Forensic Science International: Digital Investigation 40 (2022) 301338

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS 2022 EU - Selected Papers of the Ninth Annual DFRWS Europe Conference

A systematic approach to understanding MACB timestamps on Unix-like systems

Aurélien Thierry [a, *], Tilo Müller [b]

[a] Deutsche Telekom Security GmbH, Bonn, Germany
[b] Hof University of Applied Sciences, Germany

ARTICLE INFO

Article history:

Keywords:
Digital forensics
Timestamps
Unix
POSIX
Linux
OpenBSD
FreeBSD
macOS

ABSTRACT

File timestamps are used by forensics practitioners as a fundamental artifact. For example, the creation of user files can show traces of user activity, while system files, like configuration and log files, typically reveal when a program was run. Despite timestamps being ubiquitous, the understanding of their exact meaning is mostly overlooked in favor of fully-automated, correlation-based approaches. Existing work for practitioners aims at understanding Windows and is not directly applicable to Unix-like systems. In this paper, we review how each layer of the software stack (kernel, file system, libraries, application) influences MACB timestamps on Unix systems such as Linux, OpenBSD, FreeBSD and macOS. We examine how POSIX specifies the timestamp behavior and propose a framework for automatically profiling OS kernels, user mode libraries and applications, including compliance checks against POSIX. Our implementation covers four different operating systems, the GIO and Qt library, as well as several user mode applications and is released as open-source. Based on 187 compliance tests and automated profiling covering common file operations, we found multiple unexpected and non-compliant behaviors, both on common operations and in edge cases. Furthermore, we provide tables summarizing timestamp behavior aimed to be used by practitioners as a quick-reference.

1. Introduction

As files and file systems are central to most use of computing, their analysis is a cornerstone of digital forensics. A deep understanding of how files are stored on a file system, such as described by Carrier (2005), often allows to recover deleted files. File metadata, on the other hand, in particular timestamps, are key to understanding changes on a file system in order to reconstruct user actions and programmatic events. Even if a file's content is not available, for instance because it is encrypted, metadata can often be used to reconstruct user behavior, as demonstrated by Groβ et al. (2019). During investigations of security incidents, file timestamps can provide a useful view of files created or modified by unauthorized access, allowing investigators to efficiently focus on potentially malicious files, as explained by Buchholz and Spafford (2004).

That is, besides the understanding of file systems, practitioners can benefit from a deep understanding of timestamps behavior. Looking at timestamps from a research perspective, two questions arise:

1. What timestamps are modified by a given operation?
2. Given a set of timestamps, what happened on the machine on the application level?

The second question is what ultimately matters for event reconstruction, but answering it with certainty is in most cases not possible. Answers to the first question are useful for analysts when validating or refuting hypotheses related to the second question. In this paper, we look at the basics and provide an extensive explanation of timestamps on Unix-like systems *focusing on the first question*. Our contribution includes an automated test and profiling framework, which is available as open-source (os_timestamps), as well as a set of human-readable tables that summarize our results for practitioners. In order to cover applications on Linux, OpenBSD, FreeBSD and macOS alike, our work spans from POSIX specifications to the user mode, including kernel implementations, mount options and user libraries.

* Corresponding author.
E-mail addresses: aurelien.thierry@t-systems.com (A. Thierry), tilo.mueller@hof-university.de (T. Müller).

https://doi.org/10.1016/j.fsidi.2022.301338

1.1. Related work

Ridge et al. (2015) defined and published SibylFS (SibylFS), a formal specification for POSIX and real-world file systems, along with test suites and compliance checks covering Linux, BSD and macOS. It is aimed to be used for validation of file systems and operating systems in their development process. Giuliano (Giugliano, 2019) extended SibylFS to cover timestamp updates. SibylFS includes a specification and model for POSIX interfaces using higher-order logic and proof systems and thousands of automatically and hand-written tests. Tests run a sequence of POSIX interfaces and the output of each is then checked against the model.

Chow et al. (2007) investigated MAC timestamps on Windows XP for NTFS file systems. The SANS Institute researches (Knutson, 2016) and regularly publishes tables as posters (Windows Forensic Analysis) ("Windows Time Rules"), focusing on NTFS timestamps on Windows for standard operations (file copying, modification and creation). Other, similar tables can be found online, published by practitioners on blog posts. Tests are either manually conducted, or automatically run but not reproducible because the experiments' process (including source code) is not available. Galhuber and Luh (2021) reviewed timestamp updates on Windows 10, focusing on standard operations and timestamp forgery. They propose updates to the reference tables from SANS. Ho et al. (Ho et al., 1016) studied updates to timestamps on a cloud provider (OneDrive) when files are handled from a Windows machine with NTFS and from an Ubuntu computer with ext4. They published tables, useful for cloud forensics, detailing how operations modify the MACB timestamps.

Multiple approaches attempt to reconstruct events using automated profiling and comparison. Most of such approaches, as described by Soltani et al. (Soltani and Seno, 2017; Soltani et al., 2017), are based on signature systems or correlation engines. Signature systems consist in two steps: signatures are automatically generated from running applications on pre-defined scenarios, then real-world system states are matched against those signatures. Kälber et al. (2013) designed a generic application fingerprinting method for the NTFS file system, mostly based on timestamps metadata. Correlation-based systems compute distances between test scenarios and real-world states to find a fitting match.

Overall, SibilFS, which is a comprehensive approach to POSIX compliance is limited to POSIX specifications and does not cover automated profiling. Fully automated approaches have their value in automated workflows but do not help understand the underlying implementation details of timestamp updates. Furthermore, existing work into mapping timestamp updates mostly focuses on Windows environments and the NTFS file system, and are often not reproducible because implementation details are not made publicly available. They also only cover user-facing operations such as applications and standard operations from the operating system, for instance, copying a file, overlooking what happens deeper in the software stack.

1.2. Contributions

To the best of our knowledge there is no existing systematic, reproducible approach that analyzes MACB timestamps on Unix-like systems. Our contributions to the field are:

1. An open-source framework[1] able to test POSIX compliance and to profile the software stack on Linux, FreeBSD, OpenBSD and macOS for selected software libraries (Qt, GIO) and currently 15 applications.
2. An explanation of the timestamp behavior on those platforms. We highlight behavior of common operations, non POSIX-compliant behavior, as well as other unexpected behavior and possible bugs.
3. Visual tables based on the results of point 1 and 2, aimed to be used by practitioners.

For example, unexpected behaviors that could be treated as bugs occur on FreeBSD when reading a file, or on macOS when modifying timestamps. Also, BSD-based kernels do not update a symbolic link's last access timestamp when being read or followed, and FreeBSD does not update the last access timestamp of directories when performing directory listing.

[1] https://github.com/QuoSecGmbH/os_timestamps.

Table 1
Software stack and specifications.

	Examples	Specified?
Application	Vim, gedit, geany	No
POSIX Utilities	vi, cp, chmod, ls	POSIX
Standard C/C++ libraries	glibc, BSD libc, libstdc++	POSIX
Other Libraries (Middleware)	GTK, GIO, Qt, KIO	No
Operating System	Ubuntu, OpenBSD, FreeBSD, macOS	POSIX
Kernel	Linux, OpenBSD, FreeBSD, XNU	POSIX
File system	ext4, FFS1, UFS2, HFS+	No

2. Timestamps across the software stack

A software stack is a set of software subsystems that can operate without running additional software. When running an end-user application, the top of the stack is the application while the bottom of the stack is the OS kernel and drivers. We exclude what happens on the drive itself, both on the software level (firmware) and on the hardware level.

Timestamp updates are induced by the top of the stack but actual modifications happen at the very bottom when the kernel writes to the file system. The software stack relevant for timestamp updates (Table 1) contains file systems, kernels, configurations like mount options, software libraries, including the standard C and C++ libraries, as well as applications. The behavior of applications and libraries regarding timestamp updates is not specified. POSIX specifies utilities and interfaces like system calls and the standard C/C++ libraries. Regardless of other specifications, file systems do not precisely mandate usage of the metadata fields reserved for timestamps.

Because application use involves multiple chained components in the stack, typically at least the standard C library, the kernel and the file system, it is critical to understand how each of those components can alter timestamp updates. For instance understanding the standard C library yields results that can be generalized to many applications using it. Similarly, regardless of how many files applications access, the mount option used by the operating system may disable updates to the access time, making it useless for forensics purposes.

Fig. 1 shows examples of the execution of operations modifying timestamps across the software stack and the resulting timestamp updates. Users on a specific OS run applications that will induce timestamp updates. Arrows mostly represent functions, including POSIX interfaces, used between layers of the software stack. Between kernels and file systems they represent mount options. The examples and notations will be further explained in the paper.

3. POSIX

The Portable Operating System Interface (POSIX) (Standard for Informa, 2018) is a set of specifications for Operating Systems

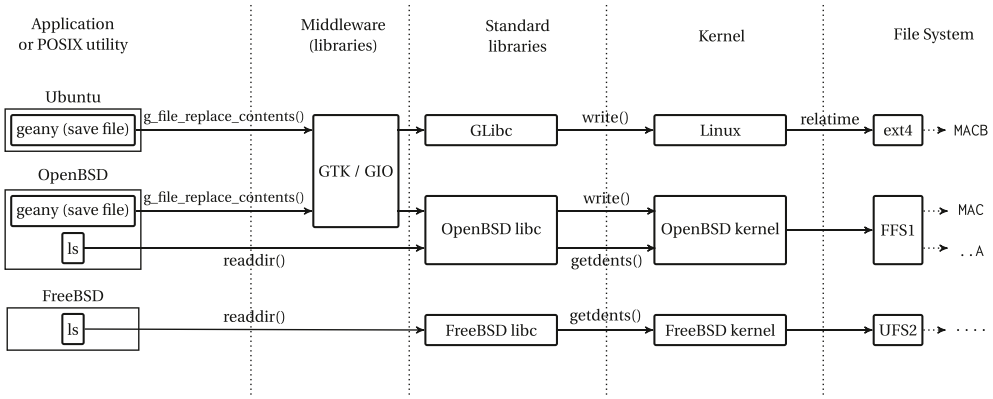


Fig. 1. Examples of application execution and updated timestamps across the software stack.

(*implementations*). It aims to promote application portability across the Unix world.

3.1. Scope

POSIX describes mandatory behavior for interfaces, shells and utilities, that are command-line programs like `cp` or `cat`. Interfaces refer to systems calls like `read()` and other functions including functions of the standard C library (libc) such as `fread()`.

Every timestamp update shall be described in the interfaces and utilities specification. Beside mandatory behavior (expressed with *shall*), optional behavior (expressed with *may* or *need not*) is also described, usually to lift requirements in edge cases. Furthermore some of the behavior is explicitly implementation-defined, leaving room for heterogeneous behavior across implementations.

Each of the evaluated operating systems implement the general design for timestamp updates (see Section 3.2) specified by POSIX. They do not claim nor aim to be fully compliant, except for macOS that is POSIX-certified[2] against UNIX 03 since version 10.5 (Mac OS X Leopard).

3.2. Timestamp principles

POSIX defines the MAC timestamps but does not specify any creation timestamp (B):

- M: Last data **m**odification timestamp.
- A: Last data **a**ccess timestamp.
- C: Last file status **c**hange timestamp.

The C timestamp is typically modified when the file's metadata (location, owner, access rights …) are changed and when the file is modified (along with the M timestamp). It does not reliably indicate when the file was created.

Timestamps shall be updated in two steps: they are first marked for update and then actually updated, meaning that the relevant metadata field on the file system is set to the current time. The time interval between the two steps is in general left to the implementation but timestamps marked for update shall be updated when the file ceases to be open (for instance following `fclose()`) or before execution of a POSIX function that directly reads or manipulates its timestamps (such as `stat()` and `futimens()`). Unless one of those occurs, the time interval between the two steps is left to the implementation.

[2] https://www.opengroup.org/openbrand/register/.

POSIX specifies precise behavior of timestamp-related interfaces and utilities. For instance, while `fopen()` called with a write mode ("w") shall mark MC for update, `fopen("r")` shall not have any impact on timestamps.

A typical file read using *libc* functions consists in the following sequence: `fopen("r")`, `fread()`, `fclose()`. In this case the access timestamp shall be marked for update when `fread()` is executed and the actual update shall happen at the latest when `fclose()` is executed.

A file overwrite consisting in `fopen("w")`, `fwrite()`, `fclose()` shall mark MC for update when `fopen()` is executed and again at some point between the execution of `fwrite()` and `fclose()`, leaving room for a buffering implementation. The actual update shall again happen at the latest when `fclose()` is executed.

3.3. C and timestomping

Every POSIX-specified operation updating the last data modification timestamp (M) also updates the last file status change timestamp (C).

Timestamp modification is possible using, for instance, `futimens()` to arbitrarily set the last access or modification timestamp. No interface allows to set the last status change timestamp arbitrarily. Setting A or M with `futimens()` always triggers an update to C, making it impossible within the POSIX specification to antedate or arbitrarily timestomp C. It is nevertheless possible with any implementation through direct file system modification using elevated privileges (typically root).

3.4. Automated compliance tests

This subsection describes how we designed our tests against POSIX. Testing compliance of actual MAC updates for a single operation against POSIX specification is described in Fig. 2. Files for which POSIX specifies updates are called watched files. We first prepare the environment and ensure watched files' timestamps that may be marked for update are actually updated (steps 1 and 2). The core steps (4, 5, 6) consist in storing the current time before the operation is run, running it, and storing the current time right after it is run. Finally the watched files' timestamps are retrieved and compared to the specification (steps 8 and 9). For instance, if a file's access timestamp shall be updated, we check that $t_1 \leq t_A \leq t_2$.

The tests rely on comparing system time (t_1 and t_2), typically gathered with POSIX functions such as `clock_gettime()` against timestamps from file systems (t_{MAC}) obtained through `stat()`. System time and file system timestamps can be based on different,

1. Prepare environment and files
2. For each watched file: `stat(file.path)`
3. `nanosleep(1.1s)`
4. t_1 = current_time()
5. **Run the tested operation**
6. t_2 = current_time()
7. `nanosleep(1.1s)`
8. For each watched file: t_{MAC} = `stat(file.path)`
9. Compare each t_{MAC}'s value related to t_1 and t_2 against the specification

Fig. 2. Steps to test POSIX compliance of an operation that shall update timestamps.

1. Prepare environment and files
2. For each watched file: `stat(file.path)`
3. `nanosleep(1.1s)`
4. t_1 = current_time()
5. **Run the tested operation**
6. For each watched file: t_{MAC} = `stat(file.path)`
8. t_2 = current_time()
9. Compare each t_{MAC}'s value related to t_1 and t_2 against the specification

Fig. 3. Steps to test POSIX compliance of an operation that shall mark timestamps for update.

coarse or truncated, values and such differences can lead a system timestamp t_1, fetched before t_A was updated an fetched, to be greater that t_A. POSIX does specify that, when updated, assigned timestamps shall be the greatest value supported by the file system (through truncation) that is not greater than the current time. It does not specify, neither for "current time" nor for file timestamps, which exact clock (real time or coarse) shall be used, nor the resolution, precision or the exact interfaces that can be used to fetch them. We looked into the implementations for timestamp updates and aimed to fetch the same clock for `current_time()`, this function is thus OS-dependent:

- Linux uses `clock_gettime(CLOCK_REALTIME_COARSE)` for timestamp updates. It has a resolution of 1 ns.[3] We used the same function for `current_time()`.
- OpenBSD uses an internal kernel function (`getnanotime()`) to update timestamps. It has a precision of 10 ms and resolution of 1 ns.[4] None of the clocks available in userland allows to fetch similar clocks. We thus used a generic solution consisting in opening, modifying, writing and closing an existing file and then fetching its M timestamps with `stat()`.
- We used the same generic solution for macOS.
- FreeBSD, by default, uses an internal kernel function (`microtime()`) to update timestamps, its resolution is 1 µs. Though none of the clocks available in userland uses the same function, the behavior is identical to using `clock_gettime(CLOCK_REALTIME_PRECISE)` and truncating its result to the microsecond resolution. We implemented this solution.

Delays (steps 3 and 7) are needed to make sure timestamps updates are not confused with environment preparation nor with the comparison steps. For instance a file timestamp marked for update (but not updated) by the operation (step 5) would be actually updated at step 8. It is thus crucial that observed current times (as previously described, including resolution and truncation issues) are different at steps 6 and 8. POSIX specifies timestamp resolutions shall not be coarser than 1 s. Additionally, resolution for system-wide clocks such as `CLOCK_REALTIME` shall not be coarser than 0.02s. We use `nanosleep(CLOCK_REALTIME)` to implement delays, asking for a 1.1s delay in order to take resolution constraints into account.

Testing operations that shall solely mark timestamps for updates (Fig. 3) can be achieved by using `stat()` before fetching t_2 (steps 6 and 8 are inverted) to force actual update of timestamps marked for update. The second delay (step 7) is rendered useless and removed because t_{MAC} is already stored and fetching t_2 will not modify it. Those tests will also match if timestamps are not only marked for update but actually updated. Neither POSIX nor studied implementations expose userland methods to observe inode flags used to mark timestamps for update, so there is no portable way to distinguish between "marked for update" and "actual update". This is not an issue in practice to test for compliance because POSIX allows implementations to immediately update timestamps that are marked for update.

The framework is written in C and covers checks against POSIX compliance as well as low-level profiling of operating systems (section 4) and libraries (section 5).

3.5. Results

We implemented a comprehensive test suite with 187 POSIX-compliance tests including 177 tests against mandatory behavior. We additionally implemented 92 tests against intuitively expected behavior or edge cases not specified by POSIX.

Table 2 shows the number of passed and failed tests for each system. Most failed tests are related to two special cases which are not interesting from a forensics standpoint. No considered implementations complies with updates to standard streams (`stdin`, `stdout`, `stderr`), whose timestamps shall be updated when reading or writing to them. Additionally, when `abort()` causes process termination after a file write, POSIX mandates that MC be marked for update, and ultimately updated when the file ceases to be opened. All considered implementations cancel the whole operation, losing the data that was only buffered and not already written, and skipping timestamp updates.

Linux, when using the `strictatime` mount option (see subsection 4.2), passes all the other mandatory tests for POSIX-compliance. OpenBSD, FreeBSD and macOS do not pass the `readlink()` tests because `readlink()` does not update the access timestamp of the read symbolic link. Moving a directory locally with `rename()` and `mv` is also not-compliant on OpenBSD and FreeBSD, as explained in subsection 4.5. Apart from this OpenBSD has a few irrelevant failed tests because some interfaces and utilities are not implemented or were deprecated, such as `gets()` and `unlink`.

Table 2
Passed and failed mandatory POSIX-compliance tests.

OS	Passed	Failed
Linux	163	14
OpenBSD	156	21
FreeBSD	161	16
macOS	142	35

[3] https://www.kernel.org/doc/html/latest/core-api/timekeeping.html#c.ktime_get_coarse_clocktai_ts64.
[4] https://github.com/openbsd/src/blob/3756ed2f07591c8bd3fd94b6d7b1a49fa7d6e042/sys/sys/time.h#L256.

Table 3
Utility `mkfifo` on macOS marks MAC for update but does not actually update them.

Time	Command	Result
12:00:08	`mkfifo fifo`	
12:00:08	`sleep 60`	
12:01:08	`stat fifo`	MAC == 12:00:34

FreeBSD and macOS are non-compliant on some irrelevant edge cases. For instance `chmod` used to set a file's mode to the same value it already has, inducing no change to the file, shall still update C but does not. Both implementations however have interesting non-compliance on some edge cases.

FreeBSD does not update a the access file of the directory when performing directory listing (`ls`). Additionally reading a file with FreeBSD's `read()` does not always trigger an update to A when the file was already recently read. The root cause of this behavior is being investigated, it is at this point not clear whether this is a bug or an undocumented feature.

On macOS, setting a file's A timestamp, for instance with `futimens`, fails unexpectedly. The behavior of utility `mkfifo` is also unique on macOS: it solely marks the created FIFO's MAC for update and does not actually update them. This is actually the mandatory POSIX behavior: interface `mkfifo()` shall mark MAC for update and utility `mkfifo` shall be equivalent to the interface. It technically does not break the requirement that files marked for update shall be updated when the file ceases to be open because `mkfifo()` does not use the `open()` interface. This results in the unexpected though arguably compliant behavior that a terminal user can create a FIFO which timestamps are later than when the command ran. Table 3 demonstrates running `mkfifo` then waiting 60 s before looking at the FIFO's timestamps, the timestamps were actually updated 26 s after the `mkfifo` command returned. Alternatively observing the timestamps immediately after `mkfifo` results in the timestamps being updated much quicker.

4. Operating systems and kernels

Linux, FreeBSD and macOS implement a fourth file timestamp to store the file creation time (B for **b**irth). The tested OpenBSD version supports file systems with this fourth timestamp but do not fill it (it always has the null value). All considered operating systems implement shared file system behavior (such as inodes and metadata) under an abstraction layer (VFS or *Virtual File System*) that can be superseded by code specific to each file system. We analyzed the behavior of each operating system on their default file systems for storage of user data.

4.1. File systems

On Linux the default file system for `/home` partitions is ext4. It allows, by default, storage for MACB timestamps with a resolution of 1 ns. If configured with smaller inodes (128 bytes instead of the default 256 bytes), MAC timestamps have resolution of 1 s and the B timestamp is omitted (Fairbanks, 2012).

FFS1 on OpenBSD allows storage for MAC timestamps with a resolution of 1 nanosecond but does not have a field for the creation timestamp (B). Although **FFS2** has a field for the creation timestamp, the kernel does not use it: it is always zero.

UFS2 on FreeBSD stores MACB timestamps and the OS can be configured to use one of the following resolutions: 1 s, **1 µs (default)**, 1 ns.

HFS+ (also called *Mac OS Extended*) on macOS stores MACB timestamps with a resolution of 1 s.

4.2. Mount options

By default Linux mounts file systems with the **relatime** option, updating the access timestamp (A) only if it was earlier or equal to M or C, or if it was at least 1 day old. Thus, by default, Linux skips most A updates and performs other updates normally. This behavior in itself makes Linux non POSIX-Compliant but is viewed as an acceptable trade-off for performance. Additional mount options on Linux are:

- **strictatime**: A updates are always performed
- **noatime**: A updates are never performed
- **nodiratime**: A updates are never performed for directories

When skipping A updates with **noatime** or **nodiratime** the A timestamp is only filled when the file created.

OpenBSD, by default, honors all MAC updates and supports a **noatime** option, which performs A updates only if M or C is also marked for update. This limits the number of updates to the access timestamps without making it totally useless.

FreeBSD and macOS, by default, honor all MACB and support a **noatime** option, which skips all A updates (like on Linux).

4.3. B and timestomping

Linux and OpenBSD do not provide a way to modify the birth timestamp, interfaces setting timestamps such as `futimens()` can only set M or A. These interfaces on FreeBSD and macOS additionally allow arbitrary modifications to the B timestamp. This possibility is directly available to users through the `touch` utility. The last file status change (C) timestamp cannot be arbitrary set and is updated when either M, A or B is set.

As a result, on FreeBSD and macOS, only the C timestamp is somewhat resistent to timestomping, while B is also resistent on Linux. On OpenBSD, the B timestamp has always the null value. Again, a user with elevated or *root* privileges is able to change any timestamp by directly modifying the file system.

4.4. Automated profiling

Tests against POSIX compliance provide an atomic view of timestamp updates. For forensics applications we are more interested in most common operations such as File Creation, Read, Write, Execute, Copy, and Delete.

Multiple implementations were written for each operation in order to automatically profile timestamp updates, using POSIX-defined interfaces and utilities. For instance File Read has two implementations:

- Interfaces: `fopen("r") + fread() + fclose()`
- Utility: `cat`

We created a virtual POSIX-compliant profile for each operation. We then ran profiling on Ubuntu Linux, OpenBSD, FreeBSD and macOS and compared them together. Automated profiling (Fig. 4) watches a selected number of files for changes and flags each timestamp with the updates it went through. Table 4 lists possible flags for each timestamp. Flags are simplified and ordered (MACB) into a single character. Profiling outputs a `.csv` file with the results. Identical results within a group are expected and merged. Fig. 5 shows the result of the File Read operation with the file (file) and its parent directory (dir/) being watched as well as the File Rename operation with the source file (src), destination file (dst) and parent directory (dir/) being watched.

Because files are watched based on their path, a renamed file is detected as having different timestamps as before the operation

1. Prepare environment and files
2. `nanosleep(1.1s)`
3. For each watched file: $t_{1,MACB}$ = `stat(file.path)`
4. t_1 = current_time()
5. **Run the tested operation**
6. For each watched file: $t_{2,MACB}$ = `stat(file.path)`
7. t_2 = current_time()
8. Compare each value of $t_{1,MACB}$ and $t_{2,MACB}$ to t_1 and t_2

Fig. 4. Steps for automated profiling.

```
PROFILE.OS.FILE.READ.dir/, ....
PROFILE.OS.FILE.READ.file, .A..

PROFILE.OS.FILE.RENAME.src,  !!!!
PROFILE.OS.FILE.RENAME.dst,  maCb
PROFILE.OS.FILE.RENAME.dir/, M.C.
```

Fig. 5. Sample CSV output of automated profiling on Ubuntu.

(`maCb`): M, A and B are inherited from the source file and C is updated. But the File Rename operation only moves the file, changing its name but keeping the same inode. The results need to be interpreted: M, A and B are actually not updated while C is updated. This interpretation is taken into accounts into our results, modifying the actual profile (`..C.`) for this operation.

The implementation is integrated into the C framework and supports C/C++ profiling tests.

4.5. Results

Table 5 and Table 6 summarize timestamp parameters and mount options across operating systems. Fig. 6 compiles what happens on the OS-level for the most common operations. Results are only divided into POSIX and implementations when they diverge. Non POSIX-compliant behavior is framed. The middle rows, for instance New File/Dir, focus on updates happening on files while the bottom rows describe updates of directories on operations involving themselves, for instance Dir Traversal, or operations to their children, such as when a file is being moved into it (see "Dir: Dir Moved into").

For instance reading a file will update its last access timestamp (A) while other timestamps will not be modified. On Linux if the default mount option (`relatime`) is used and the file's last access timestamp was not older than one day, this timestamp update will actually be skipped and the operation will update no timestamp.

Most timestamp updates specified by POSIX are straightforward. For instance a file newly created gets updated MAC and its modified parent directory sees its MC timestamps being updated. Moving files and directories locally is only partially specified. `rename()` shall update MC of the parent directories but nothing is specified for the file itself, implying that its timestamps shall not be modified. POSIX however states that, in some implementations, `mv` updates C. Implementations indeed differ on how to handle local file and directory move, macOS being the only implementation not updating C for this operation.

Additionally, moving a directory into another directory (Local Dir Move) is a tricky operation that differs from the simple Rename operation (changing its name without changing its parent directory). Firstly, implementations need to check that the source directory is not an ascendant of the target directory, otherwise parts of the file system would be orphaned and unreachable. On OpenBSD this results in all checked directories to get an updated A (see **Dir**: Dir Moved into (Local)). Secondly, the directory's entry pointing to its parent directory (`..`) is modified, updating MC on OpenBSD and FreeBSD.

Moving files and directories across volumes (Volume File Move) is an interesting operation because `rename()` does not work across volumes. Our setup consists in mounting a second partition using the same file system as the main tested partition, for instance a second ext4 partition on Linux. POSIX mandates that `mv` implements a duplication of the source file into the destination file system. The involved interfaces or utilities are not specified, macOS for instance directly uses the `cp` utility then attempts to restore the properties of the original file. The destination file shall inherit from the MA timestamps of the source file and, as with files moved locally, C may be updated. Linux attributes the destination an updated birth timestamp while on FreeBSD and macOS the destination file inherits the B timestamp from the source file.

The default mount option (`relatime`) on Linux makes it non-compliant. With the `strictatime` mount option, all profiled operations are compliant.

OpenBSD, FreeBSD and macOS have non-compliant operations. Reading of following symbolic link on these BSD-based OS do not update the last access timestamp of the link. Additionally FreeBSD does not update the last access timestamp of a directory when performing directory listing.

Table 4
Flags attributed to each {M,A,C,B} timestamp after profiling.

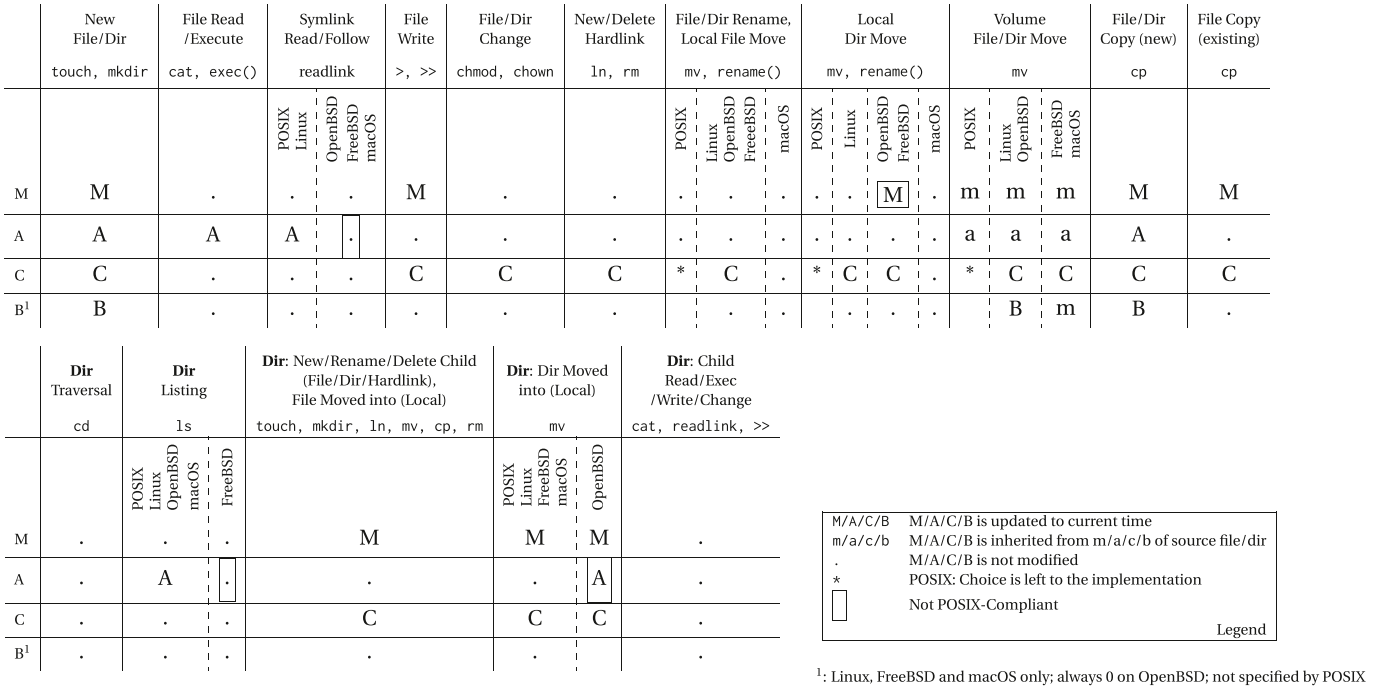
Flag Name	Single character	Description
`ERROR`	!	`stat()` failed, mostly because the file no longer exists
`ZERO`	0	Timestamp has now the null value
`UPDATE`	M/A/C/B	Timestamp was updated
`EQ`	.	Timestamp was not modified
`SAMEAS_W0_\{M,A,C,B\}`	m/a/c/b	Timestamp is now the same as the original \{M,A,C,B\} value of the first watched file (typically the *source* file)
`EARLIER`	−	New timestamp is now earlier as its original value
`LATER`	+	New timestamp is now later as its original value

Table 5
Timestamp resolution and support for the **b**irth timestamp.

	POSIX	Linux (ext4)	OpenBSD (FFS1)	FreeBSD (UFS2)	macOS (HFS+)
Timestamp Resolution (default)	≤1s	1ns	1ns	1µs	1s
Birth timestamp (B)	No	Yes	No	Yes	Yes

Table 6
Mount options related to timestamp updates. Default options are in **bold**; the relatime, nodiratime and noatime options are not POSIX-Compliant.

Mount option	Linux (ext4)	OpenBSD (FFS1)	FreeBSD (UFS2)	macOS (HFS+)
(default)	MCB updates are all performed	MAC updates are all performed, B is always 0	MACB updates are all performed	
relatime	**(default)** A updates are performed if A was earlier or equal to M or C, or at least 1 day old	(No)		
strictatime	A updates are always performed	(No)		
nodiratime	A updates are never performed for directories	(No)		
noatime	A updates are never performed	A updates are performed only if M or C is also marked for update	A updates are never performed	



[1]: Linux, FreeBSD and macOS only; always 0 on OpenBSD; not specified by POSIX

Fig. 6. OS-level MACB updates on POSIX, Linux, FreeBSD, OpenBSD, macOS.

5. Middleware

Applications make extensive use of software libraries (Middleware) providing primitives to interact with file systems. We already described and tested POSIX-compliance of the Standard C and C++ libraries. The Standard C/C++ Libraries are the only middleware specified by POSIX and, to the best of our knowledge, timestamp updates are not specified for the other software libraries described in this paper. Most Linux distributions, including Ubuntu, use the GNU C Library (glibc),[5] OpenBSD, FreeBSD and macOS each have their own libc implementation.

The investigated operating systems run complex graphical applications which are out of POSIX's scope, for instance applications belonging to popular desktop environments like GNOME and KDE. Investigating middleware allows to better understand and generalize timestamp updates. For instance two GNOME applications using the same primitives to handle file modifications will exhibit the same behavior regarding timestamp updates.

We focused on GTK for GNOME and Qt that are popular to implement graphical applications on Unix-like systems. Both are cross-platform, supporting Linux, Windows and macOS. They are also available on OpenBSD and FreeBSD. GLib is the low-level library developed for GTK+ and GNOME applications. GIO (*Gnome Input/Output*) is the C library providing GLib general purpose I/O that can be used instead of POSIX primitives, its implementation is built upon system calls and the standard C library. The Qt framework, written in C++, is extensively used by KDE applications but can also be used by other applications. Additionally to Qt, KDE has its own library implementing many file management functions, KIO.

5.1. Automated profiling

Middleware functions were tested on the same operations like File Read and Write, using the same steps used for profiling operating systems (Fig. 4). The tests are implemented within the same C/C++ framework.

5.2. Results

We implemented tests for GIO and Qt, but not yet for KIO. We ran the tests on our Ubuntu setup and compared the results to baseline OS behavior. No unexpected behavior were found when

[5] https://www.gnu.org/software/libc/.

testing the Qt functions. GIO exhibits unexpected behavior for File Copy and implements two alternatives for File Write.

5.2.1. File Copy

When copying a file into a new or an existing file using the GIO function `g_file_copy`, the destination file exhibits almost the `mACB` update pattern, preserving the access timestamp instead of updating it like `cp` does by default. Modification timestamp is inherited from the source file while the access, birth and status change timestamps are updated. Interestingly modification and access timestamps are truncated to the microsecond resolution. The truncation is an old GIO bug, first reported in2010,[6] triaged as "low priority" and being in 2021 actively worked on.

The POSIX-compliance behavior for file copy (`MACB`) differs from GIO (`mACB`), this is not a compliance issue because GIO does not aim for POSIX compliance and POSIX does not specify Middleware nor Application behavior. In combination with the truncated MA timestamps, this difference helps differentiate files copied using `cp` with default options or using a file manager implemented with GIO such as Nautilus.

5.2.2. File Write

GIO has a counterpart to the standard `fopen("rw")`, `g_file_open_readwrite`. Using this function and writing to the stream returned will update MC as expected. An alternative is to use the function `g_file_replace`, which returns an output stream for overwriting the file. Documentation[7] states it may first write to a new file before renaming it to overwrite the original file. When writing to a file using `g_file_replace`, the destination file has updated MACB timestamps, validating that it is actually a new file.

6. Applications

Forensics examiners are ultimately interested in how applications modify timestamps. We focused on file editors and file managers.

6.1. Automated profiling

Applications were tested using the same steps than for profiling operating systems (Fig. 4). In order to cover graphical applications, the implementation is a separate framework, written in *python3*, and uses *pyautogui* to simulate user input through keystrokes.

6.2. Results

Automated tests cover 15 popular file editors on Linux including Vim, Emacs, gedit, Geany, Sublime Text, Visual Studio Code and Kate. Our tests cover typical actions such as File Read and Write, along with edge cases such as reading an empty file. Except on some edge cases, including reading an empty file, editors only differ in how they write to files.

6.2.1. File modify

When saving to disk a modified file, editors either directly write to the modified file (`M.C.`) or first write to a new file before replacing the original file with the new one, typically with `rename()`, as previously described for GIO and resulting in MACB being updated. Atom and Kate also access the file when modifying it in-place (`MAC.`). Editors using GIO exhibit the `MACB` (Table 7) pattern because they use the `g_file_replace` function, but they could alternatively implement file write with low level IO functions from GIO and obtain the other behavior. Similarly, Qt and POSIX can be used to implement any of the behavior.

Table 7
Profiling text editors.

Editor	IO Middleware	Read	Modify
Vim −clean	POSIX	A	MACB
Emacs	POSIX	A	MACB
Code::Blocks	POSIX	A[a]	MACB
gedit	GTK	A	MACB
Bluefish	GIO	A	MACB
Geany	GIO	A	MACB
TeXstudio	Qt	A	MACB
JED	S-Lang	A	MACB
Kate	KTextEditor (KIO)	A	MAC
Atom	Electron	A	MAC
Vim nowritebackup	POSIX	A	MC
Nano	POSIX	A	MC
Leafpad	POSIX	A	MC
Visual Studio Code	Electron	A	MC
Notepadqq	Qt	A	MC
Sublime Text	(Unknown)	A	MC

[a] Code:Blocks, when reading an empty file, does not modify any timestamp.

6.2.2. Reading an empty file

Opening an empty file with the editor Code:Blocks does not update its access timestamps although the same operation with other editors modifies A. Most editors read files by chunks (for instance Vim uses chunks of 8192 bytes), resulting in the sequence `open(r)`, `read(count=8192)`. According to POSIX, `read()` when count is not zero shall update A, even if the number of bytes actually read is zero. Codeblocks however first uses `stat()` to determine the file size and then reads the whole file with `read(count=filesize)`, resulting in `read(count=0)` when reading an empty file. When `count` is zero, POSIX states that `read()` may return early and skip timestamp updates, explaining the different result with Code:Blocks.

6.3. Manual tests on file managers

We performed manual tests on selected file managers. Nautilus exhibits the GIO bug happening where copying a file with `g_file_copy`: modified and access timestamps are truncated to the microsecond resolution. This is also a known bug,[8] first filed in 2010. Two further file managers, Dolphin and pcmanfm, do not exhibit this exact behavior but have similar issues where nanosecond timestamps are truncated.

7. Discussion

7.1. Considerations

Our project profiles timestamps dynamically while the system is running. Noise from other programs or from the OS, such as a file indexer or an IDE (Integrated Development Environment), can alter the results and users need to carefully interpret the results to filter out the noise. Similarly the tested systems need to be configured to not skip timestamp updates. Linux in particular is tested with the non-default mount option `strictatime`, the default `relatime` options acting like a filter for timestamp updates at the bottom of the software stack.

[6] https://gitlab.gnome.org/GNOME/glib/-/issues/369.
[7] https://docs.gtk.org/gio/method.File.replace.html.
[8] ++https://bugs.launchpad.net/ubuntu/+source/nautilus/+bug/642.596.

Table 8
Tested operating systems.

System	Version	File System	Machine
Linux	Ubuntu 20.04.3 LTS, Linux 5.10.0	ext4 mounted with `strictatime`	Lenovo P1
OpenBSD	6.8	FFS1	VirtualBox
FreeBSD	13.0-RELEASE-p4	UFS2	VirtualBox
macOS	10.13.6 (Supported until 2020)	HFS+	Mac mini

Testing the same systems on various versions and new updates will be needed to keep tables up-to-date. However, apart from fixing kernel bugs and major changes such as new file systems, we expect little change to working IO implementations. Indeed applications using GIO for IO are unlikely to change because it works mostly as intended. Middleware and application bugs are not always quickly fixed and are reasonable candidates for forensics artifacts to identify application use, as attests the 11-year-old GIO bug affecting Nautilus.

7.2. Related work

SibilFS' work on POSIX-compliance for timestamps (Giugliano, 2019) was published after we began implementation of our tests on POSIX-compliance. We cover some cases that are not implemented in SibilFS, such as the `readlink()` interface which is not compliant on tested BSD systems. We believe however that SibilFS is a solid, more formal approach to POSIX-compliance, and that it would obtain similar results to ours once missing tests cases are implemented. On the practical side, contrary to SibilFS which uses OCaml, our approach of POSIX-compliance only uses system calls and the standard C/C++ library that are likely to already be present on tested systems. Pre-compiled binaries could be used to avoid installation of developer tools (GCC, CMake) and limit our footprints on the tested system. Furthermore we used POSIX specifications as a baseline to find interesting differences and combined the results with automated profiling of the software stack, which goes beyond the scope of SibilFS.

7.3. Future work

The list of tested systems, versions and configuration is given in Table 8. Future work could cover FFS2 on OpenBSD, newer macOS versions with the `APFS` file system, and add automated testing for KIO and file managers. We also want to cover Android and iOS, which are based on Linux and BSD, respectively.

8. Summary

We designed and implemented a framework to test POSIX-compliance and profile timestamp updates across Unix-like systems. We used the framework to extensively describe how Unix-like systems such as Linux, OpenBSD, FreeBSD and macOS update MACB timestamps, from a user action in an application to the kernel level and file system.

We found multiple unexpected and non-compliant behavior, both on common operations and in edge cases and provide tables that can be used by practitioners as a quick reference. Some unexpected behavior of FreeBSD when reading a file and of macOS when setting timestamps should be further analyzed as potential bugs. Operating systems based on a BSD kernel, including macOS, do not update a symbolic link's last access timestamp when being read or followed, and FreeBSD does not update the last access timestamp of directories when performing directory listing. Middleware and file editors implementing file write by writing first to a new file before moving it over the original file exhibit a different timestamp than those modifying the file in-place, and both implementations are common. A bug in the GIO library, used by the Nautilus file manager, reduces timestamp resolution to the microsecond and could be used to identify Nautilus use.

Acknowledgments

We thank FAU (*Friedrich-Alexander-Universität*) students Berenger Temgoua Dibanda and Niclas Pohl for their contribution to the framework as part of their Bachelor thesis and Professor Felix Freiling for his guidance on the project and his reviews.

References

Buchholz, F., Spafford, E., 2004. On the role of file system metadata in digital forensics. Digit. Invest. 1, 298–309. https://doi.org/10.1016/j.diin.2004.10.002.
Carrier, B., 2005. File System Forensic Analysis. Addison-Wesley Professional.
Chow, K., Law, F.Y., Kwan, M.Y., Lai, P.K., 2007. The Rules of time on NTFS file system. In: Second International Workshop on Systematic Approaches to Digital Forensic Engineering. SADFE'07, pp. 71–85. https://doi.org/10.1109/SADFE.2007.22.
Fairbanks, K.D., 2012. An analysis of ext4 for digital forensics. Digit. Invest. 9, S118–S130. https://doi.org/10.1016/j.diin.2012.05.010 the Proceedings of the Twelfth Annual DFRWS Conference. https://www.sciencedirect.com/science/article/pii/S1742287612000357.
Galhuber, M., Luh, R., 2021. Time for truth: Forensic analysis of ntfs timestamps. In: The 16th International Conference on Availability, Reliability and Security, ARES 2021. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3465481.3470016 doi.org/10.1145/3465481.3470016.
Giugliano, A., 2019. Towards Verified File Systems. Ph.D. thesis. URL. https://leicester.figshare.com/articles/thesis/Towards_verified_file_systems/10220159.
Groß, T., Ahmadova, M., Müller, T., 2019. Analyzing android's file-based encryption: information leakage through unencrypted metadata. In: Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES '19. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3339252.3340340 doi.org/10.1145/3339252.3340340.
S. Ho, D. Kao, W.-Y. Wu, Following the breadcrumbs: timestamp pattern identification for cloud forensics, Digit. Invest. 24. doi:10.1016/j.diin.2017.12.001.
Kälber, S., Dewald, A., Freiling, F.C., 2013. Forensic application-fingerprinting based on file system metadata. In: 2013 Seventh International Conference on IT Security Incident Management and IT Forensics, pp. 98–112. https://doi.org/10.1109/IMF.2013.20.
Knutson, T., 2016. Filesystem Timestamps: What Makes Them Tick?, Tech. Rep. SANS Institute.
os_timestamps: Profile timestamp updates on your Unix-like OS. https://github.com/QuoSecGmbH/os_timestamps.
Ridge, T., Sheets, D., Tuerk, T., Giugliano, A., Madhavapeddy, A., Sewell, P., 2015. SibylFS: Formal Specification and Oracle-Based Testing for POSIX and Real-World File Systems. Association for Computing Machinery, New York, NY, USA, pp. 38–53. https://doi.org/10.1145/2815400.2815411. URL.
SibylFS. https://sibylfs.github.io/.
Soltani, S., Hosseini Seno, S.A., Sadoghi Yazdi, H., 2017. Event Reconstruction using Temporal Pattern of File System Modification. IET Inf. Secur. 13. https://doi.org/10.1049/iet-ifs.2018.5209.
Soltani, S., Seno, S.A.H., 2017. A survey on digital evidence collection and analysis. In: 2017 7th International Conference on Computer and Knowledge Engineering. ICCKE2, pp. 247–253. https://doi.org/10.1109/ICCKE.2017.8167885.
IEEE Standard for Information Technology - Portable Operating System Interface (POSIX®), Tech. Rep. Base Specifications, 2018. The Open Group Standard. Issue 7.
Windows Forensic Analysis (Poster), Tech. Rep., SANS Institute.