Unifying metadata-based storage reconstruction and carving with LAYR

Janine Schneider, Hans-Peter Deifel, Stefan Milius, Felix Freiling

Angaben zur Veröffentlichung / Publication details:

Schneider, Janine, Hans-Peter Deifel, Stefan Milius, and Felix Freiling. 2020. “Unifying metadata-based storage reconstruction and carving with LAYR.” *Forensic Science International: Digital Investigation* 33 (Supplement): 301006. https://doi.org/10.1016/j.fsidi.2020.301006.

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS 2020 USA — Proceedings of the Twentieth Annual DFRWS USA

Unifying Metadata-Based Storage Reconstruction and Carving with LAYR

Janine Schneider, Hans-Peter Deifel, Stefan Milius, Felix Freiling*

Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Martensstr. 3, 91058, Erlangen, Germany

ARTICLE INFO

Article history:

ABSTRACT

Storage resources are usually organized in abstraction layers in computing systems where higher level storage (e.g. files or file systems) is constructed from lower level storage (e.g. disk volumes). Many forensic storage reconstruction techniques exist that gather data at lower layers and interpret this data to reconstruct higher layers. On the one hand, there are *metadata-based* reconstruction techniques that interpret metadata structures to precisely reconstruct upper layer content. On the other hand, there are *pattern-based* techniques (*carving*) that focus mainly on deleted files that cannot be reconstructed by other methods. Instances resembling the former approach are Carrier's *The Sleuth Kit* (TSK) as well as many commercial tools, while the latter approach is used by file carvers like Foremost and Scalpel. Based on a formalization of storage abstraction layers, we show that all these techniques can be unified within a modular reconstruction framework. We define composition operators that allow to precisely express complex reconstruction tasks that involve both metadata-based and pattern-based techniques and allow to combine their respective strengths seamlessly in forensic analysis. We present LAYR, an implementation of our approach and show that it can automatically and reliably combine different reconstruction approaches.

1. Introduction

Digital evidence comes in many different forms, be it video files stored on a hard disk, e-mails sent over a network or passwords stored in a computer's main memory. It is common practice that investigators acquire this evidence by accessing it *directly from the device*, i.e. via the device driver or other low levels of access to storage. Such low level access is considered to minimize the dangers of unintentional errors in evidence analysis that can occur if, for example, file system drivers are unreliable or there exist ambiguities in file system metadata (like multiple files with the same name).

The task to reconstruct the concrete pieces of high level evidence from the collected low level evidence has been observed to be non-trivial since it involves decoding the mapping between pieces of data on both layers and to bridge the semantic gap (Jain et al., 2014; Carrier, 2002). For example, in NTFS, a central data structure called the Master File Table (MFT) must be found and parsed to extract file metadata pointing to the starting and ending sectors of an active file [3, Chapter 11]. Metadata-based reconstruction of active content has the advantage of being *precise*: recovered content is bound to the current instance of the file system and cannot be denied. Many well-developed tools exist, e.g. a tool suite called *The Sleuth Kit* (TSK) (Carrier, 2020a) allows to reconstruct files for a multitude of files systems such as NTFS, Ext and FAT.

If the file has been deleted but the metadata is still present in the MFT, then a file might still be reconstructed as long as the sectors have not been reallocated for a different file yet. While this approach is also based on metadata, it is less precise than recovery of active content. However, TSK, for example, can reconstruct recently deleted files for Ext2 and NTFS with high precision. In file systems where block pointers are overwritten, reconstruction can be performed heuristically, e.g. in FAT if no fragmentation is assumed [3, Chapter 9].

Even if no metadata exists at all, files may still be reconstructed through a technique called *file carving* (Pal and Memon, 2009). The idea of carving is to heuristically search for typical patterns that occur at the beginning, within or at the end of files to assemble its original block sequence (Richard and Roussev, 2005; Memon and

* Corresponding author.
E-mail addresses: janine.schneider@fau.de (J. Schneider), hans-peter.deifel@fau.de (H.-P. Deifel), stefan.milius@fau.de (S. Milius), felix.freiling@fau.de (F. Freiling).

https://doi.org/10.1016/j.fsidi.2020.301006

Pal, 2006). This method is less precise than metadata-based reconstruction, i.e. it provides less information about the circumstances of the recovered file, but it is more *complete* in the sense that there is high probability that evidence is found even if it was deleted. Carving can also be used to find *metadata* structures of deleted partitions or file systems (Dewald and Seufert, 2017), thereby enabling metadata-based file reconstruction in even more cases. Common file carving tools (Richard and Roussev, 2005; Foremost, 2020; Grenier, 2020) are able to reconstruct files that have been deleted long ago, sometimes even if a volume has been reformatted with a different file system.

The above methods seem to be methodologically rather unrelated. Especially the deterministic reconstruction of active files on the one hand and non-deterministic file carving on the other hand have been treated as two separate reconstruction methods in the literature and the available tools. However, *all* the above methods basically perform the same task, namely, to interpret data on a disk and derive block sequences suspected to contain file content.

In this paper we show that all the above methods can indeed be described within the same framework allowing a much closer integration of reconstruction tasks in forensic tools and processes and thereby allowing to combine the strengths of each method within a single recovery method. We present a model of storage abstraction layers that unifies deterministic (i.e. metadata-based) file reconstruction and non-deterministic carving. Similar to work by Freiling, Glanzmann and Reiser (Freiling et al., 2017), the model is generic and is not bound to file systems only – it extends to any type of block-based storage abstraction, be it partition systems, logical volumes, virtualized file systems and even virtual memory.

1.1. Contributions

This paper deals with the forensic reconstruction of resources in arbitrary abstraction layers. We introduce a model of storage abstraction layers and heuristic reconstruction modules that cover – to the best of our knowledge – all reconstruction approaches known from the literature, be they parsing of existing data structures, classical file carving or metadata carving. Furthermore, we define composition operators for these reconstruction methods that allow to combine them in useful ways. For example, the + operator combines the results of two reconstruction modules and therefore may seamlessly produce the union of results from several individual file carvers. Another example is the – operator that is used to subtract the results of one reconstruction module from those of another. Therefore, the operator provides a simple way to remove all active files from the results of a file carver.

We transform the model and the heuristics into the object-oriented design of a tool called LAYR that is able to reconstruct abstraction layers in a modular way through a generic interface. We demonstrate the applicability of the approach by implementing several heuristics and combining them in non-trivial ways.

In a certain sense, we generalize the central methodological innovation that was introduced for file systems by Carrier in TSK (Carrier, 2020a), namely, to categorize the data that is processed in file systems and make this data accessible through a common interface. Thus, file systems are regarded as different instances of a "file system" abstraction layer. We now take the next step and harmonize the different file reconstruction methods for each layer. We also believe that tools in forensic computing should support automation by composition, much in the spirit of Garfinkel's DFXML approach (Garfinkel, 2012), only that we focus on a specific (but generic) set of objects for recovery, namely storage objects that can be modeled as a set of equal size blocks.

1.2. Paper outline

Section 2 describes our refinement of the model of Freiling, Glanzmann and Reiser (Freiling et al., 2017) of block-based storage. Section 3 then also introduces the reconstruction of storage objects using heuristics and the operators allowing to combine them in almost arbitrary fashion. Based on these results, Section 4 presents a software architecture for a forensic analysis and reconstruction tool called LAYR. Section 5 describes how LAYR is implemented while Section 6 presents the evaluation of the implemented heuristics and operators. Finally, Section 7 summarizes the obtained results and gives suggestions for possible future work.

2. A model of storage abstraction layers

The concept of abstraction layers is not unique to file systems, it can be found everywhere in computing systems, e.g. in computer networks and operating systems. Abstraction layers are a means to structure software systems, separate concerns and provide isolation.

In the analysis of file systems, Carrier (2005) systematized the analysis of data on persistent storage from the physical to the application level (see Fig. 1): Starting from the lowest layer of physical sectors, these sectors are interpreted as partitions, i.e. consecutive sequences of disk sectors that in turn can be interpreted as holding different file systems. File systems organize the sectors of the partitions into files that can be used to store data of applications. All these rules are implemented within TSK enabling consistent and reliable forensic reconstruction of each individual layer.

Freiling, Glanzmann and Reiser (Freiling et al., 2017) generalized Carrier's model to arbitrary types of *block-based storage* (e.g. on memory or disk) that is organized as a finite sequence of fixed-size memory blocks. We use slightly simplified versions of their definitions of *block* and *storage* as follows.

A *block* is a sequence of bytes of a fixed size. Blocks are identified by an index, which is a natural number. We fix a set $B \subseteq \mathbb{N}$ of possible block indices.

In contrast to previous work (Freiling et al., 2017), blocks in our model are all of the same size independent of the layer on which they are interpreted. While this makes the model less practical, this assumption substantially simplifies the formalism in that block numbers do not need to be translated between abstraction layers and blocks can generally be identified by their index in the lowest layer. This definition of blocks allows us to define higher level storage objects such as partitions and files as follows.

Definition 1. (Storage object). *A storage object is a tuple $(b_1, \ldots, b_n)$ of pairwise distinct block indices from B. We denote the set of all storage objects by*

$$[B] = \{ (b_1, \ldots, b_n) : |\{b_1, \ldots, b_n\}| = n \}.$$

We write $b \in S$ for a block index b and a storage object $S = (b_1, \ldots, b_n)$ if $b \in \{b_1, \ldots, b_n\}$. *Similarly, we write $S \subseteq T$ for storage objects if $T = (b_1', \ldots, b_m')$ with $\{b_1, \ldots, b_n\} \subseteq \{b_1', \ldots, b_m'\}$.* Informally, a level of abstraction consists of two such storage objects, one on the "upper" layer and one on the "lower" layer. Between these two layers an entity manages the allocation of upper-layer blocks to lower-layer blocks. Since block indices in our framework always refer to the lowest layer, the allocation is implicit. While many

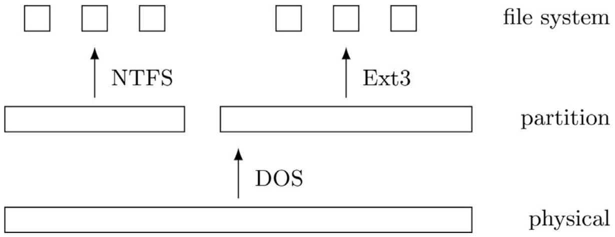

Fig. 1. Abstraction layers in file system analysis following Carrier [3, Fig. 1.3].

storage layers consist of a single storage object, in general, partitioning schemes, file systems or other abstractions consist of multiple storage objects within a single layer. Therefore, we refine the definition of a *layer* (Freiling et al., 2017) to allow multiple storage objects.

Definition 2. (Layer). *A set of storage objects is called a layer.*

We do not make any assumptions on the storage objects in a layer. In particular, two storage objects in the same layer can share blocks.

We also adapt the definition of a *relation* between storage objects from op. cit. to include the possibility of more than one storage object on a layer of abstraction. A relation captures the rules of how to interpret the lower layer to yield higher layer objects. For example, in the DOS partitioning scheme [3, Chapter 5], the relation between physical disk sectors (lower layer) and the sectors belonging to partitions (upper layer) is defined within the partition table. In the following notation, we denote upper layer symbols by $\widehat{x}$ and lower layer symbols by $\check{x}$.

Definition 3. (Level of abstraction). *A level of abstraction consists of two layers $\widehat{L}$ and $\check{L}$ and a relation $\varphi \subseteq \widehat{L} \times \check{L}$ between them.*

These concepts are illustrated in Fig. 2 where the upper layer $\widehat{L}$ consists of three storage objects (with two blocks each) and the lower layer $\check{L}$ of one storage objects (consisting of eight blocks). The relation φ between $\widehat{L}$ and $\check{L}$ indicates which lower layer storage object is used to construct which upper layer storage object and block indices are identified by their lower layer index. This model can be used to describe arbitrary file system abstractions such as those depicted in Fig. 1 (Freiling et al., 2017).

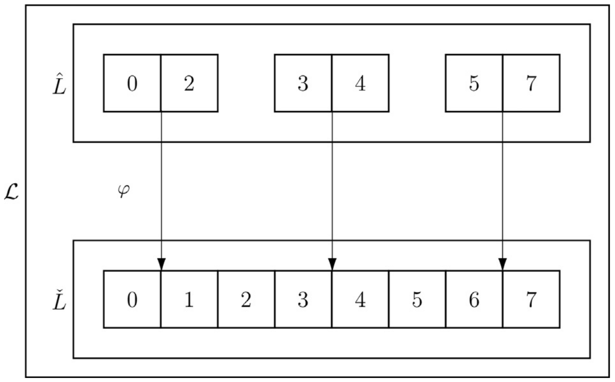

Fig. 2. A level of abstraction $\mathcal{L}$ consisting of an upper layer $\widehat{L}$ and a lower layer $\check{L}$, where the upper layer consists of three storage objects and the lower layer of one. The three pairs contained in the depicted relation φ indicate which lower layer storage object is occupied by which upper layer storage object.

3. Heuristics and their combination

We now present a general and modular framework for reconstructing storage objects of upper layers from storage objects of lower layers. It consists of two main ingredients:

(1) *Heuristics* implement specialized reconstruction tasks like parsing different file systems or carving for files using different techniques. Thus, heuristics implement the decoding rules of the relations between layers.
(2) *Operators* allow to combine different heuristics to perform more complex reconstruction tasks.

The division between heuristics and operators allows programmers to focus on the implementation of specialized heuristics for very specific reconstruction tasks while giving users the power and flexibility to arbitrarily combine these methods in novel ways. For example, if a file contains a physical copy of a hard disk (disk image), users may wish to interpret the content of a file using a specific partitioning scheme like DOS.

3.1. Heuristics

We now define an abstract interface of heuristics.

Definition 4. (Heuristic). *A heuristic is a function h of type*

$$H := [B] \to (\mathcal{P}([B]) + \{\perp\}),$$

such that the following condition holds for all $X \in [B]$:

$$\textit{if } h(X) \neq \perp \textit{ then } Y \subseteq X \textit{ for all } Y \in h(X). \quad (1)$$

Intuitively, heuristics receive one storage object of the lower layer as input (an element of $[B]$) and either fail with the special marker $\perp$ or reconstruct a set of storage objects of the higher-level layer (an element of the powerset of $[B]$). The condition (1) ensures that the reconstructed storage objects only refer to blocks of the input storage object.

The following list of heuristics is not meant to be exhaustive, but provides a few examples that show the generality of the definition.

- `DOS` is a heuristic that interprets its argument as a raw disk containing a partition table and zero or more partitions. It then returns the individual partitions as storage objects.
- `FAT`, `Ext3`, `Ext4` and `NTFS` are heuristics that interpret their input as a particular file system and return individual files.
- `Carve` is a heuristic that interprets its argument as raw disk and searches for files by known headers and footers (carving). These files are then returned.

Note that the argument and result type of heuristics make use of block indices instead of content. In fact, we do not specify how heuristics read the content of blocks at all as this is left to implementations. This also means that heuristics are mathematical functions only with respect to a fixed content and should be thought of more as procedures.

In practice, a heuristic is implemented as an algorithm that analyzes a given sequence of blocks of the lower layer and reconstructs storage objects of the upper layer, therefore operating on a single level of abstraction. In this context, reconstruction means to return one sequence of block indices that makes up a specific storage object (or multiple such sequences, respectively). Reconstructed storage objects can be active (allocated) or inactive (unallocated). In the case of active files, the heuristic algorithm could simply parse existing metadata (data management) structures. For the reconstruction of inactive files, several different carving heuristics can be implemented.

To accomplish these different reconstruction tasks, the heuristic often makes assumptions regarding the semantics of the upper layer. These assumptions can be rather strong, which aids in the reconstruction task, or they may be rather weak, which make reconstruction more of a guessing game. For example, one possible (strong) assumption could be that a layer contains a specific file system. However, the algorithm could also only assume a particular fragmentation grade or certain file content, weak assumptions which lead to reconstruction with less precision.

3.2. Operators

The power of our abstract framework lies in its modularity, and operators are the key to that. They take one or more heuristics as input and combine them to produce a new heuristic. This allows to build complex reconstruction sequences from simple primitives.

The operators that we define in this section can be divided into two different categories.

(1) *Sequential operators* execute a heuristic, inspect the result, and then execute further heuristics based on that.
(2) *Parallel operators* execute two (or more) heuristics independently from each other and combine the results of both. The heuristics' execution does not have to be concurrent but can be.

3.2.1. Sequential operators

The first operator that we consider allows to reconstruct objects across multiple layers of abstraction, by chaining different heuristics together. For example, the DOS heuristic can be used on one layer to reconstruct a partition, on which the Ext3 heuristic can then be applied to reconstruct individual files. This is done by simply using the output of one heuristic as input for another.

Definition 5. (Sequential composition). *The sequential composition operator $h_1 \gg= h_2$ first evaluates the heuristic h_1 and then evaluates another heuristic h_2 on all of the results of h_1. The final results of the h_2 invocations are then combined. It is defined as follows:*

$$
\begin{aligned}
&(\gg=) : H \times H \to H\\
&(h_1 \gg= h_2)(x) := \text{let } A := h_1(x)\\
&\qquad\qquad\qquad\quad \text{in if } A = \perp \text{ then } \perp\\
&\qquad\qquad\qquad\qquad \text{else } \bar{\cup}_{a \in A} h_2(a)
\end{aligned}
$$

where $\bar{\cup}$ is standard set union, but fails with $\perp$ as soon as one of the $h_2(a)$ returns $\perp$.

In many applications, it is not necessary or does not even make sense to consider *all* the results of the first heuristic h_1 as input for h_2. For example, an investigator might want to reconstruct only very specific parts of the evidence to avoid unnecessary violations of privacy (Stüttgen et al., 2013), or the second heuristic might only be applicable to certain results such as specific file systems. We therefore define a function choose, which models choice of output storage objects to continue processing.[1]

Definition 6. (choose). *There is a heuristic called* choose, *that selects storage objects from a set of storage objects in the following way:*

$$
\texttt{choose}(s) := \begin{cases} \{s\} & \textit{if } s \textit{ should be selected},\\ \emptyset & \textit{otherwise}. \end{cases}
$$

In practice, choose may represent either the tool users who decide during the investigation, e.g. which partition to select for further investigation. Or it could be a static list of storage objects to consider, e.g. if the investigators have already identified the relevant objects and encodes them into the analysis procedure for repeatability.

The above two definitions allow us to define a more convenient sequential composition operator:

Definition 7. (Sequential composition with choice). *The sequential composition with choice operator $h_1 \gg h_2$ evaluates a heuristic h_1, chooses a subset of the result, then evaluates another heuristic h_2 on all storage objects from this subset and combines the results:*

$$
\begin{aligned}
&(\gg) : H \times H \to H\\
&h_1 \gg h_2 := h_1 \gg= \texttt{choose} \gg= h_2
\end{aligned}
$$

Fig. 3 shows an example of a sequential execution of two heuristics. It interprets the lower layer storage object SD (a physical disk) using DOS (heuristic h_1) resulting in the active partitions P1 and P2. The heuristic choose then selects one of these partitions (FS) and applies the heuristic Ext3 to it.

The next operator makes use of the failure semantics of $\perp$ to implement a short-circuiting OR-operation. This can be used to try out a heuristic on a particular input and if that fails, try another one.

Definition 8. (Or). *The* Or *operator $h_1 \parallel h_2$ is defined as follows:*

$$
\begin{aligned}
&(\parallel) : H \times H \to H\\
&(h_1 \parallel h_2)(x) := \text{if } h_1\ (x) = \perp \text{ then } h_2(x) \text{ else } h_1(x)
\end{aligned}
$$

As an example, this operator can be used if the storage device under consideration is assumed to contain a particular file system, but the file system version is unknown. In this case, heuristics that

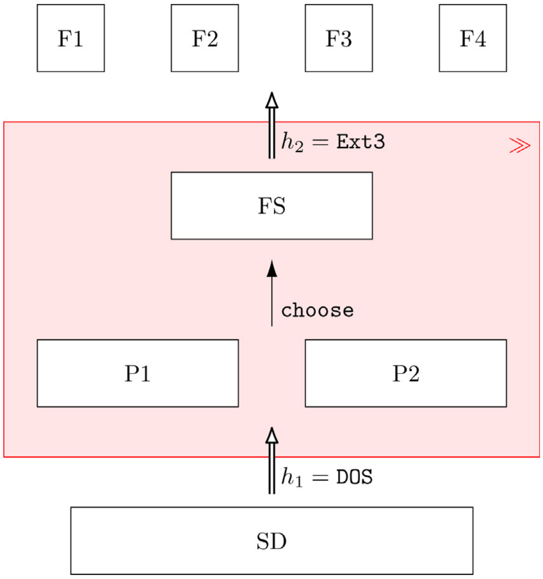

Fig. 3. Composition operator $h_1 \gg h_2$

[1] Basically, choose itself can be formalized as a heuristic, but for convenience it is implemented as a regular method, see Section 5.

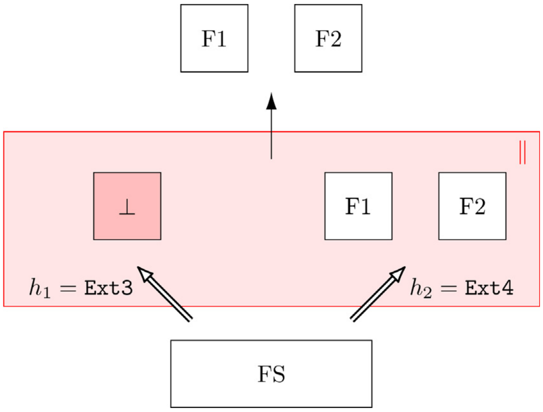

Fig. 4. Or operator $h_1 \parallel h_2$

implement the individual file system version can be tried out in sequence until one of them succeeds by combining them with Or.

Fig. 4 shows an example of how the operator is used to handle a partition containing an Ext file system but the exact file system type (Ext3 or Ext4) is unknown. First, the heuristic `Ext3` is applied. This heuristic determines that the file system is not actually of type Ext3 and fails with the special value ⊥. Thus, the heuristic `Ext4` is subsequently applied, yielding files F1 and F2, which are then returned as the final result. Note that the set of files from at most one heuristic is returned, and the heuristics themselves (or the implementer of the heuristics, respectively) define what it means for them to fail. The example can also be extended by the possibility of the storage device not containing any of the listed file systems at all. In this case, file carving could be used as a last resort.

3.2.2. Parallel operators

In addition to evaluate one heuristic after the other, it is also possible to evaluate two (or more) heuristics on the same input and combine their results with familiar set theoretic operations.

To make the definition of these operators easier, we define an auxiliary generic operator transformer called "op" of type

$$(\mathcal{P}([B]) \times \mathcal{P}([B]) \rightarrow \mathcal{P}([B])) \rightarrow (H \times H \rightarrow H).$$

It executes two heuristics on the same input and applies a given set-operation like union or intersection on the results. If one of them fails, "op" also fails. Using this, we can lift the following set operations to heuristics.

Definition 9. *We define the following* parallel operators *using* op:

(1) Union*: The* union operator $h_1 + h_2 := h_1 \text{ op}(\cup)\, h_2$ *combines the results of two heuristics by joining them together.*
(2) Intersection*: The* intersection operator $h_1 * h_2 := h_1 \text{ op}(\cap)\, h_2$ *returns all storage objects that both heuristics return in their result.*
(3) Minus*: The* subtraction operator $h_1 - h_2 := h_1 \text{ op}(\setminus)\, h_2$ *removes the results of the second heuristic from the results of the first one.*

In practice, $h_1 + h_2$ can be used to include more results than a single heuristic would. This is especially useful if the individual heuristics only return very specific results like files of a certain type, or to maximize the amount of reconstructed data, when using different reconstruction algorithms with different shortcomings, e.g. different file carvers. Fig. 5 shows the combination of an Ext3 and a carving heuristic using the + operator. It therefore yields the unified set of all active files and all carved files. Note that due to set semantics, duplicate files (in this case F2) appear only once in the final result.

In contrast to +, the operator $h_1 * h_2$ allows to narrow down results to those that are returned by *both* heuristics, thereby giving greater confidence in the validity of the results, as can be seen in Fig. 6. There, the results returned are those files that result from a consistent interpretation of the file system FS both in `Ext3` and `Ext4` (here only the file F2).

On large storage devices, the amount of reconstructable data can often be rather large. In such cases, it is useful to explicitly exclude storage objects from the result set that are not interesting. For example, when looking for deleted data with a file carver, active files can be excluded from the carver's results. This can be formulated using the Minus operator. Fig. 7 shows the result of a carving heuristic of which all active files in terms of `Ext3` have been removed.

3.2.3. Operator nesting

The ability to combine two heuristics is useful in itself, but the expressive power of our framework hinges on the fact that the result of applying an operator to heuristics is again a heuristic. Thus, operators can be applied to the result of other operators, thereby allowing to build complex nested expressions to deal with complex reconstruction problems.

As an example, consider the complex heuristic `DOS` $\gg$ ((`Ext3` $\parallel$ `Ext4`) + `Carve`) that is depicted in Fig. 8. It combines four different heuristics using three operators in the following way: The content of the disk image SD is interpreted as a DOS formatted disk, yielding a set of partitions. The user subsequently chooses a

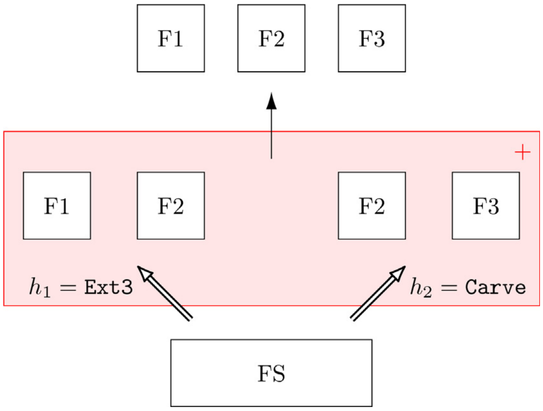

Fig. 5. Union operator $h_1 + h_2$.

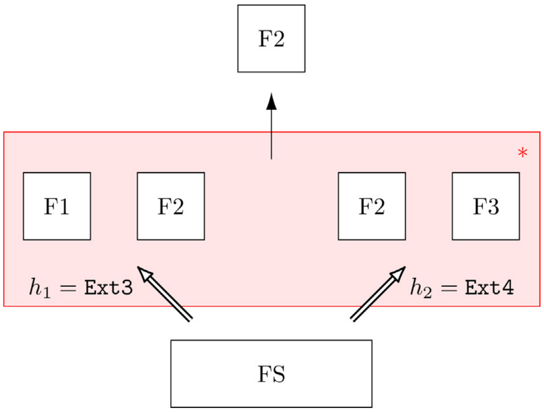

Fig. 6. Intersection operator $h_1 * h_2$.

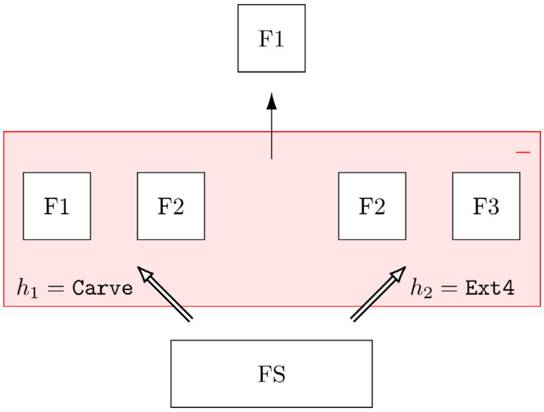

Fig. 7. Minus operator $h_1 - h_2$.

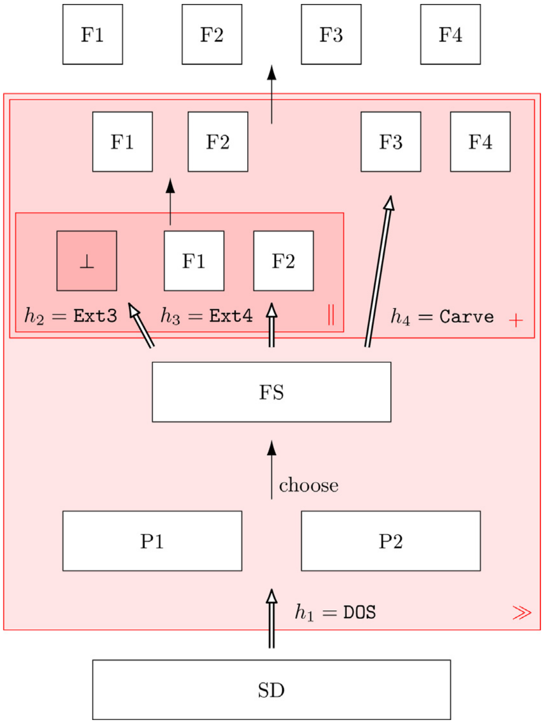

Fig. 8. Combination of four heuristics and three operators: DOS $\gg$ ((Ext3 $\parallel$ Ext4) + Carve).

particular partition and combines the results of file carving with the results from (first) interpreting the file system as Ext3 and (second, if it is not Ext3) as an Ext4 file system. The resulting set of files has a well-defined meaning and is the result of a complex sequence of reconstruction tasks.

4. Software design

To show the practical usability of the refined model, heuristics and operators, a forensic reconstruction and analysis tool called LAYR was implemented. LAYR may be regarded as complementary to tools like TSK because it implements the analysis and reconstruction of generic abstractions and addresses the problem of reliable reconstruction of deleted files. The strengths of LAYR lie in the fact that new heuristics can be easily added, and the combination of different reconstruction approaches improves reconstruction results. In contrast to TSK, which is implemented in C, LAYR is implemented in C++ which supports object-orientation, inheritance and comes with an extensive *standard template library* (STL). The usage of C++ allows us to make use of several modern advanced programming techniques, for example templates and streams. The class hierarchy of LAYR was directly derived from the definitions from Sec. 3, which shows that the definitions can be used to create a usable and generic interface for implementations of heuristics. The modular design and inheritance based interface of LAYR ensures simple extensibility, whereby developers can easily add their own heuristics specialized for specific tasks or other custom heuristic implementations. By using the implemented operators defined in Sec. 3.2 heuristics can be freely combined to solve complex reconstruction tasks.

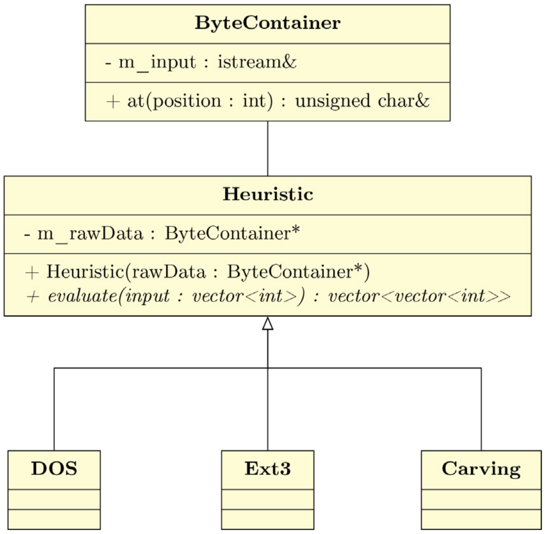
ByteContainer
- m_input : istream&
+ at(position : int) : unsigned char&
Heuristic
- m_rawData : ByteContainer*
+ Heuristic(rawData : ByteContainer*)
+ evaluate(input : vector<int>) : vector<vector<int>>
DOS
Ext3
Carving

Fig. 9. Heuristic class layout.

4.1. Class layout

Fig. 9 shows the class diagram for heuristics.[2] Each heuristic inherits from the abstract parent class `Heuristic`, which has access to the lower layer represented as `ByteContainer`. The `ByteContainer` class is an istream wrapper which provides the possibility to access the lower layer data stream through a vector like interface. For example, one single byte from the data stream can be accessed through the **at** function which behaves similar to the STL vector function **at**. To realize concrete heuristics the **evaluate** function inherited from the abstract heuristic parent has to be implemented. The function **evaluate** executes some heuristic evaluation of the given sequence of storage blocks. For active files this amounts to decoding the relation φ (cf. Def. 3).

For some abstraction layers (like file systems), it is well-known and documented how the translation from lower layer blocks (within the partition) and higher layer blocks (files) happens. In general, the ability to reconstruct upper layer storage objects is merely restricted by the ability to find and reconstruct φ on the lower layer (Freiling et al., 2017). In cases where φ is not found or cannot be decoded, other reconstruction approaches like file carving can be implemented as heuristic evaluation function.

As can be seen in Fig. 9, the **evaluate** function receives a vector

[2] Note that all class diagrams have been stripped of inessential details for illustrative purposes and better understanding.

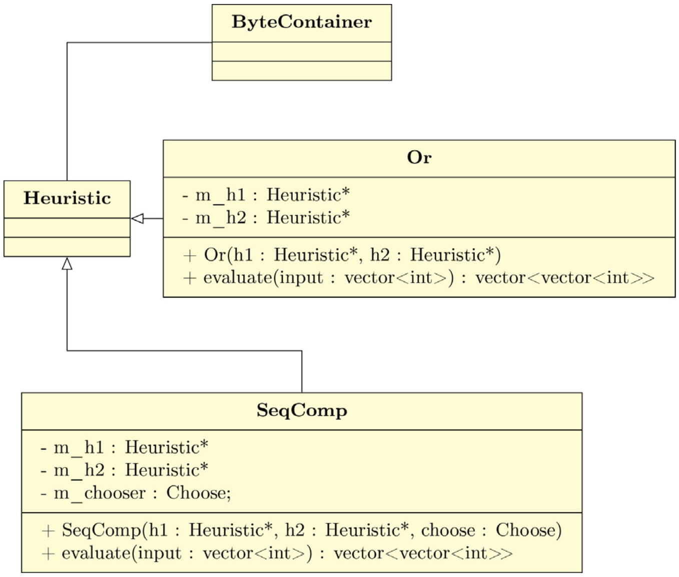
ByteContainer
Or
Heuristic
- m_h1 : Heuristic*
- m_h2 : Heuristic*
+ Or(h1 : Heuristic*, h2 : Heuristic*)
+ evaluate(input : vector<int>) : vector<vector<int>>
SeqComp
- m_h1 : Heuristic*
- m_h2 : Heuristic*
- m_chooser : Choose;
+ SeqComp(h1 : Heuristic*, h2 : Heuristic*, choose : Choose)
+ evaluate(input : vector<int>) : vector<vector<int>>

Fig. 10. Sequential operator class layout.

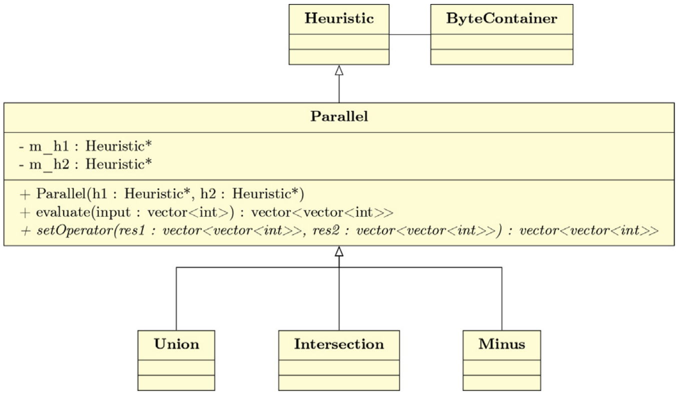
Heuristic
ByteContainer
Parallel
- m_h1 : Heuristic*
- m_h2 : Heuristic*
+ Parallel(h1 : Heuristic*, h2 : Heuristic*)
+ evaluate(input : vector<int>) : vector<vector<int>>
+ setOperator(res1 : vector<vector<int>>, res2 : vector<vector<int>>) : vector<vector<int>>
Union
Intersection
Minus

Fig. 11. Parallel operator class layout.

of integers as input and returns a vector of vectors of integers. The input vector represents the input storage object as a sequence of block indices (i.e. a vector of integers). This sequence of block indices is used to address data inside the `ByteContainer`. After successful evaluation, the function returns a sequence of storage objects represented as sequence of block indices (modeled again as a vector of vectors of integers). In case the evaluation fails an exception is thrown which corresponds to the special marker $\perp$ defined in Sec. 3.

One advantage of the LAYR framework lies in the possibility to combine different heuristics to improve the reconstruction results. This is achieved by implementing the operators defined in Sec. 3.2. As in the formal definition we distinguish between sequential and parallel operators. Sequential operators directly inherit the abstract `Heuristic` parent class, while parallel operators are derived from the `Parallel` class as can be seen in Fig. 10 and Fig. 11. Fig. 10 also shows that the sequential operator class `SeqComp` receives two heuristics and an additional choice function that corresponds to the `choose` heuristic from Sec. 3.2.1 in its constructor. The sequential operator functionality is implemented in the **evaluate** function. The same holds true for the `Or` operator, which does not need access to the **choose** function but also implements its previously described operator functionality directly inside **evaluate**.

The class `Parallel` also takes two heuristics in its constructor and extends the `Heuristic` parent class by adding the **setOperator** function. As described in Sec. 3.2.2 parallel operators use basic set operations to combine heuristic results. These set operators have to be customly implemented by the child classes `Union`, `Intersection` and `Minus`. The **evaluate** function is implemented by the parent class and can be considered as the generic operator transformer "op" from Sec. 3.2.2.

5. Implementation

For demonstration purposes, we implemented two heuristics `DOS` and `Ext3`. We also implemented several testing heuristics and the operators defined in Sec. 3.2. We did not implement a user interface yet but implemented several test scenarios instead. For the following description of the implementation of both heuristics some knowledge about the DOS partitioning scheme and the Ext3 file system is assumed (Carrier, 2005).

To implement the DOS heuristic evaluation function the partition table entries have to be found and parsed. Since we are only interested in reconstructing block sequences which represent partitions, unnecessary metadata like the partition type are skipped during the parsing of partition table entries. To generate block sequences, only the start of the partition and the partition size are processed. The function also checks for extended partitions and generates block sequences for them such that all partitions are taken into account.

The Ext3 evaluation function first extracts the necessary file system information from the superblock. This information is needed to obtain the group descriptor table containing the inode tables. Afterwards, each inode is parsed and the contained block pointers are used to generate block sequences.

Note that access of all heuristics to the lower layer is limited by the input vector. This is ensured by checking the lower and upper limits during access to the lower layer. If these limits are exceeded, an exception is thrown.

The sequential composition operator requires the possibility to choose the input for the second heuristic which is implemented by the function **choose**. This function is provided by the constructor and can either be a user input function or a hard coded choice for test purposes which is the case in the current implementation. The `SeqComp` **evaluate** function first calls the **evaluate** function of the first passed heuristic and uses the returned block sequences as input for the **choose** function which is called afterwards. The resulting choice is then used as input for the **evaluate** function of the second passed heuristic. In case several results of the first heuristic are chosen, the evaluate function of the second heuristic is called several times and the results are combined and returned.

The Or operator is implemented by executing the **evaluate** function from the first heuristic surrounded by a try block. If an exception is caught, the second heuristic is executed.

As described in Sec. 4.1 parallel operators have to implement the **setOperator** function. This is realized by using **set_union**, **set_intersection** and **set_difference** from the C++ STL depending on the child class.

6. Evaluation

Evaluation of the `DOS` and `Ext3` heuristics was done by comparing the results of an invocation of **evaluate** with the output of the TSK tools `mmls` and `istat`. For the evaluation we used two images from the Digital Forensics Tool Testing (DFTT) project (Carrier, 2020b, 2020c).

The DOS DFTT test image (Carrier, 2020b) contains three partitions and one extended partition. Furthermore, the extended

```
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

Slot   Start     End         Length      Description
000:   Meta      0000000000  0000000000  0000000001   Primary Table (#0)
001:   -------   0000000000  0000000062  0000000063   Unallocated
002:   000:000   0000000063  0000052415  0000052353  DOS FAT16 (0x04)
003:   000:001   0000052416  0000104831  0000052416  DOS FAT16 (0x04)
004:   000:002   0000104832  0000157247  0000052416  DOS FAT16 (0x04)
005:   Meta      0000157248  0000312479  0000155232  DOS Extended (0x05)
006:   Meta      0000157248  0000157248  0000000001   Extended Table (#1)
007:   -------   0000157248  0000157310  0000000063   Unallocated
008:   001:000   0000157311  0000209663  0000052353  DOS FAT16 (0x04)
009:   -------   0000209664  0000209726  0000000063   Unallocated
010:   001:001   0000209727  0000262079  0000052353  DOS FAT16 (0x04)
011:   Meta      0000262080  0000312479  0000050400  DOS Extended (0x05)
012:   Meta      0000262080  0000262080  0000000001   Extended Table (#2)
013:   -------   0000262080  0000262142  0000000063   Unallocated
014:   002:000   0000262143  0000312479  0000050337  DOS FAT16 (0x06)
```

Fig. 12. Output of TSK `mmls` applied to DFTT "Extended DOS Partition Test" image (Carrier, 2020b).

```
63 52415
52416 104831
104832 157247
157311 209663
209727 262079
262143 312479
```

Fig. 13. Extract of the output of the `DOS` heuristic evaluation test code applied to DFTT "Extended DOS Partition Test" image (Carrier, 2020b).

```
inode: 2
Direct Blocks:
165

inode: 11
Direct Blocks:
166 167 168 169 170 171 172 173
174 175 176 177

inode: 12
Direct Blocks:
1208

inode: 13
Direct Blocks:
1209 1212

inode: 14
Direct Blocks:

inode: 15
Direct Blocks:
1211
```

Fig. 14. Extract from the output of TSK `istat` applied to the inodes contained in the DFTT "EXT3FS Keyword Search #1" image (Carrier, 2020c).

partition contains two partitions and one extended partition which contains another partition. Fig. 12 shows the output of the TSK tool `mmls` applied to the DOS DFTT test image. For comparison the results of the DOS **evaluate** function were printed to standard output, which can be seen in Fig. 13. Note that only the first and the last element from the resulting block sequences are shown. The comparison shows that the `DOS` heuristic correctly reconstructs the partition block sequences.

```
330 331
332 333 334 335 336 337 338 339 340 341 342 343
    344 345 346 347 348 349 350 351 352 353 354
    355
2416 2417
2418 2419 2424 2425
2422 2423
```

Fig. 15. Output of the `Ext3` heuristic evaluation test code applied to DFTT "EXT3FS Keyword Search #1" image (Carrier, 2020c).

```
332 333 334 335 336 337 338 339 340 341 342 343
    344 345 346 347 348 349 350 351 352 353 354
    355
```

Fig. 16. Output of the sequential composition operation evaluation test code.

The Ext3 DFTT test image (Carrier, 2020c) contains six valid inodes: one for the root directory, one directory, and four file inodes, where one file is deleted. As in the `DOS` case, TSK was used to generate a comparison output. For the Ext3 evaluation TSK's `fls -r` was applied to the test image. Afterwards `istat` was used to output detailed information about the inodes, where the block pointers are of special interest. Fig. 14 shows an excerpt from the `istat` output containing the relevant block pointer passages. The results of the `Ext3` heuristic **evaluate** function were printed to standard output. Fig. 15 shows the uncut output from `cout`. Note that the standard heuristic block size is currently 512 byte. Since file 3 is deleted the `Ext3` heuristic does not create any result for this file as the block pointers are removed from the inode. This corresponds to the desired behavior.

To evaluate the functionality of the sequential composition operator an `Input` heuristic was implemented. The `Input` heuristic simply prints the passed input block sequence to `cout`. It can be used to evaluate whether the heuristic is executed correctly after the result of the first executed heuristic was chosen as input. Therefore, the `Ext3` heuristic was combined with the `Input` heuristic to test the sequential composition operator. After executing the test code, the chosen input (which is the second block sequence evaluated by the `Ext3` heuristic) is correctly printed to `cout` as can be seen in Fig. 16.

```
330 331
332 333 334 335 336 337 338 339 340 341 342 343
    344 345 346 347 348 349 350 351 352 353 354
    355
2416 2417
2418 2419 2424 2425
2422 2423
```

Fig. 17. Output of the Or operator evaluation test code.

```
330 331
332 333 334 335 336 337 338 339 340 341 342 343
    344 345 346 347 348 349 350 351 352 353 354
    355
2416 2417
2418 2419 2424 2425
2422 2423
2450
2460

330 331
332 333 334 335 336 337 338 339 340 341 342 343
    344 345 346 347 348 349 350 351 352 353 354
    355
2418 2419 2424 2425
2422 2423

2416 2417
```

Fig. 18. Output of the parallel operator evaluation test code.

Since the Or operator makes use of the ⊥ failure semantics a `Failure` heuristic was implemented. The `Failure` heuristic can be used to test if the Or operator correctly uses the second heuristic after the first heuristic failed. Therefore, the `Failure` heuristic is used as first heuristic which always throws an exception to trigger the evaluation of the second heuristic. Fig. 17 shows that the `Ext3` heuristic is correctly executed after the Failure heuristic failed.

For the evaluation of the parallel operators another heuristic was implemented which is called `AddResult` and only returns the results of the `Ext3` heuristic, without one of the original results (2416 and 2417) and extended by two additional fictitious results (2450 and 2460). To test the functionality of the parallel operators the evaluate results from `Ext3` and `AddResult` are combined successively through `Union`, `Intersection` and `Minus`. Fig. 18 shows the results of the evaluation test code printed to `cout`. The output indicates that the operators are working correctly and that the results are generated according to the definition of the operators.

7. Conclusion

We considered the problem of collecting digital evidence on systems where abstraction layers are used to organize storage. Based on a generic model of abstraction layers, we developed a modular software framework which has the power to generically combine different reconstruction approaches and techniques. In our framework, heuristics can be used to address specialized reconstruction tasks where operators can be used to combine them and to build complex reconstruction sequences. We believe that our model can be applied to most abstraction layers from the literature, and can be the basis for a library of reconstruction modules.

One minor conceptual shortcoming of LAYR is the fact that current heuristics operate on a single storage object as input. This precludes the modeling of abstraction layers that operate on multiple lower layer storage objects (such as RAID systems). Similarly, the assumption that all blocks (at any layer of abstraction) are of equal size is not practical. Both shortcomings are minor since the definition and implementation of heuristics can be easily adapted, while becoming slightly more complicated. We will consider this in future work.

Furthermore, for simplification the current version of the model does not consider metadata. Besides, LAYR currently uses a fixed standard block size of 512 byte for all layers. In the future we will implement a metadata and block sizes handling.

Additionally, LAYR can be extended to cover more file system and carving heuristics. Due to the extensibility of the framework, it might be an option to not only directly reuse code from other projects but even to integrate closed-source software by constructing wrapper classes around their interface.

Furthermore, heuristics could be used to implement complex file filters, maybe by plugging in input like search terms or regular expressions over a (graphical) user interface. Filter heuristics could be used to reduce the amount of reconstruction results to make them more specific to the current investigative question. For example, a black and white heuristic could generate a result set where only scanned documents are included. Another application could be a hash based heuristic which compares result with a database of well know illegal material. Besides, a file name or a hate speech heuristic could return files with specific names or words inside documents giving a hint of if the content of the file could be of interest.

The source code of LAYR is available at https://github.com/janineschneider/LAYR.

Acknowledgments

We wish to thank Andreas Dewald, Benedikt Lorch, Vassil Roussev and the anonymous reviewers for their helpful comments on previous versions of the paper. Work was supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Research and Training Group 2475 “Cybercrime and Forensic Computing” (grant number 393541319/GRK2475/1-2019).

References

Carrier, B., 2002. Defining digital forensic examination and analysis tools using abstraction layers. International Journal of Digital Evidence 1, 2003.

Carrier, B., 2005. File System Forensic Analysis. Addison-Wesley Pub. Co. Inc., Boston, MA, USA.

Carrier, B.. The Sleuth Kit (TSK) & Autopsy: Open source digital forensics tools. https://www.sleuthkit.org/ last access: 2020-01-20.

Carrier, B.. Digital forensics tool testing image (#1): Extended DOS partition test. http://dftt.sourceforge.net/test1/index.html last access: 2020-01-20.

Carrier, B.. Digital forensics tool testing image (#4): EXT3FS Keyword search test #1. http://dftt.sourceforge.net/test4/index.html last access: 2020-01-20.

Dewald, A., Seufert, S., 2017. AFEIC: Advanced forensic Ext4 inode carving. Digital Investigation 20 (Supplement), 83–91. DFRWS 2017 Europe. http://www.sciencedirect.com/science/article/pii/S1742287617300270.

Foremost. http://foremost.sourceforge.net/ last access: 2020-01-20.

Freiling, F., Glanzmann, T., Reiser, H.P., 2017. Characterizing loss of digital evidence due to abstraction layers. Digital Investigation 20 (Supplement), 107–115. https://doi.org/10.1016/j.diin.2017.01.012. DFRWS 2017 Europe. http://www.sciencedirect.com/science/article/pii/S1742287617300427.

Garfinkel, S.L., 2012. Digital forensics XML and the DFXML toolset. Digital Investigation 8 (3–4), 161–174. https://doi.org/10.1016/j.diin.2011.11.002. http://www.sciencedirect.com/science/article/pii/S1742287611000910.

Grenier, C.. Photorec - digital picture and file recovery. https://www.cgsecurity.org/wiki/PhotoRec last access: 2020-01-20.

Jain, B., Baig, M.B., Zhang, D., Porter, D.E., Sion, R., 2014. SoK: Introspections on Trust and the Semantic Gap. In: 2014 IEEE Symposium on Security and Privacy, pp. 605–620, 10.1109/SP.2014.45. https://ieeexplore.ieee.org/document/6956590.

Memon, N.D., Pal, A., 2006. Automated reassembly of file fragmented images using greedy algorithms. IEEE Transactions on Image Processing 15 (2), 385–393. https://doi.org/10.1109/TIP.2005.863054, 10.1109/TIP.2005.863054. https://ieeexplore.ieee.org/document/1576811.

Pal, A., Memon, N., 2009. The evolution of file carving. IEEE Signal Processing Magazine 26 (2), 59–71. https://doi.org/10.1109/MSP.2008.931081. https://ieeexplore.ieee.org/document/4806206.

Richard III, G.G., Roussev, V., 2005. Scalpel – A Frugal, High Performance File Carver. In: Proceedings of the 5th Annual Digital Forensic Research Workshop, DFRWS 2005, Astor Crowne Plaza, New Orleans, Louisiana, USA, August 17-19, 2005, p. 10. https://dfrws.org/presentation/scalpel-a-frugal-high-performance-file-carver/.

Stüttgen, J., Dewald, A., Freiling, F.C., 2013. Selective Imaging Revisited. In: 2013 Seventh International Conference on IT Security Incident Management and IT Forensics. IMF 2013, Nuremberg, Germany, pp. 45–58, 10.1109/IMF.2013.16, March 12-14, 2013. https://ieeexplore.ieee.org/document/6568553.