Domain-Specific Optimization in Digital Forensics

Jeroen van den Bos[1,2] and Tijs van der Storm[1]

[1] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[2] Netherlands Forensic Institute, Den Haag, The Netherlands
jeroen@infuse.org, storm@cwi.nl

Abstract. File carvers are forensic software tools used to recover data from storage devices in order to find evidence. Every legal case requires different trade-offs between precision and runtime performance. The resulting required changes to the software tools are performed manually and under the strictest deadlines.
In this paper we present a model-driven approach to file carver development that enables these trade-offs to be automated. By transforming high-level file format specifications into approximations that are more permissive, forensic investigators can trade precision for performance, without having to change source.
Our study shows that performance gains up to a factor of three can be achieved, at the expense of up to 8% in precision and 5% in recall.

1 Introduction

Digital forensics is a branch of forensic science that attempts to answer legal questions based on the analysis of information recovered from digital devices. These digital devices are typically computers or mobile phones confiscated from a suspect, found near a crime scene or otherwise expected to have information stored that is relevant to an investigation. In the context of this paper we are interested in *file carvers*: tools that recover data from storage devices without the help of (file system) storage metadata [16].

The current growth in size of storage devices requires that file carvers scale to analyze data in the terabyte range. Moreover, forensic investigations are often performed under very strict deadlines, making the runtime performance of such tools critical. Additionally, the large diversity in (variants of) file formats encountered on devices requires these tools to be easy to modify and extend.

Because each case may require different trade-offs with respect to precision and runtime performance, file carvers often need to be modified on a case-by-case basis. Currently, this kind of just-in-time "carver hacking" is performed by hand, which is error prone and time consuming; it is also inherently incompatible with very strict deadlines.

In previous work we have developed a model-driven approach to digital forensics tool construction [5]. In this work the file formats of interest, e.g., JPEG, GIF etc., are declaratively modeled using a domain-specific language (DSL) called Derric. These descriptions are then input to a code generator that produces highly efficient and accurate format validators that form an essential part of our file carver EXCAVATOR.

EXCAVATOR competes with file carvers widely used in practice, and is much easier to maintain due to the high-level DERRIC language. Nevertheless, the generated components encode a particular trade-off between precision and runtime performance. In this work we apply model transformations on DERRIC descriptions in order to make this trade-off configurable. We present three model transformations that successively obtain format validators that are more permissive (i.e., produce more false positives) but exhibit better runtime performance. As a result forensic investigators can choose between precision and runtime performance without having to change any code.

We have evaluated EXCAVATOR using the different format validators at each permissiveness configuration for the file formats JPEG, GIF and PNG on a representative test image of 1TB. Our results show that performance gains up to a factor of three can be achieved, at the expense of up to 8% in precision and 5% in recall.
This paper makes the following contributions:

- We present three model transformations to automatically derive format validators that trade precision for better runtime performance.
- We evaluate our approach on a representative test image in the terabyte range showing that substantial performance gains can be achieved.

Organization of this paper The rest of this paper is organized as follows. Section 2 discusses file carving and analyzes the development, performance and scalability challenges in the engineering of digital forensics software. We introduce our model-driven approach to building file carvers and discuss how it addresses the challenges. This includes an overview of DERRIC, our domain-specific language (DSL) for file format description. Section 3 defines three model transformations on DERRIC descriptions. Section 4 evaluates the effect of the model transformations on the runtime performance and precision of the generated carvers. In Section 5 we discuss our results. Related work is discussed in Section 6. We summarize our research and results in Section 7.

2 Background

2.1 File Carving

When recovering data from a storage device, all available metadata such as file system records and application logs are used to identify locations where data is stored. After this initial step, there is usually a significant amount of *unallocated space* left on the storage device. This space may contain only zeros (or some other factory default value), but may also contain deleted files, operating system caches or data that has been hidden on purpose. To recover this data, a content-based technique called *file carving* can be used.

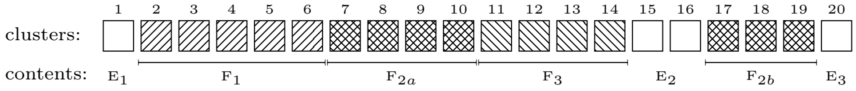

Fig. 1. An example set of contiguous clusters on a storage device

A typical modern file carver consists of a set of format validators used by one or more file reconstruction algorithms. In its most basic form the format validators consist of checking for format-specific constants at the start and end of a stream (called *header/footer matching*) and the file reconstruction algorithm simply moves through the input stream in a single pass, invoking all format validators at each offset to determine whether a file is located there. On each hit, the identified file is saved for further analysis.

Apart from generating a large amount of false positives, this approach has another drawback: it is unable to recover files that are split into multiple parts and stored in non-contiguous locations. This so-called file fragmentation is common, usually as a result of performance optimization by the operating system and implementation details of the file system.

To recover fragmented files but avoid a combinatorial explosion, file carvers implement file reconstruction algorithms, such as bifragment gap carving [10]. However, to improve precision and reduce the amount of required iterations to reconstruct a single file, they also use more advanced format validators that validate (part of) the format's structure and content.

Common optimizations include running multiple format validators on the same block of data concurrently and applying data classification techniques to reduce the search space (e.g., removing blocks of zeros). These techniques are not discussed further in this paper.

File Carving Example An example set of contiguous clusters commonly found on storage devices is shown in Figure 1. Clusters 1, 15, 16 and 20 contain only zeros. The remaining clusters contain three files: F_1 (clusters 2–6), F_2 (fragmented, clusters 7–10 and 17–19) and F_3 (clusters 11–14).

A traditional file carver that performs a single pass over the data checking for headers and footers only will probably recover F_1, since it will find a header in cluster 2 and a correct following footer in cluster 6. Fragmented file F_2 is problematic, as the first footer following the header in cluster 7 is F_3's footer in cluster 14. As a result, both F_2 and F_3 are not recovered.

A more sophisticated format validator may detect a problem around cluster 11 or 12 and report this to the file carver. The file carver can then decide to look for suitable footers within a certain range, possibly finding both F_3's footer in cluster 14 as well as F_2's footer in cluster 19. Some shuffling of the clusters between the original error location in cluster 11 and the potential footers may lead the file carver to consider clusters 7–10 and 17–19, which the format validator will accept. From the remaining clusters, F_3 will then be easy to recover as well.

2.2 Software Engineering Challenges

From a software engineering perspective, the challenges in file carver construction can be classified into three areas, described in the following subsections.

Modifiability Digital forensics tools must be continually adapted to new versions and variants of storage formats encountered during investigations. For instance, even when using a standardized format such as the JPEG image file format, different vendors of, for instance, digital cameras may store the actual files in different ways, often deviating from the standard. When forensic investigators encounter traces on some device that they want to recover or analyze, they often need to adapt their tools to these new, modified or different storage formats in order to maximize recoverable evidence.

Runtime Performance Strict time constraints means that analyses must be completed as quickly as possible, even when the amount of data to analyse grows very fast. Brute force algorithms are intractable when it comes to reconstructing a file by finding its parts in a set of millions of fragments. Hence, the challenge is to use as much domain-specific knowledge as possible for optimization. This includes knowledge about hardware, operating systems, file system implementation, file formats and typical fragmentation patterns [10].

Scalability Digital forensics tools must be scalable to deal with relatively large data sizes. Common hard drive sizes in desktop computers are already in the terabyte range. Support for these data sizes imposes additional constraints on the design and implementation of tools. Recovering evidence from a set of data of which 1% barely fits into working memory requires custom approaches. Most analyses must use a streaming architecture to collect information while reading through the data from beginning to end in a single pass.

2.3 Model-driven Digital Forensics

To address the challenges described in the previous section, we have developed a model-driven approach to file carver construction, called EXCAVATOR. The architecture of EXCAVATOR consists of three parts and is shown in Figure 2.

The first part is a domain-specific language called DERRIC that allows file formats to be specified in a declarative way. A simplified example of a DERRIC specification of the PNG image file format is shown in Figure 3, which will be discussed in more detail below. A DERRIC file format description captures the information to be used by a file carver to recognize (fragments of) files in a data stream. DERRIC file format descriptions are tailored to digital forensics applications; they may leave out details of a file format that would be relevant for implementing a file viewer, for instance, but are not important for file carving.

The DERRIC file format descriptions are input to the second component, a code generator to obtain format validators. A format validator is used to check

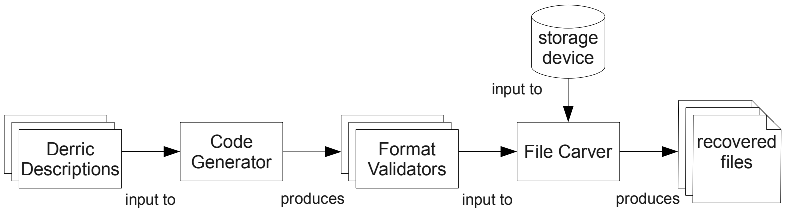

Fig. 2. Overview of the EXCAVATOR architecture

that a certain sequence of bytes indeed can be recognized as part of a file format. The code generator performs domain-specific optimizations to make the resulting code as efficient as possible, such as skipping over blocks of data that will not be interpreted and only generating variables for values read from the input data that will actually be referenced. Both the DERRIC DSL[3] and the EXCAVATOR code generator have been developed using RASCAL[4], a DSL for source code analysis and transformation [13]. The code generator produces Java source code.

The third part is the file carver itself, which employs dedicated algorithms and heuristics for locating candidate files in the data stream. This component uses the generated format validators to verify if a candidate file is an instance of a file format. This component can be considered the runtime system of EXCAVATOR. The runtime system is implemented in Java using the latest IO libraries for maximum throughput.

EXCAVATOR can be configured to run with or without file reconstruction capabilities. The algorithm it implements is bifragment gap carving with a configurable maximum gap size, with a default value of 2MB. It supports a variable cluster size with a default value of 4096 bytes. It does not support parallelism or filtering through data classification.

Our model-driven approach to digital forensics tool construction addresses the aforementioned challenges in the following way:

– **Modifiability** Using high-level file format descriptions separates the "what" from the "how": if a new variant or version of a file format has to be accomodated, only the file format description has to be changed; the code generator and runtime system remain unchanged.
– **Runtime performance** The code generator can apply sophisticated optimizations to obtain fast code. Because this concern is now isolated in the code generator, it does not affect the description of file formats. Traditionally, optimizations in digital forensics tools are tangled with the matching logic of file format structure.

[3] `http://www.derric-lang.org/`
[4] `http://www.rascal-mpl.org/`

– **Scalability** The runtime system effectively captures the way data is processed, independently from the generated validators. This means that a file carver can be made to run in streaming fashion by changing the runtime system. Additionally, state-of-the-art file carving algorithms (e.g., [8]) can be plugged into the system without affecting the other components.

Still, there is room for improvement. Digital forensics tools are often adapted to a certain situation in order to trade quality and completeness of the results for increased performance. On the one hand, if a recovery tool produces many false positives, this may be problematic, because they all have to be inspected manually. On the other hand, this may be preferable to not having any results at all before the deadline. In order to make this trade-off configurable we can apply model transformations to Derric file format descriptions to obtain a faster file carver at the cost of some precision. These transformations are described in Section 3.

2.4 Example: PNG Image File Format

As an illustration of Derric, we present a description of a simplified version of the PNG image file format in Figure 3. It omits the details of optional data structures but is complete enough to be transformed into a validator that properly recognizes PNG files.

At the beginning of the format description, the name of the format is specified (line 1) along with a set of storage-related defaults, such as string encoding (line 2) and default numerical type (lines 3–6), in this case single-byte unsigned integers.

Next is the definition of the format's *sequence* (lines 8–11), which defines the ordering of data structures in a valid file. In this example only a single operator appears (asterisk), which specifies that the structure must appear zero or more times. Additional constructs exist such as selection (parentheses), subsequencing (square brackets), optionality (question mark) and exclusion (exclamation mark).

The final part is the *structures* block (lines 13–54), defining the structures mentioned in the *sequence*. Each structure has a name and a list of field descriptions between curly braces. For example, the *Chunk* structure on lines 18–27 has four fields: *length* (line 19), *chunktype* (line 20), *chunkdata* (line 21) and *crc* (lines 22–26).

The *Chunk* structure's fields demonstrate some of Derric's specification constructs. The *length* field has the length of the *chunkdata* field as value, and its type is a 32-bit unsigned integer. The *chunktype* field is four bytes in size and may contain any value except the ASCII string "IDAT". The *chunkdata* field does not specify its value but constrains that its size must correspond to the value of the *length* field. Circular references like this are common in format descriptions and are useful in situations where only part of a data structure has been recovered; each value can be used to validate the other.

```
 1 format PNG
 2   strings ascii
 3   sign false
 4   unit byte
 5   size 1
 6   type integer
 7
 8 sequence
 9   Signature IHDR
10   Chunk* IDAT IDAT* Chunk*
11   IEND
12
13 structures
14 Signature {
15   marker: 137,80,78,71,13,10,26,10;
16 }
17
18 Chunk {
19   length: lengthOf(chunkdata) size 4;
20   chunktype: !"IDAT" size 4;
21   chunkdata: size length;
22   crc: checksum(algorithm="crc32-ieee",
23          init="allone",start="lsb",
24          end="invert",store="msbfirst",
25          fields=chunktype+chunkdata)
26        size 4;
27 }
28 IHDR = Chunk {
29   chunktype: "IHDR";
30   chunkdata: {
31     width: !0 size 4;
32     height: !0 size 4;
33     bitdepth: 1|2|4|8|16;
34     colourtype: 0|2|3|4|6;
35     compression: 0;
36     filter: 0;
37     interlace: 0|1;
38   }
39 }
40
41 IDAT = Chunk {
42   chunktype: "IDAT";
43   chunkdata: compressed(
44                algorithm="deflate",
45                layout="zlib",
46                fields=chunkdata)
47              size length;
48 }
49
50 IEND {
51   length: 0 size 4;
52   chunktype: "IEND";
53   crc: 0xAE, 0x42, 0x60, 0x82;
54 }
```

Fig. 3. Structure of the simplified PNG image file format

Finally, the *crc* field has a fixed size of four bytes and defines a value that must be calculated using the "crc32-ieee" algorithm (line 22) using the values of the *chunktype* and *chunkdata* fields (line 25).

Additionally, Derric supports structure inheritance. This is shown on line 28 where the *IHDR* structure inherits the fields of the *Chunk* structure and then overrides the *chunktype* and *chunkdata* fields (lines 29–38). Its *length* and *crc* fields remain the same as in *Chunk*.

3 Transforming Derric Models

In order to make the trade-off between precision and runtime performance configurable we have implemented three model-transformations on Derric descriptions, based on an analysis of validation techniques in file carving [3]. Each transformation removes constraints so that more permissive specifications are obtained. The transformations consist of replacing computationally expensive operations with cheaper versions that resemble the original technique, or skip over data entirely instead of processing it. They can be applied successively so that in the end four format validators can be derived from a Derric specification. The transformations are source-to-source transformations; as a result, the generic code generator of Excavator can be reused to obtain a working format validator from each transformed description.

Using the transformations, we can distinguish four configurations of format validator precision:

- **Base**: base validator (the most precise validator, based on the complete file format description).
- **NoCA**: removal of all content analysis (e.g., removal of CRC checks, data decompression, etc.).
- **NoDD**: removal of all data dependencies (e.g., a field's value becomes undefined if it used to be equal to the contents of some other field's value).
- **Header**: removal of all matching except header and footer patterns.

Although each transformation could be applied independently, for the purpose of this paper we only consider the consecutive application of each transformation. The effect of other combinations of transformations is left as future work. The transformations are described in more detail below.

Remove Content Analysis The most computationally expensive technique is content analysis, which is the interpretation and validation of a file's content, as opposed to matching structural metadata. For instance on lines 22–26 of Figure 3 a CRC32 over each *Chunk* of PNG data is defined using the **`checksum`** keyword. Additionally, lines 43–46 describe the compression scheme used by the *IDAT* structure using the **`compressed`** keyword. Removing these expensive analyses will reduce running time significantly at the cost of missing some fragmented files due to lower precision.

Removing content analysis consists of one of two rewrites, based on the field the content analysis is defined on:

- If the field has an externally defined size, i.e., if it has a fixed value (such as the CRC32's four bytes) or references an outside value (such as the *IDAT*'s reference to its *length* field), the field's value specification is removed. As a result, the data will be skipped over instead of processed.
- When the end of a field is specified by an end marker as part of the content analysis itself, the end marker is lifted out of the content analysis specification to be used to specify the end of the field.

More precisely, the transformation is defined by the following two rules:

$$f\texttt{:}\ CA\texttt{(}\overline{x}\texttt{)}\ \textbf{size}\ n\texttt{;}\ \Rightarrow f\texttt{:}\ \textbf{size}\ n\texttt{;}$$
$$f\texttt{:}\ CA\texttt{(}\overline{x}\texttt{, terminator=}c\texttt{);}\ \Rightarrow f\texttt{:}\ \textbf{terminatedBy}\ c\texttt{;}$$

The first rule replaces a fixed-length field f which requires content-analysis CA with a field of unknown data but of the same length. If the field f has no fixed length, but a terminator constant c is specified in the content-analysis, the content-analysis is removed, and field f is now **`terminatedBy`** c.

Remove Data Dependencies The second transformation removes data dependencies. All references to values or sizes defined elsewhere in the description are removed. An example of this is the *chunkdata* field as shown on line 21 in Figure 3 where *size* depends on the value of *length* on line 19. There are two types of data dependencies that are dealt with differently. First, if the contents of a field are defined by reference to another field, the reference is removed by clearing the content specification. The field's value becomes "undefined". The transformation rule implementing this transformation is as follows:

$$f\texttt{:}\ E[f']\ \texttt{size}\ n \ \Rightarrow\ f\texttt{:}\ \texttt{size}\ n\texttt{;}$$

If the value of a fixed-length field f is defined by some expression E referencing field f', the value specification is simply removed.

Second, if the size specification of a field depends on another field, the transformation is more involved. It is not possible to clear the size specification of a field just like with value dependencies, since then the position of a following field or structure becomes undefined. Instead, we remove the entire field from its containing structure. To ensure that the generated validator still works, we locate the first field f' that defines a constant value c that is required to follow the removed field f; if s does not define such a field itself, we find the first following structure that does, using the format's sequence. We replace the definition of f' with f'**: terminatedBy** c;. To prevent backtracking in the generated validator, we remove any non-mandatory structures (indicated by `*`, `?`, and `()`) inbetween f and f'. To find the first mandatory field that defines a constant, we use a simple algorithm, similar to the computation of first-sets of context-free grammars [1].

Figure 4 shows the effect of a single transformation step to remove the size dependency of the *chunktype* field of PNG's *IDAT* structure[5]. In this example the content-analysis and value dependencies have already been removed. In this step, the *chunkdata* field has been removed from *IDAT*. Additionally, the *length* field of *IEND* has been changed to include the **terminatedBy** modifier, because it is the first mandatory constant field following the removed *chunktype* field.

Reduce to Header-Footer Matching The third and last model transformation reduces a format description to two patterns: one for the beginning and one for the end of the file. This is the same strategy that is employed by the SCALPEL carver [17]. It requires file formats to have a clearly defined header and footer, using only constants. As a result, a validator based on this description will hardly ever reject data since for every header some footer is very likely to be found (assuming a large amount of files or fragments in the input data). Fragmentation in the input data will lead almost certainly to false positives. However, all recovered files are collected in a single linear pass over the input data.

The transformation operates as follows. Let S be the largest sequence of non-optional consecutive structures starting from the beginning of the sequence

[5] Note that the *IDAT* structure no longer inherits from the *Chunk* structure; the inheritance hierarchy has been flattened during normalization.

```
IDAT {
  length: lengthOf(chunkdata) size 4;
  chunktype: "IDAT";
  chunkdata: size length;
  crc: size 4;
}
IEND {
  length: 0 size 4
  ...
}
```

⇒

```
IDAT {
  length: size 4;
  chunktype: "IDAT";
  crc: size 4;
}
IEND {
  length: terminatedBy 0 size 4
  ...
}
```

Fig. 4. Example of *Remove Data Dependencies*

```
sequence
  s e

structures
  s { header: 137, 80, 78, 71, 13, 10, 26, 10; }
  e { footer: terminatedBy 0, 0, 0, 0, "IEND", 0xAE, 0x42, 0x60, 0x82; }
```

Fig. 5. Example of *Reduce to Header/Footer*

definition of the file format. Let E be a similar list of structures, but now starting backwards, from the end of the sequence definition. Now collapse both S and E into single structures s and e by taking the largest sequence of constant fields starting from the beginning and the end respectively, and concatenating consecutive field constants into single constants a and b. Then define the structures s and e as s { `header:` a; } and e { `footer:` **`terminatedBy`** b; }. Finally, construct a new file format with sequence s e. The resulting file format searches for the constant header pattern a, and (if found) subsequently searches for the constant footer pattern b.

Figure 5 shows the result of applying this transformation to the full PNG description of Figure 3. Note that all consecutive constant fields in the *IEND* structure have been merged into the single field *footer* to construct the largest possible constant.

4 Evaluation

To evaluate the effect of the transformations we have applied them on three Derric file format specifications, namely for JPEG, GIF and PNG. We have run the resulting $3 \times 4 = 12$ carver configurations on a representative disk image of 1TB, containing over a million recoverable files. We have then compared the difference in runtime performance, precision and recall between the configurations.

4.1 Development of Benchmark Disk Image

The largest publicly available disk image for exercising file carvers is 40GB in size[6]. This, however, is not large enough to properly assess how an application deals with scalability issues in practice. We have therefore developed our own 1TB test set based on data downloaded from Wikipedia. The size of Wikipedia means we could get enough files to fill at least a significant part of the 1TB data set we wanted to create. We used the latest available static dump of all images on Wikipedia, which dates from 2008[7]. Attempting to download all files from that list resulted in around 50% errors due to missing files. The end result was a usable set of over 1.2 million files with a total size of 357GB. An overview of how the files are distributed over each type (JPEG, GIF and PNG) and their total sizes is shown in the first column of Table 1.

These files were written into the test image file, spread out across the entire 1TB. Space between files (or fragments) was filled using 543GB of random data and 100GB of only zeros. Although there is little known about the amount and size of zero data blocks on hard drives, we believe 10% is a low estimate, which means the test image is more challenging for file carvers (since zeros are relatively easy to disqualify).

93% of the files have been written into the test image in contiguous blocks and are therefore not fragmented. 3% has been split into two parts and the remaining 4% has been divided into four equal size groups of 3, 4, 5–10 and 11–20 fragments, corresponding to observations of fragmentation in the wild [10]. Splitting was done at random locations in the files, but always on a cluster boundary of 4096 bytes, corresponding to the smallest common cluster size.

Format	Configuration	Running time	True positives	False positives	Precision	Recall
JPEG	Base	742m	882,511	0	100.0%	94.9%
input data:	NoCA	295m	860,022	22,007	97.5%	92.4%
total files: 930,424	NoDD	231m	837,382	46,561	94.7%	90.0%
total size: 327GB	Header	231m	837,382	46,561	94.7%	90.0%
GIF	Base	320m	34,078	0	100.0%	93.2%
input data:	NoCA	267m	33,210	702	97.9%	90.8%
total files: 36,576	NoDD	231m	32,912	2,780	92.2%	90.0%
total size: 3GB	Header	231m	32,912	2,780	92.2%	90.0%
PNG	Base	691m	222,660	0	100.0%	94.2%
input data:	NoCA	280m	219,001	8,073	96.4%	92.6%
total files: 236,457	NoDD	231m	212,911	13,905	93.9%	90.0%
total size: 27GB	Header	231m	211,790	14,577	93.6%	89.6%

Table 1. Results per configuration for all three file formats

[6] `http://digitalcorpora.org/corpora/disk-images`

[7] `http://static.wikipedia.org/downloads/2008-06/en/images.lst`

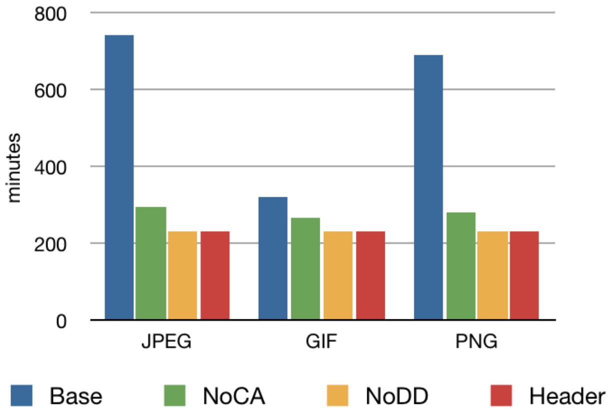

Fig. 6. Effect of each carver configuration on runtime performance

4.2 Execution of the Benchmark

The 12 carver configurations have been run on a 3.4GHz Intel Core i7-2600 with 8GB of RAM and an attached 2TB 10.000RPM SATA harddrive. The operating system used was Ubuntu Linux 11.04, with Oracle's JDK 1.6.0 update 13. The results of each run are shown in Table 1. For each file type and configuration it shows the wall clock running time in minutes in the third column. The fourth and fifth column of each table display the number of true and false positives respectively. True positive means a file has been recovered that was actually present in the disk image. False positive means that the file carver recovered a file erroneously, for instance, by combining a file header with the wrong footer. The last two columns give precision and recall percentages. An overview of the effect on runtime performance is shown graphically in Figure 6.

4.3 Analysis of Results

The fastest two configurations, NoDD and Header, require the same amount of time to complete for each format. The 231m corresponds to the time required to read through a terabyte of data on the hardware used, indicating that when using the NoDD and Header configurations, the application is bound by the read performance of the underlying platform. In other words, reading all data in a single linear pass would take the same amount of time.

Additionally, on JPEG and GIF, both the NoDD and Header configurations return exactly the same results, indicating that the final transformation does not impact the quality of the results or runtime performance. However, on PNG the situation is different: the NoDD configuration returns a little more true positives and fewer false positives.

This difference can be traced to the fact that the descriptions for JPEG and GIF both have a large variable block in the middle that is effectively eliminated by the *remove data dependencies* transformation, while the PNG description does have a fixed structure at a variable location between the first and final structure (the *IDAT* structure). This causes the PNG NoDD configuration to be more discriminating than the Header configuration. The result is slightly higher precision and recall.

For all three formats, the Base configuration returns no false positives, reaching 100% precision. The Base descriptions are complete, which leads to validation of all the contents of a candidate match. Since all three formats are compressed, even a single missing or misplaced fragment will lead to errors during validation and be rejected by the validator.

Another point of interest is the running time of the Base configuration. For JPEG and PNG, this is both at least twice the time required to run the NoCA configuration and at least three times the amount of time required to run the NoDD and Header configurations. Two factors contribute to this. The first factor is the relatively expensive operations by the validators. An example of this is CRC calculation. Although an optimized implementation is used, due to fragmentation, the CRC is sometimes calculated over large blocks that end up not being matches.

The second factor is the effect of fragment reordering in Excavator. Whenever a validator rejects a candidate match, an additional check is performed to determine whether a possible footer of the same file format is relatively close to the error location. If this is the case, the clusters between the error location and the matching footer are partially reordered and removed, running the validator on each combination to determine possible hits. To prevent a combinatorial explosion, reordering is only enabled when the distance between error location and footer is smaller than 2MB. Consequently, it is triggered by the most precise validators. In the more permissive validators the gap size is either too large or it is entirely undetected (and leads to a false positive in the results).

5 Discussion

Effects on Analysis Time It can be argued that, although more permissive validators will run faster, in practice, they may end up requiring more of the investigator's time, because there are more false positives to inspect. This time could also be spent running the analysis using a higher precision validator. Depending on the legal case, however, it might be more valuable to have results more quickly: even with more false positives, a crucial piece of evidence could be found earlier.

With our current results we believe the transformed validators are a useful alternative to the most precise validators, since the loss of precision and recall (8% and 5% respectively) is relatively small compared to the gain in performance (between 40% and 320%). For example, for PNG, the fastest carver returns 211,790 true positives and 14,577 false positives but it requires only 1/3rd of the running time of the most precise carver.

At the same time, the fastest validators do not make the original validators obsolete, considering that, after the fastest validator has finished, the most precise JPEG validator is able to recover 45,129 true positives in the extra 510 minutes.

An alternative approach is to use the more precise validators for only a short period of time and use their intermediate results when time runs out. While this is possible, there is a chance that the more precise validator will spend a lot of time near the beginning of the disk image recovering a fragmented file, while the fastest validator (which does not reject anything) will skip over it and return all the relatively simple matches directly.

Another alternative approach is to use one of the fastest validators and run the most precise validator on the results to remove false positives. This may help all carver configurations achieve 100% precision.

Other File Formats Our experiment takes three popular image file formats and shows how the described model transformations affect runtime performance and precision of the generated validators from their descriptions. A question is whether this approach works as well on other file formats. There is a strong indication that they will perform similarly, considering that most forensically interesting file formats tend to either be multimedia, document or container files. All three of these types of files often have features comparable to the image file types we used: extensive metadata, compressed contents and well-defined headers and footers. Examples of forensically interesting file types that are structured similarly are AVI and MPEG for multimedia, XLS and PDF for documents, and ZIP and RAR for containers. In future work we will apply Excavator and the model transformations on Derric descriptions of these file formats.

6 Related Work

Transformation for optimization is as old as the theory of compiler construction [2]. Moreover, transformation is considered to be one of the cornerstones of model-driven engineering [18, 4] and generative programming [9]. In both areas the objective is to specify the essential variability of an application domain at high levels of abstraction, and then generating the low-level code automatically. The commonality of an application domain is captured by such transformations. We have applied this well-known pattern in the context of digital forensics.

Domain-specific analysis, verification, optimization, parallelization and transformation (AVOPT) are well-known reasons for DSL development [14]. In particular, for optimization, the explicit representation of high-level domain concepts can be used by a compiler in order to generate code that is more efficient. Such optimizations are very hard to obtain in the context of ordinary, hand-written programs, since the high-level domain concepts are lost in low-level code. In this paper we have shown how to use domain concepts of Derric (content analysis, data dependencies and header/footer) in order to obtain faster file carvers.

In [7] the authors present a model and strategy for transforming source code in order to reduce the energy consumption of a program. It includes an explicit cost model of both the transformations and the object program. Our transformations themselves are very inexpensive, and the cost model for file carving is based solely on the most expensive operations at runtime. Another instance of applying model transformation for optimization is presented in [6]. The authors apply a number of successive transformations on BIP (Behavior, Interaction, Priorities) models to obtain a single monolithic, efficient program. The Derric model transformations operate in the same way in that they remove overhead elements from the input model. What makes our transformations different from such approaches, however, is that the transformations are not (strictly) semantics preserving, as they discard information. As such the transformations can be considered approximations, in a similar way that context-free grammars can be approximated by regular expressions [15].

Our software tool Excavator represents the state-of-the-art in digital forensics data recovery, implementing fragmented file recovery [10, 8] and a stream-based processing model [11]. Furthermore, our model-driven approach distinguishes itself by allowing high-level specification of elaborate data structures not implemented in popular file carvers. By comparison, PhotoRec [12] requires hand-written format validators and Scalpel [17] employs regular expressions for format validation.

7 Conclusion

Modifiability, runtime performance and scalability are the major challenges in digital forensics software construction. Moreover, forensic investigations are often constrained by very strict deadlines. As a result digital forensics software is often modified on a case-by-case basis. This just-in-time "carver hacking" is error prone and time consuming.

In previous work we have introduced a model-driven approach to digital forensics software development, Derric, which improves performance and modifiability by generating efficient code from high-level file format descriptions. In this paper we introduced three source-to-source model transformations on Derric descriptions in order to make the trade-off between precision and runtime performance configurable. This allows investigators to choose performance over precision if time constraints should require so, or the other way around,—without having to change any code.

The effect of the model transformations is evaluated on a 1TB disk image containing over a million recoverable files, specifically constructed to resemble a realistic file carving scenario. Our results show that performance gains up to a factor of three can be achieved. This comes at a loss of up to 8% in precision and 5% in recall.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Prentice Hall, 2 edn. (2006)
2. Allen, F., Cocke, J.: A Catalogue of Optimizing Transformations. In: Design and Optimization of Compilers, pp. 1–30. Prentice-Hall (1972)
3. Aronson, L., van den Bos, J.: Towards an Engineering Approach to File Carver Construction. In: 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW). pp. 368–373. IEEE (2011)
4. Bézivin, J.: Model Driven Engineering: An Emerging Technical Space. In: Generative and Transformational Techniques in Software Engineering, Lecture Notes in Computer Science, vol. 4143, pp. 36–64. Springer (2006)
5. van den Bos, J., van der Storm, T.: Bringing Domain-Specific Languages to Digital Forensics. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE'11). pp. 671–680. ACM (2011)
6. Bozga, M., Jaber, M., Sifakis, J.: Source-to-Source Architecture Transformation for Performance Optimization in BIP. IEEE Trans. Industrial Informatics 6(4), 708–718 (2010)
7. Chung, E.Y., Benini, L., De Micheli, G.: Source Code Transformation based on Software Cost Analysis. In: Proceedings of the 14th International Symposium on Systems Synthesis (ISSS'01). pp. 153–158. ACM (2001)
8. Cohen, M.I.: Advanced Carving Techniques. Digital Investigation 4(3-4), 119–128 (2007)
9. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison Wesley (2000)
10. Garfinkel, S.L.: Carving Contiguous and Fragmented Files with Fast Object Validation. Digital Investigation 4(S1), 2–12 (2007)
11. Garfinkel, S.L.: Digital Forensics Research: The Next 10 Years. Digital Investigation 7(S1), S64 – S73 (2010)
12. Grenier, C.: PhotoRec, `http://www.cgsecurity.org/`
13. Klint, P., van der Storm, T., Vinju, J.: Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In: Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'09). pp. 168–177. IEEE (2009)
14. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37, 316–344 (2005)
15. Mohri, M., Nederhof, M.J.: Regular approximation of context-free grammars through transformation. In: Robustness in Language and Speech Technology, chap. 9, pp. 251–261. Kluwer (2000)
16. Pal, A., Memon, N.: The Evolution of File Carving. Signal Processing Magazine, IEEE 26(2), 59–71 (2009)
17. Richard, III, G.G., Roussev, V.: Scalpel: A Frugal, High Performance File Carver. In: Proceedings of the Fifth Annual DFRWS Conference (2005)
18. Schmidt, D.C.: Model-Driven Engineering. Computer 39, 25–31 (2006)