Generation of Verifiable Evidence for Declarative Data Forensics: A File System Case Study*

Iana Boneva   Radha Jagadeesan   Corin Pitcher   James Riely

School of Computing, DePaul University

Abstract

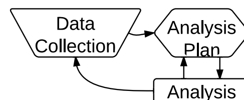
Digital forensics often requires an investigator to use multiple highly-specialized tools in the analysis phase. This process is tedious, ad hoc, and time consuming; requiring data reformatting between phases, which is often manual. As a result, it is challenging to formally codify and communicate expertise. Perhaps more importantly, the tight coupling between the tools and the search procedure complicates independent validation of the results of the forensics process, undermining confidence in the results.

In the evidence-based approach to forensics, evidence is embodied as an tangible object, such as an XML document. Others have noted that this approach facilitates composition of tools. We argue that it also has benefits for independent validation. Further, we argue that constraint-based declarative programming is a natural paradigm both for representing and manipulating evidence and for composing and coordinating multiple tools or modules. Because constraint-based programs allow high-level, modular descriptions, they are relatively easy to create, compose and validate.

We describe a proof-of-concept study, centering on semantic search of file system data structures. We develop a constraint-based search tool for generating evidence using the Gecode constraint-based framework, and provide an independent validator for existing standards for evidence concerning NTFS file systems.

1 Introduction

The lifecycle for digital forensics [Remack, 2004] consists of initial data collection, followed by iterations of planning and analysis, sometimes leading to further data collection. The lifecyle can be visualized as follows.



*Research supported by NSF 0915704.

The feedback between planning and analysis in digital forensics is labor intensive, with potentially heavy specialist intervention in these feedback loops, exploring tradeoffs between alternate ways of exploring the available digital evidence. This manual intervention is ad hoc and distracts the examiner from the real task at hand; namely the conceptual and creative analysis of the evidence available to the examiner. In this context, Garfinkel [2012] identifies *composition* of forensics tools as one of the primary challenges. Digital forensics tools encode domain-specific information and heuristics to recover data. However, the knowledge exploited by such tools is often implicit in their implementation and perforce they are often not easy to understand, to modify, or to replicate in new tools. Consequently, when tasks do not conform precisely to the results produced by such special purpose tools, it becomes necessary to consider their composition.

Composition of tools sharpens a basic issue faced by developers and users of forensic tools [Scientific Working Group on Digital Evidence, 2013]: how to avoid digital evidence discrepancies (e.g., see [Digital Detective, 2011]) and convince the end-user regarding existence ("are all reported artifacts reported as present actually present?"), no alteration ("does a forensic tool alter data in a way that changes its meaning?") and no corruption ("does the forensic tool detect and compensate for missing and corrupted data?"). Our approach is to present *verifiable* data to affirm the conclusions of the forensic analysis — this enables an end-user to validate the results of the forensic analysis without having to trust the (composition of) tools that performed the analysis. Thus we show how composition of forensic tools proceeds hand-in-hand with composition of supporting evidence generated by tools.

File carvers locate files in a disk image by their content rather than the file system data structures. We consider a simple use case of file carving to illustrate composition of forensics utilities.

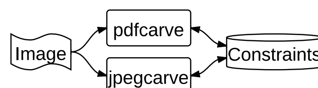
Assume we are given a tool `pdfcarve` to carve PDF files from a disk image. The affirming evidence for each result would certainly include the location of the carved PDF file, and the means used to establish that the carved file is a valid PDF. To strengthen the conclusion that the PDF file is present in the disk image, the evidence could also include the directory structure extracted from the image that leads to the extracted PDF file. Suppose that we are also given a tool `jpegcarve` that carves JPEG files, and produces supporting evidence similar to that of `pdfcarve`.

Now consider how the tools `pdfcarve` and `jpegcarve` can be combined to carve JPEG files located inside PDF files. If `jpegcarve` can use data from `pdfcarve` to limit its search, then we have the following composition pattern.

```
$ pdfcarve  image.bin > pdf-offsets.xml
$ jpegcarve image.bin < pdf-offsets.xml > jpeg-offsets.xml
```

The evidence to affirm the conclusions of such a composite tool would consist of two pieces of evidence: the set of indices for the PDF files in addition to the set of indices of the JPEG files. Garfinkel advocates the standardization of digital forensics artifacts using XML to facilitate the exchange of information between different forensic tools in such pipelines.

An alternative to pipelining is a more symmetric and flexible configuration that not only runs the two carvers in parallel, but also allows them to communicate *incremental* data.



When the incremental data is monotone, in the sense that assertions may not be withdrawn from the shared monotone database store of facts, this framework is known as constraint programming [Jaffar and Lassez, 1987; Saraswat and Rinard, 1990].

In more general examples, this store will contain expressive logical assertions about the data structures being shared by the programs.

In this paper, we present a general approach to structuring forensics tools as constraint programs. Our focus is on the use of explicit evidence, both as an intermediate artifact during programmatic composition of tools, and as an end result used for validation. We show that combining this evidence-based approach with constraint-based programming has several advantages.

- First, it permits the encoding of domain knowledge in an easily reusable form, permitting the forensic examiner to declaratively specify high level strategies in the examination of forensic data. Thus, we work towards providing an expressive variant of scripting for the entire forensic process, and introduce tool-independent specifications of file system structure.
- Second, the program itself constitutes part of the evidence provided to affirm the results of a forensic process. The repeatability and reproducibility of the results of the executable program makes it a superior witness in comparison to a textual documentation.
- Third, such an approach potentially permits the modification of existing ways to combine tools. For example, let us say that we want to search for JPEG files that are within PDF files in unallocated space (that is, in PDF files that have been deleted). In this case, it is useful to add a preprocessing filter to the composition to restrict the search only to the unallocated portions of the disk image. Thus, the tactic scripts themselves become reusable tools for a a domain expert.

The rest of the paper. In Section 2, we describe the different kinds of evidence that can arise by considering a concrete example of file system forensics in the NTFS file system. In Section 3, we describe the use of constraint programming to execute simple semantic searches on NTFS data structure while generating verifiable evidence for the results of forensics search. Our prototype implementation using Gecode and a corresponding verifier (compatible with Garfinkel's `fiwalk`) is available at `http://fpl.cs.depaul.edu/projects/forensics/`.

2 Evidence in File System Forensics: An NTFS Case Study

In this section, we provide an overview of the evidence obtained from popular digital forensics tools when analysing an NTFS file system from a Microsoft Windows machine. We build on this evidence in the examples of Section 3.

2.1 NTFS Introduction

NTFS is the default choice of file system for Microsoft Windows. Carrier [Carrier, 2005] describes the NTFS format, which, in the absence of a public specification, has been obtained by reverse engineering. Carrier describes the behavior observed when file operations are performed. Such observations are used to make predictions about the prior state of a system, e.g., a file from a rootkit was stored in the file system in the past, but it is now deleted.

The central concept in NTFS is the Master File Table (MFT). Each file or directory has an entry in the MFT (we adopt Carrier's terminology of "MFT entry" instead of Microsoft's "file record"). An MFT entry stores attributes including name(s), file size, ownership, ACLs. The contents of small files can be stored within an MFT entry (a *resident* data attribute). Otherwise, a runlist in an MFT entry points to other clusters containing the contents of the file (a *non-resident* data attribute). Each allocated cluster stores either an MFT entry or (part of) a non-resident attribute, and clusters containing MFT entries can be interleaved arbitrarily with other clusters.

We now consider examples of evidence produced in three ways: finding information from the defined file system structure; finding information from the undefined file system structure; and file carving without use of the file system structure.

2.2 Using Well-Defined File System Structure from Allocated Space

To illustrate the information provided by forensics tools for NTFS file systems, we consider a freely-available volume image prepared by the CFReDS project at NIST for pedagogical purposes [Computer Forensic Reference Data Sets (CFReDS) Project, 2007]. The 4.6GB file `original.img` represents the contents of a hard drive of the same size. It contains a DOS-style partition table [Carrier, 2005]; and the sole partition is NTFS. This can be seen using the command `mmls` from The Sleuth Kit (TSK) [Carrier, 2013]:

```
$ mmls original.img
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

     Slot     Start        End          Length       Description
00:  Meta     0000000000   0000000000   0000000001   Primary Table (#0)
01:  -----    0000000000   0000000062   0000000063   Unallocated
02:  00:00    0000000063   0009510479   0009510417   NTFS (0x07)
03:  -----    0009510480   0009514259   0000003780   Unallocated
```

Thus `mmls` claims the existence of an NTFS file system partition extending from byte offset 63*512 to ((9,510,479+1)*512)-1. This information from `mmls` can be used as input to other tools that operate on the file system in the partition, or tools that search in the unallocated space surrounding the partition. To verify the `mmls` output, the partition table in the first sector must be examined with respect to the same specification.

The file system inside the partition can be examined using the command `fls` from TSK. For example, a recursive directory listing for the file system beginning at sector 63 can be searched for filenames containing the string `cardz` using:

```
$ fls -o 63 -r original.img | grep -B5 cardz
++++ d/d 11408-144-1:          Identities
+++++ d/d 11409-144-1:         EF086998-1115-4ECD-9B13-9ADC067B4929
++++++ d/d 11412-144-1:        Microsoft
+++++++ d/d 11424-144-6:       Outlook Express
++++++++ r/r 11431-128-3:      cleanup.log
++++++++ r/r 11443-128-4:      alt.2600.cardz.dbx
```

In this listing, the file `alt.2600.cardz.dbx` is an Outlook Express database for messages from the USENET group `alt.2600.cardz`. The identifier `11443-128-4` refers to the MFT entry number

for the file. This identifier can be used to access the file, e.g., the TSK command `icat` dumps a file referenced by its identifier. Thus to search for possible subject lines that containing the strings `CVV` and `Re:` in ASCII we might use:

```
$ icat -o 63 original.img 11443-128-4 | strings | grep CVV | sed -e 's/^Re: //' | sort | uniq
CC's with CVV2 4 Sale
CVVS
More CVVs from Zee
```

This search procedure is conceptually simple, but has the drawback that it does *not* provide evidence of where the possible subject line strings occurred.

Garfinkel [Garfinkel, 2012] proposes the use of an XML-based format as a standard for exchanging evidence data between tools. For example, the `fiwalk` program from the DFXML toolset produces the following output for the same `alt.2600.cardz.dbx` file.

```
$ fiwalk -x original.img
...
<fileobject>
  <parent_object> <inode>11424</inode> </parent_object>
  <filename>Documents and Settings/Mr. Evil/Local Settings/Application Data/Identities/
            EF086998-1115-4ECD-9B13-9ADC067B4929/Microsoft/Outlook Express/alt.2600.cardz.dbx</filename>
  <partition>1</partition>
  <id>758</id>
  <name_type>r</name_type>
  <filesize>207572</filesize>
  ...
  <byte_runs>
   <byte_run file_offset='0' fs_offset='1036299776' img_offset='1036332032' len='11776'/>
   <byte_run file_offset='11776' fs_offset='365370368' img_offset='365402624' len='65024'/>
   <byte_run file_offset='76800' fs_offset='1870330368' img_offset='1870362624' len='65536'/>
   <byte_run file_offset='142336' fs_offset='885078016' img_offset='885110272' len='17920'/>
   <byte_run file_offset='160256' fill='0' len='47316'/>
  </byte_runs>
</fileobject>
...
```

The DFXML tools use of the TSK code base to traverse file system structures, so comparing the results of TSK output with DFXML output does not constitute an independent verification of correctness.

2.3 Using Undefined File System Structure from Unallocated Space

In addition to searching for information in the defined file system structure, digital forensics searches also examine data structures in unallocated space. The purpose is to make predictions about the prior state of the file system. The simplest example is finding a deleted file. In most file systems, file deletion does not erase the contents of the file from the physical media for performance reasons. Instead the areas of the media storing the file contents have their allocation status set to unallocated, and the directory entry that points to the file contents is unallocated.

However, file system information found in unallocated space does not have a defined semantics. The heuristics used to deal with these situations are complex, and usually only defined in code, thus strengthening the case for producing evidence that explains how a forensics tool concludes that a deleted file was present.

For example, in the case of NTFS, there are three relevant areas for each file:

(1) The contents of the file (stored in clusters that are not part of the MFT).

(2) The MFT entry for the file, including file name, size, timestamps, and a pointer to the file contents.

(3) The MFT entry for the directory that contains the file. It includes the file name and a pointer to the MFT entry for the file.

When a file is deleted, (1) is marked as unallocated (in a separate data structure); (2) is marked as unallocated; and (3) the entry for the file within the MFT entry for the parent directory is unallocated, and a tree containing such entries may be rebalanced. It is possible for each of (1)-(3) to be overwritten independently by file system operations that occur after the deletion.

2.4 Limiting Search of File System Structure from Unallocated Space

We now demonstrate that it is sometimes useful to ignore some or all of the file system structure. For this demonstration we use two further images, `deleted.img` and `formatted.img` (both derived from `original.img`).

The image `deleted.img` was derived by attaching a copy of `original.img` to the `E:` drive of a Virtual Box 4.1.14 VM running Windows XP SP3. The drive image was attached to a non-boot drive `E:`. We then deleted the `E:\WINDOWS` using `rmdir /s /q`. This image has approximately half the number of allocated clusters as `original.img`. The image `formatted.img` resulted from performing a quick format of `E:` inside the VM. In this image, most of the files and directory entries are intact, but some of the original directory entries may not be part of the new MFT that is written during quick formatting. That is, a search of the unallocated areas of the MFT in `formatted.img` will not find some directory entries from `original.img`, even though they exist in the image.

To see this, consider one of the deleted files, `winnt256.bmp`, with MFT entry `5717` from the `\WINDOWS` directory. This deletion can be seen using `fls`, where `*` indicates a deleted file:

```
$ fls -o 63 -r original.img | grep 5717-128-4
+ r/r 5717-128-4:       winnt256.bmp

$ fls -o 63 -r deleted.img | grep 5717-128-4
+ -/r * 5717-128-4:     winnt256.bmp
```

When the NTFS file system is quick formatted, the file contents still exists in `formatted.img`. However, the MFT is recreated and no longer includes an MFT entry `5717` for the bitmap file.

```
$ fls -o 63 -r formatted.img | grep 5717-128-4

$ icat -o 63 formatted.img 5717-128-4
Metadata address too large for image (32)
```

That is, the TSK tools operate on file systems that are mostly intact by traversing the file system's data structures, and then performing limited searches in unallocated areas, e.g., limiting search to the unallocated MFT entries rather than all unallocated clusters. These limitations placed on search provide significant performance improvements and cover many common cases. Moreover, it is unclear how search should be generalized to accommodate partial and/or conflicting file system data structures. In the sequel, we address the issue of generalization of search by providing a constraint-based programmatic interface to forensics search.

2.5 Ignoring File System Structure

File carving tools take an alternative approach to TSK. They ignore file system structure, and instead look for the contents of files by using knowledge of file formats, e.g., a bitmap image file starts with `BM`, two more bytes, and then three zero bytes. In the absence of file system structure that describes the sequence of clusters comprising a file, file carving tools must identify the sequences from the file content. This identification becomes harder as the number of fragments in files increases, but sophisticated algorithms have been developed, e.g., [Pal et al., 2008].

`foremost` is a well-known file carver that can be configured for new file formats with simple signatures (patterns) describing the start and end of files. The `audit.txt` file created by `foremost` provides evidence about where files are stored, e.g., in the results of a search for bitmap image files:

```
$ cat output/audit.txt
...
Num         Name (bs=512)          Size        File Offset        Comment
270:        01820676.bmp           47 KB         932186112          (275 x 174)
...
427 FILES EXTRACTED
```

Verification of such evidence can be decomposed into two parts. The first step is to verify that the extracted file is present in the file system image at the specified sequence of clusters. This is straightforward, because it does not depend on knowledge of any file system. The second step is to verify that the extracted file is a bitmap file. Here there are multiple possibilities, e.g., one could try to replicate the heuristic used by the file carver to recognize a file type (e.g., a check against signatures for the beginning and/or end of a file). Alternatively, the extracted file may be tested against a more complete specification of the file format.

3 Generating Verifiable Evidence

In this section, we investigate how constraint programming can be applied to: (1) execute simple semantic searches based on the NTFS data structure seen in Section 2; and (2) generate verifiable evidence for the results of forensics search. We describe a prototype search implementation using the Gecode constraint framework [Schulte et al., 2013], and a corresponding verifier that is compatible with `fiwalk`. The behavior of these tools is illustrated by searches on the volume images described in Section 2. The source code for our implementation is available at `http://fpl.cs.depaul.edu/projects/forensics/`.

3.1 Constraint Programming

We use finite domain constraint programming [Schulte and Carlsson, 2006] to execute forensics queries on file systems. Constraint programming permits the decoupling of a problem specification and the search strategy used to find solutions. Problem specifications are written declaratively using a rich selection of Boolean and arithmetic constraints (among others). This permits a direct encoding of the file system data structures of interest, and facilitates the construction of specialized forensics searches that refer to such data structures.

To convey the flavor of constraint programming and its adaptation to the forensics domain, we consider a naive search for the MFT entry start sequence "FILE" within a byte array a representing a volume image. This can be seen as a simple form of data carving, because it identifies areas of a file system image that resemble an MFT entry, without traversing the entire file system data structure. If integer variable X is the offset of an occurrence of "FILE" in a, then we consider the logical formula:

$$a[X] = \texttt{'F'} \wedge a[X+1] = \texttt{'I'} \wedge a[X+2] = \texttt{'L'} \wedge a[X+3] = \texttt{'E'}$$

This constitutes a constraint program. The constraints used in this program are addition, integer equality, array access, and conjunction. The constraint program is executed by performing a search to find a successful assignment to the integer variable X. The nodes of the search tree, often called *spaces*, store the possible domain of X. If the length of the array a is N, then the initial space of the search tree stores the constraint $X \in [0..N)$.

Constraints are implemented by propagators that examine the value of variables such as X. A propagator may: prune the search tree by failing a space (e.g., when $a[0] = \texttt{'I'}$ and $X = 0$); remove itself when it becomes entailed (e.g., when $a[0] = \texttt{'F'}$ and $X = 0$); or update the domains of variables (e.g., when $a = [\texttt{'A'}, \texttt{'B'}, \texttt{'C'}, \texttt{'D'}, \texttt{'E'}, \texttt{'F'}, \ldots]$ the domain for X can be updated from $X \in [0..N)$ to $X \in [5..N)$). Once propagators have no further action for the domain assignments to variables in a space, the search tree is extended with one or more branches. Each branch adds a constraint that shrinks the domain of one or more variables. Different search algorithms, e.g., depth-first search, can be used to explore a search tree.

A space is a *solution* if it is not failed and no branching is possible. In the context of the example above, a solution yields an offset X to an occurrence of "FILE". The supporting evidence for a proposition (constraint) includes the values assigned to variables in such a solution—in this case, the offset X.

3.2 Constraint-Based Forensics Search

We now consider execution of the simple constraint program from Section 3.1 using the Gecode constraint programming framework [Schulte et al., 2013]. This program searches for possible MFT entries within an image containing an NTFS file system.

Gecode provides a C++ library for constraint search, allowing precise control of memory management for large volume image files. The library is used by first establishing variables, constraints, branching strategies, and search engines for a problem; then iterating over solutions produced by the search engine.

A slightly simplified core of the constraint program to search for possible MFT entries appears below. This program uses integer variables `cluster` and `offset` of type `Gecode::IntVar` to represent the file system offset ($\texttt{cluster} * \mathit{clusterSize} + \texttt{offset}$). These variables range over sets of possible values, in contrast to variables of type `int` that range over single values.

```
// Search for "FILE" on cluster boundaries.
unsigned char fp_array[] = "FILE";
std::list<unsigned char> fp (&fp_array[0], &fp_array[4]); // initialize char list from char array
match (cluster, offset, fp, getInterDS);                 // constrain (cluster,offset) to "FILE" in input data source
rel (offset == 0);                                        // constrain offset to 0 (start of a cluster)
```

The `match` constraint creates a propagator that searches for `"FILE"` at the offset given by `cluster` and `offset`. This custom `match` constraint offers more possibilities for efficient propagation than the (logically equivalent) conjunction of equalities in the original formula in Section 3.1. Finally, since MFT entries are aligned on cluster boundaries, `rel(offset==0)` constrains `offset` to be zero using the standard Gecode constraint for integer equality.

As discussed in Section 2, deleting a file causes the MFT entry in an NTFS file system to be marked as unused, and the clusters that contain its contents to be marked as unallocated, but the MFT entry itself still resides in an allocated cluster. This information can be used during search: a search of allocated clusters may locate MFT entries from deleted files; and a search of unallocated clusters may locate MFT entries that existed before a formatting operation. We can extend the program above to restrict searches to clusters that are either allocated or unallocated in an NTFS file system using:

```
// Search everywhere or (un)allocated clusters?
if (area == SEARCH_ALL) {                                 // no further constraint
} else if (area == SEARCH_ALLOCATED) {
  ntfs_allocated (cluster, new NTFSBitmap (interDS));     // constrain cluster to allocated areas of file system
} else if (area == SEARCH_UNALLOCATED) {
  ntfs_unallocated (cluster, new NTFSBitmap (interDS));   // constrain cluster to unallocated areas of file system
}
```

The custom constraints `ntfs_allocated` and `ntfs_unallocated` constrain the integer variable `cluster` to represent an allocated or unallocated cluster respectively. The implementation of these constraints traverses NTFS data structures to determine the allocation status of clusters.

This simple composition of different constraints illustrates the expressiveness afforded by declarative forensics queries.

Results In our experiment, we performed an exhaustive search for the offsets of possible MFT entries within the NTFS partition. The program ran single-threaded using depth-first search. The measurements are taken using a Xeon E3-1230 3.30 GHz running Linux (kernel version 3.2.0-29) with 16GB RAM.

The following table gives the results for each of the three images. We ran three separate searches: first looking for all MFT entries, then looking for those in allocated clusters, and finally for those in unallocated clusters. For each of the nine combinations, we performed three runs and averaged the results. We report the total running time and the number of solutions found.

	all	**allocated**	**unallocated**
`original.img`	23.0s 12333 solns	11.1s 12332 solns	14.5s 1 soln
`deleted.img`	23.4s 12445 solns	11.3s 12437 solns	18.2s 8 solns
`formatted.img`	30.3s 12312 solns	0.2s 26 solns	23.0s 12286 solns

Our results are an order of magnitude slower than special-purpose tools such as the Sleuth Kit, Foremost, and Scalpel. The chief advantage we claim is not the efficiency of our prototype, but rather the declarative specification of file system forensics searches.

There are differences in the semantics of this search and similar searches conducted by `fiwalk` from the TSK, leading to different sets of results. For example, `fiwalk` returns 21 results in total for `formatted.img` where our system finds 12312 results because it conducts a wider search. We discuss this further in Section 3.3.

We can see that deleting files does not substantially change the number of MFT entries in unallocated areas, but writing a fresh MFT during quick formatting does leave old MFT entries in unallocated clusters. This causes the runtime of the search for files in the allocated clusters to complete quickly.

3.3 Verification of Evidence

In Section 3.2, we have demonstrated that declarative forensics queries involving file system structure are feasible and modular; and modularity facilitates some specialized searches that are challenging with existing tools, e.g., searching in the remnants of file system structure after quick formatting.

In current practice, the correctness of digital forensics tools is primarily addressed via testing. For this reason, a system that permits declarative forensics queries over file system structure may be considered untrustworthy precisely because new and untested queries can be constructed for specialized searches. We mitigate this concern by annotating results with sufficient information to verify the results independently of the search.

Semantics of Evidence For pragmatic reasons, we adopt the DFXML format (seen in the output of the `fiwalk` program in Section 2.2) as the basis for annotated results. This has two advantages. The first is that it facilitates composition of tools, as advocated by [Garfinkel, 2012]. The second is that a single verifier can be applied to the output of existing tools as well as the new tools built via declarative forensics queries. This is important because it is the role of the verifier to provide trust in the annotated results of forensics searches, and it is desirable to minimize the size of this trusted computing base.

We have extended our constraint-based searches with DFXML-based output. The generation of this evidence, the annotated results, is straightforward because file system structure is already identified and exposed for use in declarative forensics queries. For example, we generate the following entry for a file `paris.jpg` when searching for MFT entries in the quick formatted image `formatted.img` examined in Section 2.4.

```
<fileobject>
  <part_offset>32256</part_offset>
  <fs_offset>1080361984</fs_offset>
  <parent_object> <inode>458</inode> </parent_object>
  <filename>winnt256.bmp</filename>
  <partition>-</partition>
  ...
  <name_type>r</name_type>
  <filesize>48680</filesize>
  ...
  <byte_runs>
    <byte_run file_offset='0' fs_offset='932153856' img_offset='932186112' len='48680'/>
  </byte_runs>
```

```
  <hashdigest type='md5'>2f3cdc1d898fd25b2547f5bfeb01fd0d</hashdigest>
  <hashdigest type='sha1'>6df7047a508bd08c6c1cbdcdefb12885cb438d90</hashdigest>
</fileobject>
```

Collection of such evidence does require validity checks beyond the constraints described in Section 3.2. For example, the byte run within an MFT entry points to the file content; if the byte run is invalid, perhaps pointing outside the file system, then the file content cannot be hashed. This reduces the number of results that are produced. Often, this will remove false positives from search results. More refined specifications of evidence may be useful for searches that examine data structures that have been more heavily corrupted, e.g., the evidence might assert that a byte run is out of bounds and omit the cryptographic hashes.

There are minor differences in the semantics of the evidence produced by our constraint-based searches and those of TSK. Our declarative specifications search for more MFT-like entries anywhere in an file system or volume image, whereas TSK limits its search to the current collection of MFT entries. In Section 2.4, we showed that this choice prevents TSK from finding the file `winnt256.bmp` in `formatted.img`, and, in Section 2.5, we showed that file carving tools can be used to identify the content of the same file, but they do not recover file metadata, such as the file's name. As seen in the DFXML-based evidence above, our search does find this file and its metadata within the quick formatted image.

This difference in search coverage is also the reason that our implementation does not yield identical output to `fiwalk` (recall that `fiwalk` uses TSK's for file system search). Although `fiwalk` does not find `winnt256.bmp` in `formatted.img`, the file can be found in `original.img` at the same location, yielding the DFXML output:

```
<fileobject>
  <parent_object> <inode>458</inode> </parent_object>
  <filename>WINDOWS/winnt256.bmp</filename>
  <partition>1</partition>
  ...
  <name_type>r</name_type>
  <filesize>48680</filesize>
  ...
  <byte_runs>
    <byte_run file_offset='0' fs_offset='932153856' img_offset='932186112' len='48680'/>
  </byte_runs>
  <hashdigest type='md5'>2f3cdc1d898fd25b2547f5bfeb01fd0d</hashdigest>
  <hashdigest type='sha1'>6df7047a508bd08c6c1cbdcdefb12885cb438d90</hashdigest>
</fileobject>
```

The most obvious difference is the presence of a pathname in the `fiwalk` output. Obtaining the pathname for a file in NTFS requires finding the MFT entry for each ancestor directory of the file. These MFT entries may have been overwritten, and their location is unconstrained in general (hence, TSK's imposition of restrictions on search), so we have not searched for pathnames in our declarative queries. As a second example, `fiwalk` produces offsets relative to the start of the file system, which may be embedded inside a volume image (the file system is usually in a partition). Our searches do not rely on the existence of a single file system header that applies to all results. Instead we produce offsets relative to the start of the volume image, and an offset of the partition within the volume image for each file. This change allows our searches to return results when the volume has been repartitioned or the beginning of the file system has been overwritten (for example, by a quick format with a different file system).

Verification Verification of the DFXML-based evidence above involves two steps:

(1) reading the MFT entry at the given image offset, and checking that it is consistent with the filename and other metadata for the file; AND

(2) reading the file content at the given byte run, and checking that it has the stated cryptographic hashes.

In our experience, the code for validation of such evidence of file system data structures is significantly less complex, and faster to execute, than code that searches for file system data structures, precisely because the search behavior can be omitted.

We have developed a verifier for DFXML-based evidence from forensics searches. This verifier validates the output of our constraint-based forensics searches and those of `fiwalk` from TSK. Our verifer is aware of, and accommodates, the minor differences in the semantics of DFXML evidence.

Our verifier tests the *soundness* of evidence. Results can be omitted from the evidence without making the evidence invalid. Thus evidence may be the truth (sound), but not the whole truth (complete). This suggests that it may be beneficial to generate and verify claims about the *completeness* of the results in evidence. As we have seen, different forensics tools may differ in where they search for information, and so there may be different degrees of completeness. For example, `fiwalk`'s evidence might claim that it is complete for searching MFT entries within the current MFT of a valid file system, whereas the queries from Section 3.2 might claim to be complete for all MFT-like entries within an image that need not be a valid file system. Completeness of results is more challenging to validate, because the obvious approach requires a verifier to search for counterexamples, greatly complicating the verifier. In related work, we are investigating logical approaches to the specification and validation of completeness.

4 Related Work

There is a large literature on computer data forensics. In this survey, we focus solely on the most closely related work.

Declarative Models. [Stallard and Levitt, 2003] describes a declarative encoding of data invariants and a decision tree format to codify deductions performed by a forensic examiner. In the context of semantic search for files, we show how the search mechanisms of constraint programming achieve this aim in an executable programming language.

[Carrier and Spafford, 2006] provides a general model to compare digital forensics methodologies. It is our working hypothesis that constraint programming is expressive enough for most of the analysis techniques that they categorize.

Existing file carving tools, such as Scalpel and Foremost, use collections of patterns (byte sequences) and simple heuristics to recognize different file types. These methods are most successful when searching for files that are not fragmented. More recent approaches to file carving are able to

handle fragmentation of, e.g., JPEG, files [Garfinkel, 2007; Pal et al., 2008] by detecting fragmentation points using a variety of tests. These tests are developed by a manual analysis of the JPEG file format.

[van den Bos and van der Storm, 2011] pioneers the tool-independent specification of file formats for creation of file recognizers. In this paper, we have begun to investigate logic-based specifications of file system data structures that can be used for forensics search via constraint programming implementations.

Validation of Computer Forensics Software and Results [Guo et al., 2009] develops and classifies detailed requirements for forensic tool testing. It appears that many of their requirements map to code within our verifier.

Lyle describes undesirable and inconsistent behavior of forensics tools in [Lyle, 2008]. This has led to a subsequent program of testing [Lyle, 2010]. Lyle identifies the selection of test cases as a significant challenge. Two methodologies are identified in the testing program. The first is to construct file system images in a known manner, and to then see if tools find expected results, e.g., deleted files. The second is to compare the output of multiple forensics tools for inconsistencies. We followed both approaches in Section 2 and Section 3 for illustrative purposes. However, we also advocate independent specifications of correctness and verification of search results.

[Scientific Working Group on Digital Evidence, 2013] identifies soundness problems in the output *and* interpretation of forensics tools, as well as completeness issues discussed in Section 3.

5 Conclusions

Two of the great challenges facing computer forensics are the ability to scale (in size of data and in the complexity of analysis) and verifiability (how to convince a jury that the evidence has been analyzed properly).

We argue in this paper that declarative approaches can potentially contribute towards addressing both issues. Declarative methods support programmable compositionality of tools directly, thus enabling heuristic intermixing of specialized tools in an analysis that can be reproduced. Declarative tools can provide verifiable evidence that validates their results, thus developing users' trust in tools and their results. A declarative framework can also document the use of subcomponents that are unable to provide such evidence, so the need for faith is explicitly identified and localized.

In this paper, we have provided a prototype, proof-of-concept implementation of this program for file system search and verification of the results. A fuller realization of our program awaits future work.

References

B. Carrier. *File System Forensic Analysis*. Addison Wesley, 2005.

B. Carrier. The Sleuth Kit. `http://www.sleuthkit.org/`, 2013.

B. D. Carrier and E. H. Spafford. Categories of digital investigation analysis techniques based on the computer history model. *Digital Investigation*, 3:121–130, 2006.

Computer Forensic Reference Data Sets (CFReDS) Project. Schardt hacking case. http://www.cfreds.nist.gov/Hacking_Case.html, 2007.

Digital Detective. About digital evidence discrepancies – Casey Anthony trial. http://wordpress.bladeforensics.com/?p=357, 2011.

S. L. Garfinkel. Carving contiguous and fragmented files with fast object validation. *Digital Investigation*, pages s2–s12, 2007.

S. L. Garfinkel. Digital forensics XML and the DFXML toolset. *Digital Investigation*, 2012.

Y. Guo, J. Slay, and J. Beckett. Validation and verification of computer forensic software tools — searching function. *Digital Investigation*, 6:s12–s22, 2009.

J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL*, pages 111–119. ACM Press, 1987.

J. Lyle. Quirks uncovered while testing forensic tool. http://www.cftt.nist.gov/presentations/ENFSC-Lyle-Oct-08.ppt, 2008.

J. Lyle. Forensic tool testing results. http://www.cftt.nist.gov/presentations/NeFX-10-lyle-CFTT-test-strategy.pdf, 2010.

A. Pal, H. T. Sencar, and N. Memon. Detecting file fragmentation point using sequential hypothesis testing. *Digital Investigations*, 5:S2–S13, 2008.

S. Remack. Formalizing computer forensic analysis: A proof-based methodology. Master's thesis, North Carolina State University, 2004.

V. A. Saraswat and M. C. Rinard. Concurrent constraint programming. In F. E. Allen, editor, *POPL*, pages 232–245. ACM Press, 1990.

C. Schulte and M. Carlsson. *Handbook of Constraint Programming*, chapter 14: Finite domain constraint programming systems. Elsevier, 2006.

C. Schulte, G. Tack, and M. Z. Lagerkvist. Modeling and programming with Gecode, 2013.

Scientific Working Group on Digital Evidence. Error mitigation report. https://www.swgde.org/documents/ReleasedForPublicComment, 2013.

T. Stallard and K. Levitt. Automated analysis for digital forensic science: Semantic integrity checking. In *ACSAC*. IEEE Computer Society, 2003.

J. van den Bos and T. van der Storm. Bringing domain-specific languages to digital forensics. In *ICSE*, pages 671–680, 2011.