Digital Investigation 29 (2019) S22–S30

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2019 USA — Proceedings of the Nineteenth Annual DFRWS USA

Syntactical Carving of PNGs and Automated Generation of Reproducible Datasets

Jan-Niclas Hilgert*, Martin Lambertz, Mariia Rybalka, Roman Schell

Fraunhofer FKIE, Bonn, Germany

ARTICLE INFO

Article history:

Keywords:
Syntactical file carving
Dataset generation
Portable network graphics

ABSTRACT

File carving is a technique to recover files from a storage medium without relying on a file system or other external metadata. As long as the files have been stored contiguously, most file formats are comparatively easy to carve. The moment files have been stored fragmented, the carving process becomes a highly complicated task—even when the fragmentation scenario is relatively simple—and most file carvers available today are not capable of restoring such files correctly.

In this paper, we apply syntactical file carving, i.e. the process of utilizing the syntax of a file format to the maximum extent, to the PNG file format. By doing so, we show that the complexity of carving files even in very convoluted fragmentation scenarios can be significantly reduced. Furthermore, we provide a prototypical implementation of a syntactical PNG file carver. In our evaluation, the carver was able to restore 98% of the test files completely and correctly, while the remaining files were at least partially recovered.

Since most of the publicly available file carving datasets do not contain PNG files, we created a custom dataset for our evaluation resembling the DFRWS forensic challenges from 2006 to 2007. To ease the creation of such datasets we implemented a dataset generation framework. Using our framework it is possible to create complex fragmentation scenarios with just a few lines of code and configuration. Through this, we hope to encourage the creation of publicly available datasets and to foster further research in the area of file carving.

1. Introduction

The results of a digital forensic investigation significantly depend on the digital evidence extracted from various storage devices. In most cases, already the analysis of the latest live version of the file system reveals a lot of evidence to a forensic investigator. This data includes anything from media files and documents to saved bookmarks and visited web pages. Diving deeper and analyzing a file system even further can bring more evidence to light, including files that have been deleted. Unfortunately, a trivial recovery of these deleted files may not always be feasible due to inconsistent or missing file system meta data entries. In other cases, the file system may be damaged or completely missing and, therefore, unusable for the recovery of deleted files.

File carving is a recovery technique which does not need any existing file system meta data. A trivial approach is header-to-footer carving, where the data between two identifiable byte sequences—the header and the footer—is extracted. Unfortunately, such simple file carving methods commonly yield insufficient results as soon as files are stored fragmented.

More than ten years ago, Garfinkel already performed a large-scale analysis of fragmented files in the wild and emphasized the importance of being able to reassemble these fragments (Garfinkel, 2007). Unfortunately, carving fragmented files is far from trivial. To be able to carve fragmented files, the exact fragmentation point and the start of the next fragment have to be detected. Depending on the file type both steps can be extremely complex. Moreover, due to the ever-increasing size of storage devices, the sheer number of candidates for the beginning of the next fragment renders simple brute-force approaches useless.

In this paper, we argue that a file carving approach should take advantage of as much of the internal structure of a file type as possible to determine the fragmentation point and to find the start

* Corresponding author.
E-mail addresses: jan-niclas.hilgert@fkie.fraunhofer.de (J.-N. Hilgert), martin.lambertz@fkie.fraunhofer.de (M. Lambertz), mariia.rybalka@fkie.fraunhofer.de (M. Rybalka), roman.schell@fkie.fraunhofer.de (R. Schell).

https://doi.org/10.1016/j.diin.2019.04.014
1742-2876/© 2019 The Author(s). Published by Elsevier Ltd on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

of the next fragment. The syntax of a file is typically easier and faster to check than the actual file contents, which some approaches use to carve fragmented files. Moreover, a carved file conforming to the syntax of its format is less likely a false positive (i.e. a corrupted file). Avoiding false positives is a key aspect when file carving is performed—after all, the carved files have to be reviewed by an investigator.

Furthermore, we propose to perform the carving process in several phases. The rationale behind this is that most of the files are relatively easy to carve (e.g. non-fragmented files or simple bifragmented files), while there are comparatively few files which require methods taking most of the available blocks of a storage device into consideration. Those more costly methods should be postponed until all of the easy to carve files have been reconstructed to reduce the number of data still available.

In order to demonstrate the usefulness of syntactical file carving, we present an approach to carve fragmented PNG files. PNG is a popular image file format which is widely used especially on web pages. Moreover, there are several graphic programs which use PNG as their default output format (first and foremost screenshot programs which may have been used to capture forensically interesting data). Finally, there are various techniques to hide exploits or malicious payloads in PNG files (Shah, 2015; Marques, 2016). Therefore, we argue that PNG is a file format which is of high interest and most certainly encountered during a forensic investigation. Finally, the PNG file format is sufficiently structured to enable syntactical carving.

To evaluate our approach, we reviewed various available carving test sets but found that none of them contains PNG files. Therefore, we developed a framework to generate test datasets for the evaluation of file carvers. It allows the user to easily create different test cases reflecting various fragmentation scenarios. Using this framework we established an evaluation dataset for PNG file carvers resembling the scenarios used in the DFRWS carving challenges from 2006 to 2007 (Carrier et al., 2006, 2007).

In summary, our contributions are a phase-based, syntactical file carving algorithm for PNG files. Our prototypical implementation of the algorithm outperforms popular existing file carving tools, such as Scalpel or PhotoRec. Moreover, we provide a framework to easily generate test datasets for file carvers. Both, the prototypical implementation of our carver and the dataset generation framework will be released as open source software (Hilgert et al., 2019).

2. Portable Network Graphics

Portable Network Graphics (PNG) is an image file format which can be used to store true-color, indexed-color and greyscale images with or without an alpha channel for transparency in a lossless manner. It was designed for web applications, which is why it is streamable and supports a progressive display option. The World Wide Web Consortium (W3C) describes PNG as an *"extensible file format for the lossless, portable, well-compressed storage of raster images"* (W3C, 2003). The most recent PNG specification was published in 2003 and is the main reference to gain knowledge about the internal structure, the data streams, and the syntax of PNG files. Further resources for a better understanding of PNG include the initial RFC published in 1997 (Boutell, 1997) as well as "The Definite Guide to PNG" (Roelofs and Koman, 1999).

In this section we do not go into details of how the actual image data is encoded and stored in a PNG file, but rather focus on aspects of the file format relevant for file carving.

2.1. *Structure of a PNG file*

The basic structure of a PNG file is illustrated in Fig. 1. As depicted there, a PNG file always starts with the PNG file signature. This signature consists of the following sequence of eight bytes (represented as hexadecimal values): `89 50 4e 47 0d 0a 1a 0a`.

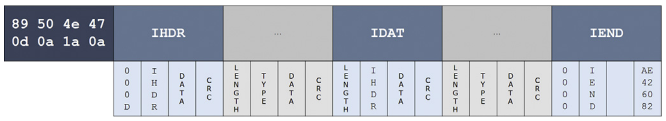

Fig. 1. Structure of a PNG file and its chunks.

Disregarding the signature, the complete PNG file is divided into multiple units referred to as *chunks*. Fig. 1 also illustrates the layout of PNG chunks. Each chunk consists of four fields: a length field (four bytes), a type field (four bytes), a data field (variable length), and a CRC field (four bytes).

The length field contains an unsigned integer specifying the length of the data stored in the data field. This value can also be zero indicating that the data field is empty. Note that the total length of a chunk is the value defined in the length field plus twelve bytes (four bytes each for the length, type, and CRC fields).

The second field defines the chunk type. Each of the four bytes stored in the type field corresponds to an ISO 646 letter from `A-Z` or `a-z`. The letters make up the name of the chunk type. As of version 1.4.6 of `libpng`, 26 different chunk types are registered (Roelofs, 2015). Furthermore, the PNG specification defines ordering constraints for chunk types restricting the position of certain chunk types within a PNG file.

If the defined length of a chunk is non-zero, the third field contains the actual chunk data. Finally, for integrity each chunk ends with a four-byte Cyclic Redundancy Check (CRC). Since the CRC is calculated over the chunk type and the data fields, it is also present for chunks without data.

2.2. *Chunk types*

PNG defines the `IHDR`, `IDAT`, `IEND` and `PLTE` chunk types as critical chunks, meaning every implementation of the PNG standard should be able to correctly parse and render these chunks. For a PNG file to be valid, exactly one `IHDR` and one `IEND` have to be present as well as one or more `IDAT` chunks. The following sections elaborate on these three chunk types in more detail.

2.2.1. `IHDR`

An `IHDR` chunk is always the first chunk within a PNG file, following the PNG signature. It defines fundamental properties of the image such as its width and height as well as the bit depth and color type and the compression, filter, and interlace methods. In total, these data fields of the IHDR chunk are always 13 bytes in size resulting in a static length field across all `IHDR` chunks.

2.2.2. `IDAT`

`IDAT` chunks store the actual image data of the PNG file as a data stream compressed by the compression method defined in the `IHDR` chunk. All `IDAT` chunks in a PNG file should be stored in a consecutive order without any other chunk type in between. PNG encoders can divide the compressed image data into arbitrarily sized chunks. As already mentioned, also `IDAT` chunks with zero data are legal according to the PNG specification.

2.2.3. `IEND`

The `IEND` chunk must be the last chunk since it marks the end of a PNG file. An `IEND` chunk does not contain any data and is, therefore, equal for all PNG files.

3. Existing file carving approaches for PNGs

To the best of our knowledge, there is no carver—either in academic publications or as a software tool—explicitly focusing on fragmented PNG files. However, there are existing approaches which can be applied to carve PNGs. These will be briefly described in what follows.

Header-to-footer carving is an obvious approach to be applied to PNG files. It searches for specific byte sequences which identify the start and end of a particular file type. Generally, header-to-footer carving benefits from longer signatures as they reduce the probability for false positives. As described in Section 2 PNG files have an eight byte signature at the beginning. In combination with the fixed length and type fields of the `IHDR` chunk, this makes a 16-byte header which can be searched for. The data of the `IEND` chunk is defined to be empty. Hence, this image trailer of 12 bytes can be used as the footer signature for PNG files. This is what the open source carver Scalpel does for example (Richard III. and Roussev, 2005; Richard III. and Marziale, 2013). As already mentioned in the Introduction, header-to-footer carving generally fails when a file is fragmented.

The same holds for header-embedded-length carving. This approach can be used when a file type has a length field in its header. After finding the header, this field is parsed to determine the length of the file. Then, the bytes from the header plus the number of bytes parsed from the length field are carved. The PNG file header does not have a global length field to indicate the size of the complete file. There are length fields within the individual chunks, though. Hence, a modified header-embedded-length carving approach could be employed here. The basic idea would be to parse the length of the first chunk, proceed this number of bytes in the disk image, parse the length field of the next chunk, and jump again. This would be carried out until a jump reaches a position without a valid chunk header. This is what Foremost (Kendall et al., 2009) does when the PNG specific extraction mode is enabled. PhotoRec (Grenier, 2015) also uses this approach. Again, this only works when a file is not fragmented.

One of the first approaches to cover fragmented files was bifragment gap carving as proposed by Garfinkel (2007). It extends the methods described above by placing a gap between the start and end of a fragmented file to be carved. This gap is subsequently moved as well as grown to find the range of bytes not belonging to the current file under reconstruction. To check whether the correct gap is found the author uses what he calls fast object validation. This is basically a file type dependent method to check whether a file is valid and includes techniques such as validating headers and footers, decompressing the file, or utilizing other characteristics of the particular file type at hand. This approach works well for files split into two fragments where the gap between the two fragments is sufficiently small. For files split into more than two fragments the algorithm is not suited.

Finally, there are various techniques which focus on the actual file contents rather than on the syntax of the file format. Examples are the publications by Memon and Pal (2006), Pal et al. (2008), and Tang et al. (2016). Here the authors propose to compare the pixel values of JPEG images in order to detect the fragmentation point and the fragment with the highest probability to be the next one in the file to be carved. In order to be able to do this, the data has to be completely decoded. That is, the file format has to be parsed, if there is any compression, it has to be removed, and any other encoding step has to be inverted as well, until the actual file contents are available. In fact, these approaches are a progression of the fast object validation mentioned in the paragraph before. However, instead of returning a binary value (“validates” or “does not validate”) the pixel comparison returns probabilities.

In summary, the existing approaches do not implement what we would consider as syntactical file carving. On the one hand, the are approaches utilizing features of the actual file contents. These features are usually computationally expensive and complex to obtain. On the other hand, there are approaches which only use the file structure in a very rudimentary way. Moreover, they typically use these features only to more precisely find the point where a file is corrupted or fragmented but do not implement any means to further reconstruct a fragmented file. We argue that the syntax of a file format can be used not only to find fragmentation points more precisely but also to aid in the reassembly of fragmented files.

4. Carving PNG files

In this section we describe our approach to phase-based, syntactical file carving of PNG files using their internal structure to facilitate the carving process.

4.1. Phase zero: signature search

During the initial step, the given disk image is searched for all occurrences of PNG-specific signatures. Besides the PNG start signature marking the very beginning of a PNG file, we also look for the 26 registered chunk types (i.e. their signatures) defined in the PNG specification. Since the PNG start signature was designed to reduce the possibility of incorrectly identifying data as a PNG file (W3C, 2003), it is used as the primary header signature during our carving process limiting the chance of finding false positives. The PNG start signature is compulsorily followed by the `IHDR` chunk. For this reason, we additionally check that the number of PNG start signatures and the number of `IHDR` signatures found are equal.

4.2. Phase one: chunk jumps

This phase is responsible for identifying contiguous sequences of PNG files. For non-fragmented PNGs, this results in carving the whole file. As already mentioned, each chunk in a PNG file starts with a four byte integer specifying its length. Assuming that there is no fragmentation, this information indicates the end of the current chunk as well as the beginning of the next chunk. Especially, it enables us to easily compute the expected location of the signature of the next chunk. Once we know this position, we check whether the bytes at this position are actually a valid PNG chunk type. If this is the case, we compute the CRC of the current chunk and compare it to the value stored at the end of the chunk.

This step, which we refer to as a *chunk jump*, is repeated until the last chunk of the PNG file, the `IEND` chunk, is reached. Whether a single chunk jump was successful or not, is determined by three different criteria:

1. After performing a chunk jump, there has to be **valid PNG chunk type signature.**
2. Valid PNG chunk signatures have to follow the **ordering constraints** defined in the PNG specification (W3C, 2003).
3. The **CRC checksum** at the end of the chunk has to match the CRC checksum calculated over the data jumped over.

If all chunk jumps starting at a PNG signature and ending at an `IEND` chunk were successful, as shown in Fig. 2, the PNG file is completely carved and extracted from the disk image. During this process, all non-fragmented files are carved. In case of a missing valid PNG chunk type after a chunk jump, a broken ordering constraint, or a CRC mismatch, the next phase of the carving process is initiated as soon as phase one is completed.

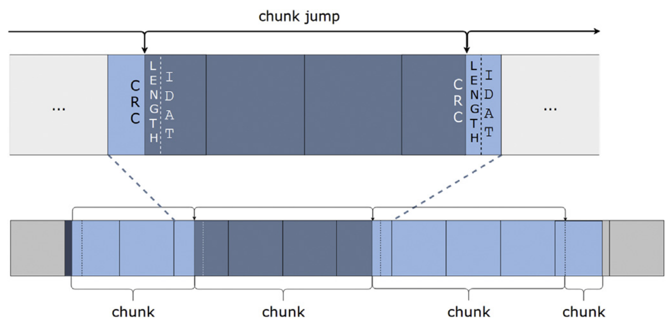

Fig. 2. Successful chunk jumps of a non-fragmented PNG file.

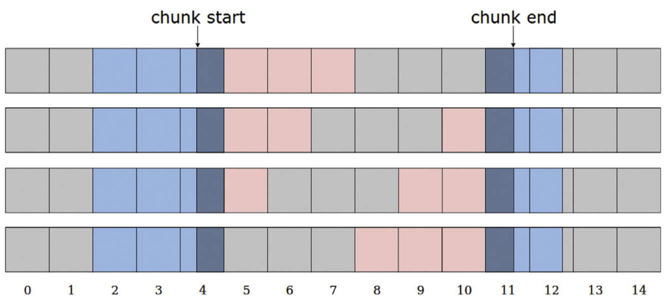

Fig. 4. Example of the bifragment block generator for combinations of three blocks of size.

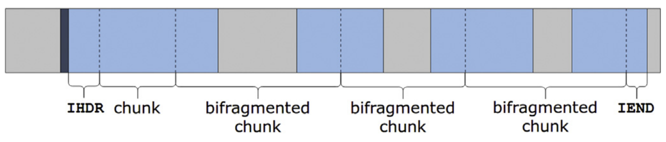

Fig. 5. Example of a PNG fragmented into four fragments.

4.3. Phase two: bifragment block generator

Fig. 3 shows an example of an unsuccessful chunk jump caused by a fragmented PNG file. Since we know that all chunk jumps, except for the last one, should end at another chunk, we are expecting a PNG chunk type signature after a valid chunk jump. We employ the database of PNG chunk type signatures created in phase zero to find the next fragment of the current file. In theory, every signature in this database could be the correct successor of our current chunk. To reduce the number of candidates we first exploit the fact that storage devices use a fixed block size when reading and writing data. On account of this fixed block size as well as the length specified in the chunk, we can compute the offset of the expected chunk type signature within its block. This allows us to filter out all chunk type signature candidates which are located at an incompatible offset. To filter out even more candidates we also consider the chunk ordering constraints. That is, only chunk types which are allowed after the current chunk are taken into account. These two filtering steps should already eliminate most of the available candidates leaving us with a comparatively small number of possible successors.

After the filtering, the combinations for the data blocks in between the known chunk start and the possible end candidates are generated. Note that we only consider data blocks of the disk image which have not been used by previously identified PNG parts. This also holds for following phases. Possible block combinations are generated by *block generators*. Given a start as well as an end block, different strategies are used to find the correct corresponding chunk data.

During phase one, the bifragment block generator generates candidates similar to the bifragment gap carving described by Garfinkel (2007). This approach exploits the fact that the majority of files are bifragmented resulting in the fragmented data being stored right after the chunk start as well as right before the chunk end. Knowing the length of the data and, thus, also the gap size, enables us to efficiently create block candidates also for non-sequential fragments. Furthermore, we use the CRC stored at the end of every chunk to validate the correct block combination. The bifragment block generator starts with the maximum number of blocks at the chunk start, which is decremented while increasing the number of candidate blocks at the chunk end, maintaining the total number of required blocks. Fig. 4 illustrates the principle of the bifragment block generator generating combinations for three blocks with start block 4 and end block 11.

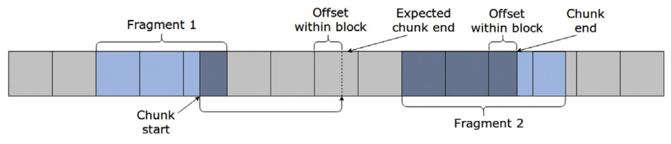

Fig. 3. Failed chunk jump of a bifragmented PNG file.

In fact, the bifragment block generator does not only succeed in the case of bifragmented PNG files. Since we are exploiting the syntax of PNG, we are able to use the detected chunk signatures as *anchor points* during the carving process. This reduces the problem of carving a whole PNG file at once to the problem of carving each chunk of the file individually. This makes the bifragment block generator applicable every time a chunk is bifragmented. As an example, Fig. 5 shows a PNG file split into four fragments. Without any additional information, finding the fragmentation points as well as reassembling the fragments is most certainly unfeasible. Using the chunk signatures of PNG, we are able to detect not only which chunks are fragmented, but also their start and possible ends. This information along with the block combinations generated by the bifragment block generator, enables us to correctly reassemble the bifragmented chunks and, finally, the whole fragmented PNG file.

4.4. Phase three: sliding window block generator

As described in the previous section, the combinations generated by the bifragment block generator succeed only for PNG files containing at most bifragmented chunks. If a chunk is split into more than two fragments, one of its fragments does not contain a PNG chunk signature and, thus, no syntax we could benefit from. Nevertheless, we still know the expected amount of data in between the first and last fragment. Moreover, the CRC checksum can still be used to identify the correct block combination.

In the very first step of this phase, a window of the size of the number of blocks required for the chunk to be complete is created. This window is then slid from the fragment containing the chunk start to the fragment containing the chunk end. An example of a sliding window block generator is displayed in Fig. 6. Note that the sliding window is also moved between the chunk end and the chunk start (not depicted in the figure) in order to handle cases where fragments are stored out of order. Also, combinations already generated during the previous phase are not generated again.

In case no correct block combination was found, the numbers of blocks following the chunk start and preceding the chunk end are fixed and increased (cf. Fig. 7). This is done since fragmentation

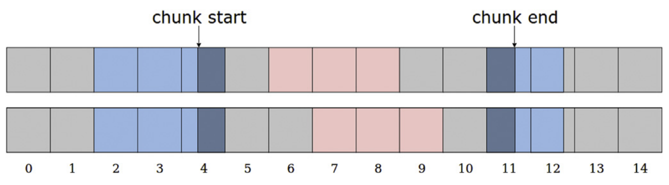

Fig. 6. Example of the sliding window combinations for three blocks.

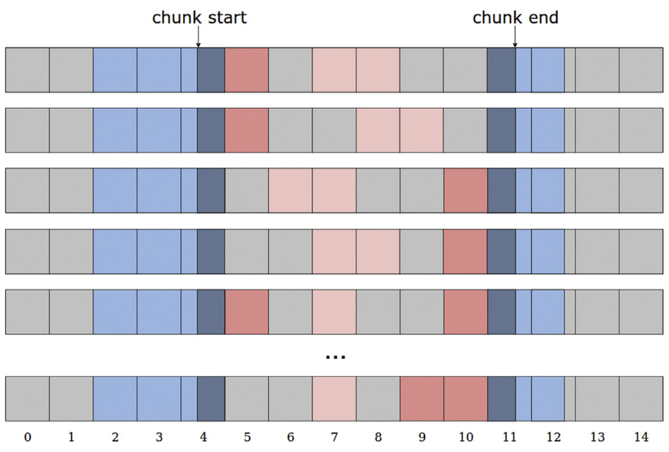

Fig. 7. Sliding window block generator for three blocks with fixed number of blocks at the start and end.

does not necessarily occur right after or before the chunk start or end block respectively. Whenever possible, the initial fixed number of blocks is chosen with respect to a logical border, e.g. the amount of blocks between the chunk start and the next PNG start signature or the end of the disk image. The size of the sliding window is adjusted according to the fixed blocks as shown in Fig. 7.

4.5. *Phase four: brute force block generator*

If none of the previous phases succeeds, the brute force block generator is initiated as a last resort. As its name suggests, all possible block combinations are brute forced and randomly generated. This approach is very time consuming even for a small number of blocks and, thus, most likely not feasible for realistic disk images. Note that we did not use this block generator in our evaluation.

4.6. *Phase five: corrupted PNGs*

In the last phase we handle corrupted PNG files. A PNG is considered to be corrupted when all block generators failed (e.g. by CRC mismatch), when the number of available blocks is smaller than the required number of blocks, or when no matching chunk signatures are left. In such cases we carve everything of the file that has been validated so far. Additionally, we append as many bytes after the fragmented chunk as indicated by its length field. Of course, this may result in random data being carved. Still, most image viewers are able to decode the image data up to the point where invalid data appears. This means that all of the correct data will still be viewable.

4.7. *Search space reduction*

One of the crucial factors to significantly improve the efficiency of the block generators—and of fragmented file carving in

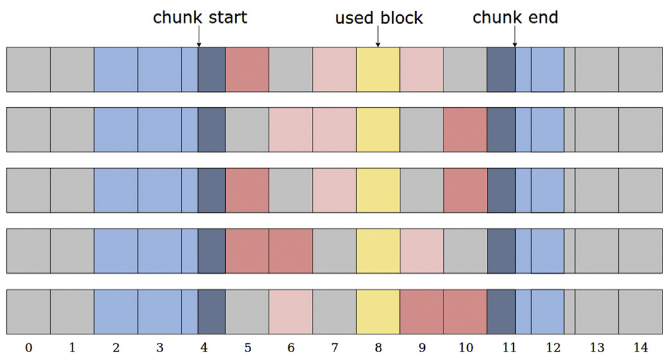

Fig. 8. Sliding window with used blocks.

general—is to reduce the number of candidates to consider when trying to find the exact fragmentation point and the beginning of the next fragment. In this section we present several techniques that we employ to enhance the efficiency of our approach.

As depicted in Fig. 8, the number of block combinations to be generated can be drastically reduced as soon as there are blocks that do not have to be considered. Such blocks are for instance those that belong to an already successfully carved file. Therefore, we keep track of all blocks that belong to parts of a file which have been validated and do not consider them any further. That is, the block generators will skip already used blocks and will not generate combinations containing them.

Starting with phase 3, we also run the block generators of the previous phases, before applying the block generator of the current phase. Due to newly added used blocks, the simpler and less time-consuming block generators may succeed now.

What is more, the previously described phases are not only started from the PNG start signatures. Instead, they are initiated for each and every detected chunk signature as well. Doing so results in already successfully validated larger parts of PNG files. This approach not only increases the number of blocks which are marked as used, it also reduces the number of chunk signatures to consider when trying to find the next fragment of a file. Considering that a large part of the files will not be fragmented or at least will not be fragmented into a lot of fragments, these techniques result in a significant reduction of the search space.

Finally, all of the operations can easily be carried out in parallel since we only work on the file structure and do not need any data from already validated parts of a file. All of this is possible due to the syntax of PNG files enabling us to identify and validate fragments independently from other carving processes.

5. File carving datasets

Grajeda et al. (2017) already emphasized that the availability of datasets for the digital forensics community is far from perfect. Less than 4% of the datasets created for research are made available and shared with the public afterwards. This thwarts reproducibility and makes a meaningful comparison between methods and tools very difficult. Moreover, the creation of datasets from scratch each and every time is not only time-consuming, but also complex and error-prone if done manually.

For the evaluation of file carvers, many different scenarios of file fragmentation and corruptions have to be created. The DFRWS forensic challenges from 2006 to 2007 provided test files containing such scenarios particularly designed to evaluate file carving approaches. Furthermore, the creators did not only provide a detailed listing of the used scenarios, but also the exact offsets of every fragment in the test files. Therefore, these files are still used nowadays to show the usefulness of new carving algorithms.

Understandably, the challenges did not cover each and every existing file type. Unfortunately, PNG was one of the file types not included in any of the challenges. For this reason, we decided to create an image generation framework specifically designed to aid in the evaluation of file carvers. Our goal was to provide an easy to use framework for the creation of even complex file carving scenarios similar to those covered in the DFRWS challenges.

5.1. *Automatic generation of file carving datasets*

Fragments are the basic building block used in our framework. A fragment can be either a `FileFragment` or a `FillerFragment`. A `File Fragment`—as the name suggests—is a fragment of a certain file and contains real data to be carved. `FillerFragments`, on the other hand, are fragments of a given size with user-defined or random data. On each of these fragments, file or filler, it is possible to perform a range of operations:

- New fragments can be created by **splitting** an existing fragment. This is possible by dividing a fragment into a user-defined number of uniformly or randomly sized fragments as well as providing the exact fragmentation offsets. This can also be done recursively, i.e. splitting a created fragment again. Since each fragmentation is carried out with block-sized granularity, only the last fragment of a file can be smaller than the specified block size.
- Another way of manipulating an existing fragment is to **remove** parts of it. This includes methods for removing the start or end of a fragment, e.g. to remove headers and footers of a file. The amount of data to be removed is specified in bytes. Also, the result of these two operations is a single fragment object. On the other hand, removing multiple blocks in the middle results in two fragments.

In the next step, fragments are combined into a **Scenario**. A scenario acts as a wrapper around its fragments making it possible to perform various operations.

- **Adding** a fragment to a scenario is the most simple operation in order to create any possible configuration of fragments manually.
- The **intertwine** operation is used to add new fragments in an alternating sequence in between the existing fragments of a scenario. E.g., it can be used to add fillers in between fragments of an existing scenario.
- Other operations on a scenario include **shuffling**, **reversing** or explicitly **changing the order** of its current fragments.

In the end, multiple scenarios are added to an **Image** used to log information about the exact offsets of the used fragments as well as to write the data to disk. Moreover, the fragments are aligned to a specified block size by padding smaller fragments when required.

5.2. *DFRWS file carving challenge*

As already mentioned, the DFRWS challenges from 2006 to 2007 featured datasets covering various scenarios for the evaluation of file carvers. Since these images did not include any PNG files, we decided to recreate the scenarios using our framework. For this purpose, we initially surveyed the fragmentation scenarios included in the challenges and organized them by dividing their characteristics into the following four categories:

- **Files:** This includes not only the *number of files* used for a scenario, but also *which files* have been selected. The maximum number of files for a scenario used throughout the challenge is four.
- **Fragments:** The files are furthermore divided into a certain *number of fragments.* For this step, the *fragmentation points* are an important property. In the DFRWS challenges, the fragmentation points appear to have been chosen at random for each fragmentation within a scenario while the maximum number of fragments created per file was four. Additionally, some scenarios had some fragments *missing*, including the start, middle, or end of a fragmented file.
- **Order:** All of the created fragments have been arranged in a certain *order.* This can also mean simply reversing the given fragments. *Intertwining* fragments also belongs into this category since it is simply a specific way of ordering fragments. For most of the scenarios in the DFRWS challenge, an exact order has been provided. The order of the fragments in the intertwined scenarios, though, does not appear to be following a particular ordering scheme.
- **Filler:** Some scenarios make use of *fillers* between fragments. Fillers in the DFRWS challenges are either random data or (parts of) other files. Aligning fragments to clusters by using *padding* is also part of this category. Each fragment in the DFRWS is padded, so that it is aligned to the given block sizes of 512 bytes.

Subsequently, we generalized the scenarios with respect to the aforementioned criteria and used them to create our evaluation scenarios containing PNG files. As in the original challenges, the exact fragmentation points of the files are chosen randomly in our scenarios. This also holds for the order and number of fragments per file in the sequential intertwine scenarios. Furthermore, the PNGs were selected randomly from a custom dataset consisting of PNG files varying in size, structure and content. These PNGs as well as any accompanying files required to recreate our scenarios are publicly available (Hilgert et al., 2019) to enable reproducibility.

Our evaluation scenarios including their characteristics are listed in Table 1. Some scenarios contain PNG files which are incomplete (indicated by a †). Moreover, scenarios 19, 20, and 22 contain PNG files missing their beginning. In these cases, it is already certain from the beginning that it will not be possible to display these PNGs even if they were carved.

6. Evaluation

For the evaluation, we chose to compare our prototypical implementation of a syntactical PNG file carver with the most prominent open source file carvers, namely Scalpel, Foremost, and PhotoRec. Scalpel uses a configuration file defining the headers as well as the footers of a certain file type. Unfortunately, in the default configuration for PNG, neither the header nor the footer byte sequences were correct. We modified the configuration file according to the PNG specification using `89 50 4e 47 0d 0a 1a 0a` for the header and `49 45 4e 44 ae 42 60 82` for the footer. Additionally, we adjusted the maximum PNG file size to 200 MB as some of the files in our dataset are larger than the default size of 20 MB. Foremost is run with its built-in PNG extractor not using the configuration file. Moreover, we enabled the options to keep corrupted or incomplete files (e.g. due to missing footers), since some of our scenarios contain incomplete files. Apart from the aforementioned options, we used the default configurations of the carvers.

For the evaluation, we created 50 instances of each of the scenarios listed in Table 1. Each of these instances is unique with regard to the fragmentation points, the used PNG files, and the padding and filler data resulting in a broader test coverage.

Table 1
Scenarios created to evaluate our carver. An asterisk (*) indicates that the value was generated randomly and may vary between the different instances of the scenario, a † indicates scenarios with missing fragments.

Scenario	PNGs	Fragments	Description	Order
1	1	1	non fragmented	1
2	1	2	non-sequential	1b 1a
3	1	2	with filler	1a FILLER 1b
4	2	2	with PNG in between	1a 2 1b
5	1	3	with filler	1a FILLER 1b FILLER 1c
6	1	3	non-sequential	1a 1c 1b
7	1	3	non-sequential	1b 1a 1c
8	1	3	non-sequential	1b 1c 1a
9	1	3	non-sequential	1c 1a 1b
10	1	3	non-sequential	1c 1b 1a
11	1	4	non-sequential	1a 1c 1b 1d
12	1†	2	missing end	1a
13	1†	3	missing middle	1a 1c
14	1†	3	missing end	1a FILLER 1b
15	1†	3	missing middle, non-sequential	1c 1a
16	1†	3	missing end, non-sequential	1b 1a
17	2	2	intertwined, non-sequential	1b 2b 1a 2a
18	2†	2	intertwined, 2nd missing end	1a 2a 1b
19	2†	2	1st missing end, 2nd missing start, continuous	1a 2b
20	2†	2	1st missing end, 2nd missing start, with filler	1a FILLER 2b
21	2†	2	1st missing end, 2nd out of order	1a 2b 2a
22	2†	2	2nd missing start and in between 1st fragments	1a 2b 1b
23	2†	2	1st missing end, with filler	1a 2a FILLER 2b
24	2	*	intertwined, sequential	*
25	3	*	intertwined, sequential	*
26	4	*	intertwined, sequential	*

All carvers were run on each scenario instance with the aforementioned configurations. Additionally, we set a timeout of 1 h per run in order to terminate cases in which a carver would take an impracticably long time.

The results of the carvers were evaluated manually and each carved file was graded into one of the following five categories:

- **Complete** carved files are those, that are viewable and show the complete image data without any alteration of colors, transparency or content. This also includes carved files with additional data after their actual end. Note that for particular scenarios (marked with a †), the term *complete* refers to the actual part of the PNG that could have been carved and displayed (e.g. only the first part of a PNG missing its middle).
- Carved files with **minor alterations** are those which can be opened with an image viewer and at least 50% of the image content has to be viewable. Moreover, only minor changes in color and transparency are allowed.
- On the other hand, carved files with **major alterations** include PNG files whose majority of the content (<50%) cannot be displayed.
- **Corrupted** or **missing** files include files which cannot be opened with an image viewer, files which do not contain any actual image data (e.g. just a transparent or black rectangle), or PNGs which are completely missing.

False positives did not occur during our evaluation and are thus not present in the table.

We used GIMP in version 2.8.22 linked with libpng in version 1.6.34 for the assessment of the carved files. GIMP proved to be very generous with respect to CRC errors in PNG, meaning that even corrupted files were decoded and displayed as far as possible.

Table 2 lists the results of the carvers. The first column indicates the scenario number and the second column the number of *expected* PNG files. Note that for the scenarios 19, 20, and 22 this number is different from the number of PNGs in this scenario (cf. Table 1). This is because these scenarios contain files with their start missing and we do not expect these files to be carved and neither do we expect them do be viewable. Each carver column is divided into five sub-columns. The first one indicates the total number of files carved, the second one the *complete* files, followed by the files with *minor alterations*, then files with *major alterations*, and finally *missing* or *corrupted* files. Since the last sub-column means corrupted or missing, the numbers of the last four sub-columns do not necessarily sum up to the number of total files carved (first sub-column).

The first thing to notice is that none of the file carvers produced any false positives in our scenarios.

Our syntactical PNG file carving approach was able to successfully carve all instances of 20 of the 26 different scenarios. We investigated the overall 38 files which have not been completely carved and found that these files were incomplete because our carver was stopped after the timeout of 1 h. A closer look at the scenario instances revealed that the files were fragmented in such a way that some of the fragments contained not enough structure for our approach to be efficient. Most of the times these timeouts occurred when a single `IDAT` was fragmented into more than two fragments. This results in at least one fragment containing only compressed data and no syntax that our carver could benefit from. In such cases, a vast number of block combinations has to be tested before the correct one is found. For single instances, we tested our carver without a time limit and found that it eventually carved the corresponding files completely.

The other evaluated carvers performed well when the data to be carved was stored contiguously, i.e. when the complete files or, in case of missing data, the remaining fragments were stored without any fragmentation and in correct order. On the other hand, even in simple fragmentation scenarios such as scenario 3, none of the other carvers was able to carve the files completely.

While Scalpel performed as expected, we found that Foremost and PhotoRec performed worse than supposed. For Foremost we found that there is a hard-coded check whether the image dimensions parsed from the `IHDR` are smaller than 3000×3000 pixels. Some of the PNGs in our dataset are larger than that

Table 2
Evaluation results. The carver sub-columns indicate the *total files* carved, *complete* files, files with *minor alteration*, files with *major alteration*, *missing/corrupted* files. A † indicates scenarios with missing fragments.

#	PNGs	pngCarver					Foremost					Scalpel					PhotoRec				
1	50	50	50	0	0	0	50	13	0	37	0	50	50	0	0	0	50	50	0	0	0
2	50	50	50	0	0	0	50	0	4	46	0	50	2	28	20	0	37	0	14	23	13
3	50	50	50	0	0	0	50	0	5	45	0	50	0	27	23	0	0	0	0	0	50
4	100	100	100	0	0	0	96	13	5	78	4	100	51	32	17	0	83	81	2	0	17
5	50	50	48	0	2	0	50	0	2	48	0	50	0	18	30	2	0	0	0	0	50
6	50	50	48	1	1	0	50	0	1	49	0	50	0	15	35	0	0	0	0	0	50
7	50	50	50	0	0	0	50	0	6	42	2	50	0	17	31	2	4	0	0	4	46
8	50	50	50	0	0	0	50	0	2	48	0	50	0	18	31	1	40	0	10	30	10
9	50	50	50	0	0	0	50	0	4	46	0	45	0	29	16	5	36	0	20	16	14
10	50	50	50	0	0	0	50	0	0	50	0	50	0	15	31	4	45	0	5	41	4
11	50	50	44	1	5	0	50	0	0	50	0	50	0	5	45	0	6	0	0	6	44
12	$50^\dagger$	50	50	0	0	0	50	9	6	34	1	50	50	0	0	0	22	22	0	0	28
13	$50^\dagger$	50	50	0	0	0	50	8	18	24	0	50	49	0	1	0	0	0	0	0	50
14	$50^\dagger$	50	37	8	5	0	50	0	11	37	2	50	0	28	21	1	7	0	4	4	42
15	$50^\dagger$	50	50	0	0	0	50	16	11	23	0	50	49	1	0	0	41	41	0	0	9
16	$50^\dagger$	50	36	7	7	0	50	0	13	37	0	50	0	32	18	0	45	0	28	17	4
17	100	100	100	0	0	0	95	0	11	79	10	100	0	44	54	2	100	0	45	53	2
18	$100^\dagger$	100	100	0	0	0	97	11	14	72	3	100	50	33	17	0	50	0	31	19	50
19	$50^\dagger$	50	50	0	0	0	50	9	7	34	0	50	50	0	0	0	0	0	0	0	50
20	$50^\dagger$	50	50	0	0	0	50	13	5	32	0	50	50	0	0	0	2	2	0	0	48
21	$100^\dagger$	100	100	0	0	0	100	17	17	66	0	100	50	29	21	0	54	5	26	23	46
22	$50^\dagger$	50	49	0	1	0	50	15	6	29	0	50	39	8	3	0	0	0	0	0	50
23	$100^\dagger$	100	100	0	0	0	97	10	15	72	3	100	50	28	22	0	50	43	7	0	50
24	100	100	100	0	0	0	95	0	13	82	5	100	0	39	61	0	50	0	19	31	50
25	150	150	150	0	0	0	145	0	15	130	5	150	0	49	99	2	75	0	23	52	75
26	200	200	200	0	0	0	193	0	30	162	8	200	0	80	119	1	104	0	49	53	98

resulting in Foremost carving only the first 1 MB of these files. For PhotoRec we did not find an obvious reason why it did not perform as expected.

7. Limitations and future work

Although our approach proved to be able to carve files correctly even in complex fragmentation scenarios, there are still some limitations which will be discussed in this section.

In cases of missing fragments in the middle of a PNG file, our approach is only restoring the first part and discarding the rest. Padding the missing parts in these cases is not useful, since such PNGs can only be displayed up to the padded part anyway. This is due to the fact that padded data most certainly corrupts the compressed PNG data stream. Here, it would be necessary to pad with data which does not interfere with the decompression of the data stream. This is a complex research topic on its own and beyond the scope of this paper.

For the detection of succeeding chunk candidates, our approach relies on the block alignment used by storage devices as described in Section 4.3. For most storage devices the fixed sector size is well-known and can, thus, be used by our implementation. Modern file systems like ZFS on the other side make use of dynamically-sized blocks. Therefore, it is necessary to examine if and how this behavior influences our approach—especially when multiple and various block sizes are used on a single file system.

Finally, our syntactical approach for PNG file carving is obviously highly dependent on the syntax of PNG. Thus, fragmented PNG files with a lack of syntactical information do not benefit from our approach and require more time to be carved. Moreover, fragmentation points at certain offsets, e.g. right before a chunk signature or in the middle of a CRC, hamper our approach and result in more block combinations having to be generated. Though these cases were not encountered during our evaluation, they need to be considered and handled in future developments.

8. Conclusion

Syntactical file carving utilizes the syntax of a file format to the maximum extent. Using PNG as an example, we showed that this approach enables effective file carving even in very complex fragmentation scenarios. Of course, syntactical file carving can only be applied to file types having a sufficiently well-defined syntax. In such cases, however, exploiting features of the syntax can drastically simplify the carving process. For instance, it is often possible to very easily carve the single parts of a fragmented file independently. Afterwards, these parts only have to be reassembled. Moreover, the operations required to apply syntactical file carving are often less complex and less computationally expensive than content-based approaches.

Since we were not able to find suitable datasets to evaluate our approach, we created a framework allowing the easy and automated creation of dataset which can be used in the assessment of file carvers. Using our framework, datasets similar to the ones used in the DFRWS forensic challenges from 2006 to 2007 can be created with just a few lines of code and configuration. We hope that this framework fosters the research of more advanced file carving approaches as well as the availability of datasets open to the public to enable the reproducibility of published evaluation results.

References

Boutell, T., 1997. PNG (Portable Network Graphics) Specification Version 1.0 (Technical Report).

Carrier, B., Casey, E., Venema, W., 2006. DFRWS 2006 Forensics Challenge. URL: http://old.dfrws.org/2006/challenge/.

Carrier, B., Casey, E., Venema, W., 2007. DFRWS 2006 Forensics Challenge. URL: http://old.dfrws.org/2007/challenge/.

Garfinkel, S.L., 2007. Carving contiguous and fragmented files with fast object validation. Digit. Invest. 4, 2–12.

Grajeda, C., Breitinger, F., Baggili, I., 2017. Availability of datasets for digital forensics–and what is missing. Digit. Invest. 22, S94–S105.

Grenier, C., 2015. PhotoRec - digital picture and file recovery. URL: https://www.cgsecurity.org/wiki/PhotoRec.

Hilgert, J.N., Lambertz, M., Rybalka, M., Schell, R., 2019. fkie-cad/png-carving. URL: https://github.com/fkie-cad/png-carving.

Kendall, K., Kornblum, J., Mikus, N., 2009. Foremost. URL: http://foremost.sourceforge.net/.

Marques, T., 2016. PNG Embedded – malicious payload hidden in a PNG file. URL: https://securelist.com/png-embedded-malicious-payload-hidden-in-a-png-file/74297/.

Memon, N., Pal, A., 2006. Automated reassembly of file fragmented images using greedy algorithms. Trans. Img. Proc. 15, 385–393. https://doi.org/10.1109/TIP.2005.863054. URL: https://doi.org/10.1109/TIP.2005.863054.

Pal, A., Sencar, H.T., Memon, N., 2008. Detecting file fragmentation point using sequential hypothesis testing. Digit. Invest. 5, S2–S13. https://doi.org/10.1016/j.diin.2008.05.015. URL: https://doi.org/10.1016/j.diin.2008.05.015.

Richard III., G., Marziale, L., 2013. sleuthkit/scalpel: Scalpel is an open source data carving tool. URL: https://github.com/sleuthkit/scalpel.

Richard III., G., Roussev, V., 2005. Scalpel: a frugal, high performance file carver. In: Proceedings of the Digital Forensic Research Conference.

Roelofs, Greg, 2015. PNG (portable Network graphics) register and extensions. URL: http://www.libpng.org/pub/png/spec/register/pngreg-1.4.6-pdg.html.

Roelofs, G., Koman, R., 1999. PNG: the Definitive Guide. O'Reilly & Associates, Inc.

Shah, S., 2015. Stegosploit – exploit delivery via steganography and polyglots. URL: http://stegosploit.info/.

Tang, Y., Fang, J., Chow, K., Yiu, S., Xu, J., Feng, B., Li, Q., Han, Q., 2016. Recovery of Heavily Fragmented Jpeg Files. Digital Investigation 18, pp. S108–S117. URL: http://www.sciencedirect.com/science/article/pii/S1742287616300512 https://doi.org/10.1016/j.diin.2016.04.016.

W3C, 2003. Portable Network Graphics (PNG) Specification, second ed. URL: https://www.w3.org/TR/PNG/.