Forensic Science International: Digital Investigation 37 (2021) 301188

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2021 USA - Proceedings of the Twenty First Annual DFRWS USA

Coffee forensics — Reconstructing data in IoT devices running Contiki OS

Jens-Petter Sandvik [a, b, *], Katrin Franke [a], Habtamu Abie [c], André Årnes [d, a]

[a] Norwegian University of Science and Technology (NTNU), Norway
[b] National Criminal Investigation Service (Kripos), Norway
[c] Norwegian Computing Centre, Norway
[d] Telenor Group, Norway

ARTICLE INFO

Article history:

Keywords:
Digital forensics
IoT forensics
Contiki
Coffee file system
File version reconstruction

ABSTRACT

The ability to examine evidence and reconstruct files from novel IoT operating systems, such as Contiki with its Coffee File System, is becoming vital in digital forensic investigations. Two main challenges for an investigator facing such devices are that (i) the forensic artifacts of the file system are not well documented, and (ii) there is a lack of available forensic tools. To meet these challenges, we use code review and an emulator to gain insight into the Coffee file system, including its functionality, and implement reconstruction of deleted and modified data from extracted flash memory in software. We have integrated this into a forensic tool, COFFOR, and analyzed the Coffee File System to reconstruct deleted and modified files. This paper presents an overview of the artifacts in the file system and implements methods for the chronological ordering of the deleted file versions, and discusses these methods' limitations. Our results demonstrate that forensic acquisition and analysis of devices running the Contiki operating system can reveal live and deleted files, as well as file version history. In some cases, a complete, chronological ordering of the version history can be reconstructed.

1. Introduction

Internet of Things (IoT) is a term spanning a wide field of technologies, systems, and application areas. IoT systems are found in a variety of application areas, from smart homes and smart cities to agriculture and environmental monitoring. Each IoT system consists of many parts, from cloud solutions and network infrastructure to resource-constrained *things* at the edge of the network. Mobile phones and computers might also be considered part of the IoT system, as they are used as gateways (e.g., wearables with a Bluetooth connection to a mobile phone) and interfaces (e.g., apps receiving updates from and controlling the IoT system). The differences in computing power and resource-constraints among the parts of an IoT system lead to different requirements for their operating systems. Many of the more powerful devices are running Linux or Windows, but specialized operating systems are necessary for resource-constrained devices.

Evidence from IoT systems is often found in the connected cloud platforms or connected applications on a mobile phone or a computer. Evidence in devices is often overlooked with the assumption that the same data is more accessible in other systems. However, there are several reasons to examine IoT devices: to validate the aggregated data's integrity and completeness, find evidence where devices themselves have been compromised, find irregularities in the network infrastructure, or collect data that has not yet been sent to a server.

The current focus on forensic examination of IoT devices is on more capable devices, such as security cameras and routers, typically running Linux. This is a natural focus as most active exploits against IoT systems target these types of devices. As latent faults in other types of devices will be exploited, it is only a matter of time before these systems also are attacked.

Resource-constrained devices are often found in devices intended to run autonomously for long periods on batteries, such as sensor systems, medical systems, and systems retrofitted with internet capabilities. The limitations are the amount of volatile and non-volatile memory, the processing power, and the devices' power usage. Contiki OS and its current active development branch, Contiki-NG, is an operating system designed for resource-

* Corresponding author. Norwegian University of Science and Technology (NTNU), Norway.
E-mail address: jens.p.sandvik@ntnu.no (J.-P. Sandvik).

https://doi.org/10.1016/j.fsidi.2021.301188

constrained devices (Dunkels et al., 2004). Contiki-NG needs about 100 kB for the code and about 10 kB of RAM for running on minimum requirements. It is found in existing products, e.g., among products using the Thingsquare platform.[1]

Contiki-NG defines an application programming interface (API) for the Contiki operating system's file system, and the Coffee File System is an implementation of, and extension to, this API. The Coffee File System introduces micro-logs that write modified pages of the original file to a micro-log file so that the file system does not need to rewrite the whole file on small changes (Tsiftes et al., 2009). Other popular file systems for resource-constrained devices include YAFFS2,[2] FreeRTOS + FAT,[3] Reliance Edge,[4] Reliance Nitro,[5] Wind-River's High Reliability File System (HRFS),[6] LittleFS,[7] and JFFS2.[8] In addition to these specialized flash file systems, conventional file systems can be used in managed flash chips. Managed flash refers to flash memory with an embedded controller for the flash management layer.

The use of these resource-optimized file systems will be more abundant with the increase of resource-constrained IoT devices in common use. Therefore, this work focuses on the Coffee File System, as implemented in Contiki-NG. The resource optimizations in file systems are often similar given similar resource constraints, which means that forensic techniques for one file system can be transferable to other, similar file systems.

Contiki is not currently a widely used operating system for IoT devices, as many of the contemporary devices have relatively powerful hardware capable of running more general-purpose operating systems such as Linux. This might change as more devices are designed to run for long periods on battery and more constraints to the power and memory usage. The Eclipse foundation's annual developer survey for IoT developers confirms that most device developers use Linux. In contrast, Contiki is used by 5% of the developers, and about the same amount as developers using ARM MBed OS, TinyOS, RiotOS, Huawei LiteOS, VxWorks, and QNX (Eclipse Foundation, 2019). To complete the list, FreeRTOS and bare-metal implementations[9] are between these and Linux in popularity.

All types of systems and tools might be used for crimes, either as a tool or as a target. In addition, the system can sense, or in other ways, gain information about crimes, even though it has not been directly involved. IoT systems are no exception to this, and the ability to find relevant evidence from IoT systems becomes increasingly important as the number of deployed systems increases. A forensic investigator's work is to ensure that all relevant aspects of the case are illuminated and that all relevant evidence is documented, both the evidence that signifies guilt and innocence.

The tools that are commonly used in forensics do not support the file system used in Contiki. The Contiki source code tree does include the tool “coffee-manager” to examine and extract files from a flash memory dump, but this only extracts existing files.

This paper examines the file system artifacts from a digital forensics perspective and creates a tool for extracting both existing files, deleted files, and previous versions of both deleted and existing files. This work has three main contributions:

1. Investigate and disseminate forensic knowledge about the Coffee File System used for non-volatile storage in Contiki OS
2. Reconstruct and chronologically order the historical file versions
3. Develop COFFOR, a tool for exporting data from the file system, both deleted and active, and sort the version history of modified files

This work's ultimate goal is to increase the value of the evidence stemming from IoT devices, as the evidence is used during investigations and in court.

The structure of the rest of this paper starts with related work in Section 2 and describes the Coffee file system in Section 3. In Section 4 two methods for reconstructing the file version history are described, and the COFFOR program for forensically extracting data from the Coffee File System is presented in Section 5. Section 6 contains the experiments. Section 7 discuss the results and future work, and Section 8 concludes the paper.

2. Related work

Forensic research in IoT systems has focused on finding aggregated data in more accessible locations, like cloud and companion apps on phones, as well as on other connected devices (Oriwoh et al., 2013; Chung et al., 2017; Kang et al., 2018; Dorai et al., 2018). Other researchers have focused on the network forensic part (Kumar et al., 2016; Rizal et al., 2018). Also, there have been studies on stored artifacts in devices, both in well-known file systems (Awasthi et al., 2018; Li et al., 2019) and in raw memory dumps without a known file system (Jacobs et al., 2017). Faults in devices can lead to erroneous data being recorded, such as failing power that can affect running clocks and timestamps in systems (Sandvik and Årnes, 2018). There has not been much research on the file systems used in resource-constrained devices. One of the reasons is that many IoT devices, especially the more powerful ones, use Linux, and therefore well-known file systems such as ext4.

While there is a lack of literature on the forensic examination of the Coffee File System, there have been studies using Contiki OS and the Cooja simulator. These studies have primarily focused on network forensics and forensic readiness systems. Kumar et al. focused on RAM acquisition and the artifacts from the IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) network architecture in RAM (Kumar et al., 2014). They later implemented rank exploitation attacks against the Routing Protocol for Low Power and Lossy Networks (RPL), showing how to detect that particular type of attack (Kumar et al., 2016). As a research platform for forensic readiness systems, Contiki with the bundled Cooja simulator has also been used for implementing a proof-of-concept system for forensic readiness and evidence storage (Hossain et al., 2018).

The Coffee File System is designed for flash memory with the peculiarities this storage technology brings. Modifying data stored on flash memory can only flip individual bits from “1” to “0”, and to flip bits back to “1”, a whole erase block needs to be reset. An erase block consists of several pages; for the Flash memory Micron M25P80, a page is 256 bytes, and an erase block is 65 kB (256 pages). Writing to flash memory thus needs special considerations, and file systems made for flash memory have similar methods for handling the flash memory peculiarities. Other research efforts into flash file system forensics have focused on the YAFFS2 file system. The YAFFS2 file system is designed for flash memory, and the file

[1] https://www.thingsquare.com, visited 2021–02–07.
[2] https://yaffs.net, visited 2021–02–07.
[3] https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_FAT/index.html, visited 2021–02–07.
[4] https://www.datalight.com/products/embedded-file-systems/reliance-edge-overview/, visited 2021–02–07.
[5] https://www.datalight.com/products/embedded-file-systems/reliance-nitro/, visited 2021–02–07.
[6] https://www.windriver.com/products/vxworks/, visited 2021–02–07.
[7] https://github.com/ARMmbed/mbed-os/blob/master/storage/filesystem/littlefsv2/littlefs/DESIGN.md, visited 2021–02–07.
[8] http://www.sourceware.org/jffs2/, visited 2021–02–07.
[9] No OS, application code runs directly in the device.

system can be found in, e.g., mobile phones. Quick and Alzaabi tested various ways of collecting the raw dump of YAFFS2 from Android (Quick and Alzaabi, 2011). Zimmerman et al. further examined the functionality of the YAFFS2 file system, focusing on the forensic analysis of the file system (Zimmermann et al., 2012).

Many options are available for extracting evidence from embedded systems, each with its own set of advantages and trade-offs (Sandvik, 2017). Physical acquisition is often considered the most forensically sound way of collecting evidence from embedded devices, but it is a resource-demanding task. Breeuwsma et al. provided an in-depth discussion of the physical acquisition of flash memory chips and an analysis of the Flash translation layer (FTL). The paper showed how the spare areas in the flash memory contain information for rebuilding the file system from raw flash dumps (Breeuwsma et al., 2007). Other authors have used statistical methods to classify pages and recreate file contents from fragmented data stored in flash memory (Park et al., 2012).

We are not aware of any paper describing the forensic artifacts of the Coffee File System used by Contiki nodes. In this study, we have therefore developed a new method for analyzing scattered evidence in the Coffee File System, using edit distance to find the correct sector order for reconstructing the version history of the files, and a tool for reconstructing file versions from a flash memory dump. We have assessed the viability of the method and under which circumstances the method has a limited effect. Our results contribute to the forensic investigator's ability to extract evidence found in IoT devices running Contiki OS and containing non-volatile memory.

3. Coffee file system

The Coffee File System was designed for sensor systems that contain flash memory, and the source code is freely available as a part of the Contiki OS (Tsiftes et al., 2009). This section is split into the file system structures in Section 3.1 and the dynamic workings in Section 3.2.

3.1. File system structures

The file system does not contain any metadata structures in any reserved areas, such as the $MFT in the NTFS file system or the superblock in the ext2/3/4 file systems. The file system metadata structures are built up at boot time by scanning the flash memory for file header structures. As the on-disk file system does not contain any metadata structures, the operating system's overview of the file system is kept in RAM.

A page is the smallest unit that can be written in flash memory, and bits can only be switched from 1 to 0 (Micron Technology, 2011). The file system inverts all bits when writing to flash, so a zero in the operating system corresponds to a one in the flash memory. The emulated flash is not inverted. A page can be written several times as long as no bits are set from zero to one, which means that appending to a file and setting (but not clearing) flags can be done. A whole erase block has to be deleted for bits to be set from zero to one, and an erase block is called a sector in the Coffee File System.

The file header structure is found in the source file cfs-coffee.c in the folder os/storage/cfs/ and the structure is shown in Table 1. The flags defined in the file header are given in Table 2. An additional file header structure appears in the log files, which is a file that stores the modifications to the contents of the base file. This header immediately follows the file header and is an index of 16-bit integers corresponding to the modified page number in the original file. The size of this index is the number of modified pages a log file contains.

Table 1
The file header structure of the Coffee File System.

Size (bytes)	Field
2	LOG_PAGE
2	LOG_RECORDS
2	LOG_RECORD_SIZE
2	MAX_PAGES
1	DEPRECATED_EOF_HINT
1	FLAGS
16/40	NAME

The log file has the same file header as a regular file but with the HDR_FLAG_LOG flag set. After the file header, a log file header consists of 16-bit page references for which pages are affected in the original file. The page references are 1-based, which means that page number 1 is the original file's first page, while 0 means that the record has not been used. After this header, a page-sized buffer for each record follows.

The number of records in the log file is given in two different ways. If the original file header has set the log records and log record size parameters, these are used for the size and number of records. Otherwise, the default settings are used. These are four pages in each log file, and each log record is one page, which means four records for each log file and five pages allocated in total. After the records in the log file are written, the last version of the file will be written to a new file, and a new log is initiated. Thus, the first full version of the file is the same as the last version in the previous log file.

One design decision in the file system is that while the header says how many pages are allocated for the file, the file size is not described anywhere. The decision reduces the amount of metadata attached to a file, minimizing disk and memory usage. The actual file size is found by searching backward from the end of the allocated file until a non-zero byte is found. This method of finding the end of a file means that files cannot end with one or more zeroes, as these will be regarded as slack space.

The file system does not support folders apart from "/" and ".". These are used for enumerating the files in the file system with the cfs_opendir and cfs_readdir functions. The lack of folders means that filenames are essentially unique in the file system. There is a theoretical possibility for two files with the same name to exist in the file system, but the identity used when opening a particular file is its name, and thus, there is no way of creating a new file with the same name using the storage API.

From the source code, we can see that the sector size is dependent on the target architecture. Within Contiki, the combination of a hardware type and configuration is called an *architecture*. The target architectures Sky, which targets the TelosB/ Tmote Sky board; Z1, which targets the Zolertia Z1 board; and native, which targets the native architecture of the computer in which the simulator is running, all have a sector size of 0x10000 bytes. The target architecture cc26x0-cc13x0, which targets the Texas Instruments' CC13xx/CC26xx microcontroller, has a sector size of 0x1000 bytes. It is possible to change this at the Contiki compile time to match the target hardware. The file name length is also dependent on the target architecture, where most have a 16-byte limit, while the cc2538 target, for Texas Instrument's CC2538 development board, has a 40-byte limit, but this is also configurable at Contiki compile time.

For reference, the list below shows the terms used in this paper for pages in the file system:

Active: Pages that have been allocated and belong to existing files.

Table 2
Flags used in the file header, from the file os/storage/cfs/cfs-coffee.c in the contiki source.

Value	Flag	Description
0x01	HDR_FLAG_VALID	Header is completely written
0x02	HDR_FLAG_ALLOCATED	File is allocated
0x04	HDR_FLAG_OBSOLETE	File marked for garbage collection
0x08	HDR_FLAG_MODIFIED	File modified, log exist
0x10	HDR_FLAG_LOG	Log file
0x20	HDR_FLAG_ISOLATED	A page from a file that started in the previous, garbage collected sector

Obsolete: Pages that have been active but currently not in use. They are available for garbage collection.
Isolated: Pages that have been active and belonging to a file starting in the previous sector. They exist as file fragments without a header at the start of a sector.
Deleted: Pages that are obsolete or isolated.
Unused: Pages that have not been written to after being erased by the garbage collector.

3.2. File system dynamics

The Coffee File System is similar to YAFFS2, where changes to files are written to new pages. The file system will scan all allocated pages and use the file version that is not marked as obsolete as the current version. In addition, there is a lightweight journal that records and stores modifications to files, called micro-logs.

When writing a file, the file system will initially allocate 0x11 × COFFEE_PAGE_SIZE bytes. The page size is typically 0x100 bytes. If the file size is too small for the file contents and header, the original file will be marked as obsolete, and a new file will be created with twice the allocated size.

Appending to a file, or opening the file for writing with the COFFEE_FD_APPEND option, will transparently append data to a file, just as an ordinary write operation. However, this write operation will not create a new version of the file unless the file content size grows more than the allocated file size. If the append extends beyond the allocated number of pages, the old file version will be marked as obsolete, and a new file will be created with twice the amount of allocated pages.

Modifying the existing contents of a file by overwriting the file or parts of the file will create log file entries if micro logs are enabled for the Contiki build. When a log file is written, the HDR_FLAG_MODIFIED will be set, and a pointer to the page where the log resides will be written in the first 2 bytes of the original file's header, in the field log_page, as described in Table 1. The log file will get the HDR_FLAG_LOG set, which indicates a log file and the same file name as the original file.

The sectors are written sequentially in increasing address order until the end of the file system is reached. A garbage collection (GC) routine deletes the sectors containing nothing but unlinked data before the file system starts writing from the first available sector. As the file system has to delete the whole sector during the garbage collection, the file versions existing in sectors that still have active pages will survive the garbage collection. The sectors containing live data at the time of the GC run will be present during the next cycle of writes, mixing sectors containing older versions of the unlinked files with the newer versions of the files. There is no shuffling of active pages to free up sectors in the file system, which saves the cost of locating live data and moves this to new sectors but results in less inactive sectors that can be collected. The ordering of write operations is, therefore, kept intact within each sector.

Coffee has *extended wear leveling* enabled by default, which will postpone garbage collection until a file allocation fails due to insufficient number of free pages. The garbage collection routine will then erase as many sectors as it can. If extended wear leveling is not enabled, the garbage collection will erase a sector when a sector becomes available for garbage collection. According to the source code comments, the extended wear leveling setting is better for wear leveling, but it can use more time before the garbage collection is finished.

Files written close to a sector boundary can start in one sector and continue into the next. If the first sector is garbage collected and erased, but the next sector is not, then the pages belonging to this file will be marked as *isolated* with the HDR_FLAG_ISOLATED flag in byte 9 of the page. This bit set on pages following a sector boundary tells the file system that the page is free to be collected.

An interesting artifact in the file system is that when a file is written the first time, a base file without a log file is written. The log file is first written at the first file modification. When all records in the log file have been filled, and the file is modified once more, the previous version is copied to a new base file, and the log file is immediately created to hold the last changes. The consequence of this copy process is that the last version of a base and log file couple is the same as the first version of the following base file and log file couple.

4. Reconstructing file versions

In this section, two methods for chronologically ordering the version history of existing and deleted files are described. The first method uses file content similarities to establish the order, while the other uses the file offset analysis.

4.1. File content similarities

Since we know that the write operation order is kept within a sector, the next question is how to find the ordering between the sectors. If content data contains timestamps or other references to external events, the content data can be used to establish a temporal order between sectors. Another way to establish the temporal order between sectors is to check the file version similarities, where the difference between two consecutive versions is smaller than for more distant versions. The last version of a file in a sector is thus more similar to the first version of the file in the following sector than in other sectors. There are several ways of measuring similarities, including the Levenshtein distance, which finds the minimal edit distance between two strings (Levenshtein, 1966), and various *diff* algorithms, such as Myers, which finds the longest common substrings between two strings together with the shortest edit script (Myers, 1986).

In general, we do not have a priori knowledge of an application's writing strategy on the device. An application can use many small write operations or process the file contents in memory and write to flash memory when it reaches checkpoints or is finished processing. If a file contains structured data with a slowly changing or static structure, we can measure differences through edit distances.

If the file is a dynamic file where huge parts change rapidly or are shuffled around, it might be better to use diff algorithms, as these tend to look for blocks of similarity rather than comparing the byte differences in a file. For the writing strategy in our experiments, there were no differences with regard to their performance. However, we note that there might be a difference between the algorithms for other file-changing patterns. For this work, we use Radare2's radiff2 program with the Myers distance. Radiff2 can also use Levenshtein distance and its own distance implementation.[10]

Edit distances are suitable for a subset of files where the changes happen gradually and change the contents away from the original. A ring buffer is an example of a file that gradually changes over time. In a ring buffer, the file is a set size. Upon reaching the end of the file, the write operations wrap around and start over from the beginning. However, it will not work for files where only a single value is updated independently of the time and earlier values or where the file changes completely between each write. In these cases, the edit distance between versions will have the same variability, whether the file versions are close or distant. For files that are gradually changing away from the original, the first and last versions within the sectors can be compared to reveal the internal order.

4.2. File offset analysis

It is possible to calculate at which offsets the files are written to a virtual infinite address space because of the deterministic allocation strategy and the lack of fragmentation, where the offset is the number of pages from the sector boundary. The allocation of a file only reserves a continuous set of pages for a file, prohibiting fragmentation of active files. This virtual, infinite address space can then be mapped to the limited address space of the physical memory, which will be reused as the writing operations reach the end of the file system and the garbage collection frees up sectors for reuse.

The default allocation size of a file is 17 pages plus five pages for the log-file, in total 22 pages, and the sector size is typically 256 pages. The files within one sector are written consecutively and are only erased when all pages in the sector are deleted. As files are written consecutively, and the first file in the first sector starts at offset 0, the 11th file will cross over into the following sector, and the first file in the new sector will start at offset $22\times 12-256=8$ in the sector for a sector size of 256 pages.

In general, the offset of the first file in a sector, n, is given by the equation:

$$A_0 = 0 \tag{1}$$

$$A_n = \left[\left(A_{n-1} + \left\lceil \frac{S_S}{S_F} \right\rceil\right) (\mathrm{mod}\ S_S)\right] (\mathrm{mod}\ S_F) \tag{2}$$

Where A_n is the offset of the first file in the virtual sector n, S_S is the sector size in pages, and S_F is the file size in pages. The ceiling function, denoted by $\lceil\cdots\rceil$, is the lowest whole number greater than or equal to its argument. Setting S_S to 256, and F_S to 22, gives a sequence of 11 possible first file offsets in consecutive written sectors: [0, 8, 16, 2, 10, 18, 4, 12, 20, 6, 14], before the sequence starts over again.

One assumption made here is that files and the log file have the same size throughout the device's lifetime. As a typical device using Contiki is resource-constrained and has limited storage, it is reasonable to believe that file writing will be held to a minimal size.

Table 3
Defaults used in *COFFOR*.

Pagesize	0x100
Sectorsize	0x10000
FS start offset	0x10000
Header size	0x1a

Another assumption made in this analysis is that all file modifications result in a base file and a log file for storing the changes. The assumption has been observed to break when the write operation causes a search for free sectors. When the search skip used sectors, some writing operations only write a base file and continue to write a new base file instead of a log file. For a single occurrence, this can be detected by the file offsets starting at an odd-numbered offset, as the sizes of the base file and the log file both being odd-numbered.

The changes in the file offsets due to the changes in the file sizes can be accounted for by calculating all possible combinations of file offsets given the possible file sizes. This can result in several possible sequences of sectors, and these can be further analyzed to filter out low probability ones, such as no log files written in the sectors.

Given a constant file size, the difference in the first file offset between two consecutive written sectors is constant, and can be found by $A_1 - A_0 = \left(\frac{S_S}{S_F}(\mathrm{mod}S_S)\right)(\mathrm{mod}S_F)$. If one sector deviates from this by missing a log file, all following offsets will be offset by subtracting the missing log file size from these sectors. The first file offset series depends on the number of missing log files, and is given by a modulus subtraction:

$$A_{n,\mathrm{new}} = (A_n - kS_L)(\mathrm{mod}S_F) \tag{3}$$

Where $A_{n,\mathrm{new}}$ is the new offset, k is the number of missing log files, and S_L is the size of the log file.

One missing log file in the first sector results in this sequence of offsets: $[0, 3, 11, 19, 5, 13, 21, 7, 15, 1, 9, 17, 3, \cdots]$. An analysis of the sector offsets can thus find the possible locations of missing log files.

5. COFFOR

For this study, we focus on the emulated flash memory from a Contiki device. For the forensic examination, a tool called COFFOR, short for Coffee Forensics, was developed. COFFOR consists of two programs: a program for extracting files and file fragments from a flash memory dump (coffee_file_extract) and a program for comparing the similarity of file versions (combinefiles.py). The program suite can be found on Github and is released under GPLv3.[11]

The program for examining the coffee file system, "coffee_file_extract", traverses a memory dump from a Contiki device. Coffee_file_extract is written in C, and it needs a flash memory dump as input. Options to the program are –pagesize if the default page size is incorrect, –sectorsize for giving the sector size, –coffeestart to set the start offset of the file system in the flash memory dump, and –directory for the output directory. The defaults are given in Table 3.

Information from Coffee_file_extract execution is written to stdout. If the –directory option is given, the files and file fragments are written to the specified directory. The files are named in

[10] https://rada.re/n/radare2.html, visited 2021–02–07.

[11] https://github.com/jenspets/coffor.

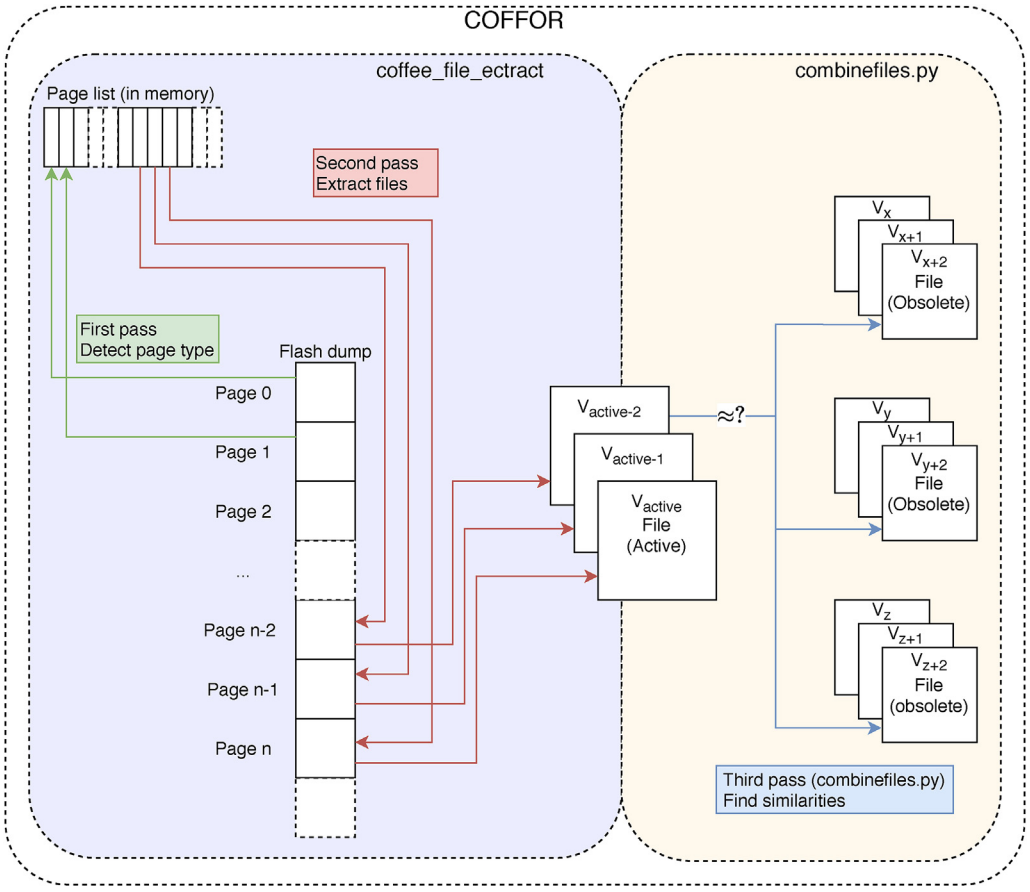

Fig. 1. The functional architecture of COFFOR, including coffee_file_extract and combinefiles.py. The main part is the array of page types and metadata that are used for analyzing the pages.

according to the format *<filename>_<status><startpage>_V<version>*. The filename is the original filename of the file, and the status is either "A" for active, "D" for deleted, or "I" for isolated pages.[12]

The architecture of the program is relatively uncomplicated. It will first map the image file in memory and then process all pages in the disk image. Each page is analyzed and categorized into page types: an active base file, an active log file, a deleted base file, a deleted log file, a file fragment, a zero-page, or an isolated page. This generated list of page types is stored as an array of page type structures, including some metadata from the pages as they are interpreted. The next step is to go through the file type list, exporting the files found to the specified directory. Fig. 1 shows this architecture.

The similarity measures described earlier were not implemented during testing for file version reconstruction, but we used the software suite Radare2's radiff2 for similarity measurements. A small Python script called *combinefiles.py* was written to combine the filenames given as command line parameters into pairs and compare these pairs of files. The comparison was made by executing the radiff2 program and reformat the output to facilitate plotting the results. An example of using the python script is combinefiles.py -sss -t a.txt_A065c_V0001 a.txt_*_V0005, where -sss is the option passed to radiff2, -t is the target file to compare all others against, and the rest is a list of the files that are to be compared against the target. We see that the last version for a particular log file (version 5: "V0005") is used to compare against the first version of the set of files belonging to the target, the active file.

[12] Isolated pages lack a file header, so these files are just without the original file name.

6. Experiments

In order to test the file system, the Cooja simulator was used. A *mote* is a short form for remote and is often used to describe a sensor node or a network-connected device with some computing abilities. A mote that will write files to the file system, append, delete, and overwrite files were created and implemented as an emulated device. For the tests, an emulated "Sky mote" was used. The emulated Sky mote offers a 1 MB M25P80 flash memory interface that can easily be exported and analyzed.

The experiments were performed to validate the documented file system functionality, describe forensic artifacts found after file operations, test the ability to reconstruct file versions' temporal order, and test the developed COFFOR tool's performance.

6.1. File operations

The following actions were taken in the first file system test:

1. File 1 was written and closed
2. File 1 was appended to and closed
3. File 2 was written and closed
4. File 2 was overwritten and closed

5. File 3–12 were written and closed
6. Odd-numbered files between 3 and 12 were deleted
7. File 13–22 were written and closed
8. Odd-numbered files between 13 and 22 were deleted
9. Flash memory was exported

The first file header starts at 0x100000, and the first 32 bytes are as shown in Fig. 2. We can see that there are 17 (0x11) pages allocated to this file, and header flags are 0x3, or Valid and Allocated, as described in Table 2. The file name is file001.txt, and the 16-byte file name field is zero-padded. The first 6 bytes of the file content here is "File1␣".

The append operation in step 6.1 resulted in a write operation on the same page as the existing page. This is possible because an append operation only affects unwritten parts of the page; the rest of the file is still the same as before. As long as the append operation does not increase the file size more than the allocated number of pages (max_pages), the append operation is indistinguishable from a single write operation.

The second file was overwritten, and the micro-logs showed several versions of the file. The logs showed, in addition to the programmed versions, several of the individual write operations. The micro-logs are written sequentially and start with a header that shows the page number of the change. The log files observed had 4 write operations recorded before a new file is written. Another set of write operations were done for establishing log file operations. These were:

1. file001.txt was written and fit in one page
2. file001.txt was overwritten
3. file002.txt was written, this file was 0x100 pages and spanned one sector
4. file002.txt was partly overwritten
5. file003.txt was written, this file spanned 6 pages
6. file003.txt was modified on page 5
7. file003.txt was modified on page 3
8. file003.txt was modified on page 5

Fig. 3 shows file003.txt that has been updated and the corresponding micro-log. At offset 0x32600, the log page field shows 0x0237, and this is the page relative to the start of the file system at 0x10000, which means that the log starts at $0x10000 + (0x0237 \times 0x100) = 0x33700$, as marked in red. After the filename field (16 bytes), the log file contains four 16-bit fields corresponding to the relative page number in the base file that is modified. In the table, the four values are 0x0005, 0x0005, 0x0003, and 0x0003, marked green. This base and log file compound are obsolete, so the table only shows the changes from step 6.1 and 6.1 listed above. One write operation in the source code resulted in two write operations, as seen by the double entries for each page in the list of relative page numbers starting at offset 0x3371a in Fig. 3. After this list, the four modified pages follow and replace the base file's corresponding pages.

The five versions corresponding to this base file and log file pair can be recreated by first restoring the original file, which is the first version. The second version is the original file with the first log-page written at the specified page number. The third version of the file is based on the second version, but with the second log-page written at the specified page number. The process continues until reaching the last log-page.

6.2. Garbage collection

To better understand the garbage collection mechanism, a new mote was created. This mote was designed to trigger garbage collection for various file write strategies. The garbage collector's default settings were tested by writing and deleting files until the write pointer reached the end of the file system. The first test showed that the garbage collection was not performed before the file writes reached the end of the file system's address space, free sectors were purged, and writing started on the first free page. This behavior was as expected. There was no sorting and copying of existing files in order to create more free sectors.

The second experiment was writing small files, deleting every second file after they had been written. This led to a state where all sectors contained both deleted and active files. When the write operation reached the end of the file system, the garbage collector was unable to free up any sectors, and the write operation failed due to no space left on the file system.

The third experiment was writing huge files, such that each file would span at least one sector. When files were deleted, the garbage collection routine was able to free up pages. Fig. 4 shows the result after garbage collection. The figure shows the end of one file, two deleted sectors, then continues in the middle of a file. The figure is shown in the format of the program hd (hexdump), where an asterisk means that the previous line, in this case, all zeroes, is repeated until the next displayed line.

6.3. File version reconstruction using file offset analysis

For this experiment, a node writing two files was used. One of the files was modified each second, while the other was modified every 100 s. This would ensure that they, for most parts, existed in different sectors.

The file os/storage/cfs/cfs-coffee.c in the Contiki OS source tree was patched to print the offset of the created base files to the Cooja simulator's log. The simulation was then run for 2 h simulation time, and at the end of the run, the flash memory was collected.

Table 4 shows the resulting data from the flash memory, and one immediate interesting finding is the odd offsets in sector 0x0d. This indicates a base file written in an earlier sector that does not include a log file. A thorough examination of the simulation log revealed that this had happened several times during the simulation where the allocation algorithm had to skip used sectors before finding the required number of consecutive pages. However, the two sectors, 0 and 1, seem to be consecutive, and an examination of the simulation log validated this.

```
00010000  00 00 00 00 00 00 11 00 00 03 66 69 6c 65 30 30  ..........file00
00010010  31 2e 74 78 74 00 00 00 00 00 46 69 6c 65 31 20  1.txt.....File1

00015300  00 00 00 00 00 00 11 00 00 07 66 69 6c 65 30 30  ..........file00
00015310  33 2e 74 78 74 00 00 00 00 00 46 69 6c 65 30 30  3.txt.....File00
```

Fig. 2. The headers of two files. At the top, a header and first 6 bytes of contents of an existing file. The second file is the header and first 6 content bytes of a deleted file, as can be seen by the flags at offset 9, marked red, of the file header.

```
00032600 37 02 00 00 00 00 11 00 00 0f 66 69 6c 65 30 30 7.........file00
00032610 33 2e 74 78 74 00 00 00 00 00 46 69 6c 65 33 20 3.txt.....File3
00033700 00 00 00 00 00 00 05 00 00 17 66 69 6c 65 30 30 ..........file00
00033710 33 2e 74 78 74 00 00 00 00 00 05 00 05 00 03 00 3.txt...........
00033720 03 00 46 69 6c 65 33 20 20 42 23 30 30 30 30 30 ..File3  B#00000
```

Fig. 3. A modified (and obsoleted) file header with a page address to the log, and the start of the microlog file.

```
00028800 30 37 30 30 30 30 30 66 66 65 00 00 00 00 00 00 0700000ffe......
00028810 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
*
00040000 30 31 30 30 30 30 30 37 39 67 30 30 30 30 30 30 010000079g000000
```

Fig. 4. Result after garbage collection. Note that the sector at 0x40000 starts in the middle of a file, without any file headers in the start of the page. The only header information is in byte at offset 9 of the isolated file, where the file header flag is set for an isolated page (marked red).

Table 4
Offsets to first and last base files in sectors containing data.

Sector	First file	Last file
0x00	0x00	0xf2
0x01	0x08	0x8c
0x06	0x04	0xf6
0x0d	0x01	0xdd

6.4. File version reconstruction using file content differences

A mote was programmed, mimicking a generic application on an IoT node, and it was run in the Cooja simulator as an emulated Sky Mote. After the mote had finished the write operations, the flash memory was exported and analyzed.

The following file operations were performed to test the viability of ordering file versions, even when missing sectors between file versions:

1. Small changes to a value between versions
2. Gradual changes over time
3. Whole file changes between each version

Old file versions disappear when a sector is erased, and the number of file versions within a sector is dependent on the file size of the files written and the number of files being written. The number of file versions in a sector is highly variable, and for our analysis, we therefore only take into account the number of missing file versions, not the number of erased sectors. The file version order is fixed within a sector, so the last version in one sector should be compared to the first versions in other sectors. For this analysis, we instead look at how many missing versions can be erased before the original order cannot be established.

The program *radiff2* from the *Radare2* software suite was used to compare the similarities between the file versions. This program can be run with Radare's original diff algorithm, the Levenshtein edit distance, or the Myers difference algorithm.

The first version in the active (last) set of base and log files was compared with the last version in all other sets of base and log files. Fig. 5 shows the similarity between file versions for three files with different modification strategies. The red line shows a 99-byte file, where a 1-byte location is changed for every write. The blue line shows a file of the same length, where only one value (a counter

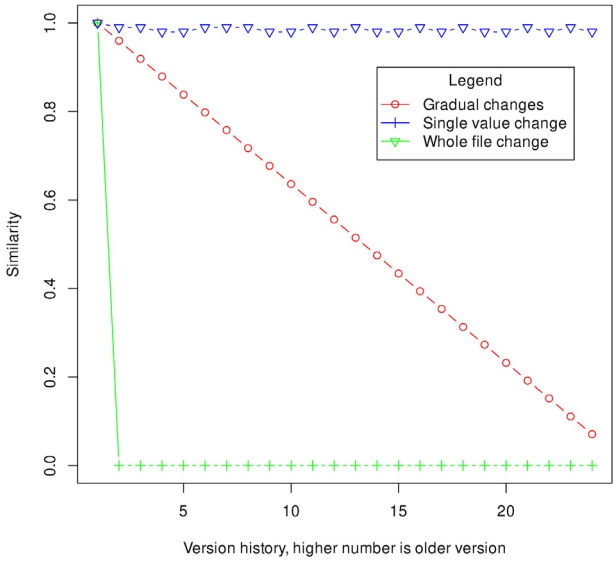

Fig. 5. File version history similarities for a file that are changing gradually from the original (red line), a file where only 1–2 bytes are changing between versions (blue line), and where the whole file changes between versions (green line). The leftmost is the newest version, and the similarity between the active version and the obsolete versions is falling gradually with higher age only for the gradually changing file.

counting from 0 to 99) is changed for every modification. The green line shows a file modification strategy where all the 99-byte file contents are modified for each writing operation.

We see that the gradually changing file is the only file writing strategy that can use the file similarities for establishing an order between versions in different sectors. There is one exception to this observation: The last version in a compound base and log file set is identical to the base file in the following version. If the last version of the last base and log file set in one sector is identical to the first base file in another sector, there is a high probability that the two sectors are chronologically ordered unless the file in question is reset to an earlier version at regular intervals. If a sector containing a file version has been deleted, the erasure will break the equality of the last and first version of the compound files.

An artifact was encountered when reconstructing files spanning more than one sector. If the file is deleted and the sector containing

0002cb00	00 00 00 00 00 00 44 00 00 07 66 69 6c 65 30 30	......D...file00
0002cb10	30 38 2e 74 78 74 00 00 00 00 30 30 30 30 30 30	08.txt....000000
00030000	00 00 00 00 00 00 11 00 00 07 66 69 6c 65 30 30	..........file00
00030010	31 33 2e 74 78 74 00 00 00 00 30 30 30 30 30 30	13.txt....000000

Fig. 6. A deleted file spanning into next sector, which has been erased and overwritten by a newer file.

the last part of the file is erased, a new file can be written at the start of the erased sector. This state leads to two hypotheses for reconstruction: The old file is either correct and continues into the following sector or overwritten by the new file in the other sector. In this case, we need to assess the probabilities for the two hypotheses by looking at sector ordering or which of the hypotheses result in the most sensible result. Fig. 6 shows an example of this in the flash memory. The value marked in blue contains the flags of the file header showing a deleted file, and the value marked in red (0x44) is the number of allocated pages for this file. The old file would thus span to $0x2cb00 + (0x44 \times 0x100) = 0x30f00$, but the newer file starts at offset 0x30000, at a sector boundary.

The coffee_file_extract program takes this into account by checking the pages in deleted files that cross the sector boundaries. If the start of the page looks like a file header, this is marked as a file header, and the program shortens the previous file accordingly with a notice that a new file header is found.

6.5. Coverage of forensic extraction

COFFOR's coverage is quantified by counting the number of pages not categorized. The coverage is given as the number of pages classified by the program divided by the total number of pages. A perfect coverage means all pages containing existing data are extracted and categorized by COFFOR.

The tests were done on ten different flash memory dumps, written by ten different nodes, and all pages in these flash dumps were correctly categorized. This result is promising, but a wider variety of tests need to be performed on different flash memories and for other types of nodes.

7. Discussion and future work

The current research has been done on an emulated device and has not considered flash-specific challenges, such as flash cell failures, bad block handling, address reordering, or other flash management artifacts. There are two reasons for this. The flash-specific challenges happen at a layer below the file system, and they are dependent on the flash memory technology/ chip manufacturer. The added complexity of this should be further analyzed in future work.

Temporal ordering of different files can be done in a similar fashion to the temporal ordering of file versions, as all the differences between files that are spanning more than one sector can create a global ordering of the sectors, thereby ordering all individual files. This technique was not pursued in this article but should be included in a future study. Ordering all files and sectors possible might help order the subset of files where consecutive differences are variable as more distant versions.

The techniques described in this study for ordering the file versions can be used for other flash-aware file systems, but it depends on how data is moved when garbage collection runs. If the garbage collection algorithm moves active pages into new erase blocks, then the historical versions in inactive pages will not be moved with it. They will, therefore, be erased during the garbage collection.

8. Conclusion

We have shown how the non-volatile memory of devices running the Contiki operating system and the Coffee File System can be examined for a forensic investigation. The file system is a minimal file system for resource-constrained devices designed for a flash memory architecture. The period from when a file is modified until the garbage collection runs, together with the probability for inactive pages to be in the same sector as active pages, make it possible for deleted pages to be available for a time after the page has become deleted.

Due to the existence of deleted pages, modified and deleted files can be reconstructed, including file version history. For files within the same erase block or sector, the file version history is chronologically ordered. The chronological order between sectors is more challenging to establish but can be analyzed given file content changes, timestamps in the file, or the other files' chronological ordering.

This study's scientific contribution is a new method of analyzing scattered evidence in the Coffee File System, using edit distance to find the correct sector order for reconstructing the version history of the files. The contribution includes an assessment of the viability of the method and under which circumstances the method has limited effect. For the subset of files that gradually change from the original, the method can establish the order between sectors, thereby substantiate the file version order for gradually changing files.

Also, the scientific contribution includes the development and documentation of COFFOR, a tool for reconstructing file versions from a flash memory dump. Coffee_file_extract, a part of the COFFOR tool, was able to extract and categorize all pages that were in the memory dump. A Python script for ordering intersector versions is also a part of COFFOR, and this script uses the radiff2 program from Radare2. The file ordering method described above is implemented in COFFOR, using the Myers diff algorithm.

The description of the functionality and artifacts of the Coffee File System, as described here, can be used as a reference for the forensic investigator facing these systems, together with a tool for extracting historical versions of the files in the file system.

This study contributes to the forensic investigator's ability to extract evidence found in IoT devices running Contiki OS and containing non-volatile memory. It also showed how to find historical versions of files stored on the device. By improving the amount of evidence from IoT devices and the investigators' ability to determine the order of file versions, court cases will be better illuminated, strengthening the rule of law.

Acknowledgement

The research leading to these results has received funding from the Research Council of Norway program IKTPLUSS, under the R&D project "Ars Forensica – Computational Forensics for Large-scale Fraud Detection, Crime Investigation & Prevention", grant

agreement 248094/O70.

References

Awasthi, A., Read, H.O., Xynos, K., Sutherland, I., 2018. Welcome pwn: almond smart home hub forensics. Proceedings of the Digital Forensic Research Conference, DFRWS 2018 USA 26, S38–S46. https://doi.org/10.1016/j.diin.2018.04.014. URL.

Breeuwsma, M., Jongh, M.D., Klaver, C., van der Knijff, R., Roeloffs, M., 2007. Forensic data recovery from flash memory. Small Scale Digital Device Forensics Journal 1 (1), 1–17.

Chung, H., Park, J., Lee, S., 2017. Digital forensic approaches for Amazon Alexa ecosystem. Digit. Invest. 22, S15–S25.

Dorai, G., Houshmand, S., Baggili, I., 2018. I know what you did last summer: your smart home internet of things and your iPhone forensically ratting you out. In: SeriesProceedings of the 13th International Conference on Availability, Reliability and Security. https://doi.org/10.1145/3230833.3232814.

Dunkels, A., Grönvall, B., Voigt, T., 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In: Proceedings - Conference on Local Computer Networks, LCN, pp. 455–462.

Eclipse Foundation, 2019. Eclipse IoT Developer Survey 2019. Tech. Rep. April. Eclipse Foundation. URL. https://www.slideshare.net/Eclipse-IoT/iot-developer-survey-2019-reporthttps://iot.eclipse.org/resources/iot-developer-survey/iot-developer-survey-2019.pdf.

Hossain, M., Karim, Y.K., Hasan, R.H., 2018. FIF-IoT: a forensic investigation framework for IoT using a public digital ledger. In: Proceedings - 2018 IEEE International Congress on Internet of Things, ICIOT 2018 - Part of the 2018 IEEE World Congress on Services, pp. 33–40.

Jacobs, D., Choo, K.K.R., Kechadi, M.T., Le-Khac, N.A., 2017. Volkswagen car entertainment system forensics. In: Proceedings - 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 11th IEEE International Conference on Big Data Science and Engineering and 14th IEEE International Conference on Embedded Software and Systems, pp. 699–705.

Kang, S., Kim, S., Kim, J., 2018. Forensic Analysis for IoT Fitness Trackers and its Application. Peer-To-Peer Networking and Applications. May.

Kumar, V., Oikonomou, G., Tryfonas, T., 2016. Traffic forensics for IPv6-based wireless sensor networks and the internet of things. In: 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT), pp. 633–638.

Kumar, V., Oikonomou, G., Tryfonas, T., Page, D., Phillips, I., 2014. Digital investigations for IPv6-based wireless sensor networks. Digit. Invest. 11 (Suppl. 2), S66–S75. https://doi.org/10.1016/j.diin.2014.05.005. URL.

Levenshtein, V.I., 1966. Binary codes capable of correcting deletions, insertions, and reversals. Sov. Phys. Dokl. 10 (8), 707–710.

Li, S., Choo, K.-K.R.C., Sun, Q., Buchanan, W.J.B., Cao, J., 2019. IoT forensics: amazon echo as a use case. IEEE Internet of Things Journal 6 (4), 6487–6497.

Micron Technology, 2011. Micron M25P80 Serial Flash Embedded Memory.

Myers, E.W., 1986. An O(ND) difference algorithm and its variations. Algorithmica 1 (1–4), 251–266. URL. http://link.springer.com/10.1007/BF01840446.

Oriwoh, E., Jazani, D., Epiphaniou, G., Sant, P., 2013. Internet of things forensics: challenges and approaches. In: Proceedings of the 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, pp. 608–615. URL. http://eudl.eu/doi/10.4108/icst.collaboratecom.2013.254159.

Park, J., Chung, H., Lee, S., 2012. Forensic analysis techniques for fragmented flash memory pages in smartphones. Digit. Invest. 9 (2), 109–118. https://doi.org/10.1016/j.diin.2012.09.003. URL.

Quick, D., Alzaabi, M., 2011. Forensic analysis of the Android file system Yaffs2. Proceedings of the 9th Australian Digital Forensics Conference (December), 100–109.

Rizal, R., Riadi, I., Prayudi, Y., 2018. Network forensics for detecting flooding attack on internet of things (IoT) device. In: International Journal of Cyber-Security and Digital Forensics (IJCSDF), vol. 7, pp. 382–390.

Sandvik, J.-P., 2017. Mobile and embedded forensics. In: Årnes, A. (Ed.), Digital Forensics. John Wiley & Sons, Ltd. Ch. 6.

Sandvik, J.-P., Årnes, A., 2018. The reliability of clocks as digital evidence under low voltage conditions. Digit. Invest. 24, S10–S17. URL. http://linkinghub.elsevier.com/retrieve/pii/S1742287618300355https://linkinghub.elsevier.com/retrieve/pii/S1742287618300355.

Tsiftes, N., Dunkels, A., He, Z., Voigt, T., 2009. Enabling large-scale storage in sensor networks with the coffee file system. In: 2009 International Conference on Information Processing in Sensor Networks, IPSN 2009, pp. 349–360.

Zimmermann, C., Spreitzenbarth, M., Schmitt, S., Freiling, F.C., 2012. Forensic analysis of YAFFS2. In: SICHERHEIT 2012 – Sicherheit, Schutz und Zuverlässigkeit, pp. 59–69.