Forensic Analysis of YAFFS2

Christian Zimmermann
IIG Telematics
University of Freiburg, Germany

Michael Spreitzenbarth Sven Schmitt Felix C. Freiling*
Department of Computer Science
Friedrich-Alexander-University Erlangen-Nuremberg

Abstract: In contrast to traditional file systems designed for hard disks, the file systems used within smartphones and embedded devices have not been fully analyzed from a forensic perspective. Many modern smartphones make use of the NAND flash file system YAFFS2. In this paper we provide an overview of the file system YAFFS2 from the viewpoint of digital forensics. We show how garbage collection and wear leveling techniques affect recoverability of deleted and modified files.

1 Introduction

The ubiquitous use of smartphones and other mobile devices in our daily life demands robust storage technologies that are both low-cost and well suited for embedded use. There are several reasons why hard disks are not at all well suited for embedded use: physical size, power consumption and fragile mechanics are just some of the reasons. That is why other technologies, namely NAND flash, became very popular and are widely used within modern embedded devices today. NAND flash chips contain no moving parts and have low power consumption while being small in size.

However, NAND flash is realized as integrated circuits "on chip" and comes with some limitations regarding read/write operations that can lead to decreased performance under certain conditions. Furthermore, flash technology is subject to *wear* while in use which may dramatically shorten the chips' lifetime. Thus, various specific techniques have been developed to overcome such shortcomings and to enable flash technology to withstand a substantial amount of read/write operations at constant speeds. Classically, these techniques are integrated into dedicated controllers that implement and enforce the above mentioned flash specific algorithms on the hardware level.

From the perspective of digital forensics, hard drives and low-level structures of various file systems are rather well studied (see for example Carrier [Car05]). The effects of NAND technologies on the amount of recoverable data on storage devices, however, is hardly understood today. Since wear leveling techniques tend to "smear" outdated data

*Contact author. Address: Am Wolfsmantel 46, 91058 Erlangen, Germany.

all over the device, it is often conjectured that digital investigations can profit from the widespread introduction of NAND flash, because it is harder for criminals to delete files and cover their traces. However, we are unaware of any related work that has investigated this question. Probably this is due to the difficulty of circumventing the controllers of NAND flash chips.

Another option, however, to implement NAND flash specific behavior is to use specifically designed file systems. These file systems are aware of the generic flash limitations and take these into account on the software level when reading and writing data from and to the chip. Such file systems are much easier to analyze since they implement techniques like wear leveling *in software*. The most common example of such a file system is YAFFS2, a file system used by the popular Android platform, which is "the only file system, under any operating system, that has been designed specifically for use with NAND flash" [Man02]. YAFFS2 stands for "Yet Another Flash File System 2" and was the standard file system for the Android platform until 2010. Allthough since the beginning of 2011 with version *Gingerbread* (Android 2.3) the platform switched to the EXT4 file system, there are still many devices in use running a lower version than 2.3 and thus using YAFFS2. Therefore, insights into the amount and quality of evidence left on YAFFS2 devices is still of major interest.

Goal of this paper. In this paper, we give insights into the file system YAFFS2 from a forensic perspective. Next to giving a high level introduction to YAFFS2, our goal is to explore the possibilities to recover modified and deleted files from YAFFS2 drives. Since there is no specific literature on this topic, we reverse engineered [ZSS11] the behavior of the file system from the source code of the YAFFS2 driver for Debian Linux running kernel version 2.6.36.

Results. As a result of our analysis, we found out that the movement of data on a YAFFS2 NAND never stops and that obsolete data (that could be recovered) is eventually completely deleted. Thus, a YAFFS2 NAND stays only for a very brief span of time in a state that can be considered a best case scenario regarding recovery of obsolete data. In one of our conducted tests, the block that held a deleted file was finally erased 7 minutes and 53 seconds after the file was deleted. Larger devices have a positive effect on this time from a forensic point of view (i.e., they potentially enlarge the time span). Therefore, the chances to recover deleted data after days or weeks, as can be done on classical hard disks [Car05], are not very high in YAFFS2.

Roadmap. We begin by giving a high-level introduction into the concepts and terminology of YAFFS2 in Section 2. In Section 3 we give insights into the inner workings of its algorithms. We construct and experimentally analyze best case scenarios in Section 4 and present some results regarding the recovery of files in Section 5. We conclude in Section 6.

2 A Brief Introduction to YAFFS2

Blocks and chunks. YAFFS2 separates storage into several areas of fixed size, called *blocks*. Within each block, again there exist several areas of fixed size, but smaller than the size of a block. These areas are called *chunks*. Following the characteristics of NAND flash, a chunk is the smallest amount of data which can be *written* whereas a block is the smallest amount of data which can be *erased* from the flash. Data can only be written to a block if the corresponding chunk was erased beforehand. A chunk that was just erased is called *free*.

Free and obsolete chunks. Since data can only be written to free chunks, modification of data is more complicated than on classical hard drives. To modify data, the data must first be read, then be modified in memory and finally be written back to a free chunk. This method is similar to the well known Copy-on-Write method. YAFFS2 writes chunks sequentially and marks all chunks with a sequence number in the flash. That way, any chunk that was associated with the original data will be identified as *obsolete* although it still holds the original (now invalid) data.

The existence of obsolete chunks is interesting from a forensic investigator's point of view: Whenever one or more obsolete chunks exist within a block, the corresponding data will still be recoverable until the respective block gets garbage collected. After this block gets garbage collected, all of its obsolete chunks will be turned to free chunks.

Header chunks and data chunks. YAFFS2 distinguishes between header chunks used to store an object's name and meta data and data chunks which are used to store an object's actual data content [Man10]. The meta data in such a header chunk describes if the corresponding object is a directory, regular data file, hard link or soft link. In Table 1 the structure of a regular file with three chunks of data and one header chunk is shown.

Block	Chunk	Object_ID	Chunk_ID	Comment
1	0	100	0	Object header chunk for this file
1	1	100	1	First chunk of data
1	2	100	2	Second chunk of data
1	3	100	3	Third chunk of data

Table 1: Structure of a file with one header chunk and three data chunks.

If a file shrinks in size, data chunks become invalid and the corresponding header chunk receives a special *shrink-header marker* to indicate this. In Table 2 we show how a deleted file looks like. In this case chunk number 5 indicates that the file had been deleted and this chunk receives the shrink-header marker. As we show below, shrink-header markers are important because object headers with this marker are prevented from being deleted by garbage collection [Man10, Sect. 10].

Block	Chunk	Object_ID	Chunk_ID	Comment
1	0	100	0	Object header chunk for this file
1	1	100	1	First chunk of data
1	2	100	2	Second chunk of data
1	3	100	3	Third chunk of data
1	4	100	0	New object header chunk (unlinked)
1	5	100	0	New object header chunk (deleted)

Table 2: Structure of the file from Table 1 after the file had been deleted.

Object_ID and Chunk_ID. Each object (file, link, folder, etc.) has its own Object_ID, thus it is possible to find all chunks belonging to one specific object. A Chunk_ID of 0 indicates that this chunk holds an object header. A different value indicates that this is a data chunk. The value of the Chunk_ID stands for the position of the chunk in the file. If you have a chunk with Chunk_ID = 1 it means, that this is the first data chunk of the corresponding object.

The tnode tree. 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* [Man10, Sect. 12.6.1].

Checkpoint data. Checkpoint data is written from RAM to a YAFFS2 NAND device on unmounting and contains information about all of a device's blocks and objects (a subset of information stored in the tnode tree). It is used to speed up mounting of a YAFFS2 NAND device.

The number of blocks needed to store a checkpoint consists of (1) a fixed number of bytes used for checksums and general information regarding the device and (2) a variable number of bytes depending on the number of objects stored on the device and the device's size. The variable part of a checkpoint consists of information on blocks, objects and tnodes.

3 Garbage Collection in YAFFS2

In YAFFS2, obsolete chunks can only be turned into free chunks by the process of garbage collection. Among all of YAFFS2's characteristics and functionalities, the garbage collection algorithm has the most significant impact on the amount of deleted or modified data that can be recovered from a NAND. In this section, we describe the different garbage collection methods (Section 3.1). An important variant of garbage collection is called *block refreshing* and explained in Section 3.2. Additionally, we describe the impact of shrink header markers on garbage collection (Section 3.3). For brevity, detailed references to the source code of the Linux driver can be found elsewhere [Zim11, ZSS11].

3.1 Garbage Collection Methods

Garbage collection is the process of erasing certain blocks in NAND flash to increase the overall amount of free blocks. Valid data that exists in blocks selected for garbage collection will first be copied to another block and thus not be erased.

Garbage Collection can be triggered either from a foreground or a background thread. The trigger within a foreground thread is always a write operation to the NAND. Background garbage collection is not directly triggered by any foreground thread, but executed even when the device is idle. Background garbage collection typically takes place every two seconds.

Interestingly, the behavior of garbage collection does not primarily depend on a device's storage occupancy. Execution rather depends on the current state of blocks regarding the amount of obsolete chunks they hold. Still, every garbage collection can be performed either *aggressively* or *passively*, depending on the device's storage occupancy. Passive background garbage collection only collects blocks with at least half of their chunks being obsolete and only checks 100 blocks at most when searching a block to garbage collect. Foreground garbage collection is executed passively if one quarter or less of all free chunks are located in free blocks and a block with seven-eighths of its chunks being obsolete can be found.

If no block of the entire flash qualifies for erasure, *oldest dirty* garbage collection is executed. This type of garbage collection selects the oldest block that features at least one obsolete chunk. It is executed every time background or foreground garbage collection have been skipped (due to the lack of qualifying blocks) 10 or respectively 20 consecutive times. 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 its variant called *block refreshing* explained below).

Aggressive garbage collection occurs if background or foreground garbage collection is performed and a device does not feature enough free blocks to store checkpoint data. Aggressive garbage collection potentially deletes a higher number of obsolete chunks per cycle than passive garbage collection and is triggered if a device features less than a certain threshold of free blocks, where the threshold depends on the size of the checkpoint data [ZSS11].

Summary from a forensic perspective. The scenario where a maximum of obsolete chunks can be recovered (and therefore the "best" case for a forensic analyst) 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. This results in a behavior in which all blocks are garbage collected as seldom as possible and still feature a high number of obsolete chunks that potentially contain evidence.

3.2 Block Refreshing

YAFFS2's only explicit wear leveling technique is *block refreshing*. Block refreshing is performed during the first execution of garbage collection after mounting a YAFFS2 NAND flash memory device and every 500 times a block is selected for garbage collection. Basically, block refreshing is a variant of garbage collection that does not pay attention to the number of obsolete chunks within blocks. Instead, its goal is to move a block's contents to another location on the NAND in order to distribute erase operations evenly. This technique enables the collection of blocks, even if they completely hold valid chunks.

Whenever block refreshing is performed, it 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.

3.3 Shrink header markers

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 until it becomes the device's oldest dirty block. Its contents can remain stored on a device for a comparatively long time without being deleted by YAFFS2's garbage collector. We practically evaluate the effects of shrink header markers on the recoverability of obsolete chunks in Section 5.1

4 Best case and worst case scenarios

All data written to a YAFFS2 NAND remains stored on the device until the corresponding blocks 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 best case scenario described above is further analyzed for its practical relevance.

4.1 Experimental Setup

All practical evaluations of YAFFS2 discussed in the following were performed on a simulated NAND flash memory device. 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.

YAFFS2 reserves five of the device's blocks for checkpoint data and uses a chunk size matching the device's page size. Hence, a chunk had a size of 2048 bytes. For ev-

ery analysis, we created images of the simulated NAND by use of `nanddump` from the `mtd-utils`.

4.2 General Considerations

As a result of previous discussions, sooner or later all obsolete chunks present on the device are deleted and thus no previous versions of modified files or deleted files exist because of the unpreventable oldest dirty garbage collection and block refreshing techniques.

Passive and oldest dirty garbage collection only collect 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 contains only one obsolete chunk, thirteen executions of passive garbage collection are necessary before the block gets erased.

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. Given the cycle time of 2 seconds for background garbage collection, even in a best case scenario, a block that features only one obsolete chunk gets erased 24 seconds at most after it was selected for oldest dirty garbage collection.

4.3 Experiments

To validate our findings about the best case, we created one file of size 124 928 bytes (respectively 61 chunks) 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 was not 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 of background garbage collection, the block was selected for oldest dirty garbage collection. Subsequently, the block was garbage collected every two seconds due to background garbage collection.

In our conducted test, 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.

4.4 Summary

Since garbage collection cannot be prevented completely, all obsolete chunks will eventually be erased. Therefore, the number of obsolete chunks that can be recovered from a YAFFS2 NAND also depends on the time span between the execution of a file deletion or modification and a forensic analysis.

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 when searching 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.

5 Data Recovery

In the following, we focus on the analysis of best case scenarios regarding recovery of deleted files. For other possible scenarios see Zimmermann [Zim11].

5.1 General Considerations

YAFFS2 uses `deleted` and `unlinked` header chunks to mark an object as deleted. Hence, an object is (at least partially) 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. YAFFS2's `deleted` header chunk is always marked with a shrink header marker. In Table 3, a selection of a block's chunks are depicted. The chunks depicted contain data of files *"fileX"* and *"fileY"*. While *"fileX"* was modified by changing its last chunk's content, *"fileY"* was deleted. As can be seen, modification of a file leads to writing of new chunks (chunk 4) replacing the chunks containing the now invalid data (chunk 1). However, deletion of a file leads to writing of `deleted` and `unlinked` header chunks with the `deleted` header chunk being marked with a shrink header marker.

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. A block containing a chunk marked with a shrink header marker is disqualified for garbage collection until the block gets the oldest dirty block. Therefore, in the best case scenario, the file's `deleted` header chunk has to be stored in the same block as all of the file's chunks in order to protect the block (and respectively the complete file) with a shrink header marker. 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

Block	Chunk	Object_ID	Chunk_ID	Comment
1	0	257	1	fileX: first data chunk
1	1	257	2	fileX: second data chunk
1	2	257	0	fileX: header chunk
1	3	1	0	root directory: header chunk
1	4	257	2	fileX: second data chunk (new content)
1	5	257	0	fileX: header chunk
1	6	258	1	fileY: first data chunk
1	7	258	2	fileY: second data chunk
1	8	258	0	fileY: header chunk
1	9	258	0	fileY: unlinked header chunk
1	10	258	0	fileY: deleted header chunk
1	11	1	0	root directory: header chunk

Table 3: The results of modification and deletion of a file

garbage collection.

5.2 Experimental Recovery of a Deleted File

We created ten files to fill exactly ten of the device's blocks with valid chunks. After creation of a stable initial state by the garbage collector by deleting all obsolete chunks created during the files' creation, 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"*

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 creation of *"fileD"* did not comply to the criteria of a best case scenario as, after the file's deletion, it contained a high number of obsolete chunks.

By analyzing the kernel log entries written by YAFFS2, we could determine that, in our conducted test, the block that held the file was finally erased seven minutes and 53 seconds after the file was deleted (see also [Zim11]). The block was selected for garbage collection after background garbage collection was skipped ten consecutive times. However, the reason for that was not, that the block, at that time being the oldest dirty block, was disqualified for regular background garbage collection. All attempts of background garbage

collection were skipped because the block was not checked for necessity of garbage collection during these attempts. Thus, it was not selected for regular background garbage collection immediately after it became the only dirty block, although that would have been possible. 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. Also, an obsolete chunk can be recovered for a longer time, if the NAND is filled to a higher degree and more blocks have to be garbage collected before the block containing the obsolete chunk in question.

Recovery of chunks that are obsolete due to file modifications differs only slightly from recovery of chunks that are obsolete due to file deletions. Modification of a file does not lead to writing of shrink header markers, except the modification creates a hole bigger than four chunks within the file. Thus, obsolete chunks of a modified file are usually not protected from garbage collection by a shrink header marker. Nevertheless, in a best case scenario, these chunks are recoverable for almost as long as obsolete chunks of deleted files (see also Zimmermann [Zim11] and Zimmermann et al. [ZSS11])).

6 Conclusion

Generally YAFFS2 allows for easy recovery of obsolete data. However, YAFFS2's garbage collector ensures that, over time, all obsolete data is erased. The amount of data recoverable depends on many factors, especially the distribution of valid and obsolete chunks within a device's blocks, the device's size and occupancy and the amount of time that has passed between modification or deletion of a file and the device's disconnection from its power source.

Acknowledgments

This work has been supported by the Federal Ministry of Education and Research (grant 01BY1021 – MobWorm).

References

[Car05] Brian Carrier. *File System Forensic Analysis*. Addison-Wesley, 2005.

[Man02] Charles Manning. YAFFS: The NAND-specific flash file system — Introductory Article. `http://www.yaffs.net/yaffs-nand-specific-flash-file-system-introductory-article`, 2002.

[Man10] Charles Manning. How YAFFS works. http://www.yaffs.net/sites/yaffs.net/files/HowYaffsWorks.pdf, 2010.

[Zim11] Christian Zimmermann. Mobile Phone Forensics: Analysis of the Android Filesystem (YAFFS2). Master's thesis, University of Mannheim, 2011. http://www1.informatik.uni-erlangen.de/filepool/thesis/diplomarbeit-2011-zimmermann.pdf

[ZSS11] Christian Zimmermann, Michael Spreitzenbarth, and Sven Schmitt. Reverse Engineering of the Android File System (YAFFS2). Technical Report CS-2011-06, Friedrich-Alexander-University of Erlangen-Nuremberg, 2011.