

Theoretically, a file can be safely deleted from a YAFFS2 NAND by bypassing the YAFFS2 file system and deleting the respective blocks directly via the NAND flash memory `iotcl` provided in the `mtd-utils` [14]. However, this requires determining in which blocks the deleted file's chunks are stored and additionally saving existing valid chunks from these blocks. This approach is already discussed in [24]. We could confirm, that erasing a block containing a deleted file's obsolete chunks by use of the tool `flash_erase` of the the `mtd-utils` safely removed a deleted file from a YAFFS2 NAND flash memory device.

6.2. Safe deletion of devices

As shown in the last section, quick and safe deletion of a specific file from a YAFFS2 NAND flash memory device is not possible without bypassing the file system. Safely deleting a whole YAFFS2 NAND flash memory device can also be performed by using YAFFS2's functionality or by bypassing the file system.

When using YAFFS2's functionality to safely delete a whole device, the approach to do so is similar to the approach to safely delete a specific file. Deleting all existing objects from a YAFFS2 NAND by use of default tools like `rm` and subsequent waiting for garbage collection to delete all obsolete chunks is sufficient to safely erase the device. This is because after deletion of all objects, only one chunk remains valid. This chunk contains the root directory's header information. However, again, this approach takes a lot of time.

A faster way to safely delete all objects from a YAFFS2 NAND requires bypassing the file system. To that purpose the NAND flash memory device's blocks can be erased by use of the NAND flash memory `iotcl` provided in the `mtd-utils`. To perform complete and safe deletion of a whole NAND flash memory device, the widely used tool `flash_eraseall` from the `mtd-utils` can be used. We could confirm that use of `flash_eraseall` completely erased all of a NAND flash memory device's blocks.

6.3. Summary

Safe deletion of a whole YAFFS2 NAND can easily be achieved by directly erasing all the device's blocks using the NAND flash memory `ioctl` provided in the `mtd-utils`. Additionally, given enough time, YAFFS2's garbage collector can be used for safe deletion of a whole device. However, depending on the device's size, this can take an unacceptable long time.

Safe deletion of a specific file from a YAFFS2 NAND flash memory device is much more complicated than simply deleting the whole device in a safe way. As for deletion of a whole device, YAFFS2's garbage collector can also be used for safe deletion of a specific file. However, depending on the device's size and the location of the deleted file's chunks this also can take an unacceptable long time. Erasing the blocks containing the deleted file's obsolete chunks directly by use of the NAND flash memory `ioctl` is viable but requires copying valid chunks from these blocks to other blocks.

7. Conclusion and Future Work

In this final chapter, we conclude this diploma thesis, starting with a short summary in Section 7.1 to recapitulate what we have learned in the last chapters. Section 7.2 highlights remaining tasks that could be the scope of future work on YAFFS2. Finally, in Section 7.3, we draw our final conclusions.

7.1. Summary

Within this diploma thesis we introduced and analyzed the file system YAFFS2 in a forensic perspective. For that purpose, we provided basic knowledge on NAND flash memory and an introduction on how to use YAFFS2 in a Linux environment in the first chapters of this diploma thesis. In subsequent chapters, we first analyzed YAFFS2's source code to reveal the file system's basic functionality and design. YAFFS2's garbage collection techniques were a major focus within this analysis, as garbage collection is the only technique YAFFS2 uses to delete obsolete data. Next, we discussed and analyzed the influences of YAFFS2's behavior on the amount of deleted or modified data that can be recovered from a YAFFS2 NAND. For that purpose, based on our analyses of YAFFS2's garbage collection and wear leveling techniques, we developed best case and worst case scenarios regarding recovery of obsolete data from a YAFFS2 NAND. We also evaluated these scenarios practically by performing file modifications and deletions on a simulated YAFFS2 NAND and analyzing the amount of obsolete data that could be recovered. Further, we compared YAFFS2 to the common file system NTFS regarding possibilities to recover obsolete data. Finally, in the last chapter of this diploma thesis, we discussed techniques to safely delete files from a YAFFS2 NAND or completely erase such a device.

7.2. Future Work

Concerning mobile phone forensics, Android OS and YAFFS2 still provide important and interesting points that should receive further investigation in future research and development.

Based on the analyses provided in this diploma thesis, an automated tool to recover deleted files and obsolete data of modified files from a YAFFS2 NAND could be developed. This would enable quicker analysis of an Android smartphone's data contents. Additionally, from a counter-forensic perspective, a tool to safely delete specific files from a YAFFS2 NAND could be developed to ensure privacy of Android smartphone users.

In the future, many Android OS smartphones will be using the file system ext4 instead of YAFFS2. Therefore, further analysis of ext4 in a forensic perspective will be necessary. Additionally, tools to recovery obsolete data from an Android smartphone that uses ext4 could be developed.

7.3. Conclusion

In our analyses of YAFFS2, we discovered that all data written to a YAFFS2 NAND can only be deleted from the NAND by YAFFS2's garbage collection techniques or by bypassing the file system and directly erasing the NAND's blocks. Because of that, deleted files and obsolete data of modified files can easily be recovered from a Android smartphone that uses YAFFS2 as a file system. However, as long as the NAND is in use, the amount of recoverable data decreases over time until no obsolete data is recoverable anymore. In Chapter 4 and Chapter 5, we showed that the time that obsolete data stays stored on a YAFFS2 NAND depends on the capacity and occupancy of the NAND as well as on the distribution of valid and obsolete chunks among the device's blocks. We also discovered, that YAFFS2 allows a higher amount of obsolete data to be recovered than NTFS does. However, while this obsolete data is erased from a YAFFS2 NAND after a certain time, potentially, it can be recovered from a NTFS device for much longer as NTFS does not feature a garbage collector.

Considering that recovery of obsolete data from a YAFFS2 NAND proved relatively easy, we discussed safe deletion of files as well as complete deletion of YAFFS2 NANDs in the last chapter of this diploma thesis. We showed, that safe deletion of files is theoretically possible but requires complex deletion techniques. As opposed to this, safe deletion of a whole YAFFS2 NAND can easily be achieved by erasing the NAND by use of common MTD tools.

A. Appendix

A.1. Used tools

A.1.1. File creation

```c
#include        <sys/types.h>
#include        <sys/stat.h>
#include        <fcntl.h>
#include        <stdio.h>
#include        <unistd.h>


/**

argv[1]: file length
argv[2]: character to write to the file
argv[3]: file name

Writes a character [argv[1]] times to a file. If the file exists, it is overwritten.
If the does not exist, it is created.

 **/

int main(int argc, char *argv[])
{

        int length = atoi(argv[1]);
        char content = (char) *argv[2];
        char name = (char) *argv[3];

        char    buf1[length];

        int             fd;
        int i;
        int j;

        for(i=0;i<=length;i++){ buf1[i] = content;}

        fd = open(argv[3], O_RDWR | O_CREAT, 0644);

        write(fd, buf1,length);

        close(fd);

        return 0;
}
```

Listing A.1: Source code of `fileWriter`

A.1.2. File modification

```c
#include        <sys/types.h>
#include        <sys/stat.h>
#include        <fcntl.h>
#include        <stdio.h>
#include        <unistd.h>

/**

argv[1]: first byte to be overwritten
argv[2]: number of bytes to be written
argv[3]: character to be written to file
argv[4]: file

Replaces [argv[2]] bytes of a file starting with byte [argv[1]]

**/

int main(int argc, char *argv[])
{

        int startPos = atoi(argv[1]);
        int length = atoi(argv[2]);
        char newContent = (char) *argv[3];
        char buf2[length];

        int             fd;
        int j;

        for(j=0;j<=length;j++){ buf2[j] = newContent;}

        fd = open(argv[4], O_RDWR);

        lseek(fd, startPos, SEEK_SET);

        write(fd, buf2, length);

        close(fd);

        return 0;
}
```

Listing A.2: Source code of `replacer`

A.1.3. File truncation

```c
#include        <sys/types.h>
#include        <sys/stat.h>
#include        <fcntl.h>
#include        <stdio.h>
#include        <unistd.h>

/**

argv[1]: new file size
argv[2]: file

Truncates a file to a size of [argv[1]] bytes

**/

int main(int argc, char *argv[])
{
        int newSize = atoi(argv[1]);
        int fd;

        fd = open(argv[2], O_RDWR);

        ftruncate(fd, newSize);

        close(fd);
        return 0;
}
```

Listing A.3: Source code of `truncater`

A.2. Nandsim parameters

When simulating a NAND flash memory device by use of `nandsim`, the following parameters can be used to create devices of different sizes [14]:

- 64 MiB, 2048 bytes pages: `first_id_byte=0x20 second_id_byte=0xa2 third_id_byte=0x00 fourth_id_byte=0x15`
- 128 MiB, 2048 bytes pages: `first_id_byte=0xec second_id_byte=0xa1 third_id_byte=0x00 fourth_id_byte=0x15`
- 256 MiB, 2048 bytes pages: `first_id_byte=0x20 second_id_byte=0xaa third_id_byte=0x00 fourth_id_byte=0x15`
- 512 MiB, 2048 bytes pages: `first_id_byte=0x20 second_id_byte=0xac third_id_byte=0x00 fourth_id_byte=0x15`
- 1 GiB, 2048 bytes pages: `first_id_byte=0xec second_id_byte=0xd3 third_id_byte=0x51 fourth_id_byte=0x95`

Bibliography

[1] McAfee Labs. Mcafee threats report: Fourth quarter 2010, 2011. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2010.pdf, Last Accessed: April 22, 2011.

[2] Charles Manning. How yaffs works, 2010. [Online] http://www.yaffs.net/files/yaffs.net/HowYaffsWorks.pdf, Last Accessed: April 22, 2011.

[3] Gartner Inc. Gartner press release, 2011. http://www.gartner.com/it/page.jsp?id=1543014, Last Accessed: April 22, 2011.

[4] Micron Technology Inc. Nand flash 101: An introduction to nand flash and how to design it in to your next product, 2010. [Online] http://www.micron.com/products/nand_flash/high_speed_nand.html, Last Accessed: April 22, 2011.

[5] Charles Manning. Yaffs 2 specification and development notes, 2005. [Online] http://www.yaffs.net/yaffs-2-specification-and-development-notes, Last Accessed: April 22, 2011.

[6] Aleph One Ltd. http://www.yaffs.net, Last Accessed: April 22, 2011.

[7] Google Inc. http://www.android.com, Last Accessed: April 22, 2011.

[8] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489 – 502, 2003.

[9] Arie Tal for M-Systems Ltd. Two technologies compared: Nor vs. nand. white paper, 2003.

[10] Paolo Pavan, Roberto Bez, Piero Olivo, and Enrico Zanoni. Flash memory cells - an overview. *Proceedings of the IEEE*, 85(8):1248 – 1271, 1997.

[11] David Woodhouse. Jffs : The journalling flash file system, 2001. [Online] http://sourceware.org/jffs2/jffs2.pdf, Last Accessed: April 22, 2011.

[12] Hynix Semiconductor, Intel Corporation Micron Technology, Inc. Phison Electronics Corp., Sony, Corporation, and STMicroelectronics. Open nand flash interface specification, 2006. [Online] http://onfi.org/wp-content/uploads/2009/02/onfi_1_0_gold.pdf, Last Accessed: April 22, 2011.

[13] Yaffs2 source code, 2010. http://www.yaffs.net/gitweb?p=yaffs2/.git;a=snapshot;h=

213dc0b42fbe8652e454bfdbf9f5c41c6eb4974c;sf=tgz.

[14] David Woodhouse. http://www.linux-mtd.infradead.org/, Last Accessed: April 22, 2011.

[15] Charles Manning. Structure of yaffs2 spare oob area, 2011. [Email communication] http://balloonboard.org/lurker/message/20110203.214218.e77b9b11.en.html, Last Accessed: April 22, 2011.

[16] IEEE Computer Society and The Open Group. Standard for information technology 1003.1 - portable operating system interface, 2008.

[17] Charles Manning. Time stamps in file header chunks, 2011. [Email communication] http://www.aleph1.co.uk/lurker/message/20110323.224326.cc219d84.en.html, Last Accessed: April 22, 2011.

[18] Taeho Kgil, David Roberts, and Trevor Mudge. Improving nand flash based disk caches. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 327–338, 2008.

[19] Charles Manning. Block refreshing, 2011. [Email communication] http://www.aleph1.co.uk/lurker/message/20110320.212508.2a671a70.en.html, Last Accessed: April 22, 2011.

[20] Brian Carrier. http://www.sleuthkit.org/, Last Accessed: April 22, 2011.

[21] Brian Carrier. http://www.sleuthkit.org/autopsy/, Last Accessed: April 22, 2011.

[22] Brian Carrier. *File system forensic analysis*. Addison-Wesley, 2009.

[23] Microsoft Corp. How ntfs works, 2003. [Online] http://technet.microsoft.com/en-us/library/cc781134(WS.10).aspx, Last Accessed: April 22, 2011.

[24] Kyoungmoon Sun, Jongmoo Choi, Donghee Lee, and S.H. Noh. Secure deletion of confidential data in consumer electronics. In *Consumer Electronics, 2008. ICCE 2008. Digest of Technical Papers. International Conference on*, pages 1 –2, January 2008.