available at www.sciencedirect.com

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

Digital Investigation

Hash based disk imaging using AFF4

Michael Cohen, Bradley Schatz*

Australian Federal Police, Brisbane, Australia

ABSTRACT

Forensic imaging has been facing scalability challenges for some time. As disk capacity growth continues to outpace storage IO bandwidth, the demands placed on storage and time are ever increasing. Data reduction and de-duplication technologies are now commonplace in the Enterprise space, and are potentially applicable to forensic acquisition. Using the new AFF4 forensic file format we employ a hash based compression scheme to leverage an existing corpus of images, reducing both acquisition time and storage requirements. This paper additionally describes some of the recent evolution in the AFF4 file format making the efficient implementation of hash based imaging a reality.

© 2010 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.

The field of digital forensic analysis has experienced rapid growth in recent years, as the use of computer forensic analysis proved invaluable in a wide range of legal proceedings. Compounding with the increased usage and collection of digital evidence is the rapidly increasing storage capacity of media such as hard disks (Turner, 2005). The rapid expansion in storage requirements is not confined to the field of forensics, with modern data reduction and de-duplication techniques widely deployed in primary enterprise storage applications (Lawrence et al., 2005).

Traditional imaging technologies consist of making bit for bit copies of all data stored on the acquired media (so called raw or "dd" images). Second generation imaging techniques improved space efficiency by introducing block based compression to the data stream (Kloet et al., 2008; Garfinkel et al., 2006). Although space requirements for image storage was reduced, this came at the cost of increased acquisition time.

The advanced forensics file format (AFF4) is a third generation forensic file format integrating multiple image streams, the expression of arbitrary information and storage virtualisation into the forensic file format itself (Cohen et al., 2009). The present work addresses the dual goals of space and acquisition time efficient storage acquisition, built atop the AFF4 file format. Our novel acquisition method leverages an existing corpus of disk images by maintaining a database of data hashes already existing in the corpus. By avoiding the acquisition of duplicate byte runs, we are able to reduce both space requirements and avoid potentially time consuming compression operations.

While applying the initial AFF4 specification to this advanced imaging application, a number of limitations were exposed, prompting an evolution of the AFF4 specification. We begin the discussion with an evolutionary review of the AFF4 file format since its initial introduction (Cohen et al., 2009), followed by a discussion of our implementation.

1. Evolution of AFF4

The AFF4 format was proposed to update previous limitations in existing forensic formats Cohen et al., 2009. The format extends the idea of incorporating arbitrary metadata and data as proposed by earlier implementations (Garfinkel et al., 2006; Schatz and Clark, 2006).

An important advance in AFF4 is the introduction of arbitrary stream types into the container format, allowing for the natural implementation of abstract mapping types, encryption

* *Corresponding author.*
E-mail address: scudette@gmail.com (M. Cohen).
1742-2876/$ – see front matter © 2010 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2010.05.015

and multiple container volume formats. As the AFF4 format was adopted to work in real world applications, the format evolved into incorporating more standardized technologies. Shortcomings were identified in scaling the format for speed critical applications.

The first part of this paper describes some of these improvements, while the second part describes a novel implementation of hash based imaging employing these novel improvements.

1.1. *RDF data model*

One of the core tenants of AFF4 is the association of arbitrary metadata with any entity within the AFF4 universe. Previously, this metadata was serialised using an ad-hoc serialisation protocol, and the attributes stored were all strings.

The current implementation uses the RDF data model for serialising information in a standard way Brickley and Guha, 2003. This allows us to use a standard library to handle serialisation of attributes (Beckett, 2010), as well as create attributes with standard or proprietary types. This data can be serialised using an array of serialisation protocols (e.g. RDFXML, Turtle, Ntriples etc).

An example of a turtle encoded *information.turtle* segment is shown in Tables 1. This example demonstrates that entity attributes can be expressed using distinct data types. For example, the aff4:createdTime attribute is stored using the <xsd:dateTime> standard type, allowing the implementation to use proper time arithmetics with this attribute.

The RDF specification also allows for arbitrary new serialisation types to be used. In our example, the bevy index is of the <aff4:integer_array_inline> type which is defined by our implementation to be simply a list of integers written inline (The bevy index is simply a list of offsets into each chunk stored in the bevy (Cohen et al., 2009)). For larger bevies however, storing the index inline within the RDF information file is inefficient. Hence, we also define an alternate binary serialisation <aff4:integer_array_binary> which stores the integer array as a list of 32 bit integers encoded in big endian encoding within another segment.

The implementation defines these alternate serialisation formats, but they are functionally equivalent - allowing us to use the most appropriate type. The implementation simply makes a request to the AFF4 resolver to obtain the <aff4:index> attribute of the bevy, and it receives an instance of an integer array. The specific serialisation of the array is opaque to the application, as long as the received object implements the array interface. We can further extend the implementation in future by providing another RDF type with the expected class interface but a different serialisation or implementation.

This flexibility allows third parties to extend the basic AFF4 types, or introduce new ones in order to deliver proprietary efficiency and functionality improvements to the AFF4 format. These alternative RDF types can be added to the standard types, so that implementations which do not support these proprietary types are able to fall back to standard types. The example shown in Tables 1 illustrates how the same object attribute can be serialised using alternate equivalent RDF types.

Table 1 – An example Turtle serialisation of an AFF4 image object. Note that attributes have their own distinct types (integers, URLs and xsd:dateTime).

```
@base <aff4://34c958c0-7955-4a72-bb9b-6b049ce3a7e9>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix aff4: <http://afflib.org/2009/aff4#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

<aff4://561301aa89c7020254dd3afed37ed7547a93dbc7/00000000>
  aff4:index "index"^^aff4:integer_array_binary.

<aff4://f9351fd873fbbcc76ad121c0a7f06596478ba83c/00000000>
  aff4:index "0,13636,"^^aff4:integer_array_inline.

<aff4://829c059dbc2788e985c68e769205d7959938968e/00000000>
  aff4:index "0,7624,"^^aff4:integer_array_inline,
    "index"^^aff4:interger_array_binary.

<aff4://f9351fd873fbbcc76ad121c0a7f06596478ba83c>
  aff4:chunk_size 32768;
  aff4:chunks_in_segment 2048;
  aff4:compression 8;
  aff4:size 49522;
  aff4:stored <>;
  aff4:createdTime "2010-02-11T13:00:25 + 00:00"^^xsd:dateTime;
  a <aff4:Image>.
```

1.2. *Alternative map implementations*

One of the core innovations in AFF4 is the integration of storage virtualisation as a fundamental building block within the container format. The map stream facilitates zero copy construction of arbitrary streams of data from other streams. Conceptually it is a description of how data from a number of streams can be combined in a piecewise linear fashion to create a logical representation of a new stream. For example, Fig. 1 illustrates a traditional map expressed in an AFF4 map stream. The map stream refers to a number of byte ranges taken from a linear stream of a hard disk image. There is no need to store a separate copy of the data within the map, as all the files data can be reconstructed from the image using the map.

Previously map definitions were serialised as a text file containing tuples of stream offset, target offset and target URL. This representation was chosen due to our design goal of human readability. However, in practice it was found to scale poorly when many points exist in the map. The resulting large maps required significant time to parse, and the duplication of the target URLs on each line resulted in storage inefficiency.

An alternative binary map serialisation method is depicted in Tables 2. The map is represented as a sorted array of

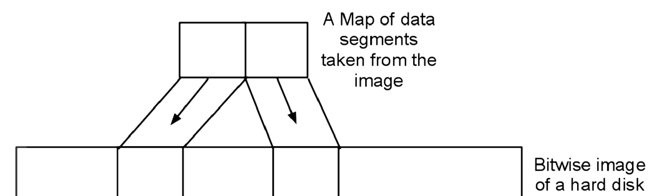

Fig. 1 – A traditional AFF4 map specifying how a file can be read from an image by combining a number of fragments.

Table 2 – Binary representation of the map. The example illustrates a map where bytes 0–299 will be read from offset 100 in target index 0, while 300 onwards will be read from offset 5000 from target index 1.

```
struct {
   uint64_t stream_offset;
   uint64_t target_offset;
   uint32_t target_index;
} map_point[];

Example (Encoded in big endian):
0,100,0
300,5000,1
```

records, ordered by stream_offset. Since the list is sorted, a binary search of the map is $O(\log n)$. The map is stored in a special segment named by concatenating the streams URL with "*/map*". Depending on other options (such as encryption, compression and remote access protocols) it may be possible to memory map the relevant segment, in which case very large maps can be managed with small physical memory and IO overheads.

The target_index is an integer index into the array of targets. The targets are saved as a null terminated array of strings into a segment named by concatenating the stream URL with "*/idx*".

1.3. In-memory map implementation

The maps are stored in memory using a treap data structure, keyed by the stream offset (Aragon and Seidel, 1989). This data structure is ideal since both insertion and retrieval of points is $O(\log N)$. The treap automatically maintains the points in sorted order, so serialisation is simply a matter of traversing the treap, an operation of the order of $O(N)$.

The most interesting property of the treap is its ability to search for the highest key below a query key and the lowest key just above a given query key. When reading from a certain offset in the map we wish to retrieve the byte range that our map offset is read from. This essentially means searching for the next highest point just below our current offset, and the map point with the lowest map offset just greater than our current offset.

In these ways the treap data structure is a perfect fit for implementing an efficient map.

2. Hash based imaging

One of the first applications of partial block based hashing within a file format was the bit torrent protocol and associated torrent file specification (Cohen, 2003). The torrent file is formed by dividing the files sent into piecewise hashes which are downloaded separately. The hash is not merely used as a check sum, but is actually used in the torrent protocol to request and address the specified piece.

Watkins et al (Watkins et al., 2009) proposed a system, named "Teleporter", for efficiently transporting forensic images over networks by avoiding transmission of common and well known files, such as common operating system binaries. Instead, they transmitted the location of such a file and a hash value of the data content of the file, reconstituting at the other end the actual data from a corpus which maps hashes to files. While the idea was primarily designed for efficient transmission of images, they also provided for the construction of "skeleton images" with some storage saving for archival purposes.

Teleporter uses a specially designed transmission protocol to communicate between client and server. The skeleton file format is proprietary and server infrastructure to manage the data is required. Yet, when the system is primed with a sufficiently large corpus of images, many hashes in newly acquired images were found to match the corpus leading to significant bandwidth saving.

Recently a standard corpus of forensic images was devised (Garfinkel et al., 2009). One purpose of which is to facilitate independent reproduction of research results. One difficulty identified with the production and sharing of standard forensic corpora was that realistic images will by necessity contain files which are copyright, potentially leading to license infringement where those files are redistributed. The solution was to redact the files using a binary scrambling technique. This solution, however, results in the corpora failing to behave correctly with regard to forensic techniques such as hash matching of redacted files. Furthermore, practitioners who do have legitimate copies of the files are unable to seamlessly plug their legitimate files into the redacted image in order to "un-redact" it, instead a new version of the entire image must be produced and redistributed, potentially resulting in significant IO overheads.

Redaction of data is essential in many other applications, such as the sharing of images containing known prohibited multimedia files, or potentially sensitive corporate information. Section 2.4 discusses how redaction can be performed in a seamless manner using hash based imaging.

The remainder of this work is devoted to the development of a hash based imaging technique using the standard AFF4 file format. We first describe the theoretic overview of a generic AFF4 hash based imager. The tradeoffs and considerations in implementing such a scheme are then explored, and finally a comparison of our scheme to other popular forensic imaging tools is made.

Our novel hash based compression is used to create one or more standard AFF4 volumes, which can be read by any tool using the regular AFF4 library (or indeed using the virtualized raw format image provided by the standard fuse implementation, as provided by libaff4 (Various file System in Userspace, 2010)). Our novel algorithm is strictly used in preparing the image, leveraging AFF4s decentralised architecture. This application is an example of the power and flexibility provided by AFF4 in assisting the development of novel algorithms and solutions.

2.1. Imaging overview

Hash based imaging relies on dividing the hard disk into a sequence of byte ranges. The general approach is depicted in Fig. 2. The hard disk is represented as an *image map*, addressing many targets. We refer to each target as a *Byte Stream*, since it is merely a stream created by collecting byte ranges from the disk. The map essentially provides a recipe for

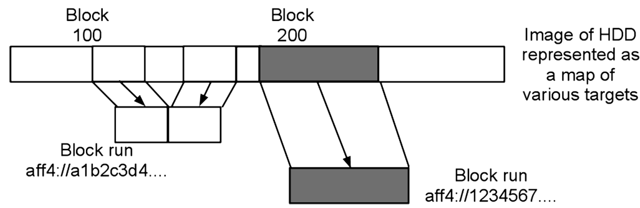

Fig. 2 – A conceptual diagram depicting the hash based compression data-flow.

reading each image byte from one of many distinct *byte streams*.

The *byte streams* themselves are stored with a URL that reflects their hash value (for example aff4://sha1_hash). Each of those streams is stored within some AFF4 volume accessible to the local AFF4 implementation.

In the example shown in Fig. 2, a read request issued for block 100, results in the stream aff4://a1b2c3d4 being fetched and opened. A read request for block 200 is fetched from aff4://1234567. The specific streams fetched may reside on the same or different AFF4 volumes, since AFF4s distributed architecture allows any stream to be stored in any valid accessible volume as long as the local resolver is aware of its location.

Our goal in imaging is to divide the disk into many byte ranges and store each run separately in an AFF4 volume. The image map is then created in such a way that the disk image can be faithfully reconstituted from the logical map at a later stage. The byte ranges themselves can be divided arbitrarily using a *segmenting algorithm*.

The end result is one or more AFF4 volumes containing a large number of byte streams with a URL of the form aff4://encoded_hash and a single AFF4 map object representing the hard disk image.

For disk images which contain filesystems, an effective segmenting algorithm uses filesystem allocation information to select byte ranges which approximate the actual allocation of files. This maximises the probability that byte streams derived from the same files contain the same data for many disks within the corpus.

It is important to emphasise that our algorithm works on the block level, and each byte stream does not actually contain the same data as the file it represents within the filesystem. Most modern filesystems present files as a logical abstraction independent from their actual block level storage. In general, files are linearly mapped to the block level storage, however, in a significant number of cases they are not. For example, a compressed NTFS files data does not correspond to the data stored in its byte stream. In this case it is not useful for our algorithm to store the files data in a separate stream because we can not map the disk back to it (it is not a linear transformation). However, if we store the byte stream itself using its hash, it would be likely to contain the same data as another instance of this compressed file on another image (but may not match the same file in an uncompressed state).

For these reasons byte stream hashes do not correspond to file hashes in the general case, and therefore are not a substitute to forensic file based hashing analysis. Our goal is merely to recreate a forensically valid image of the source drive rather than perform forensic analysis.

Once byte ranges are generated from allocated files, the space between the allocated blocks can be used to form *residual byte range streams*. The residual byte streams do not necessarily only reflect unallocated files, simply those bytes which do not belong to previous byte streams.

Section 2.3 discusses the tradeoffs and design of the segmenting algorithm.

2.2. *Imaging efficiency*

In modern acquisition tasks, acquisition schemes must be compared by both storage efficiency and acquisition time. Although available processing power increases rapidly, it is usually divided between several cores. This trend encourages multithreaded acquisition tasks, where processor intensive tasks are performed by different cores.

An obvious benefit to our scheme is the advantage that possessing a large corpus of images brings. Clearly the larger the corpus, the more likely we are to already posses the copied byte stream in one of the images within it. For example, assume our corpus contains an image a WinXP SP2 system. When imaging another such system, many of the system files will already exist within the corpus and will not need to be acquired.

Similarly, if data is duplicated within the same image (e.g. through multiple copies of the same file), multiple references are made to the same AFF4 hash stream in the image map. This is especially interesting when considering that most modern drives are very large, and will typically have many runs of zeros or another constant value. Since the hash calculated on unallocated byte ranges will be the same, we will only create a single compressed data stream of the constant residual stream.

Another benefit with our scheme is the ability to differentially compress different parts of the image. Traditionally, the full hard disk image can either be compressed or non-compressed. However, in our scheme, the byte stream is stores in the AFF4 volume as an independent stream, allowing us to choose in advance if it should be compressed. For example a fast entropy estimator can be used to decide if the sequence is likely to be compressible at all and only attempt compression if the entropy is deemed low enough. Alternatively we can make this choice based on filesystem filename or another quickly attainable property of the stream. This flexibility effectively allows us to adapt our compression dynamically – optimising both acquisition speed and storage requirements.

2.3. *Segmenting algorithm*

The segmenting algorithm divides the disk into byte range streams. The algorithm must balance competing requirements:

2.3.1. *Maximal size of byte streams*

Since each byte stream must be hashed first, and then potentially compressed it is inefficient to re-read the byte

stream from disk in two passes. It is better to read the byte range data into memory at once, then hash it and compress it from memory. This inherently limits the maximum size of the byte stream. If block runs are derived from large files allocated on the filesystem, they must be subdivided to prevent byte streams from being too long. This maximum size imposed on byte streams helps to increase the probability of some part of the file matching an existing hash, even if the entire file does not – for example, if the file contains large runs of zeros.

2.3.2. *Minimal size of byte streams*

In a multithreaded implementation the goal is to keep cores busy at all time, minimising context switches. The AFF4 resolver is used by all threads and can become a source of contention in a heavily multithreaded program. It is more efficient therefore, to amortize resolver access by tasking each core with processing a larger byte stream. Using larger streams also reduces seek times, thread context switches and increases compressibility.

2.3.3. *Selective compressibility*

When segmenting the disk based on filesystem allocation information we can make an educated guess of the potential compressibility of the data before attempting to compress it. For example when segmenting a file with a file extension such as .mp3 or .avi, it is extremely unlikely to be compressible and we can avoid spending time compressing it by dumping it as an uncompressed AFF4 byte stream. Thus we have the ability to compress selected parts of the image – a capability which is not present in current imaging formats which must attempt to compress all parts of the image.

2.3.4. *Byte stream reuse*

Maximum efficiency is achieved when byte stream containing the same data are collected within the same corpus. The algorithm aims to maximise the probability that a particular byte stream exists within the corpus by using the filesystem to extract byte ranges corresponding to block allocation of stored files. If the same file is present in multiple images in the corpus, the byte stream corresponding to each appearance is likely to contain the same data, regardless of the specific filesystem layout within each image.[1]

2.3.5. *Sequential disk access optimization*

A common optimisation for IO intensive tasks is to minimise disk head seeks by ordering disk reads in consecutive order. This ensures that the disk readahead cache is full, and maximises the probability of byte ranges read from cache. A good optimisation for the segmentation function is therefore to emit the byte streams in order of their appearance within the disk. This ensures that the disk is read sequentially and minimises overall seek time provided the byte streams are not too fragmented.

[1] An exception to that is the case where the same file is transformed by the filesystem in some way – for example NTFS compressed files, EFS encrypted files, and small files resident in MFT entries.

2.3.6. *Redaction considerations*

An exception to the above is the case where files must be redacted from the image. As described in Section 2.4, redacted files must be contained within their own byte stream, regardless of their size, so that they may be conditionally included in the redacted volume. The segmentation algorithm might check candidate files for redaction at this stage, and include them within a block run, even if they are smaller than the minimum block run.

Our segmenting algorithm operates in two passes. On the first pass the filesystem is analysed and block allocation information for all allocated files is extracted. This allocation information is converted to byte ranges by splitting sequential file allocations such that they do not exceed the maximum byte segment length. Short byte streams (for example small NTFS MFT resident files) are also discarded at this stage as they will be merged into *Residual Byte Streams* in the second pass. Any necessary redaction is also performed at this stage.

The second pass then creates *Residual Byte Streams* by collating the byte ranges not present in the byte streams generated in the first pass into new byte ranges of the constant maximal length. This effective combines smaller "holes" in the allocated blocks into larger byte streams increasing the probability of a hash match. Finally, All byte streams are read in order of their appearance on the disk and are farmed off to multiple threads to hash and compress as needed.

An example of our segmentation algorithm is seen in Fig. 3. In this example, an allocated file is found to be fragmented. We create a byte stream corresponding to the allocated file by merging the two fragments byte ranges. In the second pass, the unallocated blocks before the first fragment, between the two fragments, and some unallocated data after the allocated file are merged to form a single residual byte stream. This stream is of constant length, sufficiently long to reduce context switching related overheads.

One might be tempted to conclude that the extra level of processing is bound to result in a time penalty during acquisition. The amount of time taken during acquisition however, is a balance of competing effects – some increase acquisition time, while others reduce it. In the case that a file on the acquired hard disk already exists in our corpus, we still need to read it in order to calculate its hash, but we avoid compressing it and writing it to the output media. On the other hand if the file does not exist in the corpus we are penalised by having to calculate its hash in addition to compressing and

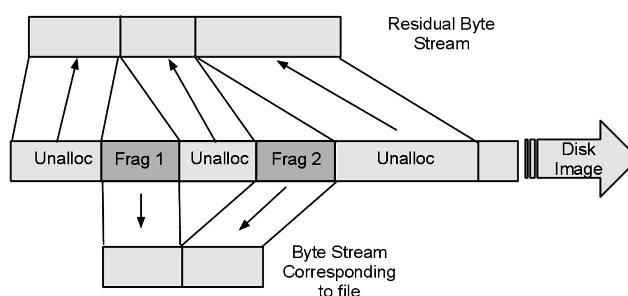

Fig. 3 – An example of our segmentation algorithm.

writing it onto the output media. Anecdotally it was found that hashing is an order of magnitude faster than compression in the typical case, leading to a significant saving when we are able to avoid compression.

Acquisition time will be minimised when the acquired image contains mostly zero blocks or standard operating system files, and a similar OS build exists in our corpus. In this case imaging simply boils down to a filesystem traversal, followed by a hashing operation of each file and unallocated data, an operation potentially faster than compression. In this case a significant saving in storage space is expected.

On the other hand if the image contains unique data (e.g. pseudo random data as in an encrypted or an already compressed hard disk) our imaging technique will suffer the additional penalty of hashing beyond the need to compress the acquired files. In addition, a slight penalty in storage space will result due to the extra space required to store the map itself.

Most hard disks fall somewhere between these two extremes. In our experience, most real hard disks contain a great deal of similar or repetitive data. Our technique has the potential to provide significant savings to acquisition time and storage requirements.

Clearly the segmentation algorithm must balance many competing needs in order to efficiently de-duplicate the image data. It is important to emphasise however, that ultimately the compression and speed efficiency of the compressor are independent of the end user decompressor. The user simply uses the standard AFF4 library to read the image. Third party optimisations and more sophisticated segmentation algorithms may be devised in future to produce even better imagers, but backwards compatibility is assured since the basic building blocks of this scheme exist within the AFF4 file format.

2.4. *Redaction of byte ranges within images*

Garfinkel et al (Garfinkel et al., 2009) have argued the importance of a standard corpora of forensic images in advancing the field. Building and sharing such a corpora, in a manner that the images within the corpora are sufficiently representative of real world images, is significantly complicated by redaction oriented concerns. Sharing of forensic images, in both civil and criminal matters, is similarly complicated. Forensic images will typically contain files which, should they be copied or possessed, may breach confidentiality concerns, violate licensing terms, or break laws.

Redaction (selective censoring) of files within images might appear deceptively straight forward at first. However, a number of non-trivial challenges appear when one attempts to redact specific files from the filesystem. These primarily stem from the complex mapping between the logical file presented by the filesystem, and its block level representation. For example in the case of versioned filesystems (such as the volume snapshots used by windows 2003 and above), the blocks underlying a single file instance may be shared by multiple versions of the same file, and the same file may have redundant copies on disk. Modern filesystems also perform block level de-duplication (e.g. ZFS (Sun Microsystems, 2010)) causing a single block to be shared by multiple unrelated files. In addition, files may be arbitrarily located or backed up in other storage containers, such as PST files or archives.

Due to the significant challenges with file based redaction, we have in the context of the work described in this paper, focused exclusively on redaction of blocks of data, rather than files within an image. We consider block level redaction an essential (but insufficient) building block for implementing complex, filesystem aware, redaction algorithms.

Current generation forensic container formats provide no means to describe which byte ranges are redacted and thus invalid. For example, Garfinkel et als images simply replaced the redacted files with corrupt versions of these files. There is no way to tell that these are redacted within the confines of the forensic container. Similarly, currently there is no way to automatically "un-redact" the image by providing just the redacted blocks – an entirely new image must be created and distributed.

On the other hand, the hash based imaging approach described in this paper seamlessly supports redaction of byte runs. Our redaction aware imager operates with a segmenting algorithm as described above, however the bytes to be redacted are stored in a stream within a separate redaction evidence container. The final product is therefore an AFF4 volume containing the map representing the hard disk and most of the byte streams, and a separate redacted volume representing the redacted files. These volumes can be distributed separately or protected using the fine grained encryption policy already inherent in the AFF4 file format Cohen et al., 2009.

In addition to storing the byte streams in a different container, a fact would be asserted within the AFF4 resolver – and the RDF serialisation, of the reason to the redacted streams absence. Should an attempt be made to read the blocks corresponding to the redacted byte ranges, without access to the redacted volume, this operation will fail. A compliant tool can obtain the reason for the redaction and report it to the user by querying the AFF4 resolver. Simply obtaining the redacted volume and adding it to the local AFF4 resolver at any time, will allow these reads to succeed seamlessly.

2.5. *Remote transmission of images*

Many digital forensic laboratories operate in a distributed environment. Acknowledging the need to collaborate with images stored in remote locations, a number of advanced techniques have been developed. For example, the Teleporter tool uses a database of hashes on the acquisition tool to selectively transmit images to a remote server (Watkins et al., 2009). Teleporter is focused on the problem of image transmission from a remote location rather than persistent access. Once the image skeleton is delivered it must be recreated on the server in order to be used, but the image skeleton may be kept for archival purposes.

AFF4 is a forensic storage format with a focus on distributed storage. For example, suppose examiners work in two locations A and B which are geographically remote. The examiners in location A used their corpus of hashes to acquire

a new hard disk. Those files not present in the corpus were copied and appended to the corpus.

An examiner in location B wishes to examine the disk image acquired at A. They download the hash map representing the new disk image, and open the hash map using their local AFF4 stack. The resulting AFF4 map handle can be wrapped using the following pseudo code listed in Table 3

In this example, the AFF4 objects read() method is wrapped such that if the map attempts to open a hash stream not available to the local implementation, we attempt to fetch it from the remote server by other means, and add the new hash data stream to our local corpus. A single read request may require many hash data streams to be fetched. In effect, unique files from the image will be fetched on demand and cached locally.

On the other hand, if the examiner in location B does not need to fetch the entire image, the interaction can be very quick and bandwidth efficient – only fetching the required data. Fetching the entire image from location A to location B involves transferring only those files which are not present in the corpus in location B. This should be contrasted with Teleporter technology where the client needs to know in advance what hashes are present on the server in order to prepare a transmission package to the server omitting those file already present on the server.

2.6. *Potential risks*

Hashing is inherently susceptible to collisions simply due to the reduced entropy. Although cryptographic algorithms are specifically designed to make it difficult to engineer a collision, random collisions can occur with a probability of 2^{-n} where n is the bit length of the hash. Widely identified weaknesses in hash algorithms such as MD5 and SHA1 further increase the odds of a collision.

While the possibility of a hash colliding can not be discounted, but it can be reduced by increasing the effective bit length. This can be achieved by combining several hash algorithms or using more complex algorithms.

Using more complex hashing algorithms can increase the cost of calculating a hash. On modern multicore systems however, this increased cost is unlikely to impact acquisition speed since calculating the hash is purely a processor intensive task. In our implementation, hashing occurs by other threads while the main thread is waiting on IO. We did not find significant performance differences between using MD5, SHA1 or SHA256.

Ultimately, for a forensically valid imaging application, the reproduced image must pass the *Ultimate Test* (Turner, 2006), and produce exactly the same behaviour which a raw image produces.

Table 3 – An example of how the AFF4 map object can be wrapped to fetch missing hash data streams on demand.

```
class WrappedMap(Map):
    def read(self, length):
      while 1:
          try:
              return Map.read(self, length)
          except IOError,e:
              if not self.fetch_from_remote_server(e):
                      raise e
```

3. Method

Our implementation was written in Python using the standard AFF4 library python bindings. We used the pytnon bindings for Sleuthkit (Carrier, 2003) as provided by the PyFlag project (Cohen, 2008).

To gain comparative insight of our technique we converted a dd image of a 4 Gb Windows XP SP2 system. The system imaged was a typical workstation with applications such as Visual Studio, OpenOffice and FireFox. The imaging machine was a Dell Inspiron 6400, dual core Intel T2250 with 2 Gb of RAM, running Ubuntu 9.04. Timing was measured using the time command.

The image was first converted to the legacy AFF format using AFFLIB 3.3.3, and to the EWF format using libewf Version 20091224. In both these cases we chose the fastest compression setting available by the tool (ewfacquirestream -c fast and afconvert -x1).

We then applied our tool to the same image without a corpus loaded. This produced a worst case scenario where none of the byte streams could be skipped. Some byte streams naturally appeared in the image more than once and were deduplicated.

To estimate an upper bound on our tools performance we repeated the acquisition, this time with the same image already added to the corpus. This test ensures that all byte stream hashes will always be present in the corpus – hence no byte streams will actually be saved to the AFF4 volume.

Realistic images will perform somewhere between the best case and worst case. This allows us to measure upper and lower bounds on performance of our tool.

4. Results

Both CPU time and wall clock time taken for imaging are recorded in Table 4. Total resulting image size is also reported. Note that for multithreaded applications CPU time can exceed wall clock time since there are two cores in this system. The standard AFF4 imager is multithreaded and therefore already has a much higher throughput than the standard AFFLIB or EWF imagers. Since this system has two cores we expect it to be approximately twice as fast as similar implementations.

5. Discussion

Our prototype implementation was found to perform well against other imaging tools. At worst our performance is similar to the popular *ewflib* acquisition tool in both resulting image and acquisition time. However, when a significant

Table 4 – Comparison of imaging times and resulting volume size for a variety of imaging tools.

Acquisition Method	Total Image Size (Bytes)	Elapsed Time	User CPU time
Hash based Imaging (No corpus)	1,586,315,782	7m12s	5m42s
Hash based Imaging (Full corpus)	108,573	4m41s	0m51s
EWF – fast setting	1,622,220,104	7m11s	5m23s
AFFLIB Compression level 1 (fastest)	1,590,285,671	9m55s	7m7s
AFF4 imager	1,621,922,977	4m37s	5m43s

portion of the corpus matches the acquired image we are able to improve our acquisition speed significantly. The size of the resulting volume is reduced significantly. Indeed in this pathological case, the resulting volume contains only the map object. Real world images are likely to range in size between the two extremes – on both acquisition time and storage space.

Another interesting observation is the performance enhancement that a multithreaded implementation such as the standard AFF4 imager gains on multicore architectures. Since the standard AFF4 imager spends most of its CPU intensive time in the compression loop it is able to parallelise very well with very little mutex contention. This results in more CPU time consumed than real time (i.e. performance scales with extra cores).

On the other hand, our hashing implementation performs less well in this regard, failing to scale with the number of cores. When the full corpus is available, our application uses little CPU time but still takes as long as the AFF4 imager does. This is possibly due to increased mutex contention arising from heavy utilisation of the AFF4 resolver by all threads. This suggests IO or mutex contention bottlenecks which still need to be addressed.

Although our implementation can clearly be improved, it already produces comparable results to existing tools in performance, while providing significant de-duplication in storage. Possible improvements can be made to the segmentation algorithm, increasing the probability of de-duplication in typical images. Other optimizations include balancing compression with speed by selectively choosing to compress only compresseable byte runs, and adapting our segmentation algorithms to the specific type of image acquired.

The real strength of our technique is in producing a flexible framework for hash based imaging with the advantages of de-duplication and transparent redaction built into the file format. This framework can easily be built on to provide more complex imaging capabilities.

REFERENCES

Aragon CR, Seidel R. Randomized search trees. In: Proc. 30th symp. foundations of computer science. Washington, D.C.: IEEE Computer Society Press; 1989. p. 540–5.

Beckett D. Raptor rdf parser library, http://librdf.org/raptor/libraptor.html; Feb 2010.

Brickley D, Guha RV. RDF vocabulary description language 1.0: RDF schema. W3C working draft, http://www.w3.org/TR/2003/WD-rdf-schema-20030123/; January 2003.

Carrier B. The sleuthkit, http://www.sleuthkit.org/sleuthkit/; 2003 [accessed on 02.2010].

Cohen M, Garfinkel S, Schatz B. Extending the advanced forensic format to accommodate multiple data sources, logical evidence, arbitrary information and forensic workflow. Digital Investigation 2009;6:S57–68.

Cohen B. Incentives build robustness in bittorrent. In: Proceedings of workshop on economics of peer-to-peer systems; 2003.

Cohen MI. PyFlag: an advanced network forensic framework. In: Proceedings of the 2008 digital forensics research workshop. DFRWS; Aug. 2008.

Garfinkel S, Malan DJ, Dubec K-A, Stevens CC, Pham C. Disk imaging with the advanced forensic format, library and tools. In: Research advances in digital forensics (Second annual IFIP WG 11.9 international conference on digital forensics). Springer; Jan 2006.

Garfinkel S, Farrella P, Roussevc V, Dinolta G. Bringing science to digital forensics with standardized forensic corpora. Digital Investigation 2009;6:S2–11.

Kloet B, Metz J, Mora R-J, Loveall D, Schreiber D. libewf: project info, http://www.uitwisselplatform.nl/projects/libewf/; 2008.

Lawrence LY, Kristal TP, Long DE. Deep store: an archival storage system architecture. In: Proceedings of the 21st international conference on data engineering. IEEE; 2005.

Schatz BL, Clark A. An information architecture for digital evidence integration. In: Australian security response team annual conference; 2006.

Sun Microsystems. Solaris zfs, http://www.sun.com/software/solaris/zfs.jsp; Feb 2010.

Turner P. Unification of digital evidence from disparate sources (digital evidence bags). Digital Investigation 2005;2(3):223–8.

Turner P. Selective and intelligent imaging using digital evidence bags. Digital Investigation 2006;3:59–64.

Various, “Filesystem in userspace”, http://fuse.sourceforge.net/; Feb 2010.

Watkins K, McWhorte M, Long J, Hill B. Teleporter: an analytically and forensically sound duplicate transfer system. Digital Investigation 2009;6:S43–7.