International Journal of Digital Crime and Forensics
Volume 14 • Issue 3

Golden Eye:
An OS-Independent Algorithm for Recovering Files From Hard-Disk Raw Images

Fan Zhang, Mathematics and Computer Science School, Wuhan Polytechnic University, China*
Wei Chen, Nanjing University of Posts and Telecommunications, China
Yongqiong Zhu, School of Art, Wuhan Business University, China

ABSTRACT

File systems are important sources of intelligence information and digital evidence. They have long attracted the interest of researchers in recovering files that are deleted from a hard disk. Existing file recovery studies rely heavily on an operating system (OS). However, it is often encountered that OS services are not available, making existing file recovery approaches unusable. To address this issue, the authors design and implement an OS-independent file recovery algorithm named Golden Eye (GE) by targeting the EXT4 file system. Fed the raw image obtained from a (sanitized) hard disk, GE can automatically recover any designated file or even the whole EXT4 file system. GE is based on the understanding of the file disk layout of EXT4 and does not need any support from additional hardware or software. Experimental results prove the feasibility and correctness of GE. This work not only solves the OS dependency problem that most existing file recovery work suffers from but also reveals the fact that even sanitized hard disks are still at risk of leaking sensitive data.

KEYWORDS

Data Security, Digital Forensics, EXT4, File Recovery, File System Security

INTRODUCTION

File systems are important sources of confidential and private information. Various types of data (e.g., documents, audio, videos, and pictures) are stored in file systems. Data are often intentionally deleted or unintentionally lost, and therefore different methods of file recovery have been developed for various purposes, such as digital forensics and file rescue.

Depending on whether utilizing the file system metadata, existing file recovery approaches can be divided into two categories: Metadata-based file recovery (MFR) (Dewald & Seufert, 2017; Fairbanks, 2012; Jo et al., 2018; Kim et al., 2021; Lee et al., 2020; Lee & Shon, 2014) and carving-based file recovery (CFR) (Garfinkel, 2007; Garfinkel & McCarrin, 2015; Gladyshev & James, 2017; Golden & Vassil, 2005; Hand et al., 2012; Pal et al., 2003; Pal et al., 2008; Tang et al., 2016). MFR is fast and accurate because it can leverage file system metadata to interpret user data. However, MFR cannot work if metadata are missing or corrupted. Different from MFR, CFR does not rely on metadata. It

DOI: 10.4018/IJDCF.315793 *Corresponding Author

leverages syntactic signatures (e.g., file header-footer pairs) (Tang et al., 2016), semantic structures (e.g., explicit control flow paths within a binary executable) (Hand et al., 2012), heuristic technologies (Garfinkel & McCarrin, 2015; Gladyshev & James, 2017; Pal et al., 2008), timestamps (Nordvik et al., 2020; Portera et al., 2021) or deep learning technologies (Heo et al., 2019; Mohammad & Alqahtani, 2019) to restore files. Unlike MFR, which can precisely recover a file under the "direct guidance" of metadata, CFR "indirectly infers" which data blocks belong to the file to be recovered. Therefore, CFR suffers from problems such as false positives and higher time overhead. In summary, both MFR and CFR have their advantages and disadvantages. They complement each other, and neither can take the place of the other.

Although researchers have conducted in-depth and extensive research, there are still issues to be addressed for MFR and CFR. A critical issue is that most existing approaches rely heavily on services from an operating system (OS) (Fairbanks, 2012; Garfinkel, 2007; Garfinkel & McCarrin, 2015; Golden & Vassil, 2005; Hand et al., 2012; Jo et al., 2018; Kim et al., 2021; Lee et al., 2020; Lee & Shon, 2014; Pal et al., 2003; Pal et al., 2008; Tang et al., 2016). However, in many cases an OS is not available. For example, when a hard disk fails, or a hard disk is sanitized based on American federal NIST 800-88 (Kissel et al., 2014), the hard disk can no longer be mounted to other machines to boot their OSs nor boot its own OS, which renders existing approaches (Fairbanks, 2012; Garfinkel, 2007; Garfinkel & McCarrin, 2015; Golden & Vassil, 2005; Hand et al., 2012; Jo et al., 2018; Kim et al., 2021; Lee et al., 2020; Lee & Shon, 2014; Pal et al., 2003; Pal et al., 2008; Tang et al., 2016;) unusable.

Researchers found that raw images can be obtained from a hard disk by using modern technologies such as magnetic force microscopy (Hughes et al., 2009; Kanekal, 2013). However, as for the next step of mapping the raw image back into original files, the authors have found little related work. This has motivated them to develop an OS-independent algorithm that can restore files based merely on the raw hard disk image. This study targets the EXT4 file system in Linux. The authors do not target Windows file systems, because, on the one hand, Windows file systems are relatively well studied, on the other hand, Linux has been more widely deployed than Windows in the server market, and EXT4 is the mainstream and default file system in a Linux OS.

The authors' contributions are as follows:

1. They propose a file recovery algorithm called Golden Eye (GE) for the EXT4 file system. So long as given a hard-disk raw image and the corresponding global sector addresses, GE can automatically recover files in the absence of an OS. The authors' work does not need any support from additional hardware and software (including OS), thereby solving the OS dependence problem from which currently most existing MFR/CFR approaches suffer.
2. The authors' work, together with Kanekal's research (2013), constitutes a closed-up loop for file recovery from a sanitized hard disk. Specifically, first of all, Kanekal (2013) argued that raw images can be acquired from a hard disk that is sanitized based on the American federal NIST 800-800. Then, the authors' algorithm GE further proved that files can be recovered from hard-disk raw images. Therefore, sensitive files in a sanitized hard disk may still be at risk of being recovered. This suggests that the authors should re-examine the NIST 800-88. Specifically, for high-security risk storage devices, they recommend the cryptographic erase (CE) be mandatory before applying any other sanitized approaches provided in the NIST 800-88.
3. To prove the feasibility and correctness of GE, the authors illustrate how to "manually" derive the exact blocks where a file is stored in the absence of an OS (see Appendix). Experiments showed that the authors' result is identical to what was returned by the debugfs (i.e., a widely deployed Linux file system tool).

RELATED WORK

Research on File Recovery

In the following, the authors will introduce related studies about MFR and CRF, respectively.

Generally, MFR is a process of restoring files under the guidance of metadata. Lee and Shon (2014) introduced an EXT2/3 file recovery approach. The main contribution of their work is it allows to find the exact original data blocks to which the indirect block originally points if the indirect block pointer is zeroed due to a file deletion operation. Later, Lee et al. (2020) extended Lee and Shon's (2014) work to an EXT4-based file recovery framework that could be compatible with all EXT file systems. Kim et al. (2021) introduced an EXT4 file system forensic framework, which can recover deleted EXT4 files based on the EXT4 journal information. Compared with the authors' work, Kim et al.'s (2021), Lee et al.'s (2020), and Lee and Shon's (2014) studies had to rely on semantic information provided by an OS. However, this is infeasible in the context of the authors' research, because. in their attack model, they assume that there is nothing but a raw image that has no semantics at all, let alone an OS. Jo et al. (2018) studied the privacy protection issue in an Internet of things (IoT) platform from the perspective of digital forensics. They illustrated that data recovery is an important means to undermine privacy protection by manually restoring a deleted passport file on an EXT4 file system. They claimed that they would establish an automated recovery system in the future, while the authors have already implemented the automated file recovery algorithm GE. Furthermore, Aine et al. (2020) discussed the challenges and considerations of cybercrime investigations and digital forensics of IoT. Fairbanks (2012) presented a detailed introduction to the low-level data structures of the EXT4 file system from the perspective of digital forensics. However, Fairbanks merely described the data structures of EXT4, rather than presenting a specific recovery algorithm. Dewald and Seufert (2017) described an EXT4 file reconstruction approach, which can work under the situation that the super block (i.e., an essential metadata structure for an EXT4 file system to interpret data) is corrupted or overwritten. The limitation of their work is that it can only recover files on a file system that keeps a journal and enables the content data mode. In other words, their work is also OS-dependent.

Generally, the CFR is a process of reassembling files from fragments in the absence of metadata of a file system. In the early stage of file carving investigations, researchers utilized the "header-footer" pair to identify file boundaries (Garfinkel, 2007; Golden & Vassil, 2005). This approach can only restore nonfragmented files (i.e., files stored with consecutive data blocks). However, file fragmentation is common in practice, especially for large files. To address the fragmentation issue, Garfinkel (2007) introduced a brute-force searching solution for bifragmentation files. This solution exhaustively checks every combination of bifragmentations until the combination passes some decoder's verification algorithm, but it still suffers from problems of scalability and false positives. About heuristic-based or semantic-based exploration, Pal et al. (2003) provided a fragmentation-resilient carving approach graphically. Later, Pal et al. (2008) improved their work by using the sequential hypothesis test to move through block by block to pinpoint the original file. Gladyshev and James (2017) viewed file carving as a decision problem and gave a formal definition of decision-theoretic file carving. Their work can effectively speed up the carving time, which is important to the resource-constrained file carving where a reduction in carving time is preferred to completeness. Regarding the situation that little effort was put forth toward the recovery of binary executable files, Hand et al. (2012), for the first time, provided a solution that leverages the combination of both road map information defined in executable file headers and explicit control flow paths within the binary code. Nordvik et al. (2020) argued that existing CRF methods do not consider how to recover file-related metadata, which, such as timestamps, have greater value for complete forensics. For this reason, they proposed a generic timestamp metadata carving method for the first time. However, this method can only find metadata records with multiple equivalent timestamps and therefore misses metadata records and

files with different but similar timestamps. To address this issue, Portera et al. (2021) introduced a prefix-based potential timestamp carving approach, which greatly improves the recall of files and corresponding timestamped metadata without reducing accuracy. With the development of machine learning techniques, deep learning-based methods are also being applied to file recovery (Mohammad & Alqahtani, 2019). Heo et al. (2019) described a long short term memory-based approach for audio file recovery. In contrast to traditional methods, In contrast to traditional file recovery methods, Heo et al.'s method aims to recover audio that can be distinguished by the human ear, rather than recovering the original exact data as in traditional methods.

Secure Sanitization and Raw Images Obtaining

In this section, the authors will introduce the four main sanitization approaches provided in the American federal guideline NIST 800-88 (Kissel et al., 2014), and explain why raw images can still be obtained even if a storage device is sanitized.

Cryptographic Erase

CE (Hughes & Coughlin, 2006; Kissel et al., 2014; Zhang et al., 2018) achieves data erasure by first encrypting all sensitive data on a hard disk, and then permanently discarding the encryption key. Recovering data from a cryptographically erased hard disk is essentially equivalent to attacking cryptography algorithms, and thus the authors do not take CE into account in this paper.

Overwritten

Overwritten (Kanekal, 2013; Kissel et al., 2014) overwrites sensitive data with random data once or multiple times, attempting to erase every bit of the original sensitive information. Overwritten is not secure. For example, reallocated blocks (i.e., error blocks) cannot be erased (Hughes & Coughlin, 2006) and sensitive data may be left intact in the guardbands between tracks (Hughes et al., 2009; Kanekal, 2013). It is possible to recover sensitive data by using the spin-stand or the magnetic force microscopy technique in practice.

Degaussing

Degaussing leverages degaussers (i.e., coil, capacitive, and permanent magnet), to reset the magnetization of individual domains in the storage media, thereby eliminating sensitive data (Kanekal, 2013; Kissel et al., 2014). However, degaussing has become increasingly difficult due to the rapid progress of magnetic media technology. Modern magnetic media have such high coercivity (i.e., magnetic force) that existing degaussers may not have sufficient force to effectively degauss the media, which results in the retention of sensitive data (Kanekal, 2013).

Physical Destruction

Physical destruction (PD) is the highest level of data sanitization in NIST 800-88 (Kissel et al., 2014). PD requires that the storage media be ground into small enough pieces; otherwise, data may still be recovered from media fragments (Kanekal, 2013). Unfortunately, with the development of electronic storage media (e.g., flash) technology, flash chips become increasingly smaller and harder. This poses a major challenge to PD in practice: Grinders may even be damaged due to the high hardness of flash chips (Kanekal, 2013).

Consequently, overwritten, degaussing, and PD are not reliable ways to completely sanitize sensitive data from a storage device (Kanekal, 2013). It is theoretically feasible to recover raw image data from a sanitized hard disk by using modern technologies such as magnetic force microscopy (Kanekal, 2013). Since raw images can be obtained from a sanitized hard disk, it is feasible to acquire raw images from any hard disk.

PROBLEM STATEMENT AND ASSUMPTIONS

Problem Statement

The authors' goal is to design and implement an OS-independent file recovery algorithm, which inputs merely the raw image of a hard disk and then outputs the recovered files.

Assumptions

The authors assume that they can obtain the raw image of a hard disk as well as the corresponding global linear sector addresses of the raw image.

Assumption One: Obtaining a Hard-Disk Raw Image

Firstly, the authors have described that it is feasible to obtain raw images from any hard disk. In this regard, it is important to highlight the difference in requirements between the authors' work GE and existing approaches. Unlike existing approaches that require data to be accompanied by OS or file system semantics, GE requires only raw images and hence removes the OS-present restrictions.

Assumption Two: Obtaining Global Linear Sector Addresses of the Raw Image

The raw image alone is not sufficient to recover a file. Theoretically, block information is another prerequisite for file recovery. In EXT4, a block is a file system concept. A block is a minimum unit of file access, which consists of several consecutive sectors. Every block in EXT4 has a unique block number, and every block number can be equivalently converted to consecutive global linear sector numbers. A block number and its corresponding global linear sector numbers are semantically equivalent, that is, both can equivalently locate the same addresses on a storage device. The difference between the two is that a block number is OS-dependent, while a global linear sector number is OS-independent.

Since the "block number" and the "global linear sector numbers" are semantically equivalent, it would appear that assumption two is merely a simple equivalent conversion to the way of locating files of existing research and therefore is unnecessary. However, this is not true. The authors highlight that their assumption two renders all existing block-based MFR approaches (Dewald & Seufert, 2017; Jo et al., 2018; Lee et al., 2020; Lee & Shon, 2014) a subset of the authors' study. This is because the block number is a file system concept, which must be provided in the presence of an OS. Since the OS is present, the block number can be easily converted to its equivalent global linear sector numbers, and thus GE can also run normally in the OS-present context of existing MFR approaches. In contrast, the global linear sector number is OS-independent. Since an OS is absent, the global linear sector number cannot be converted to its equivalent block number at all, and thus existing block-based MFR approaches are surely not able to work in the OS-absent context of GE.

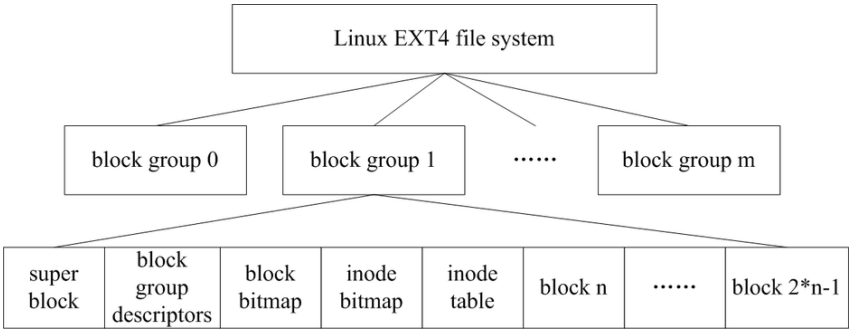
From a practical point of view, it is feasible to obtain the global linear sector addresses of a hard disk. Initially, the global linear sector address is a standard addressing method of a modern hard disk, and hence it is a universal procedure to obtain the global linear sector address. Furthermore, even if the target is a sanitized hard disk, the authors can still obtain the global linear sector addresses based on existing studies (Hughes & Coughlin, 2006; Hughes et al., 2009; Kanekal, 2013; Kissel et al., 2014).

INTRODUCTION TO THE EXT4 FILE SYSTEM

The Hard-Disk Layout of an EXT4 File System

In the following, the authors will use the terms "target hard disk" (THD) (in which files to be recovered) and "target operating system " (in which files to be recovered reside) (TOS). Figure 1 gives the layout of an EXT4 file system on a hard disk.

Figure 1. The Hard Disk Layout of an EXT4 File System



Starting Address of a Linux Partition: SEC_START

Each OS partition starts from a particular sector of a hard disk, which the authors call *SEC_START* in this paper. The exact SEC_START can be obtained from the disk partition table of the master boot record (MBR) ("Logical block addressing," 2022; Sedory, 2013). A modern hard disk adopts "global linear addressing" to number all sectors starting from 0. Therefore, once the authors obtain the global linear sector addresses occupied by a given file *F*, file *F* can be directly read out from the hard disk without any support from the OS. Importantly, the SEC_START is the starting address of an OS partition, and thus it can be treated as the "base address." Then, by analyzing the metadata of the TOS, the authors can obtain the "offset address(es)" of *F* within the TOS. Finally, by adding the two, that is, SEC_START (base address) + offset address(es), the authors can obtain the global linear sector address(es) that *F* occupies, and hence file *F* can be recovered in the absence of an OS.

Block and Block Group

To improve the efficiency of reading and writing files, EXT4 aggregates a fixed number of sectors to form a minimum read/write unit, namely, a *block* (Poirier, 2019; The Linux Kernel Development Community, 2022). A fixed number of blocks are then aggregated into a *block group*. For example, in Figure 1, the entire Linux partition is divided into m+1 number of block groups, that is, "block group 0~block group m." Block group 1 includes consecutive and adjacent blocks (i.e., "block n~block2*n-1").

Super Block

The *super block* (i.e., s_blocks_per_group structure) (Poirier, 2019; The Linux Kernel Development Community, 2022) is a special block dedicated to storing the key metadata of EXT4. EXT4 leverages the super block to understand and manage the file system.

Inode

A Linux system abstracts everything as a "file," and no longer distinguishes between a regular file and a directory. For example, a regular file "/etc/passwd" is a file; a directory "/home" is also a file. Every "file" *F* in EXT4 has a one-to-one corresponding *inode* structure named EXT4 inode (Poirier, 2019; The Linux Kernel Development Community, 2022), from which the authors can obtain all the blocks where *F* is stored.

Block Number and Inode Number

In EXT4, all blocks are globally numbered, starting from 0 and in turn incremented by 1, according to the order in which they appear on a hard disk. Similarly, all inodes are globally numbered, starting

from 1 and in turn incremented by 1, according to the order in which they appear on a hard disk. Besides, in EXT4, a block number is also called a block pointer, and an inode number is also called an inode pointer. The *block number* and *block pointer*, and the *inode number* and *inode pointer*, can be used equally.

Given any inode number, the authors can calculate to which block group the inode number belongs. Generally, without loss of generality, the given inode number may be inode_num; then, the block group descriptor number (blk_grp_num) to which it belongs, as well as the offset (offset_num) in the inode table, can be calculated using the following formulas:

blk_grp_num = inode_num / s_inodes_per_group (1)

offset_num = inode_num % s_inodes_per_group /* block group is numbered starting from 0 */ (2)

if (0 == offsetNum) blk_grp_num (3)

Poirier's (2019) and The Linux Kernel Development Community's (2022) work provide further information about the layout of an EXT4 file system.

Key Data Structures in an EXT4 File System

In this subsection, the authors will introduce some key data structures that are critical to file recovery.

Group Descriptor Structure

In the group descriptor structure (struct EXT4_group_desc) (Poirier, 2019; The Linux Kernel Development Community, 2022), for each block group, there is a one-to-one corresponding group descriptor in which management information of the block group is stored. For every group descriptor, there are three key pointers (in EXT4, "a pointer" is in fact "a block number," in this paper, the authors use two terms equivalently), namely, block bitmap, inode bitmap, and inode table:

```
struct EXT4_group_desc {
__u32 bg_block_bitmap;/* Block bitmapptr */
__u32 bg_inode_bitmap;/* Inode bitmap ptr*/
__u32 bg_inode_table;/* Inodes table ptr*/
    ......                  /* Can be ignored */
};
```

Among these pointers, the inode table pointer is essential to file recovery. All inode structures in a block group are consecutively and adjacently stored in the inode table. If the inode table is N_b, then starting from N_b, reading data sequentially in steps of 128 bytes (an inode structure is 128-byte long) (Poirier, 2019; The Linux Kernel Development Community, 2022), the authors can in turn obtain all inode structures in a block group.

Inode Structures

Inode structures are struct EXT4_dir_entry_2 and struct EXT4 inode (Poirier, 2019; The Linux Kernel Development Community, 2022).

Linux abstracts everything into "files." Every "file" has a unique inode number, and every inode number has a one-to-one corresponding inode structure. Among these inode structures, the directory inode structure EXT4_dir_entry_2 and the regular file inode structure EXT4_inode are vital to file recovery.

The directory inode structure EXT4_dir_entry_2 stores information about subdirectories or regular files in a directory. Concretely, every subdirectory or regular file in a directory has a corresponding EXT4_dir_entry_2 structure. This structure records the name, the inode number of the subdirectory or the regular file, and the length of the structure itself. During file recovery, the EXT4_dir_entry_2 structure is leveraged to recursively traverse all (sub)directories in an EXT4 file system to find all regular files as well as their inode numbers.

The regular file inode structure EXT4_inode is used to locate where a regular file exactly resides. Specifically, by reading the "__u32 i_block[15]" array within the EXT4_inode structure, the authors can acquire all of the numbers of blocks in which a regular file is exactly stored.

The directory inode structure is as follows:

```
struct EXT4_dir_entry_2{
      __u32 inode;         /* Inode number */
      __u16 rec_len;       /* Total length of the structure */
      __u8 name_len;       /* Name length */
      __u8 file_type;      /* 01 regular file; 02 directory */
      char name[EXT4_NAME_LEN];/* File name */
};
```

Any directory may contain subdirectories or regular files. Then, for whatever is inside a directory (i.e., whether it is a subdirectory or a regular file), it must have a corresponding EXT4_dir_entry_2 entry. The meaning of every member in an EXT4_dir_entry_2 structure is as follows:

1. **EXT4_dir_entry_2:** This is itself a variable-length structure, because names of different regular files and subdirectories may have different lengths. EXT4_dir_entry_2 uses "rec_len" to indicate the total length of the structure itself, and uses "name_len" and "name[EXT4_NAME_LEN]," respectively, to give the name length and the exact name of the regular file or subdirectory inside.
2. **File_type:** It gives the type of the EXT4_dir_entry_2 structure. If file_type = 02, then this is a subdirectory structure; if file_type = 01, then this is a regular file structure.
3. **Inode:** It gives the inode number of the subdirectory (if file_type = 02) or the regular file (if file_type = 01).

The regular file inode structure is as follows:

```
struct EXT4_inode {
     ......                /* Can be ignored */
     __u32 i_block[15];    /* file block pointer */
     ......                /* Can be ignored */
};
```

In the regular file inode structure EXT4 inode, the i_block[15] array stores all of the numbers of blocks occupied by a regular file. Specifically, in i_block[15]:

1. **i_block[0]~i_block[11]:** They are direct block pointers (block numbers). The direct block pointers store the numbers of blocks where file data are directly stored.
2. **i_block[12]:** It is an indirect block pointer. The indirect block pointer stores the number of a block where direct block pointers are stored; i_block[13] is a double indirect block pointer. The double indirect block pointer stores the number of a block where indirect block pointers are stored; i_block[14] is a triple indirect block pointer. The triple indirect block pointer stores

the number of a block where double indirect block pointers are stored. By dereferencing these (double/triple) indirect pointers, the authors can obtain the exact block numbers in which a file is stored.

In summary, for any file *f*, the authors can start from the root directory "/", and recursively analyze the EXT4_dir_entry_2 directory inode structure to obtain the inode number of *f*. Without loss of generality, supposing the inode number of *f* is f_i and then reading the EXT4_inode regular file inode structure of *f* based on f_i, the authors can obtain the i_block[15] array. Finally, converting the i_block[15] block numbers to their equivalent global linear sector addresses, the authors can obtain the entire data of *f* by directly reading these sectors, thereby recovering *F* in the absence of an OS. Below, Example 1 illustrates the above process.

Example: Suppose a regular file *f* named "/home/sec/passwd" is in the TOS. Try to recover *f* in the absence of an OS.

Solution: See below:

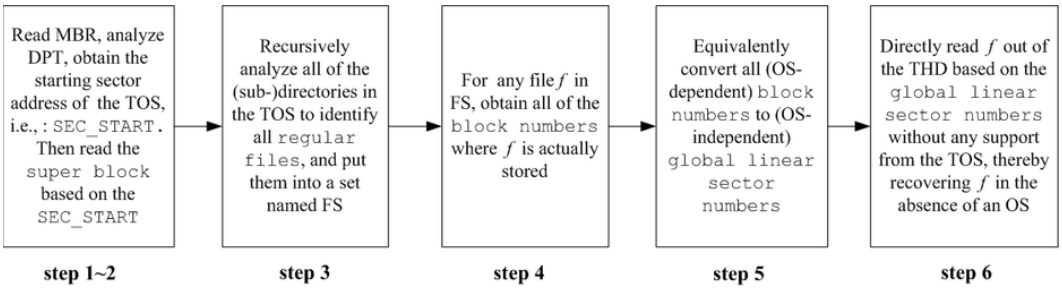
1. Initially, the inode number of the root directory "/" is fixed to 2 in Linux. Read the #2 inode structure to obtain all EXT4_dir_entry_2 entries in "/" (as the authors mentioned before, every subdirectory or regular file in "/" has its EXT4_dir_entry_2 entry), from which if name[EXT4_NAME_LEN] = "home," name_len = 4, and file_type = 02, then the inode is the inode number of the subdirectory "/home;" suppose the inode number is N_h.
2. Read the $\#N_h$ inode structure to obtain all EXT4_dir_entry_2entries in "/home," from which if name[EXT4_NAME_LEN] = "sec," name_len = 03, and file_type = 02, then the inode is the inode number of subdirectory "/home/sec;" suppose it is N_{hs}.
3. Read the $\#N_{hs}$ inode structure to obtain all EXT4_dir_entry_2 entries in "/home/sec," from which, if name[EXT4_NAME_LEN] = "passwd," name_len = 6, and file_type = 01, then the inode is the inode number of the regular file "/home/sec/passwd;" suppose it is N_{hsp}.
4. Note that N_{hsp} is the inode number of file *f*. Therefore, reading the $\#N_{hsp}$ inode structure, the authors can obtain the EXT4_inode structure of *F*. Then, reading the i_block[15] array in the EXT4_inode structure and converting the block numbers to their equivalent global linear sector addresses, the authors can obtain the entire data of *F* by directly reading these sectors, thereby recovering *F* in the absence of an OS.

SYSTEM DESIGN OF GOLDEN EYE

The example above gives the intuition of the authors' work. Figure 2 shows the detailed design of GE:

1. Firstly, the authors calculate the starting sector address of the TOS partition, that is, SEC_START.

Figure 2. System Design of GE



The SEC_START can be obtained by analyzing the DPT of the MBR ("Logical block addressing," 2022; Sedory, 2013). Generally, SEC_START has two main usages. One is to locate the super block (see step 2), and the other is to convert the OS-dependent block numbers to their equivalent OS-independent global linear sector numbers. Once the authors obtain the global linear sector numbers, they can read the file content out of the TOS in the absence of an OS (see step 5).

2. Then, they identify the super block.

The super block contains metadata of an EXT4 file system, which is essential to the interpretation of raw images in the file recovery process. The super block resides in a fixed position of a partition. Concretely, the super block is 1024 bytes away from SEC_START, and is 1024 bytes long. Therefore, by reading [SEC_START+2, SEC_START+3], a total of two sectors, the authors can obtain the super block structure EXT4_super block (Poirier, 2019; The Linux Kernel Development Community, 2022).

3. Next, the researchers identify all of the regular files as well as their corresponding inode numbers.

Starting from the root directory "/", and recursively analyzing the EXT4_dir_entry_2 structure to traverse all of the subdirectories in the TOS, they can finally identify all of the regular files as well as their corresponding inode numbers in the TOS.

For every regular file identified in the subdirectory traversing process, the authors put it into a special set called *FS* (regular File Set). This regular file identification process continues until all subdirectories in the TOS are traversed (and hence all regular files are also identified). Then, they repeat the following steps 4~7 to recover every regular file in the *FS*.

4. For any identified regular file *f* in *FS*, the researchers obtain all of the numbers of blocks in which *f* is stored, based on the inode number of *f* obtained in step 3.

As the authors mentioned before, the block numbers are stored in the i_block[15] array. Therefore, once the authors acquire the EXT4_inode structure of *f*, they can immediately obtain all of the block numbers that *f* occupies. They obtain the EXT4_inode structure of *f* as follows. Firstly, they calculate blk_grp_num and offset_num based on the inode number they obtained in step 3 by applying formulas (1), (2), and (3). Next, they read the #blk_grp_num block group descriptor structure, through which they can obtain the inode table pointer (i.e., bg_inode_table) (Poirier, 2019; The Linux Kernel Development Community, 2022). Finally, according to the offset in the inode table indicated by the offset_num, they can read out the exact EXT4_inode inode structure of *f*.

5. The researchers convert all of the block numbers obtained in step 4 to their equivalent global linear sector numbers.

The intuition of the conversion is straightforward. It is important to note that the SEC_START obtained in step 1 is the base address, and the block number is the offset to the SEC_START (Poirier, 2019; The Linux Kernel Development Community, 2022). Therefore, by adding the two, namely, SEC_START (base address) + block number (offset), the researchers immediately obtain the equivalent global linear sector numbers of every block number obtained in step 4.

6. The researchers directly read the data stored in the sectors, which are specified by the global linear sector numbers, out of the THD without any support from the TOS, thereby successfully recovering *f* in the absence of an OS.

7. They delete *f* from *FS*. If *FS* is not empty, they return to step 4. Otherwise, the whole recovery process terminates.

SYSTEM IMPLEMENTATION OF GOLDEN EYE

Figure 2 shows that GE consists of five main steps, and algorithm 1 details its implementation. Before explaining algorithm 1 in detail, the authors will briefly introduce data structures in GE:

1. **Recovered File Set (RFS):** RFS contains all of the regular files recovered by GE. When GE terminates, RFS is returned.
2. **SubDirectory Set (SDS):** SDS contains all of the subdirectories identified in the running of GE. The SDS is a first in first out (FIFO) queue so that GE can traverse all the subdirectories of the TOS in a breadth-first manner. The SDS is initialized to the root directory "/".
3. **Regular File Set (FS):** FS contains all of the regular files as well as their inode numbers in the running of GE. If all the subdirectories in the SDS are traversed, then all regular files in the TOS are sure to be identified and put into the FS. Each file *f* in the FS will be recovered and put into the RFS.
4. **Raw Image of a Disk (RD):** RD contains the raw image that is restored from a (sanitized) hard disk.
5. **Blocks Occupied Set (BOS):** When recovering any file *f* in the FS, BOS contains all of the numbers of blocks in which *f* is stored. BOS is a FIFO queue, and all block numbers of *f* are stored in strict order from head to tail according to the order that they compose *f*.
6. **Sectors Occupied Set (SOS):** When recovering a file *f*, all block number(s) in the BOS are converted to equivalent global linear sector addresses. SOS is a FIFO queue, and all sectors are stored in strict order from head to tail, according to the order that they compose *f*.

Algorithm 1: The Detailed Implementation of Golden Eye

```
Input: The RD
Output: The RFS

    Initialization.
    /* Get the partition address of the TOS. */
    SEC_START → GetTOSPartitionAddress(RD);
    /* Get the metadata, i.e., the super block, to interpret the raw image */
    sb →GetSuperBlock(SEC_START, RD);
    /* Analyze the super block, i.e., sb, to get the following key parameters */
    block_size;                  //the byte size of a block
    blocks_per_group;            //the number of blocks in a block group
    inodes_per_group;            //the number of inodes in a block group
    sectors_per_group;           //the number of sectors in a block
    /* Get the starting address of the  #0 (first) block group
    descriptor structure */
    grp_descrptr_struct_start_addr → SEC_START + 2 + sectors_per_block;
    /* Initialize the SDS (Sub Directory Set) to the root directory "/" */
    SDS → ∅;
    subdir.name  → "/";       //the root directory
    subdir.inode  → 2;        //the inode number of the
    root directory is fixed to 2
    SDS  → subdir;
    /* Initialize the FS (regular File Set to empty) */
```

```
19   FS  → ∅;
20   Body.
21   /* Find all regular files in the TOS */
22   FS  → FindAllRegularFiles(SDS, RD);
23   /* If there are still regular files in FS to be recovered */
24        while (FS ≠ ∅) do {
25        f → arbitrarily takes one regular file f out of the FS;
26        FS → FS − {f};
27        /* Find all of the blocks in which f is stored based on
          the inode number of f and then put all of the identified
          blocks into the BOS (Blocks Occupied Set) */
28        BOS → ∅;
29        BOS → FindBlocksOccupiedByInode(f.inode);
30        /* For every block number b in BOS, convert b to its
          equivalent global linear sector numbers, and then put
          into the SOS (Sectors Occupied Set) */
31        SOS → ∅;
32        SOS → ConvertBlkNums2GlobalLinearSecNums(BOS);
33        /* Directly read out f based on the OS-independent
          global linear sector addresses in SOS, and thereby
          recovering f in the absence of an OS */
34        f → RecoverSingleRegularFile(SOS);
35        RFS  → RFS + {f};
36   }    //end of while
37   Termination.
38   return RFS.
```

Explanation of the Golden Eye Algorithm

In the following, the authors will detail the GE algorithm.

Explanations to Lines 1-18 of Golden Eye (Steps 1~2 of Figure 2). This subsection details lines 2-3, 4-5, 6-12, and 13-19, respectively.

Lines 2-3. Firstly, it is necessary to derive the partition starting address of the TOS, that is, the SEC_START.

The partition starting address is stored in the partition table of the MBR. Every MBR has four partition entries, whose offsets (in bytes) in the MBR are, respectively, 0x01BE-0x01CD, 0x01CE-0x01DD, 0x01DE-0x01ED, and 0x01EE-0x01FD (Sedory, 2013). For each partition entry, the 0th byte indicates whether the partition is active (0x80 yes, 0x00 no); the 1st, 2nd, and 3rd bytes, respectively, give the starting head, sector, and cylinder (CHS) information of the partition. The CHS tuple can be equivalently converted to the global linear sector address based on the following formula ("Logical block addressing," 2022; Sedory, 2013):

$$GLSA = (C * HPC + H) * SPT + (S-1) \tag{4}$$

where C, H, and S can be directly read out from the MBR; HPC is the maximum number of heads per cylinder; SPT is the maximum number of sectors per track. HPC and SPT are reported by a hard disk, but they are usually 16 and 63, respectively, for a modern hard disk. Generally, it is possible to use 16 and 63 as default.

Substituting CHS, HPC = 16, and SPT = 63 into formula (4), the authors obtain SEC_START = GLSA.

Lines 4-5. Then, the GE algorithm reads the metadata of EXT, that is, the super block.

The super block always resides in a fixed location (i.e., it is of 1024-byte offset to the SEC_START, and is 1024-byte long). Therefore, it is possible to get the super block by reading the following two consecutive sectors: [SEC_START + 2, SEC_START + 3].

Lines 6-12. Next, the GE algorithm initializes global parameters that are used to interpret the raw image.

Lines 6-10 can be directly read out from the super block (Poirier, 2019; The Linux Kernel Development Community, 2022).

In lines 11-12, the first (i.e., #0) block group descriptor is adjacently and consecutively stored to the super block. It is important to note that the super block is 1024 bytes (i.e., two sectors) offset to the SEC_START, and the super block must occupy a whole block (i.e., sectors_per_block number of sectors); therefore, line 12 gives the exact starting global linear sector address of the structure of block #0 group descriptor.

Lines 13-19. the GE algorithm initializes the SDS and FS.

Lines 13-17 initialize the SDS to the root directory "/". In line 21, GE iteratively traverses the whole subdirectories starting from "/" in a breadth-first manner to identify all regular files in the TOS. The authors will introduce details about SDS in the following explanation to line 21.

Explanations to Lines 20-22 of Golden Eye (Step 3 of Figure 2). This subsection details lines 20-22.

Lines 21-22. Starting from the root directory "/", the function FindAllRegularFiles traverses all subdirectories to identify all regular files to be recovered in the TOS.

FindAllRegularFiles achieves this through a while loop in a breadth-first manner. Specifically, FindAllRegularFiles continues performing the following operations until SDS is empty. Firstly, FindAllRegularFiles takes out a subdirectory subdir from the head of SDS. Then, it reads out the EXT4_dir_entry_2 structure based on the inode number of subdir. Next, it analyzes the EXT4_dir_entry_2 structure of subdir to identify all subdirectories and regular files inside the subdir: If a new subdirectory *s* is identified, put *s* to the tail of SDS for the future traversal by FindAllRegularFiles; if a new regular file *f* is identified, put *f* into FS.

Explanations to Lines 23-29 of Golden Eye (Step 4 of Figure 2). This subsection details lines 23-26 and 27-29, respectively.

Lines 23-26. The FS currently contains all of the regular files to be recovered in the TOS. Line 24 uses a while loop to recover files one by one from FS until FS is empty.

Lines 27-29. For each regular file f to be recovered, GE invokes the function FindBlocksOccupiedByInode to obtain all of the blocks in which f is stored.

Firstly, FindBlocksOccupiedByInode reads out the EXT4_inode structure of f. Then, as the authors mentioned before, the i_block[15] array in the EXT4_inode structure contains all of the numbers of blocks that *f* occupies, and therefore, in turn, puts i_block[0], ..., i_block[14] into BOS; BOS finally contains all of the block numbers that make up f. It is important to note that i_block[12], i_block[13], and i_block[14] are, respectively, indirect, double indirect, and triple indirect pointers. These indirect pointers must be dereferenced to derive the real block pointers. Finally, ReadInodeStructureByInodeNumber returns BOS and terminates

Explanations of Lines 30-32 of Golden Eye (Step 5 of Figure 2). This subsection details lines 30-32.

Lines 30-32. BOS is a FIFO queue, and block numbers are in turn stored from the head to the tail of BOS in the order that they compose f. Therefore, Line 31 invokes ConvertBlkNums2GlobalLinearSecNums, which, in turn, from the head to the tail of BOS, takes out a block number block_num and converts the block_num to its equivalent global linear sector addresses.

It is known there are " sectors_per_block" number of sectors in a block (Line 10 of GE). Therefore, for every block_num taken out from BOS, once the GE algorithm acquires the starting sector address of block_num, without loss of generality, if it is the blk_2_sec_start_addr, then starting

from blk_2_sec_start_addr, reading the adjacent and consecutive " sectors_per_block " number of sectors, the GE algorithm can obtain the equivalent global linear sector addresses of block_num.

The block_num is a block-counted offset to SEC_START, whose equivalent sector-counted offset is block_num * sectors_per_block. Therefore, the global linear sector addresses of block_num are: Starting from the sector whose number is blk_2_sec_start_addr = SEC_START + block_num * sectors_per_block, the adjacent and consecutive sectors_per_block number of sectors, namely, [blk_2_sec_start_addr, blk_2_sec_start_addr+sectors_per_block-1].

Explanations of Lines 33-36 of Golden Eye (Step 6 of Figure 2). This subsection details lines 33-35 and 36, respectively.

Line 33-35. As the authors mentioned before, SOS is a FIFO queue. All sectors are in turn stored from head to tail of the SOS based on the order in which they compose f. Therefore, by directly reading the data from sectors that are in turn stored from head to tail of SOS, it is possible to recover f. The recovered f is then put into the RFS.

Line 36. When all the regular files are covered (FS is empty; line 24 of GE), GE returns RFS and terminates.

EXPERIMENTAL RESULTS

As the authors mentioned previously, it is theoretically possible to recover the raw image from an (even sanitized) hard drive by using modern techniques such as magnetic force microscopy (Kanekal, 2013). However, how to obtain the raw image based on magnetic force microscopy techniques is beyond the scope of this paper; Kanekal's (2013) study provides relevant details.

In their experiments, the authors formed the raw image by directly copying each sector from the THD using a hard disk sector copying tool. Then, they used the copied raw image for manual parsing, as well as automatic GE parsing, to verify the feasibility and correctness of GE (see Appendix), and to compare with existing work (Table 1). In addition, they presented a practical application of GE for sensitive data protection (Figure 3).

Feasibility and Correctness Experiments

In this section, the authors give a running example of GE. Specifically, they arbitrarily select a regular file *f* and "manually recover" it based on GE without any support from the OS.

The selected regular file *f* is "/home/hky/getblk" (it does not matter to select which file *f* and the recovery process is similar). The researchers first analyze the MBR to identify the starting address of the TOS partition (i.e., SEC_START), and then read out the metadata (i.e., the super block). Next, starting from the root directory "/", they iteratively analyze the EXT4_dir_entry_2 structure to obtain the inode number of *f* and, based on it, they read out the EXT4_inode inode structure of *f*. Finally, the block number(s) occupied by *f* are stored in the i_block[15] array within the EXT4_inode inode structure of *f*, and thus the authors convert the block(s) in i_block[15] to equivalent global linear sector address(es). By reading data stored in the global linear sector address(es), the file *f* can be recovered.

Due to space limitations, the authors detailed the "manual recovery process" in the Appendix. The experiment showed that the authors' recovery result is the same as what was the standard Linux file system tool debugfs returned, proving the feasibility and correctness of GE.

Comparisons With Existing Work

The authors compared their work with existing work in Table 1. In Table 1, Jo et al.'s (2018), Kim et al.'s (2021), Lee et al.'s (2020), Lee and Shon's (2014), and the authors' work are pure MFR approaches; Garfinkel's (2007), Garfinkel and McCarrin's (2015), Gladyshev and James's (2017), Golden and Vassil's (2005), Hand et al.'s (2012), Pal et al.'s (2003), Pal et al.'s (2008), and Tang et al.'s (2016) work are pure CFR approaches; Dewald and Seufert's (2017), Nordvik et al.'s (2020), and

Portera et al.'s (2021) work are mixed approaches that utilize both MFR and CFR technologies. The difference between the authors' work and Jo et al.'s (2018), Kim et al.'s (2021), Lee et al.'s (2020) and Lee and Shon's (2014) research is that the authors' work recovers files without any support from an OS, while the the others cannot.

The explanations of the comparison results in Table 1 are as follows:

1. **Efficiency:** As the authors mentioned before, CFR approaches must leverage syntax/semantics information and heuristic methods to indirectly infer which file fragmentations (file blocks) belong to the same file to recover it. Since inference is a time-consuming process, it makes pure CFR approaches slower (L) than pure MFR approaches (H).

Table 1. Comparisons With Existing File Recovery Work

Approach	Efficiency	No false positive	No false negative	OS-independent	Algorithm	Support sanitized devices	Support types of files
Kim et al. (2021)	H	✓	✓	×	×	×	Any
ExtSFR (Lee et al., 2020)	H	✓	✓	×	×	×	Any
Jo et al. (2018)	H	✓	✓	×	×	×	Any
Lee and Shon (2014)	H	✓	✓	×	×	×	Any
Gladyshev and James (2017)	L	×	×	×	✓	×	Image
Tang et al. (2016)	L	×	×	×	✓	×	Image
Garfinkel & McCarrin (2015)	L	×	×	×	✓	×	Known files with hash
BinCarver (Hand et al., 2012)	L	×	×	×	✓	×	Binary
Pal et al. (2008)	L	×	×	×	✓	×	Image
Scalpel (Garfinkel, 2007)	L	×	×	×	✓	×	Files with built-in signatures
Foremost (Golden & Vassil, 2005)	L	×	×	×	✓	×	Files with built-in signatures
Pal et al. (2003)	L	×	×	×	✓	×	Files with built-in signatures
Portera et al. (2021)	L	×	×	✓	✓	✓	Any
Nordvik et al. (2020)	L	×	×	✓	✓	✓	Any
AFEIC (Dewald & Seufert, 2017)	L	×	×	×	×	×	Any
Heo et al. (2019)	L	×	×	✓	✓	✓	Audio
The authors' approach GE	H	✓	✓	✓	✓	✓	Any

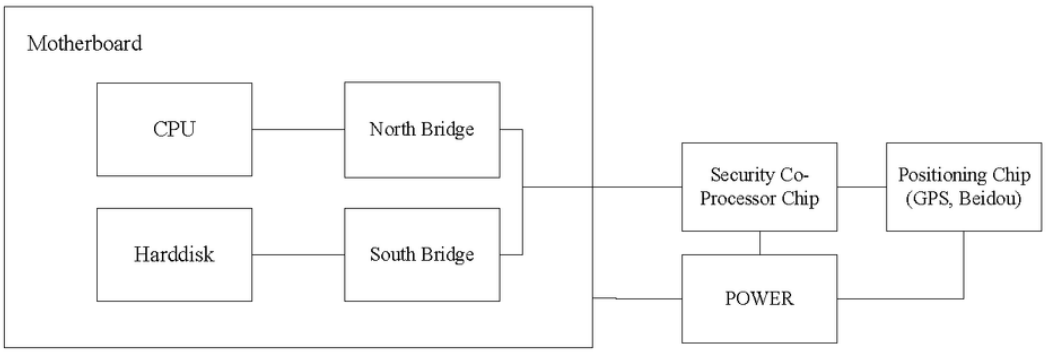
2. **False-Positive and False-Negative:** MFR approaches leverage the metadata to guide the file recovery process; hence, MFR can precisely interpret every bit of data to be recovered. Therefore, there are no false positives or false negatives for pure MFR approaches. Conversely, CFR approaches recover files based on indirect inference methods; hence, false positives and false negatives are inevitable. Heo et al. (2019) recovered audio clips to sounds recognizable to the human ear based on deep learning techniques, rather than accurately recovering the original audio file, and thus their work also suffered from false positives and false negatives.
3. **OS Independence:** The implementations of Jo et al. (2018), Kim et al. (2021), Lee et al. (2020), and Lee and Shon (2014) are on top of the existence of an OS, and, hence, they are OS-dependent. Dewald and Seufert's (2017), Garfinkel's (2007), Garfinkel and McCarrin's (2015), Gladyshev and James's (2017), Golden and Vassil's (2005), Hand et al.'s (2012), Pal et al.'s (2003), Pal et al.'s (2008), and Tang et al.'s (2016) studies are block-based to identify file (fragmentation) boundaries to carve files. It is important to note that block is a file system concept, therefore it must run in the presence of an OS.
4. **Algorithm Implementation:** Garfinkel (2007), Garfinkel and McCarrin (2015), Gladyshev and James (2017), Golden and Vassil (2005), Hand et al. (2012), Pal et al. (2003), Pal et al. (2008), and Tang et al. (2016) also introduced in detail file recovery algorithms as the authors of this study did. However, unlike GE, which can recover any type of file based solely on raw images, each algorithm has its drawbacks. Specifically, files that can be recovered are limited to specific types, such as images (Gladyshev & James, 2017; Pal et al., 2008; Tang et al., 2016), known files (i.e., files whose hashes have already been put into the hash database in advance) (Garfinkel & McCarrin, 2015), binary files (Hand et al., 2012), and built-in types of files that have specific headers, footers, and internal data structures (Garfinkel, 2007; Golden & Vassil, 2005; Pal et al., 2003). In addition, none of the algorithms presented can recover files from sanitized devices as GE does.

Golden Eye Application: Location-Based Sensitive Information Protection System

The goal of the location-based based sensitive information protection system is to ensure that sensitive data in the authors' secure computer can only exist in a designated security zone. Once the security computer is out of the security zone, sensitive data in the security computer must be immediately and automatically deleted (regardless of whether the system is powered on or not).

Figure 3 shows the architecture of the authors' security computer. The motherboard is specially designed to interface with a security coprocessor. There is a positioning chip (e.g., GPS and Beidou) inside the case. When the security computer is powered off, the motherboard, the security coprocessor, and the positioning chip are powered by the built-in power supply. The security zone is written into the

Figure 3. The Architecture of the Authors' Security Computer



security coprocessor in advance. The positioning chip continuously monitors the current coordinates of the security computer and sends it to the security coprocessor. Once the security coprocessor detects that the security computer is out of the security zone, it loads the GE code (here GE receives the absolute path name of sensitive files and returns their corresponding global linear sector addresses) into memory and transfers control to the CPU. The CPU generates a noise-based random number as the key to encrypt all the sectors returned by GE, and then permanently discards the key, thereby implementing the CE (Richard et al., 2014) to sensitive files. Since the security computer is powered off, the authors' algorithm GE takes the place of the file system to locate arbitrary sensitive files on the hard disk and deletes them cryptographically in the absence of an OS.

CONCLUSION

Aiming at the "OS-dependence" problems from which most existing file recovery investigations suffer, the authors proposed an OS-independent file recovery algorithm GE. So long as given the raw image of a hard disk, as well as the global linear sector addresses of the raw image, GE can automatically recover files in the absence of an OS.

The authors' work not only solved the OS-dependence problem, but, and together with the research that acquires on-disk raw images from sanitized storage devices (Kanekal, 2013), also revealed the fact that the risk of sensitive files being recovered from sanitized hard disks does exist. Therefore, regarding any high-security risk storage device, the authors suggest the CE (i.e., firstly encrypt the full storage device, and then securely and permanently discard the encryption key) be enforced first before applying other sanitization approaches recommended in the NIST 800-88.

The authors' approach shows some limitations as well. For example, the extent to which files can be recovered depends on the raw image obtained from the THD. In general, the more complete the raw image is, the more files GE can successfully recover. Particularly, if the full raw image of the THD is given, then GE can successfully recover the whole file system of the THD. How to get a better raw image from a hard-disk platter is beyond the authors' study.

In the future, the authors will extend their work to mobile devices. They will explore the OS-independent file recovery approaches for EXT4 and Yaffs2, and apply them to areas such as intelligence acquisition, digital forensics, file rescue, and trust measurement.

ACKNOWLEDGMENT

This work was supported by the Hubei Provincial Natural Science Foundation of China (2020CFB761), the Research and Innovation Initiatives of WHPU (2021Y38) and Digital Media Arts Academic Team of Wuhan Business University(No.2020TD001).

REFERENCES

Aine, M., Thar, B., Paul, B., Farkhund, I., & Qi, S. (2020). The Internet of Things: Challenges and considerations for cybercrime investigations and digital forensics. *International Journal of Digital Crime and Forensics, 12*(1), 1-13. 10.4018/IJDCF.2020010101

Dewald, A., & Seufert, S. (2017). AFEIC: Advanced forensic EXT4 inode carving. *Digital Investigation*, *20*, S83–S91. doi:10.1016/j.diin.2017.01.003

Fairbanks, K. D. (2012). An analysis of EXT4 for digital forensics. *Digital Investigation*, *9*, S118–S130. doi:10.1016/j.diin.2012.05.010

Garfinkel, S. L. (2007). Carving contiguous and fragmented files with fast object validation. *Digital Investigation*, *4*, S2–S12. doi:10.1016/j.diin.2007.06.017

Garfinkel, S. L., & McCarrin, M. (2015). Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb. *Digital Investigation*, *14*, S95–S105. doi:10.1016/j.diin.2015.05.001

Gladyshev, P., & James, J. I. (2017). Decision-theoretic file carving. *Digital Investigation*, *22*, 46–61. doi:10.1016/j.diin.2017.08.001

Golden, G. R. III, & Vassil, R. (2005). Scalpel: A frugal, high performance file carver. In *Proceedings of the 5th Annual Digital Forensic Research Workshop* (pp. 1-10). ACM Press.

Hand, S., Lin, Z., Gu, G., & Thuraisingham, B. (2012). Bin-Carver: Automatic recovery of binary executable files. *Digital Investigation*, *9*, S108–S117. doi:10.1016/j.diin.2012.05.014

Heo, H.-S., So, B.-M., Yang, I. L.-H., Yoon, S.-H., & Yu, H.-J. (2019). Automated recovery of damaged audio files using deep neural networks. *Digital Investigation*, *30*, 117–126. doi:10.1016/j.diin.2019.07.007

Hughes, G., & Coughlin, T. (2006). *Tutorial on disk driver data sanitization*. https://cmrr.ucsd.edu/_files/data-sanitization-tutorial.pdf

Hughes, G., Coughlin, T., & Commins, D. M. (2009). Disposal of dis and tape data by secure sanitization. *IEEE Security and Privacy Magazine*, *7*(4), 29–34. doi:10.1109/MSP.2009.89

Jo, W., Chang, H., & Shon, T. (2018). Digital forensic science approach by file recovery research. *The Journal of Supercomputing*, *74*(8), 3704–3725. doi:10.1007/s11227-016-1909-2

Kanekal, V. (2013). *Data reconstruction from a hard disk drive using magnetic force microscopy* [Master thesis]. University of California, San Diego. https://escholarship.org/uc/item/26g4p84b

Kim, H., Kim, S., Shin, Y., Jo, W., Lee, S., & Shon, T. (2021). EXT4 and XFS file system forensics framework based on TSK. *Electronics (Basel)*, *10*(18), 2310. doi:10.3390/electronics10182310

Kissel, R., Regenscheid, A., Scholl, M., & Stine, K. (2014). *NIST special publication 800-88 (Revision 1): Guidelines for media sanitization*. National Institute for Standards and Technology, U.S. Department of Commerce. doi:10.6028/NIST.SP.800-88r1

Lee, S., Jo, W., Eo, S., & Shon, T. (2020). ExtSFR: Scalable file recovery framework based on an Ext file system. *Multimedia Tools and Applications*, *79*(23-24), 16093–16111. doi:10.1007/s11042-019-7199-y

Lee, S., & Shon, T. (2014). Improved deleted file recovery technique for Ext2/3 file system. *The Journal of Supercomputing*, *70*(1), 20–30. doi:10.1007/s11227-014-1282-y

Logical block addressing. (2022). In *Wikipedia*. https://en.wikipedia.org/wiki/Logical_block_addressing

Mohammad, R. M. A., & Alqahtani, M. (2019). A comparison of machine learning techniques for file system forensics analysis. *Journal of Information Security and Applications*, *46*, 53–61. doi:10.1016/j.jisa.2019.02.009

Nordvik, R., Portera, K., Toolan, F., Axelssona, S., & Franke, K. (2020). Generic metadata time carving. *Forensic Science International: Digital Investigation*, *33*, S1–S10. doi:10.1016/j.fsidi.2020.301005

Pal, A., Sencar, H. T., & Memon, N. (2008). Detecting file fragmentation point using sequential hypothesis testing. *Digital Investigation*, *5*, S2–S13. doi:10.1016/j.diin.2008.05.015

Pal, A., Shanmugasundaram, K., & Memon, N. (2003). Automated reassembly of fragmented images. In *Proceedings of the 2003 International Conference on Multimedia and Expo*. IEEE Press. doi:10.1109/ICME.2003.1220995

Poirier, D. (2019). *The second extended file system internal layout*. https://www.nongnu.org/ext2-doc/ext2.html

Portera, K., Nordvikab, R., Toolanb, F., & Axelssonac, S. (2021). Timestamp prefix carving for filesystem metadata extraction. *Forensic Science International: Digital Investigation*, *38*, 1–13. doi:10.1016/j.fsidi.2021.301266

Sedory, D. B. (2013). *MBR/EBR partition tables*. https://thestarman.pcministry.com/asm/mbr/PartTables.htm

Tang, Y., Fang, J., Chow, K. P., Yiu, S. M., Xu, J., Feng, B., Li, Q., & Han, Q. (2016). Recovery of heavily fragmented JPEG files. *Digital Investigation*, *18*, S108–S117. doi:10.1016/j.diin.2016.04.016

The Linux Kernel Development Community. (2022). *Ext4 data structures and algorithms*. https://www.kernel.org/doc/html/latest/filesystems/ext4/index.html

Zhang, Q., Jia, S., Chang, B., & Chen, B. (2018). Ensuring data confidentiality via plausibly deniable encryption and secure deletion: A survey. *Cybersecurity*, *1*(1), 1–20. doi:10.1186/s42400-018-0005-8

APPENDIX

Without loss of generality, suppose the targeting file *f* is "/home/hky/getblk." In the following, the authors will illustrate how to manually obtain its global linear sector addresses based on their algorithm GE.

Step 1: Read MBR to calculate SEC_START; read super block. The offsets of 0x01BE-0x01CD, 0x01CE-0x01DD, 0x01DE-0x01ED, and 0x01EE-0x01FD in the MBR are four DPT entries. If the 0th byte of the DPT entry is 0x80, then this is an active partition. If the 0th byte of the DPT entry is 0x00, then this is an inactive partition.

The following shows the four DPT entries in the authors' experiment hard disk. Their CentOS occupies the first DPT entry (bold font):

```
80 20 21 00 83 93 29 2a -- 00 08 00 00 00 68 0a 00
00 b4 09 2a 05 fe ff ff -- fe 77 0a 00 02 80 75 02
00 00 00 00 00 00 00 00 -- 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 -- 00 00 00 00 00 00 00 00
```

An analysis of the DPT entry evidences that the 1st~3rd bytes, namely, 20 21 00, give the starting Head, Sector, and Cylinder, respectively. Therefore, the CHS tuple is (0x00, 0x20,0x21). Substituting the C = 0, H = 32, S = 33, HPC = 16, and SPT = 63 into formula (4), the result is:

$$\text{SEC\_START} = \text{GLSA} = (0 * 16 + 32) * 63 + (33 - 1) = 2048 \tag{5}$$

It is necessary to read the [SEC_START * 512 + 1024, SEC_START * 512 + 2047] bytes, a total of 1024 bytes, to obtain the super block, which provides the following critical parameters of EXT4 files system: The number of inodes per group is s_inodes_per_group = 0x7fc0 = 32704 and the size of a block is block_size = 4096 bytes. Hence, this corresponds to blk_size_N = 4096/512 = 8 in the authors' experiment, that is, every 8 sectors form a block.

Step 2: Obtain the block number(s) of *f* and equivalently convert to corresponding global linear sector address(es). In the following, the authors manually obtain all block(s) occupied by *f* and then convert them to global linear sector address(es). Importantly, the data are in Little-Endian format.

1. Read the inode structure of the root directory '/'

Firstly, the inode structure of '/' is fixed to 2 in Linux, and the authors have obtained the number of inodes per group s_inodes_per_group = 32704. Then, the calculation is:

```
blk_grp_num = inode_num / s_inodes_per_group;
Offset_num = inode_num % s_inodes_per_group;
if (0 == OffsetNum) blk_grp_num--;
the result is: blk_grp_num = 0, offset_num = 2.
```

Therefore, the EXT4_dir_entry_2 inode structure of inode #2 is stored in block group #0 (blk_grp_num = 0) and is the second inode structure in the inode table (offset_num = 2). The EXT4_group_desc block group descriptor structure of block group descriptor #0 has the following (every group descriptor structure is fixed to 32-byte long in EXT4):

```
01 04 00 00 02 04 00 00 -- 03 04 00 00 73 78 ea 1f
02 00 04 00 00 00 00 00 -- 00 00 00 00 00 00 00 00
```

The inode table of block group 0 is stored in the block (bold font): 0x00000403=1027.

In the block #1027, its global byte offsets in the hard disk is: ranging from "starting_byte = (SEC_START + 1027 * blk_size_N) * 512 " to " ending_byte = (SEC_START + 1028 * blk_size_N) * 512 ". Since offset_num = 2, then skipping the first inode structure (i.e., skipping 128 bytes from the beginning of block #1027, because every inode structure is fixed to 128 bytes in EXT4) to read the second inode structure, the result is:

```
ed 41 00 00 00 10 00 00 -- 55 69 03 5d 53 69 03 5d
53 69 03 5d 00 00 00 00 -- 00 00 17 00 10 00 00 00
00 00 00 00 00 00 00 00 -- 01 08 00 00 00 00 00 00
00 00 00 00 00 00 00 00 -- 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 -- 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 -- 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 -- 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 -- 2a 90 00 00 00 00 00 00
00 00 00 00 00 00 00 00 -- 00 00 00 00 00 00 00 00
```

The i_block[15] array data are in bold. The "/" occupies only one block i_block[0] = 0x00000801 = 2049 (0x00000000 indicates that the block is not used).

2. Analyze the EXT4_dir_entry_2 structure of the root directory "/" and obtain the inode number of the subdirectory "/home".

Specifically, if the the block is #2049, a total of 4096 bytes (because the block size is 4096 bytes from the super block), the EXT4_dir_entry_2 structure is "/".The authors' experiment OS includes a total of 12 subdirectories and regular files in the root directory "/". Therefore, there are correspondingly 12 continuous and adjacent EXT4_dir_entry_2 structures stored in the EXT4_dir_entry_2 structure of "/". For the sake of simplicity, the authors only list the "home" subdirectory-related EXT4_dir_entry_2 structure in the following:

01 fb 09 00 0c 00 **04** -- 68 6f 6d 65

The first 4 bytes (grey background with italic) indicate the inode number of "/home": inode = 0x0009fb01 = 654081.

The following 2 bytes (underlined font) indicate the exact length of the EXT4_dir_entry_2 structure of "/home": rec_len = 0x000c = 12, a total of 12 bytes. As the authors mentioned before, every EXT4_dir_entry_2 is a variable-length structure. The exact length of every EXT4_dir_entry_2 structure is indicated by the 5th and 6th bytes of the structure, that is, the underlined font "0c 00" in the above example.

The 7th byte (bold font) indicates the name length of the subdirectory: name_len = 0x04 = 04, which equals the length of "home".

The 8th byte indicates the file (with a box around) indicates the file type: file_type = 0x02 = 02, which means this is a subdirectory.

The last four bytes indicate the exact name of the subdirectory. Specifically, the 0x68, 0x6f, 0x6d, and 0x65 are, respectively, the ASCII codes of characters 'h', 'o', 'm', and 'e'. Therefore, this is exactly the subdirectory "home" that the authors are looking for, and thus the inode number of "/home" is 654081.

3. Obtain the inode number of "/home/hky/getblk".

Similarly, it is necessary to repeat steps (1) and (2) to recursively parse the "/home/hky/getblk" and finally obtain the inode number of "/home/hky/getblk". The process is as follows.

Firstly, it is necessary to read the inode structure 654081# (as step b does) and analyze the EXT4_dir_entry_2 structure of "/home/" to obtain the inode number of the subdirectory "/home/hky" (as step b does). The result is that the inode number of subdirectory "/home/hky" is 654082.

Then, it is necessary to read the inode structure 654082# (as step b does) and analyze the EXT4_dir_entry_2 structure of "/home/hky" to obtain the inode number of the regular file "/home/hky/getblk" (as step b does). The result is that the inode number of the regular file "/home/hky/getblk" is 654151.

4. Obtain the block number(s) of "/home/hky/getblk"

The following have to be calculated with inode_num = 654151 and s_inodes_per_group = 32704:

```
blk_grp_num = inode_num / s_inodes_per_group;
Offset_num = inode_num % s_inodes_per_group;
if (0 == OffsetNum) blk_grp_num--;
The result is: blk_grp_num = 20, offset_num = 71.
```

When reading the EXT4_group_desc structure of group descriptor 20#, the result is:

```
00 00 0a 00 01 00 0a 00 -- 02 00 0a 00 9d 7a 1c 7f
41 00 00 00 00 00 00 00 -- 00 00 00 00 00 00 00 00
```

The inode table of the group descriptor 20# is (in bold font): bg_inode_table = 0x000a0002 = 655362.

In the block 655362#, because offset_num = 71, skipping the former 70 inode structures and reading the 71st inode structure, the result is:

```
fd 41 f4 01 00 10 00 00 -- b6 35 05 5d 20 50 f1 5c
20 50 f1 5c 00 00 00 00 -- f4 01 02 00 10 00 00 00
00 00 00 00 00 00 00 00 -- 07 70 0a 00 00 00 00 00
00 00 00 00 00 00 00 00 -- 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 -- 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 -- 00 00 00 00 00 00 00 00
00 00 00 00 fc a1 ec 24 -- 02 04 0a 00 00 00 00 00
00 00 00 00 00 00 00 00 -- 00 00 00 00 00 00 00 00
```

Read the i_block[15] (in bold font) and we have: the targeting regular file "/home/hky/getblk" occupies only one block, i.e., i_block[0] = 0x000a7007 = 684039.

This result is the same as the result returned by the standard Linux file system tool debugfs.

5. Convert the block number to global linear sector addresses.

In the authors' experiment platform, a block contains blk_size_N = 8 sectors. Therefore, for any block number nblock, the nblock corresponds to the following consecutive and adjacent sectors: Starting from the sector file_secs_start = SEC_START + nblock * blk_size_N, and consecutive blk_size_N = 8 sectors. Consequently, for i_block[0] = nblock = 684039, the nblock corresponds to

the following sectors: Starting from the sector with the number file_secs_start =2048 + 684039 * 8 = 5474360 and 8 consecutive sectors, namely, sectors of [5474360, 5474367].

This is the global linear sector address of the file "/home/hky/getblk".

By reading these 8-sector data, the authors manually obtain the file content of "/home/hky/getblk" without any help from additional hardware or software. Furthermore, the block information that they "manually" obtained is exactly consistent with the result returned by the standard file system tool debugfs. Therefore, the authors algorithm is correct.

Manual analysis illustration ends.

Fan Zhang received his Ph.D. degree in Computer Science from Wuhan University, in 2009. He is currently an associate professor at the Mathematics and Computer Science School, Wuhan Polytechnic University. His research interests include information system security, vulnerability mining, and artificial intelligence security.

Wei Chen received his Ph.D. degree from Wuhan University, in 2005. He is currently a professor in the Nanjing University of Posts and Telecommunications. His research interest is in the field of information security, including Web security, IoT security, and mobile system security.

Yongqiong Zhu (corresponding author) received her Ph.D. degree in Computer Science from Wuhan University, in 2013. She is currently an associate professor at the Art of School, Wuhan Business University. Her research interests include information system security, digital forensics, and artificial intelligence.