Digital Investigation 11 (2014) 224–233

Contents lists available at ScienceDirect

Digital Investigation

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

The Linux FAT32 allocator and file creation order reconstruction

Wicher Minnaard

Netherlands Forensic Institute, Department of Digital Technology & Biometry, PO Box 24044, 2490 AA The Hague, The Netherlands

ARTICLE INFO

Article history:
Received 20 February 2014
Received in revised form 28 May 2014
Accepted 26 June 2014
Available online 5 August 2014

Keywords:
Linux kernel
TomTom
File systems
File allocation
The Sleuth Kit (TSK)
FAT16
FAT32
Event order reconstruction
Antedating

ABSTRACT

The allocation algorithm of the Linux FAT32 file system driver positions files on disk in such a way that their relative positions reveal information on the order in which these files have been created. This provides an opportunity to enrich information from (carved) file fragments with time information, even when such file fragments lack the file system metadata in which time-related information is usually to be found.

Through source code analysis and experiments the behaviour of the Linux FAT allocator is examined. How an understanding of this allocator can be applied in practice is demonstrated with a case study involving a TomTom GPS car navigation device. In this case, time information played a crucial role. Large amounts of location records could be carved from this device's flash storage, yielding insight into the locations the device has visited—yet the carved records themselves offered no information on *when* the device had been at the locations. Still, bounds on the records' time of creation could be inferred when making use of filesystem timestamps related to neighbouring on-disk positions.

Finally, we perform experiments which contrast the Linux behaviour with that of Windows 7. We show that the latter differs subtly, breaking the strong relation between creation order and position.

Introduction

Commonly, user space applications writing to files in file systems cannot influence where the data ends up on the block device that underlies the file system. It is the file system allocator in a kernel's file system driver that locates the storage spaces in which the data is written. Allocation information (or a way of tampering with it) is usually unavailable to user space. This article shows how a careful interpretation of allocation patterns can supply a forensic investigator with an interesting side channel of event order information.

Willassen, 2008 combines the order of NTFS MFT entries and some form of sequence numbers—*generation markers*—that occur in the entries, to detect the antedating of data. While his area of research—NTFS, and *allocated* files—is quite different from the situation which we will turn to in our case study, it is an example of how information on allocator behaviour can be used to reconstruct the order of events.

The relation between file data location and file creation order on FAT32 file systems has been documented in an MSc thesis by Tse, 2011. He describes the phenomenon, formulates statistical correlations between file location and metadata items, and classifies systems according to their behaviour. The underlying mechanisms are, however, not investigated.

To the best of our efforts, we could find no scientific literature succinctly addressing FAT32 allocation patterns in a context other than observation of a manifested relation between creation order and offset. For our purposes we

E-mail addresses: wicher@holmes.nl, wicher@nontrivialpursuit.org.

http://dx.doi.org/10.1016/j.diin.2014.06.008
1742-2876/

required a deeper understanding of the causes of this relation. The section "Case study: TomTom Go" documents a case, involving a TomTom car navigation device, which we have been able to solve through an understanding of the Linux FAT32 allocator.

Allocators are discussed in the section "Allocator models", and the Linux FAT32 allocator implementation source code is discussed in the "Linux FAT allocator source code analysis" section.

The "Experiments" section details the experimental verification of the allocator model derived from the kernel source code, which combined with an analysis of the way in which the TomTom uses the filesystem offers a full explanation of the correlation between creation order and on-disk location.

The "Analyzing TomTom application system calls" section shows how the model can be applied in an investigation. The analysis method is shown in detail, and can be applied to other cases with little change.

The section "The Windows FAT32 allocator" contrasts the Linux behaviour with that of Windows 7. Demonstrated are the reasons why the Windows 7 FAT32 driver does not allow inferring time order from disk order to the same extent as the Linux FAT32 driver does.

The "Discussion" section addresses possible pitfalls in the application of the methods, and shows how the methods could be applied to other cases. The "Conclusion" section summarizes the findings and underlines their expected relevance for forensic investigations involving time-related traces on embedded devices running Linux.

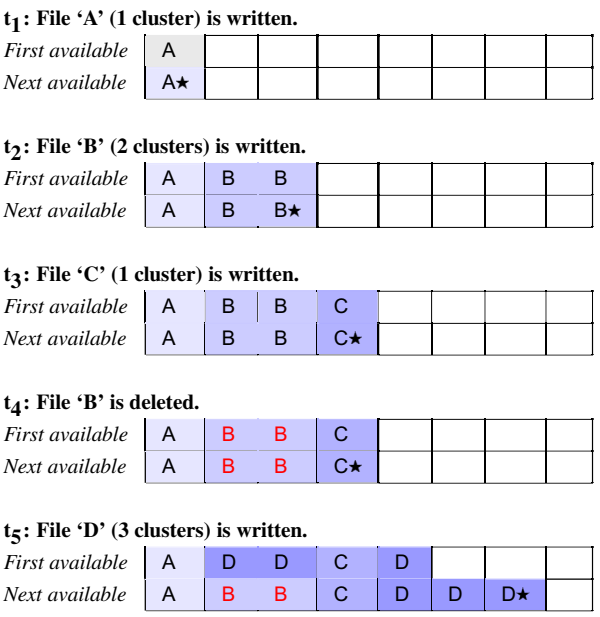

Fig. 1. Resultant file layouts—*first available* versus *next available* allocator. For the next available allocator, the star symbol (★) signifies its reference to the cluster allocated last. A red-coloured file name signifies that the file is deleted, but that the data previously associated with it still exists.

Allocator models

In *File System Forensic Analysis* (Carrier, 2005, pp.179–181), Carrier illustrates the workings of three types of allocator algorithms; *first available*, *next available* and *best fit*. He mentions that in his tests, the Windows 98 and XP operating systems appear to use a *next available* algorithm for FAT file systems (Carrier, 2005, p.224).

The FAT filesystem derives its name from the file allocation table structure (FAT), which is essentially an array of annotated implicit references[1] to the *clusters* (groups of consecutive disk sectors) within the file system. For each file, the FAT contains a linked list of clusters—these lists are called *FAT chains*. For details of the FAT, see Carrier's work.

With a *next available* algorithm, whenever space needs to be allocated to a file, an empty cluster is found by linearly scanning the FAT table for cluster references marked as free. The use of a reference to a position from where to restart a search is what makes a FAT allocator algorithm of the stateful *next available* type. If it would not use such a reference, the allocator algorithm would degenerate to the *first available* type. Fig. 1 illustrates the difference between the two types.

The diagrams show that storage space freed by deleting files is greedily reused by a *first available* allocator. Whether a file system volume handled by such an allocator will show any relation between file offset and creation order will strongly depend on usage patterns; file deletion will quickly confuse the relation. A *next available* allocator, on the other hand, will not reuse space freed by deleting files until it wraps around the last cluster of the volume. With such an allocator type the relation is therefore much more robust with respect to file deletion.

In a FAT32 file system, a reference to the cluster allocated last is made persistent by way of the FSI_Nxt_Free field of the FSINFO structure. In Microsoft's specification (Microsoft Corporation, 2000) the field is described as follows:

> This is a hint for the FAT driver. It indicates the cluster number at which the driver should start looking for free clusters. Because a FAT32 FAT is large, it can be rather time consuming if there are a lot of allocated clusters at the start of the FAT and the driver starts looking for a free cluster starting at cluster 2. Typically this value is set to the last cluster number allocated by the driver. If the value is 0xFFFFFFFF, then there is no hint and the driver should start looking at cluster 2. Any other value can be used, but should be checked first to make sure it is a valid cluster number for the volume.

The FSI_Nxt_Free hint is not present in FAT16 file systems; a FAT16 file system does not contain the FSINFO structure. For FAT16, the upper limit on the size of the FAT is just 128 KiB (for 2^{16} entries of 2 bytes each). FAT32, on the other hand, can address 2^{28} clusters and uses 4 bytes to address each, thus its FAT could

[1] A position in the array corresponds to the disk offset of the cluster it describes, but there is no direct reference to a cluster number.

theoretically be as large as 1GiB—indeed a non-trivial amount of data to scan through, even if it would need to be done only at mount time. Hence making the hint persistent makes sense for FAT32. Considering that a file system may be remounted many times over its lifetime, this apparently innocuous optimization can have serious consequences for the relation that we are interested in. Fig. 2 shows how a remount can affect a *next available* allocator that uses a persistent hint versus one that does not.

These diagrams illustrate that allocator state which persists across remounts is essential for maintaining the correlation between file offset and file creation order.

Linux FAT allocator source code analysis

As Carrier mentions, the OS is free to use any allocator algorithm—as long as the allocator can make use of available disk space and does not corrupt the file system (by double-booking sectors, for instance), the approach it takes does not matter from a functional perspective. Performancewise, though, the allocator matters; for instance, a *first available* algorithm will in many workloads result in a filesystem with heavily fragmented files.

For the TomTom case study, it is the Linux FAT implementation that we are interested in. Its code is open source and thus easily inspected. An annotation shows how simple a file system kernel driver allocator implementation can be.

Listing 1 exhibits the fat_alloc_clusters() function, in which the building blocks of a *next available* allocator are recognizable. The state of the allocator (the hint) is restored at line 23, and stored at line 44. FAT entries are traversed linearly (line 62) by calling the function shown in Listing 2.

While traversing the FAT in this manner (do…while, lines 33, 62), the allocation state of entries is evaluated (line 34). An entry marked as free will become allocated (lines 38–40). Searches continue until the requested amount of clusters is satisfied (line 50). If a search reaches the end of the volume, it is wrapped around (line 25 and 26); the search will then be restarted at the beginning of the data area of the volume (cluster 2).

It is worth mentioning that after having been initialized (in inode.c/fat_fill_super(), not shown), the hint variable is never modified outside of this allocator function. The hint value will thus only ever increase (modulo wraparounds), and so will the starting point of a search for available clusters.

The revision log of the Linux kernel shows that these functions have changed little since Linux source code moved out of BitKeeper and into Git (linux/fs/fat/fatent.c revision log; Revision-annotated linux/fs/fat/fatent.c). This indicates that kernels since 2005 (and possibly earlier) have been exhibiting the same behaviour as the current kernel, and that millions of devices are running this code.

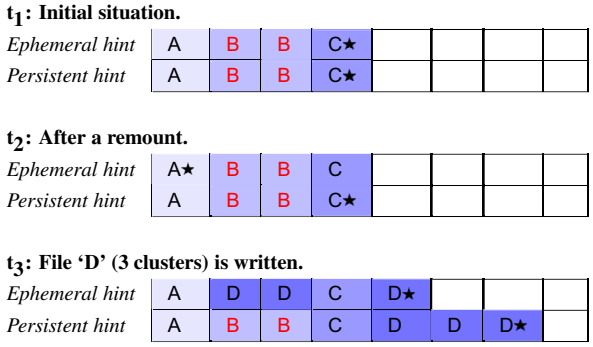

Fig. 2. Resultant file layouts—*next available* allocator, with and without persistent hint. As in the previous diagram, the star symbol (★) signifies a reference to the cluster allocated last. A red-coloured file name signifies that the file is deleted, but that the data previously associated with it still exists.

Case study: TomTom Go

TomTom forensics

The subject device of our research is a TomTom Go 930 equipped with version 8.014 of the TomTom navigation software. It is an ARM embedded system running a Linux kernel (compiled for the 'ARMv4T' subarchitecture), with a fairly conventional user space environment consisting of the proprietary and closed source core navigation software, a BusyBox userland, the GNU libc6 C library, and some other open source libraries (TomTom International BV, 2014). Due to reverse engineering efforts by open source enthusiasts, users can quite easily run custom programs and even custom kernels on the device (OpenTom.org, 2013)—permitting development of forensic RAM acquisition software as described in Roeloffs & van Eijk, 2010.

The Go 930 is not a novel model. Neither is forensic analysis of the workings and contents of this type of TomTom navigation devices. In 2008, Nutter provided a comprehensive overview of the Go model range, detailing functionality, identifying types of data stored, and explaining the implications of retrieved information for casework (Nutter, 2008). Crucially, she details how to interpret the binary format of the '.cfg' file. The .cfg file is commonly found as 'MapSettings.cfg' in the file system directory of the map currently in use on the device. It can hold a list of entered addresses, a home location set by the user, a list of recently used addresses, a route origin, a route destination, and the last recorded position ('fix'). The purpose of this file appears to be to store variables required for route planning using the device's touch screen interface. It does not contain timestamps or sequence numbers, and it only stores one fix at a time—not a sequence of them. In other words, *it is not a log file*. Route log files *can* be created by opting in to share data with the TomTom company, in which case the device creates files called 'triplogs' in the 'statdata' directory. These files seem to be encrypted, and they are uploaded to the TomTom company when the user interfaces with the device using the 'TomTom Home' application on a PC.

Listing 1. The Linux FAT cluster allocator from linux-3.13.2/fs/fat/fatent.c. © 2004 OGAWA Hirofumi, released under GPL v2.

```
int fat_alloc_clusters(struct inode *inode, int *cluster, int nr_cluster)
{
    struct super_block *sb = inode->i_sb;
    struct msdos_sb_info *sbi = MSDOS_SB(sb);
    struct fatent_operations *ops = sbi->fatent_ops;
    struct fat_entry fatent, prev_ent;
    struct buffer_head *bhs[MAX_BUF_PER_PAGE];
    int i, count, err, nr_bhs, idx_clus;

    BUG_ON(nr_cluster > (MAX_BUF_PER_PAGE / 2));    /* fixed limit */

    lock_fat(sbi);
    if (sbi->free_clusters != -1 && sbi->free_clus_valid &&
        sbi->free_clusters < nr_cluster) {
        unlock_fat(sbi);
        return -ENOSPC;
    }

    err = nr_bhs = idx_clus = 0;
    count = FAT_START_ENT;
    fatent_init(&prev_ent);
    fatent_init(&fatent);
    fatent_set_entry(&fatent, sbi->prev_free + 1);
    while (count < sbi->max_cluster) {
        if (fatent.entry >= sbi->max_cluster)
            fatent.entry = FAT_START_ENT;
        fatent_set_entry(&fatent, fatent.entry);
        err = fat_ent_read_block(sb, &fatent);
        if (err)
            goto out;

        /* Find the free entries in a block */
        do {
            if (ops->ent_get(&fatent) == FAT_ENT_FREE) {
                int entry = fatent.entry;

                /* make the cluster chain */
                ops->ent_put(&fatent, FAT_ENT_EOF);
                if (prev_ent.nr_bhs)
                    ops->ent_put(&prev_ent, entry);

                fat_collect_bhs(bhs, &nr_bhs, &fatent);

                sbi->prev_free = entry;
                if (sbi->free_clusters != -1)
                    sbi->free_clusters--;

                cluster[idx_clus] = entry;
                idx_clus++;
                if (idx_clus == nr_cluster)
                    goto out;

                /*
                 * fat_collect_bhs() gets ref-count of bhs,
                 * so we can still use the prev_ent.
                 */
                prev_ent = fatent;
            }
            count++;
            if (count == sbi->max_cluster)
                break;
        } while (fat_ent_next(sbi, &fatent));
    }

    /* Couldn't allocate the free entries */
    sbi->free_clusters = 0;
    sbi->free_clus_valid = 1;
    err = -ENOSPC;

out:
    unlock_fat(sbi);
    mark_fsinfo_dirty(sb);
    fatent_brelse(&fatent);
    if (!err) {
        if (inode_needs_sync(inode))
            err = fat_sync_bhs(bhs, nr_bhs);
        if (!err)
            err = fat_mirror_bhs(sb, bhs, nr_bhs);
    }
    for (i = 0; i < nr_bhs; i++)
        brelse(bhs[i]);

    if (err && idx_clus)
        fat_free_clusters(inode, cluster[0]);

    return err;
}
```

Listing 2. Companion function to the allocator as in linux-3.13.2/fs/fat/fatent.c. This function is called from a loop in fat_alloc_clusters() to traverse the FAT. The FAT16 variant is very similar. © OGAWA Hirofumi. License: GPL v2.

```
static int fat32_ent_next(struct fat_entry *fatent)
{
    const struct buffer_head *bh = fatent->bhs[0];
    fatent->entry++;
    if (fatent->u.ent32_p < (__le32 *)(bh->b_data + (bh->b_size - 4))) {
        fatent->u.ent32_p++;
        return 1;
    }
    fatent->u.ent32_p = NULL;
    return 0;
}
```

The problem at hand

Carving for .cfg files, and the decoding of their contents, can be performed in an automated way, for instance using the TomTology forensic software (Forensic Navigation Ltd., 2014). In the reports this software generates, information decoded from records in carved .cfg files is reported ordered by their offset into the storage medium. While the company developing TomTology is making no claims as to the meaning of this ordering, examiners may easily mistake it for a temporal ordering—especially since the ordering often aligns well with corroborating location information for which a temporal ordering *is* available, such as call detail records.

However, as argued, the location information is not at all comparable to a well-defined log, and we cannot treat it as such. The location records are carved from unallocated space, and the ordering of the previous versions of the .cfg files is never meant to convey any meaning. Rather, it would simply be just a circumstancial effect of particular system interactions that the records' positional ordering relates to a temporal ordering. Two approaches of investigating the correlation are available.

One approach would be to treat the TomTom device as a black box which as one of its inputs takes location information from GPS, and outputs a result in the form of an on-disk order of location records. The rules of the transformation of input to output may be approximated by inductive reasoning; we can perform a black box experiment once, twice, or N times, controlling for as many possible inputs to the transformation rule system as we can think of. The drawback of this approach is that it is very hard to cover corner cases, or to even know which cases out of the total spectrum have been covered in the experiments (Vickers, 2014). To improve the admissibility of TomTom location record ordering in court of law, it would be of help to uncover the rules themselves—since we are dealing with automated systems, these rules are ought to be found somewhere. Once the rules are known (or thought to be), we can devise specific experiments that test our understanding of the rules.

Concretely, while we understand from examining the Linux FAT allocator source code why more recently created .cfg files would be stored at ever greater offsets into the storage medium, we do not yet understand why there are so many incarnations of this file. Suppose file contents are written using operating system calls as in Listing 3.

Listing 3. System calls for writing and overwriting file data (demonstrated in the Python interactive shell).

```
>>> import os
>>> fh = os.open('somefile', (os.O_RDWR | os.O_CREAT))
>>> os.write(fh, b'yesteryear')
11
>>> os.fsync(fh)
>>> os.lseek(fh,0,0)
0
>>> os.write(fh, b'zoological')
11
>>> os.fsync(fh)
>>> os.close(fh)
```

Now, the expectation is that between the first and second call to fsync(), the string 'yesteryear', as it appeared on disk, will have been overwritten with the string 'zoological'. The TomTom application is not updating the contents of .cfg files in this way—for if it would be, we would not be able to find so many previous instances of the .cfg file; older data would have been overwritten with newer data.

One could suppose that the carved .cfg files are the remnants of temporary files, but then one would also expect to be able to find at least some (deleted) directory entries associated with such temporary instances. We find none of these. There is only one directory entry that can be associated with .cfg file data, and that is the entry for the currently allocated .cfg file.

To answer the question, we need to examine the behaviour of the closed source software running on the TomTom device with respect to its handling of the .cfg files. The section "Analysing TomTom application system calls" details this research.

Experiments

Analysing TomTom application system calls

Turning towards the question of why are there so many incarnations of the .cfg file, it makes sense to investigate the boundary between the user space TomTom application and the kernel. On Linux, the interface for using the file system is provided through system calls such as open(), read(), write() and seek().

'strace' is a system call tracer that can log such interactions (Various contributors, 2014). Using the cross compiler toolchain provided by TomTom, version 4.7 of strace was compiled to run on the TomTom device. Minor patching of the source code was necessary to accommodate for the version of the GNU libc6 C library deployed on the TomTom (Minnaard, 2014). The target process for strace was the 'ttn' process; this is the closed source component that is found running once the system is booted. To attach strace to the ttn process, a shell script was launched through the 'TTConsole' program, which is a terminal emulator that allows a user to open and command a shell using touch screen input (Hoffmann, 2014). The shell script reads as follows:

```
#!/bin/sh
for pid in `pidof ttn`; do
    /mnt/sdcard/bin/strace -f -ff -y -q -ttt \
    -o /mnt/sdcard/ttnstrace.$pid -p $pid &
done
```

Important options are **-f**, to also trace any processes spawned by the traced processes, and **-y**, to print the file paths associated with file descriptors. After connecting strace to the ttn processes in this way, the TTConsole application was exited and a route destination was set using the user interface. Upon arrival at the destination 1.5 h later, the strace log files were copied off the TomTom device.

Interim results. Since the system call log statements include file paths, filtering out calls that have anything to do with the MapSettings.cfg file was straightforward. Only one of three 'ttn' processes that strace was attached to had made any such system calls. Of the 1823177 logged calls, 54 concern the MapSettings.cfg file. Listing 7 in Appendix A details these. Two patterns emerge. For reading from the .cfg file, the process performs, in succession:

open(), stat(), seek(), read(), close()

For writing to the file, the process calls:

open(), stat(), write(), fsync(), close()

Note that the file writing system call pattern is very similar to the example shown in Listing 3, which served to illustrate how writing to a file would result in the *overwriting* of previous contents. The similarity is remarkable; the file writing system calls performed by the ttn application should, given the occurrence of the multitude of .cfg file incarnations, result in just the opposite: the *appending* of data. The essential difference between the two sets of calls is in the flags passed in the open() call.

Replaying system calls

This experiment aims to establish whether we the log-like phenomenon can be reproduced by replaying the observed file writing system calls. The call pattern was emulated using the Python OS interface, using the following script:

Listing 4. tomtomulate.py: Emulating ttn file writing system calls.

```
#!/usr/bin/env python3
import sys,os
FLAGS = (os.O_WRONLY | os.O_CREAT | os.O_TRUNC)
for word in iter(sys.stdin.readline, ''):
  fd = os.open(sys.argv[1], FLAGS)
  os.write(fd, word.encode())
  os.fsync(fd)
  os.close(fd)
```

For each line read from standard input, this script opens the file specified as the first argument using the same flags as observed in the system call traces log. According to the kernel manual pages (Linux open() system call manpage), the meaning of these flags is as follows:

O_WRONLY	Access mode, requests opening the file write-only.
O_CREAT	If the file does not exist it will be created.
O_TRUNC	If the file already exists and is a regular file and the open mode allows writing (i.e., is O_WRONLY or O_RDWR) it will be truncated to length 0.

The script then writes the line into the (truncated) file. The modifications are flushed to storage, after which the file is closed. The FAT32 file system on which the script operated was created and mounted on a Linux system (kernel 3.13.2) using the following shell commands:

```
dd if=/dev/zero of=/dev/shm/ttfat.img bs=1M count=1024
mkfs.vfat -F 32 -n ttfat /dev/shm/ttfat.img
sudo mount -o loop,uid=${UID} /dev/shm/ttfat.img /mnt/tmp0/
```

Using 10-letter words from a word list in a file '10lw.in', the emulation script 'tomtomulate.py' (Listing 4) was run with these shell commands:

```
tomtomulate.py /mnt/tmp0/simfile < 10lw.in
sudo umount /mnt/tmp0
sync
```

The words were then retrieved from the file system image using the 'srch_strings' program from the Sleuth Kit (TSK) suite (version 4.1.2, Various contributors, 2013).

```
srch_strings -n 10 -t d /dev/shm/ttfat.img \
 | egrep -o '[0-9]+ [a-z]{10}$' \
 | fgrep -f 10lw.in \
 | tee 10lw.pos \
 | fgrep -o -f 10lw.in \
 > 10lw.out
```

There are now three lists:

10lw.in	The original list of 4893 10-letter words.
10lw.out	This file will contain any and all occurrences of the above words on the file system image, in the same order as in which they are found, including any duplicates.
10lw.pos	The above list, prefixed with the byte offset into the file system volume.

Simulation results. Analysis of the three generated lists shows the following:

1. 10lw.in and 10lw.out are identical. This means that the *complete* content history of the 'simfile' file—our stand-in for the MapSettings.cfg file—has been preserved. Moreover, the temporal ordering of writes *completely* corresponds with the on-disk order of content.
2. 10lw.pos features a distance between successive offsets of, invariably, 4096 bytes—a distance corresponding with the 8-sector cluster size of the volume.

The O_TRUNC flag is crucial in attaining these results. Upon adapting the script to drop this flag from the open() system call, only the very last content written to the 'simfile' file was retrievable from the file system image. For this allocator, O_TRUNC in an open() call on an *existing* file can be seen to make the system essentially behave as if a *new* file is being allocated, except that instead of creating a new directory entry for the new file, it updates the existing file's entry to refer to the new file's cluster chain entry point in the FAT. In the simulation, this can be witnessed by tracking the 'simfile' directory entry using TSK's 'fiwalk' and 'istat' tools. The file's inode number never changes, while the position on disk (shown in the form of allocated disk sectors) is advanced with each write(). So is the persistent FSI_Nxt_Free hint, which TSK's 'fsstat' shows as 'Next Free Sector (FS Info)'.

For a FAT32 system, the results were observed to be invariant under remounts. With FAT16, they were not. Consistent with theory discussed in the "Allocator models" section, the FAT16 file system only preserved the history of content created since it was last mounted, overwriting history from the previous 'session'.

The source code of the vendor-supplied kernel on the TomTom device (TomTom International BV, 2014) was examined for changes to the allocator function, and a variant of the allocation experiment was run on the device itself to verify that the same results can be obtained with its particular configuration.

Time bounding carved .cfg files

The experiments have demonstrated the causes for the correlation between the storage location of carved .cfg files and the order in which they are created. We can take this knowledge one step further. What if the TomTom device, over the course of its lifetime, not only writes to .cfg files, but also creates other files on the FAT32 file system? These files would have creation timestamp metadata associated with them, and we would find these files' data interspersed among the carved .cfg files' data. Since we now know how storage order and storage location relate, we can use the creation timestamps of such allocated files to formulate lower and upper bounds on the creation time of the carved location data demarcated by these allocated files. Two sources of information are used to accomplish this:

1. Start- and end-offsets of records in carved .cfg files. Taking the end offset of one .cfg file and combining it with the start offset of the subsequently located .cfg file yields a list of inter–.cfg–file *gaps* in which allocated files could reside.
2. A mapping of storage location to associated allocated files, so that given a volume offset, the metadata of the file (if any) to which the data at this offset belongs can be retrieved.

Combining these two produces the time bounds.

Generating the 'gap list'

A convenient way to generate this list is by querying the SQLite database (.gpsc) which the TomTology software uses as a "case file". The 'tomtomlocations' table contains the record offset information, in a somewhat peculiar and denormalized manner. For instance, the 'CFGID' field contains the sequence number assigned to carved .cfg files—except when it is 0, in which case the associated data comes from the *allocated* ("live") version of MapSettings.cfg, or when it is −1, in which case the associated data are "orphan" records that could not be tied to any intact .cfg file

at all. Such idiosyncrasies make for a peculiar query.[2] Listing 5 shows a SQL query that generates the gap list:

Listing 5. Creating an inter-.cfg-file gap list using the TomTology database.

```
SELECT cfgid-1 AS cfga, cfgid AS cfgb,
  (SELECT MAX(offset) FROM tomtomlocations
   WHERE cfgid=ttla.cfgid-1 AND offset > 0)
   AS gapstart,
MIN(offset) AS gapend FROM tomtomlocations ttla
WHERE cfgid > 1 AND offset > 0 GROUP BY cfgid;
```

This query generates rows such as shown in Table 1:

Table 1
Example gap list fragment.

cfga	cfgb	gapstart	gapend
1474	1475	183029858	183132175
1475	1476	183141024	183242767
1480	1481	183287202	183389509

Finding file times associated with data in gaps

TSK's 'tsk_loaddb' generates a SQLite database of file system information, from which a mapping of file data offsets to file metadata can be retrieved. For instance, to query for file metadata based on the first two gaps shown in Table 1, one could use the 'gapstart' and 'gapend' values in a query such as shown in Listing 6.

Listing 6. Finding file metadata by file storage location.

```
SELECT byte_start, name,
 DATETIME(crtime,"unixepoch","localtime") AS crtime
FROM tsk_files WHERE
  (byte_start > 183029858 AND byte_start+byte_len < 183132175)
 OR
  (byte_start > 183141024 AND byte_start+byte_len < 183242767)
AND has_layout IS NULL;
```

This selects unfragmented files of which the data is allocated between the records of carved .cfg files with numbers 1474, 1475 and 1476, respectively. A query result is shown in Table 2.

This query result signifies that .cfg file #1474 was written *before* the 'triplog-2014-02-14.dat' file was created, which is at 2014-02-14 17:22:11, while .cfg file #1475 has been written *after* that time, but *before* 'triplog-2014-02-15.dat', which is at 2014-02-15 13:14:15. Combining information for multiple gaps yields many more time bounds for location fields from .cfg files, the most important of which is the 'fix' field—so pinning the device to specific places, within specific time frames.

The Windows FAT32 allocator

The support for FAT32 in the popular Windows family of operating systems has quite possibly contributed to the prevalence of this filesystem. As there is no public reference implementation, it is interesting to compare the *de facto* standard FAT32 driver with the Linux implementation to see if we can find corresponding regularities.

Table 2
Example gap metadata.

byte_start	name	crtime
183037952	triplog-2014-02-14.dat	2014-02-14 17:22:11
183144448	triplog-2014-02-15.dat	2014-02-15 13:14:15

Setup

A Windows 7 instance[3] was launched under the Linux KVM hypervisor. Through the VirtIO mechanism (Jones, 2010), this virtual machine had access to a block device backed by a 128 MiB file, filled with zeroes, on the Linux host's filesystem. The block device was initialized through the diskmgmt.mmc management console snap-in and formatted to FAT32 with a 1 KiB cluster size. The Cygwin environment (Cygwin) was employed to provide a POSIX API so that the methods shown in the section "Replaying system calls" could be applied unmodified from within the Windows environment. To inspect the results of actions taken within the Windows VM, the FAT32 filesystem was unmounted by taking it offline with diskmgmt.mmc. The file underlying the block device was then analysed, again using the same methods as shown in the section "Replaying system calls".

Tests and findings

Allocation contiguity with O_TRUNC

On Linux, we had observed that passing the O_TRUNC flag to open() caused the file system to allocate clusters as if a new file is being created. Running the 'tomtomulate.py' script on Windows shows that it behaves differently; only the very last string written to the file is retrievable from the file system image.

Allocation contiguity when allocating multiple files

In this test, words from a word list were written to files, one word per file. The files were created using open(), write(), fsync() and close(). Naturally, all of these words then appeared on the file system image—and just as on Linux, they were found at 1-cluster intervals and the on-disk order was in complete agreement with their creation order. TSK's 'fsstat' showed the FSI_Nxt_Free hint to point to the cluster[4] *after* the cluster allocated last—instead of to the cluster allocated *last*, as on Linux.

Allocation contiguity after deleting files

From the disk image created in the previous test, the file that was created as the 720th (out of 4893) was deleted. Through this action, the value of the FSI_Nxt_Free hint had been decreased to point to the cluster occupied by the file just deleted. Deleting file 719 decreased the value by one more. Deleting file 721 or file 803 or any other file occupying a cluster *beyond* the hint did not affect the hint. This

[2] Indeed, the .gpsc database does not seem to be intended to be queried by the end user, and the investigator who uses it directly is solely responsible for the interpretation of its structure.

[3] 64-bit 'Ultimate' edition, version 6.1, build 7601.

[4] Nota bene: fsstat expresses the FSI_Nxt_Free hint in units of *sectors* rather than units of *clusters*.

makes the Windows FAT32 allocator reuse freed space more greedily than the Linux allocator.

Write-size dependent allocation contiguity

A write() of a 3 KiB buffer to a newly created file was issued on the filesystem in its state after the previous test. This write would fit neatly in the three clusters deallocated through the previous file deletion of file 719, 720, and 721. And this is indeed where the file data ends up.

However, on issuing a write() of a 4 KiB buffer in the same situation, the data ends up past the three empty clusters left by deletion of files 719–721, and also past the single cluster previously occupied by file 803. Conversely, dividing up the 4 KiB write into two subsequent write calls results in occupation of clusters previously allocated to files 719 and 720, but not in occupation of the clusters previously allocated to files 721 and 803. Instead, the second write ends up in pristine space, after all other data.

It appears that the allocator tries to avoid fragmented writes, skipping small spaces where the data would not fit anyway.[5] Replaying this scenario on Linux *does* result in full reoccupation of the deallocated space.

Impact of the differences on event order reconstruction

Whether the behaviour varies between Windows versions and types of storage media is not investigated, but in these tests, the Windows FAT32 allocator has not shown itself to be a pure *next-available* type allocator. The decisions it takes are much tied into the history of the filesystem; file deletions and the sizes of past writes both influence its state, making it much harder to reason backwards from the current filesystem layout. Furthermore, it does not feature the log-like behaviour when using the O_TRUNC flag—had the TomTom from our case study been running on some Windows device with this allocator, we would not be seeing the hundreds of previous incarnations of the .cfg file that we rely on for past location data.

Discussion

Though the mechanism by which temporal ordering can be reconstructed seems straightforward, there are many pitfalls. In the particular example case of the TomTom, we know that both .cfg file data and triplog files are created by the TomTom device itself, using its local clock of known behaviour, and using an allocator of known behaviour. These favourable circumstances will not always be met for arbitrary FAT32 file systems. A memory card exchanged between a digital camera and a PC will be affected by both, and it will therefore always be necessary to determine the mechanisms by which data ended up on the volume as closely as possible in order to make sensible claims about the order in which events have taken place. But there are some simple consistency checks we can avail ourselves of. First off, we can inspect the value of the persistent FSI_Nxt_Free hint. If any data is stored beyond the location pointed to by this hint, we can assume that the allocator has wrapped around the end of the volume at least once—greatly complicating event order reconstruction. To the same end, we can inspect file slack space; data found there can reveal if (and possibly when) the allocator had previously allotted that storage space.

The simple techniques demonstrated can be applied to many current and future devices. For instance, Samsung has just launched a high-end camera line running its 'Tizen' Linux platform (Lukas, 2014). Quite probably, it will support FAT32 for its SD card media. Unless Samsung has been making modifications to the allocator (which should be detectable[6]), the methods demonstrated in the TomTom case study are just as applicable to a camera device such as the NX300M. Consider an antedating scenario, in which the camera's clock is set to a certain time to make a picture appear to have been taken at an earlier time than it really has. The picture, taken with the sham clock setting, is downloaded through the camera's WiFi interface to later be used as fabricated evidence, and is subsequently deleted from the camera. Afterwards the camera clock is reverted to the correct time, and some further pictures are taken. In a court case, the validity of the picture (brought forward as evidence) is put into question. By locating the picture in the SD card's unallocated space, an investigator can make use of the timestamps associated with neighbouring sectors to reveal traces that not only indicate tampering, but can also be used to approximate the true time at which the falsification was fabricated.

Conclusion

The Linux FAT32 file system driver, due to its strict 'next available' type allocator with state preserving property, may be a special case in allowing event order reconstruction. The examination of a Windows 7 allocator shows that even for a simple filesystem such as FAT, allocators are not necessarily as simple and predictable as this Linux driver's allocator.

In the case study of the TomTom navigation device, we have uncovered the causes of peculiar allocation behaviour through analysis of allocator and application behaviour. A relation between spatial and temporal ordering of location records has been thoroughly proven to exist—since the causes of the phenomenon are known, conclusions drawn upon location record ordering in certain TomTom devices are stronger than when they would be based on a mere statistical correlation. The studied behaviour even allows for the time bounding of carved location data, yielding highly valuable information on *when* a device has been at a particular place. Furthermore, we know more about the circumstances in which the relation

[5] This begs the question: what happens when the file system is only half full, but layed out in such a way that single free clusters are interleaved with single allocated clusters? A write larger than one cluster would not fit anywhere unfragmented. Will the allocator then keep seeking, giving up only when it has wrapped around the FAT and is back in the same spot? Or will it time out somewhat earlier?

[6] The terms of the GNU Public License require the release of modifications to GPL-licensed code if that code (and its modifications) is compiled and redistributed. Therefore, vendor-supplied FAT32 filesystem driver code can be examined for changes that could affect allocator behaviour. In the case of the NX300M camera the source code is available at http://opensource.samsung.com/.

will *not* hold up—the corner cases—and expect to be able to detect some of these.

FAT32 is compatible with many consumer-facing operating systems; it is the greatest common file system denominator between Linux, Windows and MacOSX. For many use cases its lack of access controls is a feature in itself; devices can be shared without conflicts. Even though it is ageing, it is still very popular for portable media such as generic USB flash drives, MP3 players, and SD cards for photo cameras. Due to the presence of Android and other Linux derivatives in the embedded device space the Linux FAT32 driver is one of the (if not *the*) most widely deployed filesystem drivers for removable media storage. More Linux derivatives, such as Sailfish OS (Sailfish, 2014) and Firefox OS (Mozilla, 2014), are entering the mobile device field, making the findings of significant value.

Given its age—measurable in decades—a discussion of the extent to which event order heuristics can be based on different FAT32 allocator behaviours seems due. As it is an interesting side channel of event order information, we hope to see discussions of allocator mechanisms in future forensic literature on file systems.

Acknowledgements

I would like to thank the open source community in general for creating all the excellent tools that I have made use of, and for offering a wealth of easily accessible information. My thanks also go out to my colleagues, especially M. Kaart, R. van der Knijff and S. Laraghy, who were happy to engage in discussions of the topic.

Appendix A. 'ttn' system call log

Listing 7: System calls associated with the MapSettings.cfg file, performed by the 'ttn' process. The call tracer output has been altered to fit the page width; the full

```
1392050953.498317 stat64("MapSettings.cfg", {st_mode=S_IFREG|0755, st_size=5447, ...}) = 0
1392050953.503775 access("MapSettings.cfg", R_OK) = 0
1392050953.515388 open("MapSettings.cfg", O_RDONLY) = 48
1392050953.528403 fstat64(48<MapSettings.cfg>, {st_mode=S_IFREG|0755, st_size=5447, ...}) = 0
1392050953.532051 _llseek(48<MapSettings.cfg>, 0, [0], SEEK_CUR) = 0
1392050953.533257 fstat64(48<MapSettings.cfg>, {st_mode=S_IFREG|0755, st_size=5447, ...}) = 0
1392050953.535251 _llseek(48<MapSettings.cfg>, 4096, [4096], SEEK_SET) = 0
1392050953.536630 read(48<MapSettings.cfg>, "#DATA#"..., 1351) = 1351
1392050953.539129 _llseek(48<MapSettings.cfg>, 0, [0], SEEK_SET) = 0
1392050953.540362 read(48<MapSettings.cfg>, "#DATA#"..., 4096) = 4096
1392050953.543063 read(48<MapSettings.cfg>, "#DATA#"..., 4096) = 1351
1392050953.544575 close(48<MapSettings.cfg>) = 0
1392050953.546768 open("MapSettings.cfg", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 48
1392050953.549323 fstat64(48<MapSettings.cfg>, {st_mode=S_IFREG|0755, st_size=0, ...}) = 0
1392050953.552824 write(48<MapSettings.cfg>, "#DATA#"..., 4096) = 4096
1392050953.554697 write(48<MapSettings.cfg>, "#DATA#"..., 2342) = 2342
1392050953.556555 fsync(48<MapSettings.cfg>) = 0
1392050954.682734 close(48<MapSettings.cfg>) = 0
1392051053.287808 stat64("MapSettings.cfg", {st_mode=S_IFREG|0755, st_size=6438, ...}) = 0
1392051053.290278 access("MapSettings.cfg", R_OK) = 0
1392051053.301007 open("MapSettings.cfg", O_RDONLY) = 48
1392051053.302611 fstat64(48<MapSettings.cfg>, {st_mode=S_IFREG|0755, st_size=6438, ...}) = 0
1392051053.306448 _llseek(48<MapSettings.cfg>, 0, [0], SEEK_CUR) = 0
1392051053.307632 fstat64(48<MapSettings.cfg>, {st_mode=S_IFREG|0755, st_size=6438, ...}) = 0
1392051053.309610 _llseek(48<MapSettings.cfg>, 4096, [4096], SEEK_SET) = 0
1392051053.311820 read(48<MapSettings.cfg>, "#DATA#"..., 2342) = 2342
1392051053.313451 _llseek(48<MapSettings.cfg>, 0, [0], SEEK_SET) = 0
1392051053.314696 read(48<MapSettings.cfg>, "#DATA#"..., 4096) = 4096
1392051053.316226 read(48<MapSettings.cfg>, "#DATA#"..., 4096) = 2342
1392051053.317811 close(48<MapSettings.cfg>) = 0
1392051053.319870 open("MapSettings.cfg", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 48
1392051053.321713 fstat64(48<MapSettings.cfg>, {st_mode=S_IFREG|0755, st_size=0, ...}) = 0
1392051053.325206 write(48<MapSettings.cfg>, "#DATA#"..., 4096) = 4096
1392051053.327533 write(48<MapSettings.cfg>, "#DATA#"..., 2346) = 2346
1392051053.329225 fsync(48</MapSettings.cfg>) = 0
1392051054.182433 close(48</MapSettings.cfg>) = 0
1392053912.694084 stat64("/MapSettings.cfg", {st_mode=S_IFREG|0755, st_size=6442, ...}) = 0
1392053912.696612 access("/MapSettings.cfg", R_OK) = 0
1392053912.707609 open("/MapSettings.cfg", O_RDONLY) = 8
1392053912.709145 fstat64(8</MapSettings.cfg>, {st_mode=S_IFREG|0755, st_size=6442, ...}) = 0
1392053912.712632 _llseek(8</MapSettings.cfg>, 0, [0], SEEK_CUR) = 0
1392053912.713812 fstat64(8</MapSettings.cfg>, {st_mode=S_IFREG|0755, st_size=6442, ...}) = 0
1392053912.715882 _llseek(8</MapSettings.cfg>, 4096, [4096], SEEK_SET) = 0
1392053912.717056 read(8</MapSettings.cfg>, "#DATA#"..., 2346) = 2346
1392053912.719558 _llseek(8</MapSettings.cfg>, 0, [0], SEEK_SET) = 0
1392053912.720811 read(8</MapSettings.cfg>, "#DATA#"..., 4096) = 4096
1392053912.723380 read(8</MapSettings.cfg>, "#DATA#"..., 4096) = 2346
1392053912.724934 close(8</MapSettings.cfg>) = 0
1392053912.726942 open("/MapSettings.cfg", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 8
1392053912.729503 fstat64(8</MapSettings.cfg>, {st_mode=S_IFREG|0755, st_size=0, ...}) = 0
1392053912.733031 write(8</MapSettings.cfg>, "#DATA#"..., 4096) = 4096
1392053912.734911 write(8</MapSettings.cfg>, "#DATA#"..., 2254) = 2254
1392053912.736675 fsync(8</MapSettings.cfg>) = 0
1392053913.590372 close(8</MapSettings.cfg>) = 0
```

path to the MapSettings.cfg file is removed, and the data in reads and writes is replaced with "#DATA#".

Listing 7.

References

Carrier B. File system forensic analysis. Addison-Wesley Boston; 2005.

Cygwin. URL http://cygwin.com/.

Forensic Navigation Ltd. TomTology2. Visited Feb 20th; 2014. URL, http://forensicnavigation.com/index.php/products/tomtology2.

Hoffmann M. TomTom console. Visited Feb 20th; 2014. URL, http://sourceforge.net/apps/mediawiki/x11-basic/index.php?title=Tom-Tom_Console.

Jones TM. Virtio: an I/O virtualization framework for Linux; January 2010. URL, http://www.ibm.com/developerworks/library/l-virtio/index.html.

Linux open() system call manpage. URL http://man7.org/linux/man-pages/man2/open.2.html.

linux/fs/fat/fatent.c revision log. URL https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/log/fs/fat/fatent.c.

Lukas G. Hacking the Samsung NX300 'Smart' Camera. Visited May 13th; 2014. URL, http://op-co.de/blog/posts/hacking_the_nx300/.

Microsoft Corporation. Microsoft extensible firmware initiative FAT32 file system specification; December 2000. URL, http://msdn.microsoft.com/en-us/library/windows/hardware/gg463080.aspx.

Minnaard W. Patch for compiling strace against the TomTom SDK; February 2014. URL, http://nontrivialpursuit.org/ttt/strace-4.7-TT.patch.

Nutter B. Pinpointing TomTom location records: a forensic analysis. Digit Investig 2008;5(1):10–8. http://dx.doi.org/10.1016/j.diin.2008.06.003.

OpenTom.org, OpenTom home page. Archived by The Internet Archive at June 6th; 2013. URL, http://web.archive.org/web/20130606020011/http://www.opentom.org/Main_Page.

Revision-annotated linux/fs/fat/fatent.c. URL https://github.com/torvalds/linux/blame/455c6fdbd219161bd09b1165f11699d6d73de11c/fs/fat/fatent.c.

Roeloffs M, van Eijk O. Forensic acquisition and analysis of the Random Access Memory of TomTom GPS navigation systems. Digit Investig 2010;6(3):179–88. http://dx.doi.org/10.1016/j.diin.2010.02.005.

TomTom International BV. README for the source code of the parts of TomTom GO software that fall under open source licenses, TomTom GO version 8.x* software. Visited Feb 20th; 2014. URL, http://www.tomtom.com/page.php?Page=GPL80.

Tse WHK. Forensic analysis using FAT32 cluster allocation patterns. Master's thesis. The University of Hong Kong; April 2011. URL, http://hdl.handle.net/10722/143258.

Various contributors, The Sleuth Kit (TSK) & Autopsy: open source digital forensics tools. Visited Feb 20th, 2014, URL http://sleuthkit.org/.

Various contributors. Strace. Visited Feb 20th; 2014. URL, http://sourceforge.net/projects/strace/.

Vickers J. The problem of induction. In: Zalta EN, editor. The Stanford Encyclopedia of Philosophy. Spring 2014 Edition 2014.

Willassen SY. Finding evidence of antedating in digital investigations. In: 2012 Seventh International Conference on Availability, Reliability and Security; 2008. pp. 26–32. http://dx.doi.org/10.1109/ARES.2008.149.