Forensic Science International: Digital Investigation 48 (2024) 301689

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS EU 2024 - Selected Papers from the 11th Annual Digital Forensics Research Conference Europe

Ubi est indicium? On forensic analysis of the UBI file system

Matthias Deutschmann [*], Harald Baier

University of the Bundeswehr Munich, Research Institute CODE, Munich, Germany

A R T I C L E I N F O

Keywords:
UBI
UBIFS
MTD
Digital forensics
Data recovery

A B S T R A C T

Crimes involving Internet of Things (IoT) or embedded devices like drones are on the rise. A widespread class of file systems for storing data on embedded devices are flash file systems (FFS). FFS are optimized to manage conceptual limitations and characteristics of raw flash memory, i.e., memory that is not managed by an additional hardware controller that hides the characteristics of flash (called the Flash Translation Layer). Thus, FFS incorporate mechanisms and structures, which are not part of traditional block-based file systems like NTFS, APFS, or ExtX. Regarding analyses of FFS-based embedded devices, digital forensics tools handling FFS are needed. Unfortunately, currently available tools are not able to analyze FFS or raw flash images in general. In this paper, we provide a concept and an open-source implementation of a digital forensics tool bridging this gap for the widespread UBI File System. Our concept is inspired by the well-known Sleuth Kit and reflects the different abstraction layers of a digital forensics analysis (e.g., the storage device level, the volume level, the file system level). We provide an open-source tool of our concept, which we call UBI Forensic Toolkit (UBIFT). In contrast to previous work, UBIFT is able to parse file system structures like the directory tree or the UBIFS journal to recover deleted files including the respective metadata. We show the usefulness of UBIFT by a twofold evaluation: we first apply our tool to a publicly available Internet camera flash dump to perform a forensically sound analysis of the flash device. Our second evaluation comprises both a methodology for creating adaptable flash dumps in general and the comparison of our tool to competitors with similar functionality on the basis of self-generated flash dumps. Finally, we address the usability aspect of UBIFT by providing an Autopsy plugin of our tool.

1. Introduction

The increasing number of IoT devices demands storage solutions that are cost-effective while offering ample space for extensive data. The most common technologies used in conjunction with IoT or embedded devices are NOR and NAND flash memory, respectively (Brewer and Gill, 2008). In both cases FFS play a pivotal role in managing flash-specific characteristics not found on traditional hard disks, they are used to mitigate the complexities by efficiently managing flash-specific characteristics. An alternative approach to manage a flash-based storage chip is the usage of a Flash Translation Layer (FTL). An FTL is tasked to emulate a block device to outside clients, thus hiding the physical properties of the underlying flash. As a consequence, an FTL enables the use of traditional block-based file systems like FAT, NTFS, or ExtX on top. FTLs are mostly put into practice as hardware components on chips, thus essentially being black boxes and their algorithms vendor secrets, additionally hampered by patent restrictions. FTLs are present in various peripherals, including SSDs, SD/MMC cards, and eMMC chips. Unlike FTLs, which require an additional file system on top, FFS are explicitly designed for use on raw flash devices, therefore eliminating unnecessary translation layers (Boukhobza, 2017; Liu et al., 2010).

Axis Communications AB introduced one of the first FFS, called the Journaling Flash File System (JFFS), in 1999 under the GNU General Public License (Woodhouse, 2001). Today, there exist a diverse array of FFS, showcasing the rapid evolution of storage technology. FFS can be found within various consumer products, from smartphones and tablets to digital cameras and USB drives. A notable example of an FFS is YAFFS (YAFFS, 2023) (Yet Another Flash File System). YAFFS finds extensive utilization across diverse areas such as communications (radios, mobile phones, cell towers), industrial equipment, avionics and satellite equipment, sewing machines, office copiers, transport (bus, train, traffic-signal management) and many more (Woodhouse, 2001).

The Unsorted Block Images File System (UBIFS) represents a more recent development in the domain of FFS, it has first been introduced in 2008 in Linux kernel 2.6.27 and has since then been constantly updated. Benchmarks show that it has several advantages over other FFS,

* Corresponding author.
E-mail address: matthias_de@gmx.net (M. Deutschmann).

https://doi.org/10.1016/j.fsidi.2023.301689

particularly in terms of performance (Olivier et al., 2012). Similar to YAFFS, UBIFS can be found in manifold devices, ranging from robot vacuums, Internet security cameras to drones, or routers (Moonbeom Park, 2017). Moreover, UBIFS is an integral component of OpenWRT, a Linux-based operating system designed for embedded devices (OpenWrt, 2023). OpenWRT is a versatile operating system widely deployed in various devices, including routers, NAS systems, smart TVs, VPN servers, and more (Zeifman and Rossi, 2023).

In the context of digital forensics, it is key to understand the fundamental concepts and mechanisms of FFS in order to successfully retrieve valuable information. A significant problem regarding FFS in the field of digital forensics is the lack of practical tools for in-depth forensic investigations. While FFS such as YAFFS2 have been well-studied in regard to digital forensics, providing theoretical background for recovering deleted data (Zimmermann et al., 2012), UBI/UBIFS is currently poorly documented and out-of-scope of the digital forensics community. As a consequence both established open-source tools like *The Sleuth Kit* by Brian Carrier or commercial applications like Cellebrite UFED or MSAB XRY do not provide any support for UBI/UBIFS, leading to the inability of forensic practitioners to retrieve crucial information from raw flash chips from drones, which have become a popular tool for criminals to perform manifold illegal activities. Moreover, currently available open-source tools for UBIFS, such as the UBI Reader,[1] exhibit notable deficiencies in forensic capabilities, such as the recovery of deleted files. Their usability is constrained to a command-line interface without a graphical solution, which could overwhelm forensic practitioners unfamiliar with the intricacies of UBIFS.

In this paper we present a forensic analysis of the UBI ecosystem, comprising different abstraction levels like MTD (Memory Technology Device), UBI and UBIFS. Our prototypical implementation, the UBI Forensic Toolkit (UBIFT), offers robust analyses of UBI/UBIFS structures, enabling the extraction and recovery of deleted files and associated metadata. UBIFT is publicly available at https://github.com/matthias-deu/ubift.

UBIFT is based on established digital forensics concepts introduced by Brian Carrier, which he heavily utilizes in his Sleuth Kit. UBIFT bridges the gap of other tools lacking functionality such as the recovery of deleted data and by providing a structured way of performing a digital forensics analysis. We further strengthen UBIFT's usability by providing a plugin for the established Autopsy[2] digital forensics platform, allowing investigators who may not be familiar with UBIFS to retrieve possibly crucial information. We show the usefulness of UBIFT by a twofold evaluation: we first evaluate our implementation based on a freely available Internet camera flash dump that was extracted from a regular device. Additionally, we provide a methodology to generate artificial NAND flash dumps based on freely available tools in Linux. We finish our evaluation by comparing our tool to related ones with similar functionality.

The subsequent sections are structured as follows: Section 2 introduces the necessary fundamental concepts of FFS in general and UBIFS in particular. We emphasize key forensically relevant data structures and mechanisms of UBIFS, with a specific focus on recovering deleted files. Next, in Section 3, we present the conceptual idea of our developed tool. We further outline how we included established concepts defined by Brian Carrier. Section 4 describes implementation details and introduces the functionality and commands offered by our prototypical implementation. We then evaluate our approach in Section 5 and conclude our paper in Section 6.

[1] https://github.com/onekey-sec/ubi_reader.
[2] https://www.autopsy.com/.

2. Forensically relevant data structures in UBI

As UBIFS is currently not documented with respect to its forensically relevant data structures, we first review general mechanisms of FFS in Section 2.1 that are highly relevant from a digital forensics perspective. In Section 2.2 we turn to important structures and mechanisms of UBIFS in the context of our work. Our analyses of the UBI ecosystem as presented in this section are based on the Linux kernel version `6.2.0-rc7` and will be used in our implementation in Section 4.

2.1. Flash file systems

Unlike traditional file systems like ExtX, FFS have specific requirements resulting from the limitations and degradations of flash memory (Micheloni et al., 2010; Micron Technology et al., 2010). More specifically, flash memory is made up of several blocks of memory cells. Such blocks are the minimal unit for erasure, hence called *erase blocks*. The *erase* operation is unique to flash and not found on hard disks. A block consists of consecutive pages, which are the minimal unit for *write* and *read* operations. Pages also incorporate a spare area, referred to as out-of-band (OOB) data. Its layout differs among manufacturers, housing per-page metadata like error-correcting codes (ECC) or indicators of whether the block has gone *bad*. Flash blocks wear out over time, and when a block goes bad, it cannot be used anymore due to the physical conditions of the flash. Non-empty pages cannot be directly modified in-place due to the physical restrictions, thus their *whole* corresponding block has to be erased first, causing all data of that block to be lost if not saved. A naïve approach of saving data before erasure is given by reading a whole block into a buffer, replacing the contents of a specific page and writing everything back to flash. This approach takes roughly 100 times longer than directly writing the updated page to another empty place (Hunter, 2008). Thus, FFS employ an **out-of-place** update approach. Fig. 1 highlights this concept: page modifications are performed by writing updated pages to another, empty space. The old page is marked as obsolete.

This mechanism is highly relevant in the context of digital forensics, as old pages are typically not immediately deleted, thus their contents can be recovered. Since overall space on flash is limited and obsolete data increases over time, FFS utilize a garbage collector (GC) that is tasked to free up obsolete pages. The implementation of a GC is specific to individual FFS, thus no general conclusion can be drawn regarding the recoverability of data. We performed an analysis of the GC in UBIFS to derive metrics regarding the recoverability of data in UBIFS.

2.2. UBIFS

Fig. 2 shows the UBIFS architecture within the Linux kernel. UBIFS relies on additional layers called **UBI** and **MTD** (MTD, 2023).

Memory Technology Devices (MTD) is an abstraction layer for raw flash devices (MTD, 2023). It is implemented as a Linux subsystem. Its primary aim is to provide a generic interface between hardware drivers

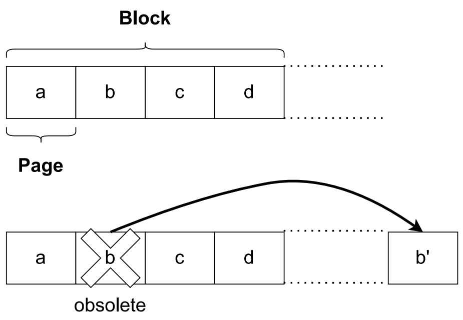


Fig. 1. Flash file system out-of-place update approach.

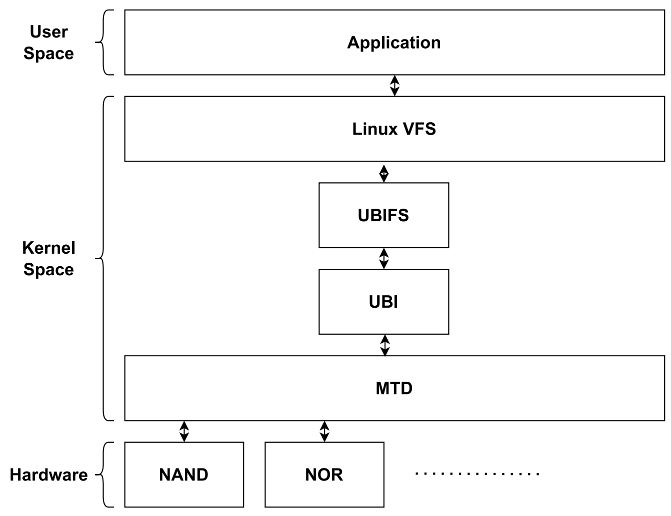


Fig. 2. Architecture of UBIFS (Bharadwaj and Singh, 2019).

and upper layers within the Linux architecture. As such, an MTD device provides uniform access to underlying flash devices. MTD further enables the partitioning of a device into multiple static MTD-partitions. Generally, raw flash devices are not partitioned like hard disks, which contain well-defined partition tables within the Master Boot Record (MBR). Instead, there are three methods of defining partitions within a raw flash device (Abbott, 2013):

1. Hard-coding partitions in the source code of a driver
2. Defining partitions within a bootloader which can then pass its partition layout to the kernel command-line
3. Describing the partition layout via device trees

As a result, deriving partitioning information for raw flash devices in a digital forensics tool may require multiple approaches and distinct heuristics rather than direct extraction from specified locations.

Unsorted Block Images (UBI) is a flash management layer above the MTD layer (Bityutskiy, 2007; UBI, 2020). UBI provides management of multiple flexible volumes that consist of several consecutive logical erase blocks (LEB). LEBs are dynamically mapped to physical erase blocks (PEB) within the MTD layer, thus providing an address translation functionality. Various other flash-related mechanisms such as wear-leveling, bad block handling and scrubbing are also implemented within the UBI layer.

The address translation feature of UBI is shown in Fig. 3: LEBs within the UBI layer are dynamically mapped to PEBs in the MTD layer. Bad blocks are transparently handled by UBI and its wear-leveling mechanism aims to evenly distribute erase operations across PEBs by transferring the contents from an underused PEB to a free one that has undergone a higher number of erasures compared to other blocks[3].

UBI attaches two additional headers to a PEB in order to implement its mechanisms, as shown in Fig. 4: an *erase counter header* (EC) and a *volume identifier header* (VID). The EC header is used by the wear-leveling subsystem of UBI. Its magic bytes of value `0x55424923` (ASCII “UBI#”) are of particular importance in the context of digital forensics, as they mark the start of an LEB, thus they enable the identification of all available LEBs within a device. Further information can then be derived from the additional VID header, which stores a volume ID to indicate to which volume the LEB belongs to, and an LEB number to indicate the particular mapping between LEB and PEB. Respectively, this information can be found in its `vol_id` and `lnum` fields. Similar to the EC header, the VID header has magic bytes of value `0x55424921` (ASCII “UBI!”).

While the EC header is always available (unless there are faulty conditions), the VID header can be missing if a PEB is currently not referenced by an LEB. UBI's volume management is implemented by providing a reserved volume with ID `0x7fffefff` and name `layout volume`. Hence, it can be identified by scanning for LEBs and comparing their `vol_id`. That way, all available UBI volumes can be identified. As instances of UBIFS reside within UBI volumes, it may occur that multiple UBIFS instances are present across multiple UBI volumes.

The UBI File System builds on the UBI layer, relying on its mechanisms such as the translation of LEBs to PEBs. Unlike its predecessor JFFS2, UBIFS stores the file-index on flash, enabling the mounting of large flash memories as the index is not limited to the size of the RAM anymore. However, UBIFS introduces additional complexity in managing the on-flash index. Its file-index structure is realized by a B^+-Tree, which is a specific kind of balanced search tree that is not limited to two children per node. The amount of a node's children is given by its branching factor, also called *fanout*. Per default, UBIFS utilizes a fanout of eight. Compared to a regular B-Tree, a B^+-Tree strictly keeps data in leaf nodes only. Inner nodes, also called indexing nodes, are used to store keys and pointers. UBIFS' file system objects can therefore be found in leaves. Fig. 5 illustrates the structure of a B^+-Tree in UBIFS with a fanout of three. It is noteworthy that index nodes and leaf nodes are strictly kept in separate erase blocks on flash.

UBIFS distinguishes between 14 different node types. Every node type has distinct purpose with varying forensic relevance. All nodes have a common header (`ubifs_ch`), which contains magic bytes of value `0x06101831` within its field `magic`. This enables a straightforward scanning approach: every available node can be found by searching for common headers, which can then be further processed depending on their concrete node type stored in the common header's `node_type` field. Each node within the B^+-Tree has a unique key. Even though

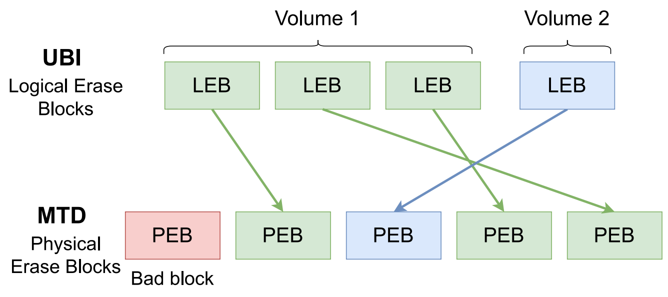


Fig. 3. UBI layer: mapping LEBs to PEBs.

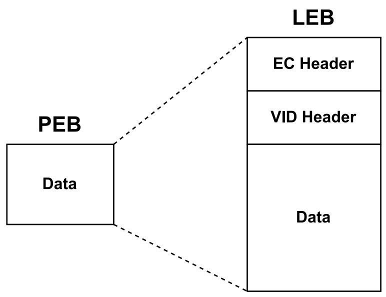


Fig. 4. UBI headers attached to a PEB.

[3] https://bootlin.com/blog/managing-flash-storage-with-linux/ubi/.

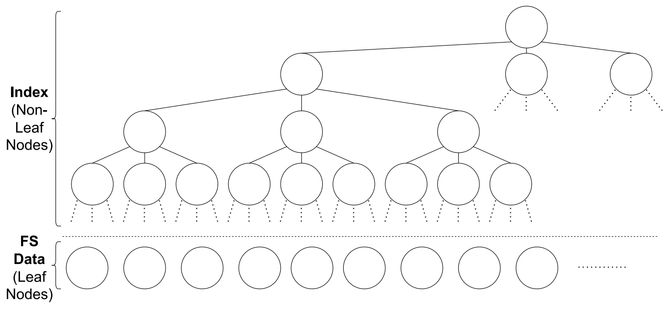


Fig. 5. UBIFS B^+-tree.

UBIFS provides the ability of using multiple key schemes, the only available one is the *Simple key scheme* shown in Fig. 6: keys consist of 64-bits divided into inode number (32-bits), key type (3-bits) and variable content that depends on the type of node (29-bits).

UBIFS adheres to the Linux Virtual File System (VFS) abstraction, hence providing structures needed to integrate with it. One central structure of VFS being inodes. The VFS inode structure (`struct inode`) is wrapped by UBIFS in its `ubifs_inode_node` node. Thus, many digital forensics relevant artifacts found within the original inode structure such as MAC (Modification, Access, Change) timestamps are also available in UBIFS. Regarding timestamps, there is one notable difference: the access timestamp is not written by default to prolong the flash's lifetime. Even though the timestamp is not updated, a test showed that it is initially written on file creation, thus it can be used to derive a file's creation time. Besides the inode node, there are several other forensic relevant node types:

Directory Entry Node A directory entry node (ubifs_dent_node), or *dent* for short, associates an inode number with a name/directory. In UBIFS, the root directory is associated with inode number 1. Generally, there are two inode numbers associated with a dent node: an `inum` refers to the inode number of the file (or directory) that the dent node is associated with, while the inode number encoded in the first 32-bits of the key is the inode number of the parent directory (enabling the recursive determination of the full path).

Data Node A data node (`ubifs_data_node`) resembles a chunk of data that an inode is associated with. By default, a single data node can store the maximum amount of 4096 bytes, defined by the constant `UBIFS_BLOCK_SIZE` in linux/fs/ubifs/ubifs-media.h. The last 29-bits of a data node's key are reserved for its block number, indicating which specific chunk of a file it is.

Master Node Master nodes (ubifs_mst_node) record important positions such as the location of the root index node of the B^+-Tree.

UBIFS is divided into areas that have distinct LEB locations, set during file system creation. Fig. 7 provides an overview of all areas.

For digital forensics, the *Log Area* is of particular importance. The log is part of the journal. The main goal of the journal is to reduce the number of write accesses to the flash. Generally, UBIFS utilizes a *wandering tree* mechanism to implement its out-of-place update mechanism: every time a node within the tree is updated, it is written to another, empty space. As its parent node's reference to the node is now outdated, it is also updated in similar fashion. Hence, nodes are updated out-of-place by also updating all nodes along their parent chain. To minimize

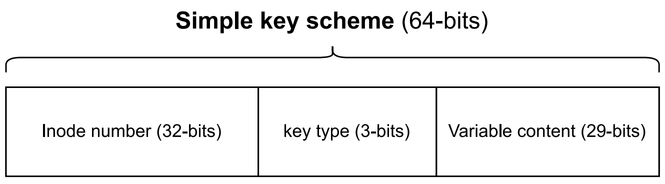


Fig. 6. Simple key scheme.

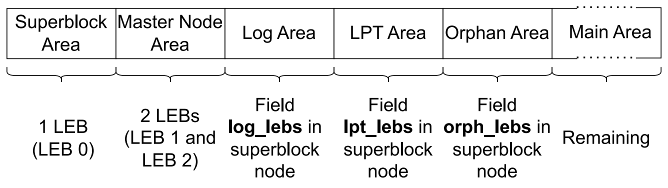


Fig. 7. Areas in UBIFS.

these cascading write accesses to the flash, the journal acts as write cache. All changes to the file system are buffered to the journal before updating the tree. Once the journal reaches a certain size threshold, a **commit** operation is performed, updating all nodes within the file-index tree based on the contents of the journal. This has grave implications for forensics, as mere traversal of the file-index tree would miss files currently only buffered within the journal. As of time of writing this paper, we could not identify any other open-source tool that analyzes the journal. The log as part of the journal contains so-called *reference nodes*. Reference nodes point to LEBs in the main area containing nodes of the journal. Such nodes can be inode nodes, data nodes and so on. The referenced LEBs are also called *buds*. Fig. 8 illustrates this concept.

File deletions and truncations are also buffered within the journal. The handling of file deletions varies depending on whether it involves the deletion of an inode or a directory entry. If an inode is deleted, an inode node with a link count of zero is written to the journal, basically resembling a deletion marker. Upon journal commit, during processing of the deletion marker, the corresponding inode and links pointing to it will be deleted. Directory entry deletions are handled similarly. A directory entry node is written to the journal with its inode number set to zero, also resembling a kind of marker. For truncations, specific truncations nodes (`ubifs_trun_node`) are written to the journal. Old inode and directory entry deletion markers within the journal can be recovered unless the GC has cleaned them, owing to the out-of-place mechanism that preserves old data. Generally two approaches can be used to find such markers.

First, the file-index can be completely disregarded, essentially scanning the whole image. This brute-force approach is straightforward, as it would simply parse every node in the main area. A parsed node can then be checked whether it is an inode node with a link count of zero or a directory entry node with an inode number of zero. The second approach uses structures of the file system to find obsolete nodes. For instance, old reference nodes within the log may be used to find already committed buds. This approach is not very promising, as a new, empty log LEB is used after a commit. Thus, the old log LEB containing obsolete reference nodes is very likely already unmapped and scheduled for erasure.

The quantity of remaining relevant obsolete nodes depends on the GC. If UBIFS runs out of space, the GC follows a greedy approach to turn dirty space, i.e., space occupied by obsolete nodes, into free space. It finds the LEB that has the most dirty space via the LEB property tree (LPT). The LPT stores per-LEB information such as their free and dirty

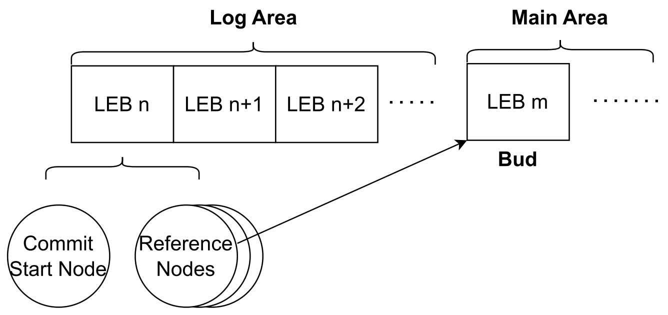


Fig. 8. UBIFS journal.

space. The GC itself is implemented in linux/fs/ubifs/gc.c. Its function `ubifs_garbage_collect` (line 670) is used as entry point. It is called from several other places:

- In the journal implementation `journal.c` (line 142) if no free space is available when trying to allocate space for it.
- In `recovery.c` (line 1193) to allocate a new empty LEB to be used by the GC in the case of a recovery.
- In `budget.c` (line 72). The file `budget.c` is an implementation of the budgeting subsystem in UBIFS. The budgeting subsystem is used to determine how much free space is available. This is, for instance, used to assure that a RAM cache never grows bigger than the available space on flash. This process is further complicated due to various reasons in UBIFS. One reason being that UBIFS performs on-the-fly compression.

In summary, the GC operates passively, only being used when there is no more available space on flash during allocation processes. There is another edge case in which LEBs with obsolete data are erased. If an LEB consists of only dirty and free space, it is considered a *freeable* LEB. A freeable LEB can safely be erased, as it contains no more valid data. The erasure of freeable LEBs happens whenever the journal is committed via the `do_commit` function in linux/fs/ubifs/journal.c (line 97). This function, among other things, invokes the GC's `ubifs_gc_start_commit` function in linux/fs/ubifs/gc.c (line 876). The function's purpose is to unmap and therefore erase all freeable LEBs, causing notable implications for the recoverability of deleted data. Suppose a file is deleted, and its contents are contained within LEBs that become freeable. Consequently, the file cannot be recovered anymore, as the resulting freeable LEBs will be erased.

3. The concept of the UBI Forensic Toolkit

UBIFT provides various functionality to explore and analyze the UBI File System, including methods to restore deleted files based on the previously mentioned structures. Its architecture is based on the established concepts of TSK (especially the use of abstraction layers) by Brian Carrier (2003). As data in its most basic form is only available as collection of bits and bytes, a complexity problem arises. Carrier solves this issue by introducing layers, each layer interpreting data at a higher abstraction level. Thus, data can be viewed either at the lowest abstraction level as bits and bytes if necessary, or at higher abstractions like the file system level as files or other file system related structures.

Carrier's original concept is heavily biased towards traditional block based file systems, therefore we adapted it to fit FFS. One particular issue was the need for an additional layer to encapsulate UBI, as it is a layer not found on traditional file systems. Fig. 9 shows our adapted concept. The lowest accessible layer is the Memory Technology Device (MTD), providing access to physical erase blocks which can be further divided into pages with possibly additional out-of-band areas. Instances of UBI reside in MTD-partitions and finally instances of UBIFS are contained within UBI volumes.

The layers are taken as basic building blocks for the implementation of our tool. We decided to implement a standalone tool instead of extending TSK, as it lacks internal structures for the representation of flash specific structures. Reviewing the TSK source code revealed challenges in extending it to work with FFS like UBIFS, primarily due to its inherent bias towards traditional file systems. Adapting TSK would require extensive changes, which was out of scope for this project. Nevertheless, as our implementation adheres to the concepts of TSK, there is the possibility that it may be integrated into TSK in the future, or it may remain as a standalone framework for flash file systems, resulting in a flexible compromise.

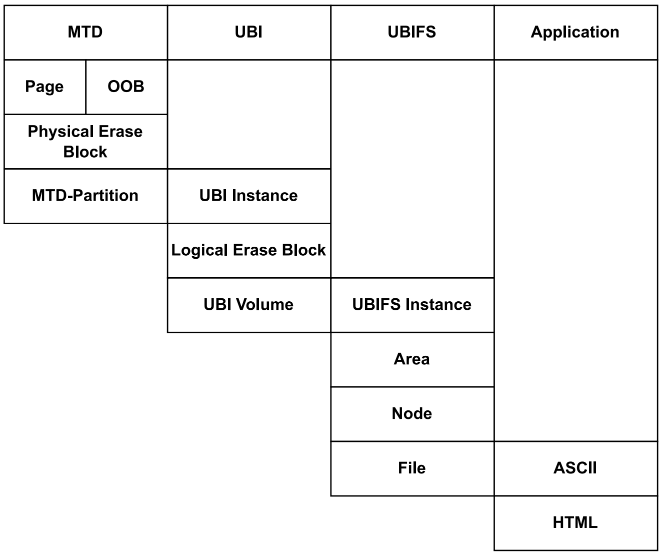


Fig. 9. UBIFT abstraction layers inspired by (Carrier, 2003).

4. Implementation

In this section we provide details on how we put our concept from Section 3 into practice. We make use of Python 3 for the implementation of our tool.

Much like TSK, UBIFT organizes its commands into layers using prefixes and suffixes. The prefix signifies the layer, while the suffix specifies the command to be executed. The currently available commands are detailed in Table 1. Refer to Section 2.2 for a detailed description of the different UBI layers and to Section 5.1 for a sample use of various UBIFT commands on a sample flash dump.

Most commands support specifying an additional `scan` or `deleted` parameter. These parameters cause UBIFT to utilize the previously mentioned scanning mechanism, instead of traversing the file-index, to find nodes that are not part of the B^+-Tree anymore, i.e., deleted content. The `deleted` differs from the `scan` parameter in that it exclusively shows nodes not found within the B^+-Tree, while the other parameter includes those within the tree as well. Listing 4.1 shows an exemplary invocation of the `fls` command plus the `deleted` parameter.

Notable commands in the context of file recovery are `fls` and `ils`, respectively. As in TSK, `fls` outputs information about file names and metadata addresses. Listing 4.1 shows a sample use of `fls` to extract deleted files from a sample flash dump, where the type, metadata address, parent directory and file name are output, respectively. As all extracted files are deleted, we assign an inode address of zero to them.

```
$ python ./ubift.py fls flash_dump.bin -o 0 -n data
↪  --deleted
Type    Inode    Parent    Name
file    0        105       secret.txt
dir     0        104       secret_folder
file    0        107       secret_image1.jpg
file    0        107       secret_image4.jpg
```

Table 1
Available UBIFT commands.

Prefix	Suffixes	Prefix	Suffixes
mtd	ls, cat	fs	stat
peb	cat	i	ls, cat, stat
ubi	ls, cat	f	ls, find
leb	ls, cat	j	ls

Listing 4.1. Exemplary fls output. The inode number of zero is a marker for deleted directory entries within the journal.

ils offers a comprehensive overview of all available inodes within UBIFS and displays important forensic metadata such as user/group IDs, timestamps, file type (a sample output of ils for an Internet camera is depicted in Listing 5.4). When used in conjunction with the optional deleted parameter, a tabular view of deleted inodes is obtained, offering valuable insights into their metadata.

UBIFT provides two additional commands for convenience and further analysis:

ubift_info This command presents information on the recoverability of deleted inodes. It calculates the cumulative amount of deleted data by summing up the size field values for each identified deleted inode.

ubift_recover This command enables the convenient export of all files (including deleted ones) from an image into a local directory.

We streamline the UBIFT process through our Autopsy plugin, which seamlessly integrates with the UBIFT CLI in the background. By leveraging the ubift_recover command for image processing, the plugin efficiently handles the entire procedure. This involves copying the provided image to a temporary folder, executing the UBIFT CLI, and capturing its output as a comprehensive report appended to the Autopsy case. Thus, digital forensics practitioners can utilize the familiar Autopsy file browser for a user-friendly investigation. Tested with Autopsy 4.20.0, the plugin is available for review on https://github.com/matthias-deu/ubift.

5. Evaluation

We present a twofold evaluation of our tool UBIFT. First, in Section 5.1, we apply UBIFT to a publicly available flash dump of a Foscam R2 camera[4] to demonstrate its practicality and value in digital forensics investigations. Second, in Section 5.2 we provide a methodology of creating scenario-based adaptable flash dumps in Linux together with the respective ground-truth. We generate diverse dumps to minimize the likelihood of potential implementation errors, demonstrating the robustness of UBIFT and highlighting its superiority to competitors like the UBI Reader (GitHub, 2023-08-13b).

UBIFT also underwent additional evaluation in collaboration with a third party within a confidential security research project. This comprehensive evaluation involved a complete hardware forensics work cycle, encompassing the chip-off process of a commercial drone's NAND flash and subsequent analysis of the extracted data using UBIFT. The project served as a primary reference for defining UBIFT's feature set and established a baseline for its capabilities. In contrast to UBIFT, both UBI Reader and UBIFS Dumper failed to work correctly with the chip-off dump. Because disclosing detailed information on this depends on the lifting of a confidentiality agreement, we are planning to conduct further UBIFT evaluations on additional open data sets in the future.

5.1. Use case study

The flash dump of the Foscam R2 camera is provided as several binary files, one for each MTD partition. As it is rather uncommon to know the actual MTD partitioning of an acquired flash dump beforehand, we concatenate each binary file, increasing the difficulty imposed on our tool.

We initiate the analysis of the flash dump by employing the mtdls command to identify all MTD partitions, encompassing those housing UBI instances. The results are presented in a comprehensive tabular format. An overview of the Foscam partitioning determined by UBIFT is shown in Listing 5.1. A partition designated as *Unallocated* signifies the absence of UBI, though it may still contain other data, such as a bootloader. For further manual analysis, such partitions can be extracted using the mtdcat command.

```
$ python ./ubift.py mtdls foscam.bin
[...]
        Start    End     Length    Description
000:    0        212     213       Unallocated
001:    213      610     398       UBI
002:    611      657     47        UBI
003:    658      996     339       UBI
[...]
```

Listing 5.1. UBIFT mtdls output.

Subsequently, the UBI instances that have been identified can undergo more in-depth analyses, involving the exploration and analysis of their UBI volumes or individual LEBs. Similar to the mtdls command, the ubils command provides a comprehensive overview of available UBI volumes within the analyzed UBI instance. Listing 5.2 shows the existence of a dynamic (i.e. writable) UBI volume within the UBI instance starting at PEB 213.

```
$ python ./ubift.py ubils foscam.bin -o 213
[...]
        Start          End            Length
0000:   000213         000610         000398
|
|       Volumes
|       Index    Reserved PEBs   Type        Name
|       0        0000000360      DYNAMIC     rootfs
```

Listing 5.2. UBIFT ubils output.

On the next abstraction layer the identified UBI volumes can be subject to a more granular investigation utilizing one of UBIFT's diverse commands designed for performing analyses on the file-system layer. An investigator may seek to obtain a comprehensive overview of all files that can be found via scanning approach, including all recoverable files, in order to retrieve important evidence.

An illustrative example of this process is shown in Listing 5.3, which demonstrates the usage of the fls command used in conjunction with the scan option. The tabular overview provides investigators with a comprehensive view of available files. An inode number of 0 serves as a clear indicator that the corresponding directory entry has been deleted. As demonstrated in Listing 5.3, our scanning approach unveils the existence of a bash history file .ash_history. Remarkably, this file remained undetected by the **UBI Reader** and **UBIFS Dumper**, underscoring UBIFT's distinctive scanning technique. The bash history is an important artifact in digital forensics investigations, as it provides a record of commands executed by users, offering valuable insights into their activities, intentions, and potential evidence of malicious actions or system compromises.

```
$ python ./ubift.py fls foscam.bin -o 213 -i 0
↪  --scan
Type    Inode    Parent    Name
file    0        269       lighttpd.pem
file    0        215       .hwdb.binnSQTCJ
file    0        80        group.31Kh44
file    0        80        passwd.2730LI
file    0        80        .passwdIeH9nr
file    4025     1         .ash_history
[...]
```

Listing 5.3. UBIFT fls output.

More information, including important metadata like timestamps, can be obtained using the ils command. Similar to fls, the command will output a comprehensive table as shown in Listing 5.4. It is worth noting that after the initial identification, a total of 18 data nodes can still be recovered from the previously identified bash history associated

[4] https://github.com/timawesomeness/foscam-r2-fw.

with inode number `4025`.

```
$ python ./ubift.py ils foscam.bin -o 213 -i 0
↪  --scan
[...]
inum|uid|gid|timestamps|mode|nlink|size|data|dent
4023|0|0|2000|2000|2000|LINK|rwxrwxrwx|1|35B|0|1
4024|0|0|2000|2000|2000|LINK|rwxrwxrwx|1|43B|0|1
4025|0|0|1970|1970|1970|FILE|rw-------|1|530B|18|1
4026|0|0|1970|1970|1970|DIR|rwxr-xr-x|0|160B|0|1
111|0|0|2017|2019|2019|FILE|rw-r--r--|1|17.1KiB|5|1
[...]
```

Listing 5.4. UBIFT `ils` output (truncated).

Finally, the contents of the bash history can be retrieved using the `icat` command, as shown in Listing 5.5. We see that the user opened a bash shell and inspected the system files showing all users (`/etc/passwd`) and their respective hashed credentials (`/etc/shadow`).

```
python ./ubift.py icat foscam.bin -o 213 -i 0 --scan
↪  4025
[...]
cat /etc/passwd
cat /etc/shadow
[...]
```

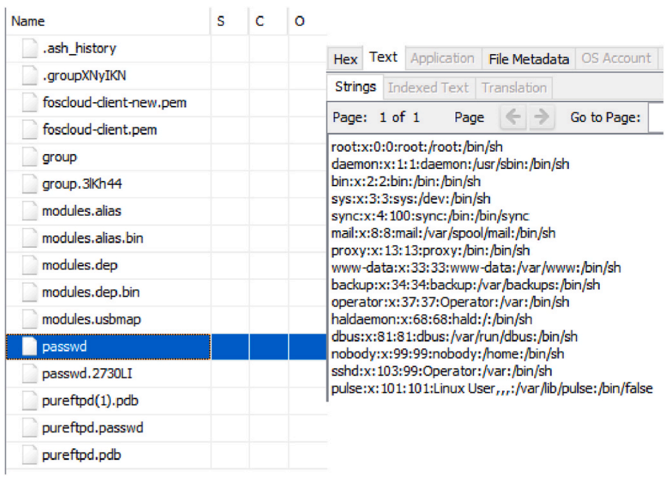
Listing 5.5. UBIFT `icat` output.

UBIFT stands out from other tools like the **UBI Reader** and **UBIFS Dumper** not solely for its capability to retrieve information about deleted inodes, but also for its structured examination features, which collectively play a pivotal role in recovering evidence that could ultimately lead to identifying and apprehending a perpetrator.

As a manual analysis requires some degree of knowledge about the file system, we provide an Autopsy ingest module. The plugin facilitates graphical exploration of all available files, including recoverable deleted files. In Fig. 10, a visualization in Autopsy of recoverable files within UBIFS of the first UBI instance is presented. The seamless integration of UBIFS into Autopsy empowers investigators, even those without prior knowledge of this file system, to conduct purposeful and effective forensic examinations. To the best of our knowledge, no other tool currently offers a graphical interface for UBIFS.

5.2. *Artificial flash dumps*

Since publicly available flash dumps of embedded devices are sparse, we propose a methodology for creating artificial flash dumps that can be used for the evaluation of arbitrary tools. The methodology relies on



```
.ash_history
.groupXNyIKN
foscloud-client-new.pem
foscloud-client.pem
group
group.3lKh44
modules.alias
modules.alias.bin
modules.dep
modules.dep.bin
modules.usbmap
passwd
passwd.2730LI
pureftpd(1).pdb
pureftpd.passwd
pureftpd.pdb
```

```
root:x:0:0:root:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:100:sync:/bin:/bin/sync
mail:x:8:8:mail:/var/spool/mail:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
operator:x:37:37:Operator:/var:/bin/sh
haldaemon:x:68:68:hald:/:/bin/sh
dbus:x:81:81:dbus:/var/run/dbus:/bin/sh
nobody:x:99:99:nobody:/home:/bin/sh
sshd:x:103:99:Operator:/var:/bin/sh
pulse:x:101:101:Linux User,,,:/var/lib/pulse:/bin/false
```

Fig. 10. Recovered files in Autopsy.

open-source tools available within the `mtd` and `mtd-utils` Linux packages. The `mtd` package contains the `nandsim` module, capable of simulating NAND flash in RAM with various options. Listing 5.6 shows the command necessary to emulate a `16 MiB` NAND flash with a page size of `512 Bytes`.

```
# modprobe nandsim id_bytes=0x20,0x33
[...]

# dmesg
[1465.819526] nand: device found, Manufacturer ID:
↪  0x20, Chip ID: 0x33
[1465.819527] nand: ST Micro NAND 16MiB 1,8V 8-bit
[1465.819527] nand: 16 MiB, SLC, erase size: 16 KiB,
↪  page size: 512, OOB size: 16
[...]
```

Listing 5.6. `nandsim` Linux module and `dmesg` output.

After creation of the flash in RAM our methodology contains the following steps:

1. Create an arbitrary amount of UBI instances within the MTD device using the `ubinize` tool within `mtd-utils`. The `ubinize` tool requires a config file as parameter which describes the volumes in the UBI instance. Contents of a volume can be specified in the config file using the `image` parameter.
2. Either create a UBIFS image beforehand and supply it to a UBI volume via the `image` parameter, or directly format UBI volumes. Both is achieved using the Linux `mkfs.ubifs` tool.
3. UBI may be attached used the `ubiattach` tool within `mtd-utils`, to mount available UBIFS instances within UBI volumes. Once mounted, the file system can be manipulated to create artificial crime or testing scenarios.
4. Dump the contents of the MTD device into a file, so it may be used for evaluation. This is done via `nanddump` tool of `mtd-utils`.

In order to streamline the process, we offer a Python script that automates the majority of the tasks involved. It can be found in the ubigen folder of our repository.[5] It is implemented as a CLI offering the following sub-commands:

`create` Creates a NAND device in RAM, followed by generating an arbitrary folder structure using names extracted from an external text file. A subsequent loop randomly selects and places files from the local directory into corresponding folders, ensuring a diverse arrangement. Currently, the random seed value responsible for the process cannot be altered, but the number of files taken can be supplied via `file_count` parameter. The created folder structure forms the root directory for the subsequently generated UBIFS image, for which a UBI instance containing one volume is then created.

`mount` Takes the path to an image and attempts to mount it. Using the previously created UBI image file via `create`, this creates the device **/dev/ubi0_0**, which will be mounted as UBIFS at the mount point **/mnt**. A forensic practitioner may then manually manipulate its contents.

`simulate` Simulates a specified number of random file deletions and copies, determined by a `count` parameter, on a given mounted UBIFS instance. This can be used to evaluate the recoverability of deleted files by producing obsolete data on flash, so that the GC is forced to become active. Thus, it facilitates the evaluation of the extent to which the GC affects the recoverability of deleted files. Currently, the operations are only displayed on the console. In future script iterations, incorporating a comprehensive log to track the ground truth, detailing the files copied and deleted, would be beneficial. Presently, the user cannot set the random seed here either.

In Listing 5.7, we provide an excerpt of the `mount` command's

[5] https://github.com/matthias-deu/ubift.

implementation, wherein the requisite Linux tasks are programmatically invoked. This automation simplifies tasks, enabling users to perform them without delving into technical details. Similarly, the other commands allow users to focus on essential image creation tasks, avoiding technical intricacies.

```
ubiformat = ["ubiformat", f"/dev/mtd{mtdn}", "-f",
 image_path, "--vid-hdr-offset", "2048"]
print(self._execute_command(ubiformat))

ubiattach = ["ubiattach", "/dev/ubi_ctrl", "-m",
 str(mtdn), "--vid-hdr-offset", "2048"]
print(self._execute_command(ubiattach))

mount = ["mount", "-t", "ubifs", "-o", "sync",
 "/dev/ubi0_0", "/mnt"]
print(self._execute_command(mount))
```

Listing 5.7. Excerpt of the `mount` command's implementation in `ubigen`.

The commands are designed so that multiple forensic scenarios may be created. A user can employ the `create` command to generate a foundational image, subsequently utilizing the `mount` command for mounting and customizing it according to distinct use-case scenarios. These resultant scenarios not only serve as valuable test environments for forensic tools but also as invaluable educational assets for aspiring forensic practitioners.

As of now, the script is limited. Parameters such as the geometry of the created NAND in RAM are hard-coded. Nevertheless, it proved to be a valuable factor in the evaluation of our tool. Further development can potentially benefit many projects related to flash. In future iterations, there is potential to elevate the tool's capabilities by extending support to various flash file systems, while also providing flexibility for users to customize the flash and file system geometries according to their specific requirements.

5.3. Related tools

Lastly, we compare UBIFT to two of the most established tools on GitHub which provide similar functionality.

UBI Reader (GitHub, 2023-08-13b): The UBI Reader is a Python module and script collection designed for extracting data and analyzing UBI/UBIFS images. The extensive array of standalone scripts within the UBI Reader presents challenges in terms of usability. To accomplish specific tasks, users must initially locate the relevant script and explore its functionality, introducing potential complexity. When applied to an artificially generated flash dump devoid of structural errors, UBI Reader produces outcomes akin to those of UBIFT. Using its script `ubireader_extract_files` on the Foscam dump, it creates three folders containing the UBI/UBIFS instances with their respective files, similar to the UBIFT `ubift_recover` command. Nevertheless, UBI Reader lacks features for the recovery of deleted files and does not consider the file system journal—a known issue documented on the author's GitHub page. Consequently, uncommitted data within the journal that is potentially crucial to digital forensics investigations remains overlooked, possibly resulting in the omission of evidence crucial to the success of an inquiry. As an illustration, our attempts to recover the previously mentioned bash history from Section 5.1 using UBI Reader proved unsuccessful.

UBIFS Dumper (GitHub, 2023-08-13a): The UBIFS Dumper is provided as a single Python script that is implemented as a CLI. It provides the functionality to view or extract the contents of UBIFS images. However, when employed on the Foscam dump, the tool encounters issues, yielding the output message *Unknown file type*. Conversely, it operates effectively when used on a single MTD partition, implying a possible limitation in handling multiple UBI instances. Furthermore, similar to the UBI Reader, the tool lacks journal support, rendering it unable to recover deleted files.

In summary, UBIFT distinguishes itself from tools such as the **UBI Reader** and **UBIFS Dumper** by introducing novel features and capabilities:

UBIFT is based on established TSK layer concept Incorporating concepts from Brian Carrier's Sleuth Kit aims to enhance the acceptance of UBIFT within the digital forensics community. However, integrating it into The Sleuth Kit appears questionable at present, given the additional layers introduced by the UBI ecosystem.

Comprehensive recovery of deleted files UBIFT sets itself apart as the only currently available tool that enables the recovery of deleted files by leveraging the journal of UBIFS. UBIFT's distinctive scanning mechanism enables the recovery of data that other methods, such as relying solely on file index tree, fail to uncover.

Ease of use UBIFT prioritizes usability, making it a standout choice for digital forensics practitioners, as it offers a streamlined solution with a unified CLI and an integrated Autopsy plugin.

6. Conclusion and future work

In this paper, we presented UBIFT, a Python toolkit providing the ability to perform in-depth digital forensics evaluations on UBIFS. We showed that UBIFT enables a structured analysis of an image while being compliant with established concepts introduced by Brian Carrier. We further provided an analysis and evaluation of UBIFS, highlighting the fact that deleted data can be recovered due to its journal and the out-of-place update approach generally employed by flash file systems. To the best of our knowledge, no other tool is capable of recovering deleted data in UBIFS.

However, there are several limitations. First, UBIFT is not able to handle all kinds of flash dumps, e.g., flash dumps that contain erroneous structures or bit flips may cause unforeseen issues. This is a consequence of UBIFT mainly being tested with self-created dumps containing no erroneous structures whatsoever. Many edge cases such as encrypted instances of UBIFS have not been tested either. In future works, we aim to enhance the robustness and versatility of UBIFT by subjecting it to a more comprehensive array of data sets, including those featuring edge cases such as erroneous structures or encrypted instances of UBIFS.

Second, we plan to extend the capabilities of UBIFT beyond its current support for UBIFS. Drawing inspiration from related works, such as the forensic analysis conducted by Zimmermann et al. (2012) on YAFFS2, we envision the evolution of UBIFT into a versatile tool capable of supporting various flash file systems.

Acknowledgement

The authors thank the anonymous reviewers for their detailed comments and Frank Adelstein for his valuable advice in the generation of the final version.

References

Linux for embedded and real-time applications. In: Abbott, D. (Ed.), 2013. Embedded Technology Series, third ed. Elsevier Science, Burlington.

Bharadwaj, N.K., Singh, U., 2019. Acquisition and analysis of forensic artifacts from raspberry pi an Internet of things prototype platform. Recent Findings in Intelligent Computing Techniques 707, 311–322. URL. https://www.researchgate.net/publication/328732814_Acquisition_and_Analysis_of_Forensic_Artifacts_from_Raspberry_Pi_an_Internet_of_Things_Prototype_Platform_Proceedings_of_the_5th_ICACNI_2017_Volume_1.

Bityutskiy, A., 2007. UBI - unsorted block images. URL. http://www.linux-mtd.infradead.org/doc/ubi.ppt. (Accessed 30 September 2023).

Boukhobza, J., 2017. Flash Memory Integration: Performance and Energy Issues. Energy Management in Embedded Systems Set. Elsevier Science, San Diego. URL. http://www.sciencedirect.com/science/book/9781785481246.

Brewer, J.E., Gill, M., 2008. Nonvolatile Memory Technologies with Emphasis on Flash: A Comprehensive Guide to Understanding and Using NVM Devices.

Carrier, B., 2003. Defining digital forensic examination and analysis tools using abstraction layers. URL. https://www.utica.edu/academic/institutes/ecii/publications/articles/A04C3F91-AFBB-FC13-4A2E0F13203BA980.pdf.

GitHub, 2023-08-13. ubidump: tool for viewing and extracting files from an UBIFS image. URL. https://github.com/nlitsme/ubidump.

GitHub, 2023-08-13. ubi_reader: collection of Python scripts for reading information about and extracting data from UBI and UBIFS images. URL. https://github.com/onekey-sec/ubi_reader.

Hunter, A., 2008. A brief introduction to the design of UBIFS: UBIFS whitepaper. URL. http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf.

Liu, S., Guan, X., Tong, D., Cheng, X., 2010. Analysis and comparison of NAND flash specific file systems. Chin. J. Electron. 19 (3), 403–408.

Micheloni, R., Crippa, L., Marelli, A., 2010. Inside NAND Flash Memories. Springer Netherlands, Dordrecht.

Micron Technology, Inc. jcooke, dberrett (TW), vschulthies, 2010. Nand Flash 101: an Introduction to Nand Flash and How to Design it into Your Next Product. URL. https://user.eng.umd.edu/~blj/CS-590.26/micron-tn2919.pdf.

Moonbeom Park, S.J., 2017. Iot hacking & forensic with 0-day. URL. https://troopers.de/downloads/troopers17/TR17_What_happened_to_your_home.pdf.

MTD, 2023. Memory technology device subsystem for linux. URL. http://www.linux-mtd.infradead.org/. (Accessed 30 September 2023).

Olivier, P., Boukhobza, J., Senn, E., 2012. On benchmarking embedded linux flash file systems. ACM SIGBED Rev. 9 (2), 43–47, 9. URL. https://arxiv.org/pdf/1208.6391.pdf.

OpenWrt, 2023. Openwrt project. URL. https://openwrt.org/docs/techref/filesystems#ubifs. (Accessed 30 September 2023).

UBI, 2020. Unsorted block images. URL. http://www.linux-mtd.infradead.org/doc/ubi.html. (Accessed 30 September 2023).

Woodhouse, David, 2001. Jffs : the journalling flash file system. URL. https://sourceware.org/jffs2/jffs2.pdf.

YAFFS, 2023. Yet another flash file system. URL. https://yaffs.net/. (Accessed 25 September 2023).

Zeifman, I., Rossi, B., 2023. OpenWrt OS: how it works, challenges, security concerns and alternatives. URL. https://sternumiot.com/iot-blog/openwrt-how-it-works-challenges-and-alternatives. (Accessed 30 September 2023).

Zimmermann, C., Spreitzenbarth, M., Schmitt, S., Freiling, F.C., 2012. Forensic analysis of YAFFS2. In: Suri, N., Waidner, M. (Eds.), SICHERHEIT 2012 – Sicherheit, Schutz und Zuverlässigkeit. Gesellschaft für Informatik e.V., Bonn, pp. 59–69.