Advanced forensic recovery of deleted file data in F2FS

By:
Junghoon Oh, Hyunuk Hwang

From the proceedings of
The Digital Forensic Research Conference
DFRWS APAC 2025
Nov 10-12, 2025

Forensic Science International: Digital Investigation 54 (2025) 301976

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS APAC 2025 - Selected Papers from the 5th Annual Digital Forensics Research Conference APAC

Advanced forensic recovery of deleted file data in F2FS

Junghoon Oh *, Hyunuk Hwang

The Affiliated Institute of ETRI, 1559 Yuseong-daero, Yuseong-Gu, Daejeon, South Korea

ARTICLE INFO

Keywords:
F2FS
Deleted file
Forensic recovery

ABSTRACT

Flash-Friendly File System (F2FS) is a file system optimized for flash memory-based storage devices and is used in a wide range of devices including Android smartphones, drones, in-vehicle infotainment systems and embedded devices. Therefore, from a digital forensic perspective, a recovery technology for deleted file data in F2FS is needed. However, as far as research on deleted data recovery from F2FS is concerned, only basic research has been conducted on deleted data recovery from F2FS, and no specific recovery algorithms have been published. Even in the case of tools that support deleted file data recovery from F2FS, a significant proportion of deleted file data could not be recovered in tests, which limits their usefulness in real-world digital forensic investigations. Therefore, this paper proposes a deleted file data recovery algorithm based on file system metadata carving and virtual address table creation to overcome the limitations of existing research and tools. The proposed recovery algorithm is implemented as a recovery tool and used for performance evaluation with existing forensic and data recovery tools. The performance evaluation results proved the superiority of the recovery algorithm, with the proposed algorithm showing superior recovery performance compared to existing tools.

1. Introduction

Flash-Friendly File System (F2FS) is a file system based on the log-structured file system (Rosenblum, 1991) developed by Samsung and is optimized for flash memory-based storage devices (Lee et al., 2015). As a result, F2FS is used in a variety of devices that use flash memory-based storage, such as Android smartphones, drones, in-vehicle infotainment systems, and embedded devices (Matei, 2019; Azjar et al., 2018; Shin et al., 2022).

Therefore, when conducting digital forensic investigations on these devices, recovery techniques for deleted data in F2FS are required. However, only basic research on deleted data recovery for F2FS has been conducted, and no specific recovery algorithm has been published. Furthermore, tools that support deleted file data recovery for F2FS failed to recover a significant portion of deleted file data in recovery tests, making them of limited use in real-world digital forensic investigations.

In this paper, we studied a deleted file data recovery algorithm based on file system metadata carving and virtual address table creation to overcome the limitations of existing research and tools. To this end, we developed a detailed non-allocated area creation algorithm, a metadata carving algorithm, and a virtual address table creation algorithm and designed a deletion data recovery algorithm by integrating these detailed algorithms. To test the performance of the recovery algorithm, we implemented it as a tool and conducted performance evaluation against existing forensic and data recovery tools. The performance test results showed that the algorithm recovered fragmented deleted file data and deleted file path information that were difficult to recover with existing tools, demonstrating superior performance compared to existing tools.

The contributions of this paper are as follows.

- Proposed an advanced metadata-based recovery algorithm for F2FS. This enables the recovery of deleted file data that is fragmented, has duplicated metadata, or the path information of deleted files, all of which are difficult to recover with existing tools.
- Presented the recovery performance of existing tools for F2FS and conducted a comparative evaluation with the proposed algorithm.

The rest of this paper is organized as follows: Chapter 2 reviews existing research related to F2FS, and Chapter 3 provides background knowledge to understand the recovery algorithm proposed in this paper. Chapter 4 proposes a detailed algorithm for deleted file data recovery, and Chapter 5 describes the experimental design and experimental results for testing the performance of the proposed recovery algorithm. Chapter 6 describes various considerations related to the recovery algorithm presented in this paper. Finally, Chapter 7 summarizes the

* Corresponding author.
E-mail addresses: blueangel@nsr.re.kr (J. Oh), hhu@nsr.re.kr (H. Hwang).

https://doi.org/10.1016/j.fsidi.2025.301976

Available online 3 November 2025

results of this study.

2. Related works

Currier (2022) performed an analysis of the F2FS basic structure and metadata from a digital forensic perspective and an analysis of the metadata that are changed when a file is deleted. In addition, he created test cases for deleted file recovery and performed recovery tests using the XRY tool (MSAB). However, the author did not present a specific algorithm for deleted file recovery.

Tools that support deleted file recovery in F2FS include XRY (MSAB), a mobile forensic tool, and UFS Explorer Professional Recovery (SysDev Laboratories), a file recovery tool. Although the detailed method used for deleted file recovery is not disclosed for either tool, it appears that recovery is performed based on file system metadata rather than on the carving technique because metadata (filename, timestamp) for deleted files is recovered. However, neither tool can recover the full path of the deleted file, and the results of the recovery tests described in this paper showed that these tools failed to properly recover a substantial proportion of deleted files.

Looking at the research to date on deleted file recovery for F2FS, only basic research has been conducted, and no research on specific recovery algorithms has been published. In addition, tools that support F2FS deleted file recovery also fail to recover the full path of deleted files. In the recovery tests performed in this paper, a significant number of deleted files were not properly recovered, indicating that these tools are insufficient for use in actual digital forensic investigations.

Therefore, this paper proposes a deleted file data recovery algorithm based on file system metadata carving and virtual address table creation that can overcome these limitations of existing research and tools. As for carving-based recovery, the proposed technique can be used regardless of the file system, and because carving-based techniques cannot recover fragmented files and cannot recover several important pieces of information (file name, timestamp, full path, etc.), recovery using these techniques was excluded from this study.

3. Background knowledge

This section provides background knowledge to understand the recovery algorithms studied in this paper.

3.1. Overall layout

The data units used in F2FS are as follows (Lee et al., 2015).
Block: Basic unit of data storage (Default: 4 KB).
Segment: A collection of blocks (Default:512 blocks, 2 MB).
Section: A collection of segments.
Zone: A collection of sections.

The overall structure of F2FS is shown in Fig. 1. The superblock stores basic information about the file system, such as area partitioning details and internal parameter values. The checkpoint area stores the current state of the file system, including block allocation, node allocation, and the status of currently active segments. The Segment Information Table (SIT) contains bitmap information that identifies used and unused blocks in the main area. The Node Address Table (NAT) is a structure used to manage nodes, consisting of a table with the physical addresses of nodes. The Segment Summary Area (SSA) is used to manage mapping between physical and logical addresses. The main area is composed of data blocks and node blocks; data blocks contain directory information or user file data, whereas node blocks store inodes or indices of data blocks (Lee et al., 2015).

3.2. F2FS allocation management

The SIT stores block allocation information for the main area, where the actual file metadata and data of F2FS are stored. The location of the SIT can be found using the block address stored in the *sit_blkaddr* field within the superblock. The SIT is organized in block units, and each block is composed of entries. Each entry stores the allocation information of 512 blocks, which make up a single segment in the main area in the form of a bitmap. If the checkpoint area is in compact mode, an additional structure called the SIT journal is used to record the most recent changes in allocation information (Lee et al., 2015). Fig. 2 shows the overall structure of the SIT.

3.3. Building a file/directory tree

The main area is composed of data blocks and node blocks. Data blocks store the actual file data, whereas node blocks store inode information on files and directories or index information of file data. The inode information consists of metadata such as the file/directory name,

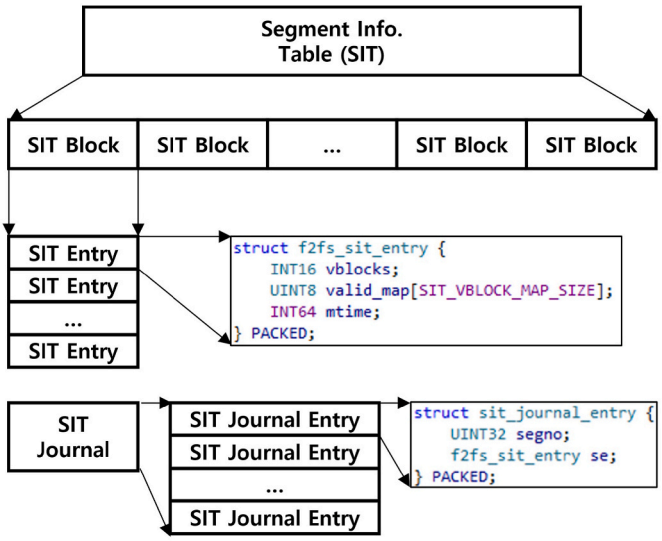

Fig. 2. Structure of segment information table.

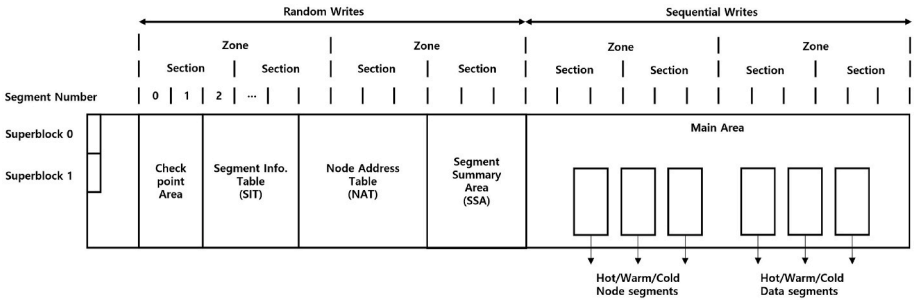

Fig. 1. Overall layout of F2FS.

file data size, and timestamps. In addition, in the case of directory inodes, the inode numbers of child files and directories are stored using a *dentry* structure. The *dentry* structure stores the inode number information (ino) of each child file/directory in units of *f2fs_dir_entry* entries, and the activation/deactivation status of each entry is recorded in a bitmap format using the *dentry_bitmap* (Lee et al., 2015).

The NAT is a table that maps the node identifier (nid) to the corresponding block address of the node block. Similarly to the SIT, the NAT is organized in blocks, and each block consists of entries. Each entry is assigned a nid in sequential order; however, this value is not stored as a field within the entry itself. Instead, the position of the entry within the NAT determines its nid. For example, the first entry in the first block of the NAT represents the entry with nid 0. Each NAT entry stores the inode number of the associated file or directory, as well as the block address of the corresponding node block. If the metadata of a specific file span multiple node blocks, the NAT entries for those node blocks will share the same inode number. If the checkpoint area is in compact mode, an additional structure called the NAT journal is used to store the most recent changes in mapping information. Fig. 3 shows the overall structure of the NAT (Lee et al., 2015).

The method for constructing the file/directory tree in F2FS is as follows. First, the inode number of the root directory is obtained using the *root_ino* field in the superblock. Once the inode number of the root directory is obtained, it is used as the nid to locate the corresponding NAT entry in the NAT. From this NAT entry, the node block address is retrieved, and the inode information of the root directory located at that block address can be accessed. Because the inode information of a directory contains the inode numbers of its child files and folders, the same procedure is repeated using the NAT to obtain the inode information for those child files and folders. By repeating this process, a file/directory tree structure starting from the root directory can be constructed. In addition, if a NAT journal exists, the NAT entries in the journal are used with higher priority to retrieve inode information (Lee et al., 2015). Fig. 4 illustrates the overall process of constructing the file/directory tree.

3.4. Structure of file data

The structure for storing file data is largely divided into inline data and file data block addresses. Inline data is a method of storing file data within an inode when the file data size is smaller than the free space excluding the basic metadata (file name, file size, timestamp, etc.) in the inode. The file data block address is a structure that stores the addresses of the blocks in which the file data are stored if the file data are too large to be stored as inline data. This structure is largely divided into direct pointers and file data index structures. Direct pointers is a structure in

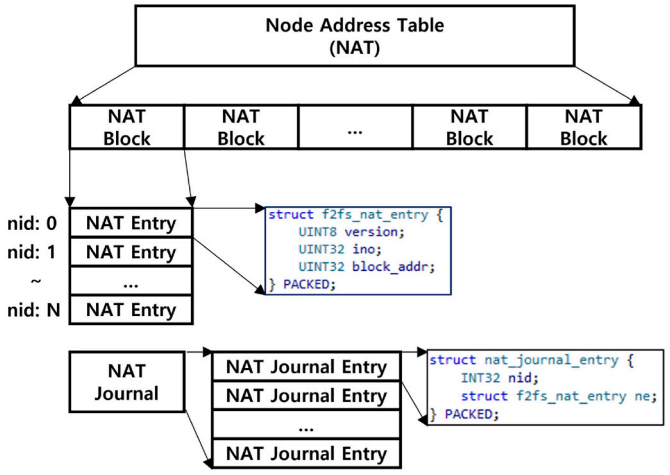

Fig. 3. Structure of node address table.

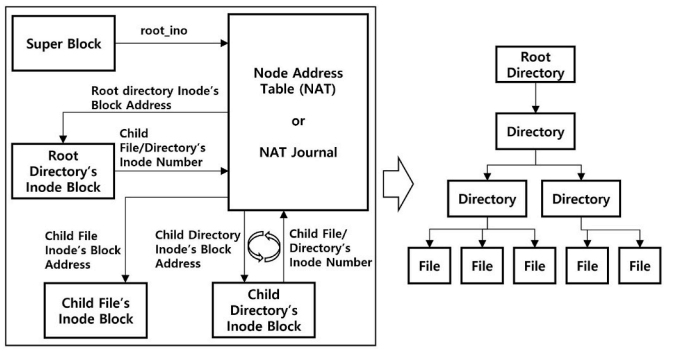

Fig. 4. Process of building a file/directory tree.

which the list of block addresses where file data are stored is kept directly within the inode. If the file data are too large to be stored in the direct pointers structure, the file data index structure from single-indirect to triple-indirect is used in order. In the file data index structure, the direct node block stores a list of block addresses where file data are stored, and the indirect node block stores a list of nids of direct node blocks or indirect node blocks. The locations of direct node blocks and indirect node blocks can be found by converting the nid value to a block address using the NAT (Lee et al., 2015). Fig. 5 shows the structure for storing file data.

4. Deleted file and directory recovery

This section describes how to recover data from deleted files and directories. First, before explaining the detailed recovery algorithm, we will look at the metadata that are changed by the deletion operation and explain the analysis of the unallocated area required for the recovery algorithm. Then, based on this, we will explain the detailed recovery algorithm.

4.1. Metadata changes of the delete operation

When a file or directory is deleted, the bit of the corresponding *f2fs_dir_entry entry* for the deleted file or directory in the *dentry_bitmap* of the dentry structure within the parent directory's inode is set to 0, deactivating the associated *f2fs_dir_entry entry* entry. However, the *f2fs_dir_entry entry* data of the deleted file or directory is not reinitialized and remains intact. In the case of the NAT, the NAT entries that stored the mapping information of the deleted file or directory are reinitialized by setting the values of the *block_addr* fields to 0, as shown in Fig. 6. Finally, the inode block, data blocks, and direct/indirect node blocks of the deleted file or directory are marked as unallocated areas, but the actual data within these blocks remain intact until F2FS garbage collection is performed.

Therefore, when a file or directory is deleted, it is possible to obtain the inode number information of the deleted child file/directory from

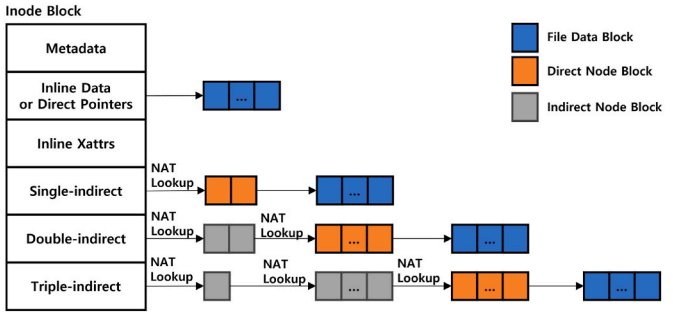

Fig. 5. Structure of file data.

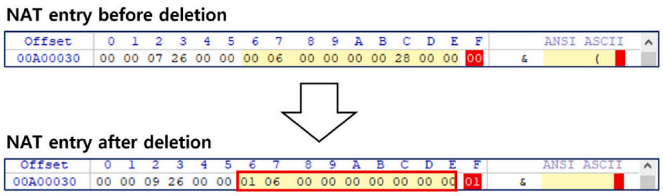

Fig. 6. Changes in NAT entries caused by delete operations.

the parent directory's *dentry* structure. However, the *block_addr* value in the corresponding NAT entry has been reinitialized, making it impossible to determine the block address where the inode is stored. Even if the inode block is carved from the unallocated area, if the file size exceeds a certain threshold (approximately 3.7 MB), the file uses a data index structure. In such cases, if the *block_addr* value of the NAT entry has been reinitialized, the data in the deleted file cannot be properly recovered.

4.2. Recovery algorithm

The process of recovering deleted files/directories is as follows.

4.2.1. STEP 1: Collecting information on unallocated areas

The first step is to collect information on unallocated areas within the main area. Collecting information on unallocated areas is a task of analyzing SIT to obtain information on unused blocks within the main area. The process of obtaining information on unallocated areas is as follows. First, the block address where the SIT data are located is obtained through the value of the *sit_blkaddr* field of the superblock. Next, the SIT data located at the corresponding block address are accessed in blocks, and the bitmap information of the *valid_map* field of the SIT entry in the block is analyzed. The *valid_map* field is 64 bytes in size and stores 512 bits. Therefore, one SIT entry has allocation/non-allocation information for 512 blocks in the main area. Because a segment is generally composed of 512 blocks, one SIT entry can be considered to manage the block allocation/non-allocation information of one segment in the Main Area. In addition, the bitmap information in the *valid_map* field is stored in the Big Endian format. Finally, after the bitmap information of all blocks in the main area has been analyzed, information about unallocated blocks is collected to obtain unallocated area information. In addition, if there is an SIT journal, the unallocated area information is obtained by adding the most recent unallocated block information of the SIT journal entry to the unallocated block information generated earlier. Fig. 7 shows the process of obtaining unallocated area information.

The unallocated area information collected in this way can be used to specify the area to be processed in the next step, which is node block carving, and can also be used to recover deleted file data through the carving technique.

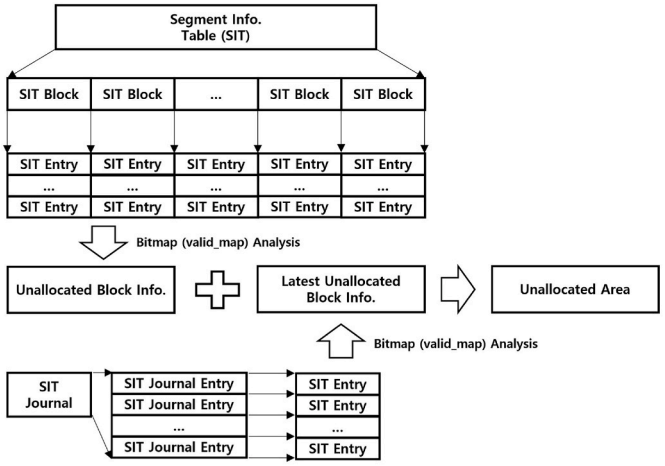

Fig. 7. Collecting information on unallocated areas.

4.2.2. STEP 2: Carving node blocks

The second step is to perform node block carving using the collected unallocated area information. To carve a node block in unallocated area, the node footer located at the end of the node block must be checked. As shown in Fig. 8, the node footer is 24 bytes of data located at the end of the node block and uses the *node_footer* structure.

Therefore, by examining the last 24 bytes of a block where the node footer is located, and verifying that all conditions listed in Table 1 are satisfied, it can be determined whether the block is a node block.

The *node_footer.cp_ver* value is the checkpoint version at the time the corresponding node block was modified. This value cannot be 0 and cannot be greater than the *checkpoint_ver* value in the header of the current checkpoint. Note that if the CP_CRC_RECOVERY_FLAG (0x40) bit is set in the *ckpt_flags* field of the checkpoint header, the upper 32 bits of the *node_footer.cp_ver* value are set to the CRC32 value for the current checkpoint, while only the lower 32 bits are used [2]. Therefore, the *node_footer.cp_ver* value can be compared with the *checkpoint_ver* value in the header of the checkpoint only when the CP_CRC_RECOVERY_FLAG bit is not set. In addition, even if the CP_CRC_RECOVERY_FLAG bit is set, if the *checkpoint_ver* value of the current checkpoint header is less than or equal to the 32-bit maximum value (0xFFFFFFFF), the condition can be used because it can be compared with the lower 32-bit value of *node_footer.cp_ver*. Algorithm 1 is a pseudo-code that compares *node_footer.cp_ver* with the *checkpoint_ver* of the current checkpoint header.

Algorithm 1. Checking node_footer.cp_ver in Node Block Carving

```
1:  if CP_CRC_RECOVERY_FLAG is set in the check_point_header.ckpt_flags
    then
2:      if check_point_header.checkpoint_ver <= 0xFFFFFFFF then
3:          low_32bit_cp_ver ← node_footer.cp_ver & 0xFFFFFFFF
4:          if low_32bit_cp_ver <= check_point_header.checkpoint_ver  then
5:              node_footer.cp_ver is vaild.
6:          else
7:              node_footer.cp_ver is invaild.
8:          end if
9:      else
10:         The validity of node_footer.cp_ver cannot be determined.
11:     end if
12: else
13:     if node_footer.cp_ver <= check_point_header.checkpoint_ver then
14:         node_footer.cp_ver is vaild.
15:     else
16:         node_footer.cp_ver is invaild.
17:     end if
18: end if
```

The *node_footer.nid* means the nid of the current node block, which is larger than the nid (0–3) value used by default when formatting and cannot be larger than the maximum nid value in the current file system. The maximum nid can be calculated as shown in the following equation using the number of NAT segments, block size, and NAT entry size (9 bytes) obtained from the *segment_count_nat* field in the superblock:

$$\text{Max nid} = ((\text{NAT Segment Count}/2)*512*(\text{Block Size})) / 9$$

The *node_footer.ino* refers to the inode number of the file/directory associated with the current node block and must be greater than the ino (0–3) value used by default when formatting. *node_footer.next_blkaddr* is the block number of the next node block, which cannot be zero and cannot be greater than the maximum block number in the current file system. The maximum block number can be found in the *block_count*

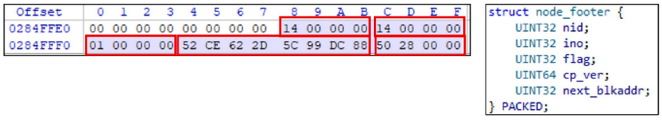

Fig. 8. Node footer & *node_footer* structure.

Table 1
Conditions for carving node blocks.

No	Conditions
1	*node_footer.cp_ver* != 0
2	*node_footer.cp_ver* <= Current Checkpoint Version
3	*node_footer.nid* >= 3
4	*node_footer.nid* <= Max nid
5	*node_footer.ino* >= 3
6	*node_footer.next_blkaddr* != 0
7	*node_footer.next_blkaddr* <= Max Block Number

field of the superblock.

4.2.3. STEP 3: inode identification

The third step is to identify the inode block among the carved node blocks. To determine whether the carved node block is the block with inode data, the conditions of the field values of the *f2fs_inode* structure used by the inode must be checked. Table 2 shows the conditions for identifying the inode block.

The *f2fs_inode.i_mode* field represents the file type and follows the standard Linux file types. When a bitwise AND operation is performed with *i_mode* and S_IFMT, the result will be one of the following values: S_IFIFO, S_IFCHR, S_IFDIR, S_IFBLK, S_IFREG, S_IFLNK, S_IFSOCK, or S_IFWHT (Linux, b). *f2fs_inode.ilinks* refers to the number of links in the target file/directory. For a file, the ilinks value is always greater than or equal to 1. For a directory, the ilinks value is always greater than or equal to 2 because the number of subdirectories is stored as the number of links, and two subdirectories, '.' and '..', always exist. *f2fs_inode.namelen* gives the length of the file/directory name, and because the size of *f2fs_inode.i_name[]* in which the file/directory name is stored is 255 bytes, it can have a value in the range of 1–254. In addition, the actual length of the name of the stored file/directory in *f2fs_inode.i_name[]* must match *f2fs_inode.namelen*. Therefore, if the carved node block satisfies all the conditions in Table 2, the target node block is assumed to be an inode node block.

4.2.4. STEP 4: node block deduplication

The fourth step is to remove any duplicates of the carved node blocks. When a node block is carved based on the node footer information, several node blocks with the same nid and ino can be carved in the unallocated area. This occurs because when data are changed in F2FS, a new block is allocated using the Copy-On-Write method to save the changed data, and the block containing the existing data is deallocated. When recovering a file/directory, if there are multiple node blocks with the same nid and ino, the most recently saved node block should be used. To identify the most recently saved node block among the node blocks with the same nid and ino, use the *cp_ver* in the node footer. In this case, if the CP_CRC_RECOVERY_FLAG bit is not set in the *ckpt_flags* field of the Checkpoint header as described above, the node block with the largest *cp_ver* value is assumed to be the most recently changed node block. On the contrary, if the CP_CRC_RECOVERY_FLAG bit is set, only the lower 32-bit value of *cp_ver* is compared to determine the node block with the largest value as the most recently changed node block. The reason why only the lower 32-bit value of *cp_ver* can be used is that F2FS performs garbage collection periodically, so that the node blocks with the same nid and ino remaining in the unallocated area do not have enough *cp_ver* difference to use the upper 32-bit part of *cp_ver*. Algorithm 2 is a pseudo-code that uses *node_footer.cp_ver* to identify the most recently changed node block.

By applying the method in Algorithm 2, the most recently modified node blocks among those with identical nid and ino values are identified from the carved node blocks, and the remaining node blocks are discarded.

Table 2
Conditions for identifying inode blocks.

No	Conditions
1	(*f2fs_inode.i_mode* & S_IFMT) in {S_IFIFO, S_IFCHR, S_IFDIR, S_IFBLK, S_IFREG, S_IFLNK, S_IFSOCK, S_IFWHT}
2	*f2fs_inode.i_links* >= 1
3	if (*f2fs_inode.i_mode* & S_IFMT) == S_IFDIR, then *f2fs_inode.i_links* >= 2
4	1 <= *f2fs_inode.namelen* <= 254
5	len(*f2fs_inode.i_name*) == *f2fs_inode.namelen*

Algorithm 2. Finding the Latest Node Block in Carved Node Blocks with Same nid and ino

```
1:  latest_node_block ← NULL
2:  for node_block in carved_node_blocks_with_same_nid_ino[] do
3:      if latest_node_block == NULL then
4:          latest_node_block ← node_block
5:          continue
6:      end if
7:      if CP_CRC_RECOVERY_FLAG is set in check_point_header.ckpt_flags
            then
8:          low_32bit_cp_ver_1 ← (node_block.node_footer.cp_ver
            & 0xFFFFFFFF)
9:          low_32bit_cp_ver_2 ← (latest_node_block.node_footer.cp_ver
            & 0xFFFFFFFF)
10:         if low_32bit_cp_ver_1 > low_32bit_cp_ver_2  then
11:             latest_node_block ← node_block
12:         end if
13:     else
14:         if node_block.node_footer.cp_ver > latest_node_block.node_footer
            .cp_ver  then
15:             latest_node_block ← node_block
16:         end if
17:     end if
18: end for
```

4.2.5. STEP 5: Generating virtual NAT

The fifth step is to create a virtual NAT using the carved node blocks that have been deduplicated. As explained earlier, when a file/directory is deleted, all the block address information of the NAT entry related to the deleted file/directory is reinitialized. Therefore, to recover the deleted file/directory, information to replace the reinitialized NAT entries is needed. In this paper, a virtual NAT is created using the nid and ino information stored in the node footer of the carved node block and the location information of the carved node block. The created virtual NAT consists of key and value mapping information, using the nid + ino value as the key information and the address of the carved node block as the value information. The mapping information of the virtual NAT created in this way replaces the information in the reinitialized NAT entries when recovering deleted files/directories. Fig. 9 shows the overall process of creating a virtual NAT.

The detailed process of recovering deleted files and directories using the mapping information of the virtual NAT will be explained in the following step.

4.2.6. STEP 6: directory & file recovery

The sixth step involves recovering the deleted files and directories based on the various pieces of information generated in the previous

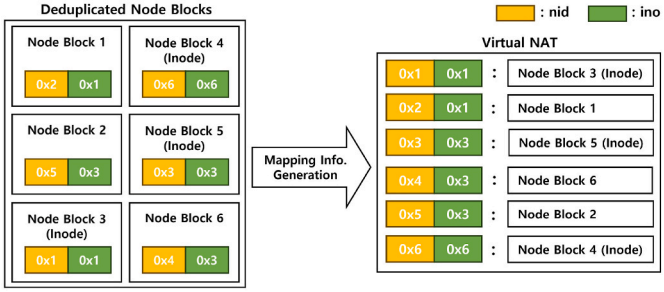

Fig. 9. Generating virtual NAT.

steps. The recovery process for the deleted files and directories is as follows.

The first task is to traverse the file system tree and analyze the *dentry* structures within directories to identify inactive *f2fs_dir_entry* entries. These inactive entries contain inode number information corresponding to deleted files or directories. Once an inactive *f2fs_dir_entry* is found, its inode number is used as both the nid and ino to generate a nid + ino key. This nid + ino key is then used to retrieve the corresponding value, the block address, from the virtual NAT generated in the previous step. If the data located at the retrieved block address are inode data, it must be verified whether these data truly correspond to the inode of the deleted file or directory. This verification process uses the *file_type*, *name_len*, and *hash_code* fields of the *f2fs_dir_entry* entry. First, the file type obtained from the *i_mode* field of the inode is compared with the *file_type* value of the *f2fs_dir_entry*. Next, the *i_namelen* value from the inode is compared with the *name_len* field of the *f2fs_dir_entry*. As for the *hash_code* field, it represents a hash value of the file name and is calculated using the *f2fs_dentry_hash()* function (Google). Therefore, the hash value of the file name stored in the inode is computed and compared with the *hash_code* field. If all three comparisons match, the inode corresponds to the deleted file or directory originally pointed to by the inactive *f2fs_dir_entry*.

If the acquired inode data represent a file, file data recovery proceeds in the following steps. First, if the file data size is small and the file data are in the inline data format (about 3 KB or less) or the direct pointer format (about 3.6 MB or less), file data recovery is possible using only the data in the inode (file data or block address). However, if the file data size is larger, the file data index structure must be analyzed. At this time, the nid corresponding to the direct/indirect node block to be searched and the inode information of the file to be recovered are used as ino to create the nid + ino key, and then the block addresses mapped in the virtual NAT created in the previous step are obtained. If all the file data index structures of the target file can be analyzed with the block addresses obtained through the virtual NAT, then the entire file data can be recovered. On the other hand, if all the file data index structures cannot be analyzed, recovery is performed with only the analyzed file data block addresses, and the unanalyzed parts are treated as sparse areas. Finally, if the file data index structure cannot be analyzed at all, recovery is started using the block addresses at the beginning of the file stored in the inode, and then the remaining part is assumed to be in a non-fragmented state and is recovered by sequentially reading it as much as the file size. The reason for performing this recovery method is that F2FS is a log-structured file system (Rosenblum, 1991) that uses a policy of minimizing file data fragmentation (Currier, 2022). A file that has been completely recovered in this way can know its full path because it can know the parent directory exactly.

If the acquired inode data is a directory, all child files and directories have been deleted, so all *f2fs_dir_entry* entries in the dentry structure of the inode data are analyzed, and the inode data of the deleted child files and directories are acquired through the virtual NAT. After this, the recovery operation is performed by recursively repeating the method described above. Similarly, because the restored directory knows the exact parent directory, it can know its full path.

Finally, if the recovery operation is completed by traversing all the file system trees, the recovery operation is performed on the inode blocks that have not yet been recovered in the carved node block. In this case, the recovered file/directory is set to the orphaned file/directory state because the exact path is unknown. Fig. 10 shows the process of recovering deleted files/directories using the virtual NAT.

Fig. 11 illustrates the overall process of the recovery algorithm as a flowchart based on the descriptions provided so far.

5. Experiment and evaluation

This section describes the experiments and evaluation results used to assess the performance of the recovery algorithm proposed in this paper.

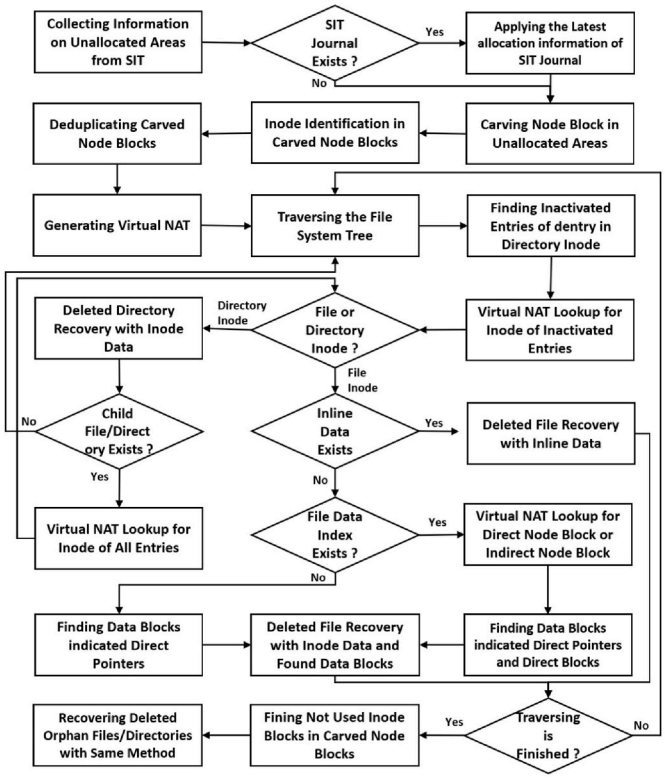

Fig. 11. Flowchart of the recovery algorithm.

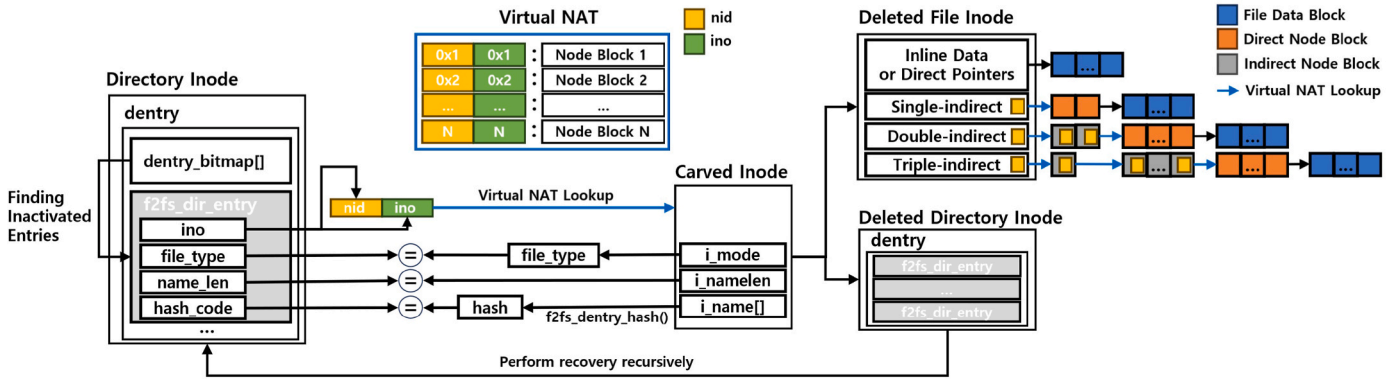

Fig. 10. Process of recovering deleted file and directory using virtual NAT.

5.1. Experiment design

The performance evaluation experiment for the recovery algorithm proposed in this paper was conducted as follows. First, on a Linux system using the Ubuntu 22.04.5 LTS operating system, a 4 GB USB storage device was formatted with F2FS, and 10 test files were created in the 'test_folder_1' and 'test_folder_2' directories, as shown in Table 3. Test files are named in the format 'test[N]_[FileSize].txt' and are configured to use various inline data, direct pointers, direct node blocks, and indirect node blocks, which are methods used by F2FS to store file data. The 'File Data Storage Method' column in Table 3 indicates which storage method is used for each test file, and in the case of direct and indirect node blocks, it additionally indicates the number of blocks used to store file data. This makes it possible to identify each file data storage method in the recovery test and to verify whether the file data have been recovered properly. The data in the test files were saved in text format (file name without extension) to avoid recovery through the carving method; this enables testing of file data recovery through F2FS metadata analysis. Next, deletion of the test files was performed by deleting all test files using the 'rm' (Linux, a) command, which is the delete command in the Linux system shell. In the deletion process, the files under the 'test_folder_1' folder were deleted without deleting the parent folder, and the files under the 'test_folder_2' folder were deleted along with the parent directory, 'test_folder_2'. This makes it possible to verify that the directory is being restored and to test whether the path information of the restored files can be obtained. Next, after the deletion process was complete, the USB storage device was unmounted and then connected to the recovery test system through Tableau Forensic (OpenText, b), which is a write blocker, An image file for the USB storage device was also created in raw format using the FTK Imager (Exterro, a). Finally, the recovery test was performed using the image file created, and if the image file input was not supported (e.g., XRY), the recovery test was performed using the USB storage device directly. This experiment aims to test the recoverability of deleted files under different data storage methods. Cases in which metadata and data of deleted files remaining in unallocated area are overwritten in real-world scenarios are not considered. In other words, the experiment tests whether deleted files can be fully recovered when both their metadata and data remain intact in the unallocated area. To this end, the file system was unmounted immediately after deleting files and directories, without performing any further operations. Table 3 shows the file name, path, file data storage method, number of file data fragments, number of inode blocks, and whether the parent directory was deleted for the test files deleted from USB storage media formatted with F2FS. The number of inode blocks refers to the number of such blocks identified when the inode block of the file is searched in an image file.

The recovery algorithm proposed in this paper was implemented with the F2FS_Recover tool, and existing forensic and recovery programs were used to compare and evaluate its performance. The existing forensic and recovery tools used in the experiment were the following:

- Encase Forensic v24.4 (OpenText, a)
- Magnet AXIOM v8.8 (Magnet Forensics)
- X-Ways Forensics v21.3 (X-Ways)
- FTK v8.1 (Exterro, b)
- Autopsy v4.21 (Sleuth Kit Labs)
- Cellebrite Inseyets PA v10.5 (Cellebrite)
- XRY v10.12 (MSAB)
- UFS Explorer 10.11 (SysDev Laboratories)

5.2. Evaluation

The results of evaluating the performance of each tool through the deleted file recovery experiment are shown in Table 4. First, among the world's most widely used forensic tools, including Encase Forensic, Magnet AXIOM, X-Ways Forensics, FTK, Autopsy, and Cellebrite Inseyets PA, none supported analysis and recovery of F2FS, except for Magnet AXIOM. In the case of Magnet AXIOM, it only supports file system tree analysis for F2FS and creation of unallocated area and does not support recovery of deleted files.

XRY supports file system tree analysis and deleted file recovery, but does not support the generation of unallocated area. The deleted file recovery function recovered all metadata (file name, timestamp, etc.) for 20 deleted test files, but it failed to recover any data for nine files in which multiple inode blocks were identified. The data sizes of the nine files were zero. In addition, XRY does not support deleted file path information recovery or directory recovery.

UFS Explorer, like XRY, provides file system tree analysis and deleted file recovery, but does not support generation of unallocated area. The deleted file data recovery function supports metadata recovery and file data recovery for non-fragmented files only. Accordingly, recovery was performed only for the 14 files among the test files that had not been

Table 3
Test files for deleted file recovery experiment.

No	File Name	File Path	File Data Storage Method				Fragment Count	Inode Block Count	Parent Directory Deletion Status
			Inline Data	Direct Pointers	Direct Node Block	Indirect Node Block			
1	test1_3 KB.txt	/test_folder_1/test1_3 KB.txt	O	X	X	X	1	1	Not Deleted
2	test2_1 MB.txt	/test_folder_1/test2_1 MB.txt	X	O	X	X	1	1	Not Deleted
3	test3_5 MB.txt	/test_folder_1/test3_5 MB.txt	X	O	O (1)	X	2	1	Not Deleted
4	test4_10 MB.txt	/test_folder_1/test4_10 MB.txt	X	O	O (2)	X	1	1	Not Deleted
5	test5_20 MB.txt	/test_folder_1/test5_20 MB.txt	X	O	O (5)	O (1)	1	2	Not Deleted
6	test6_30 MB.txt	/test_folder_1/test6_30 MB.txt	X	O	O (7)	O (1)	2	2	Not Deleted
7	test7_40 MB.txt	/test_folder_1/test7_40 MB.txt	X	O	O (10)	O (1)	10	2	Not Deleted
8	test8_50 MB.txt	/test_folder_1/test8_50 MB.txt	X	O	O (12)	O (1)	13	2	Not Deleted
9	test9_60 MB.txt	/test_folder_1/test9_60 MB.txt	X	O	O (14)	O (1)	15	2	Not Deleted
10	test10_70 MB.txt	/test_folder_1/test10_70 MB.txt	X	O	O (17)	O (1)	15	2	Not Deleted
11	test11_3 KB.txt	/test_folder_2/test11_3 KB.txt	O	X	X	X	1	1	Deleted
12	test12_1 MB.txt	/test_folder_2/test12_1 MB.txt	X	O	X	X	1	1	Deleted
13	test13_5 MB.txt	/test_folder_2/test13_5 MB.txt	X	O	O (1)	X	1	1	Deleted
14	test14_10 MB.txt	/test_folder_2/test14_10 MB.txt	X	O	O (2)	X	1	1	Deleted
15	test15_20 MB.txt	/test_folder_2/test15_20 MB.txt	X	O	O (5)	O (1)	1	1	Deleted
16	test16_30 MB.txt	/test_folder_2/test16_30 MB.txt	X	O	O (7)	O (1)	1	1	Deleted
17	test17_40 MB.txt	/test_folder_2/test17_40 MB.txt	X	O	O (10)	O (1)	1	2	Deleted
18	test18_50 MB.txt	/test_folder_2/test18_50 MB.txt	X	O	O (12)	O (1)	1	2	Deleted
19	test19_60 MB.txt	/test_folder_2/test19_60 MB.txt	X	O	O (14)	O (1)	1	2	Deleted
20	test20_70 MB.txt	/test_folder_2/test20_70 MB.txt	X	O	O (17)	O (1)	1	2	Deleted

Table 4
Performance evaluation results.

	Encase Forensic v24.4	Magnet AXIOM v8.8	X-Ways Forensics v21.3	FTK v8.1	Autopsy v4.21	Cellebrite Inspect PA v10.5	XRY v10.12	UFS Explorer 10.11	F2FS_Recover (Our tool)
Filesystem tree analysis	X	O	X	X	X	X	O	O	O
Unallocated area generation	X	O	X	X	X	X	X	X	O
Deleted file's metadata recovery	X	X	X	X	X	X	O (20/20)	△ (14/20)	O (20/20)
Deleted file's data recovery	X	X	X	X	X	X	△ (11/20)	△ (14/20)	O (20/20)
Deleted file's path recovery	X	X	X	X	X	X	X	X	O
Deleted directory recovery	X	X	X	X	X	X	X	X	O

fragmented. In addition, path information recovery and directory recovery of deleted files are not supported.

In contrast, F2FS_Recover, which implements the recovery algorithm proposed in this paper, supports all key functionalities including file system tree analysis, creation of unallocated area, deleted file recovery, recovery of deleted file paths, and recovery of deleted directories. In the deleted file recovery test, it successfully recovered both the metadata and file data for all 20 deleted test files, regardless of the number of data fragments or identified inode blocks. Moreover, by recovering the path information of deleted files, it could accurately determine their original locations. Deleted directories were also successfully recovered, enabling restoration of full paths for files that were deleted along with their parent directories. Fig. 12 shows the deleted file recovery results of the F2FS_Recover tool. The detailed recovery results for each test file using XRY, UFS Explorer, and F2FS_Recover are provided in Table 5.

6. Discussion & limitation

6.1. Garbage collection

F2FS performs garbage collection to improve file system performance. F2FS's garbage collection task selects the target section based on information such as the number of valid blocks or block age, moves the data of allocated blocks within the section, and registers the target section as a free section. Sections registered as free sections are set as unallocated space, enabling new data to be written. Garbage collection operations can be performed by the user in device settings, when free space is insufficient, or periodically as a background task. In the case of background garbage collection operations, the execution interval varies depending on the system's resource status and file system usage patterns (Currier, 2022). The recovery algorithm proposed in this paper has the limitation that it performs recovery operations only on data remaining in unallocated areas before garbage collection is performed.

6.2. Fragmentation

In F2FS, file data fragmentation varies depending on the logging mode. F2FS uses two logging modes: append logging and thread logging. The append logging mode writes data sequentially to clean segments, whereas the thread logging mode writes data by locating unallocated blocks. As a result, file data are not fragmented in append logging mode, but fragmentation occurs in thread logging mode. Typically, when sufficient free space is available on the storage device, append logging mode is used; otherwise, thread logging mode is applied (Currier, 2022). Therefore, in most cases, file data are stored contiguously without fragmentation in Append Logging mode. For this reason, the recovery algorithm proposed in this paper adopts a method that sequentially recovers data from the starting location of the file over the size of the file, in cases where the data location of the deleted file cannot be accurately determined due to damage of the metadata left in the unallocated area.

7. Conclusion

F2FS is a file system optimized for flash memory-based storage devices and is used in various devices such as Android smartphones, drones, IVI, and embedded equipment. Therefore, from a digital forensics investigation perspective, recovery techniques for deleted data within F2FS are necessary. However, only basic research on deleted data recovery in F2FS has been published, and existing tools that support recovery of deleted file data from F2FS have demonstrated poor performance in recovery tests, failing to recover a significant portion of deleted data.

Therefore, this paper proposes a deleted file data recovery algorithm based on file system metadata carving and virtual address table generation to overcome the limitations of existing research and tools. The proposed algorithm was implemented as a tool and used in performance comparison experiments with existing forensic and recovery tools. In these experiments, the implemented tool demonstrated superior

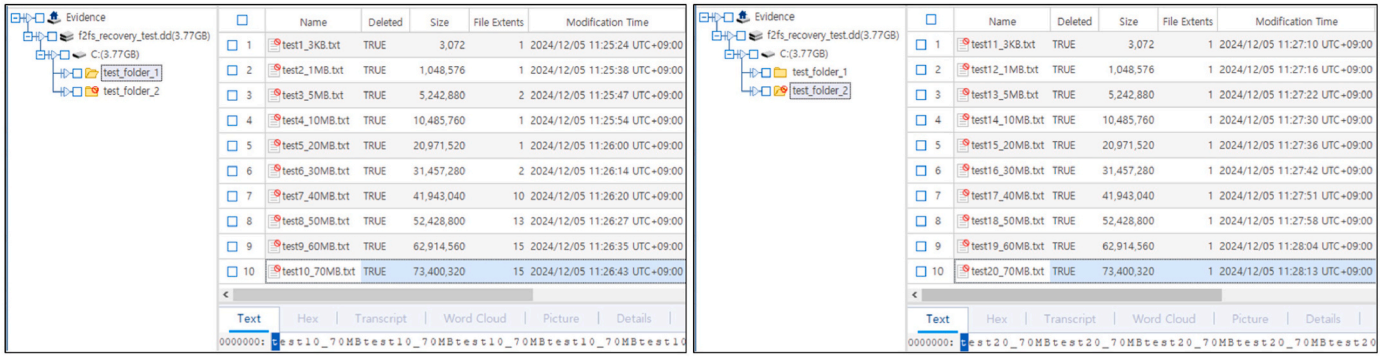

Fig. 12. Recovery result of F2FS_Recover.

Table 5
Detailed recovery result of tools supporting F2FS deleted file recovery.

No	File Name	File Data Storage Method				Fragment Count	Inode Block Count	XRY v10.12		UFS Explorer v10.11		F2FS_Recover (Ours)	
		Inline Data	Direct Pointers	Direct Node Block	Indirect Node Block			Meta Data	File Data	Meta Data	File Data	Meta Data	File Data
1	test1_3KB.txt	O	X	X	X	1	1	O	O	O	O	O	O
2	test2_1MB.txt	X	O	X	X	1	1	O	O	O	O	O	O
3	test3_5MB.txt	X	O	O (1)	X	2	1	O	O	X	X	O	O
4	test4_10MB.txt	X	O	O (2)	X	1	1	O	O	O	O	O	O
5	test5_20MB.txt	X	O	O (5)	O (1)	1	2	O	O	O	O	O	O
6	test6_30MB.txt	X	O	O (7)	O (1)	2	2	O	X	X	X	O	O
7	test7_40MB.txt	X	O	O (10)	O (1)	10	2	O	X	X	X	O	O
8	test8_50MB.txt	X	O	O (12)	O (1)	13	2	O	X	X	X	O	O
9	test9_60MB.txt	X	O	O (14)	O (1)	15	2	O	X	X	X	O	O
10	test10_70MB.txt	X	O	O (17)	O (1)	15	2	O	X	X	X	O	O
11	test11_3KB.txt	O	X	X	X	1	1	O	O	O	O	O	O
12	test12_1MB.txt	X	O	X	X	1	1	O	O	O	O	O	O
13	test13_5MB.txt	X	O	O (1)	X	1	1	O	O	O	O	O	O
14	test14_10MB.txt	X	O	O (2)	X	1	1	O	O	O	O	O	O
15	test15_20MB.txt	X	O	O (5)	O (1)	1	1	O	O	O	O	O	O
16	test16_30MB.txt	X	O	O (7)	O (1)	1	1	O	O	O	O	O	O
17	test17_40MB.txt	X	O	O (10)	O (1)	1	2	O	X	O	O	O	O
18	test18_50MB.txt	X	O	O (12)	O (1)	1	2	O	X	O	O	O	O
19	test19_60MB.txt	X	O	O (14)	O (1)	1	2	O	X	O	O	O	O
20	test20_70MB.txt	X	O	O (17)	O (1)	1	2	O	X	O	O	O	O

performance compared to existing tools that support F2FS deleted file recovery, thereby proving the effectiveness of the proposed recovery algorithm. Consequently, the recovery algorithm presented in this paper is expected to be highly beneficial for recovering deleted data from devices using F2FS in the context of digital forensic investigations.

References

Azjar, M., et al., 2018. Drone forensic analysis using open source tools. Journal of Digital Forensics, Security and Law 13. URL: https://commons.erau.edu/cgi/viewcontent.cgi?params=/context/jdfsl/article/1513/path_info=V13N1_03_Azhar.pdf.

Cellebrite. Cellebrite inseyets pa. https://cellebrite.com/en/cellebrite-inseyets-powered-by-pa. (Accessed 30 July 2025).

Currier, C., 2022. The flash-friendly file system (f2fs). In: Mobile Forensics - the File Format Handbook. Springer, pp. 69–118. URL: https://link.springer.com/chapter/10.1007/978-3-030-98467-0_3.

Exterro, a. Accessed: 2025-July-30.

Exterro, b. Forensic toolkit. https://www.exterro.com/digital-forensics-software/forensic-toolkit. (Accessed 30 July 2025).

Google. Google, hash.c. https://android.googlesource.com/kernel/msm/+/android-8.1.0_r0.17/fs/f2fs/hash.c. (Accessed 30 July 2025).

Lee, C., et al., 2015. F2fs: a new file system for flash storage. In: 13th USENIX Conference on File and Storage Technologies (FAST 15). URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee.

Linux, a. Linux manual page, rm(1). https://man7.org/linux/man-pages/man1/rm.1.html. (Accessed 30 July 2025).

Linux, b. Linux, stat.h. https://github.com/torvalds/linux/blob/master/include/uapi/linux/stat.h. (Accessed 30 July 2025).

Magnet Forensics. Magnet axiom. https://www.magnetforensics.com/products/magnet-axiom/. (Accessed 30 July 2025).

Matei, M., 2019. Galaxy note 10 uses f2fs, not ext4 file system: what's the difference? URL: https://www.sammobile.com/news/galaxy-note-10-uses-f2fs-not-ext4-file-system-whats-the-difference/. online article.

MSAB. Xry. https://www.msab.com/product/xry-extract/xry-pro/. (Accessed 30 July 2025).

OpenText, a. Encase forensic. https://www.opentext.com/products/forensic. (Accessed 30 July 2025).

OpenText, b. Tableau forensic. https://www.opentext.com/products/tableau-forensic. (Accessed 30 July 2025).

Rosenblum, M.O.J., 1991. The design and implementation of a log-structured file system. In: SOSP '91: Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, pp. 1–15. URL: https://dl.acm.org/doi/abs/10.1145/121132.121137.

Shin, Y., et al., 2022. Digital forensic case studies for in-vehicle infotainment systems using android auto and apple carplay. Sensors 22, 7196. URL: https://www.mdpi.com/1424-8220/22/19/7196.

Sleuth Kit Labs. Autopsy. https://www.autopsy.com/. (Accessed 30 July 2025).

SysDev Laboratories. Ufs explorer professional recovery. https://www.sysdevlabs.com/product.php?id=ufsxpuser_cat=techos=win. (Accessed 30 July 2025).

X-Ways. X-ways forensics. https://www.x-ways.net/forensics/. (Accessed 30 July 2025).