2009

Extending the Advanced Forensic Format to accommodate Multiple Data Sources, Logical Evidence, Arbitrary Information and Forensic Workflow

DIGITAL INVESTIGATION 6 (2009) S57–S68

available at www.sciencedirect.com

ScienceDirect

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

Digital Investigation

Extending the advanced forensic format to accommodate multiple data sources, logical evidence, arbitrary information and forensic workflow

Michael Cohen, Simson Garfinkel, Bradley Schatz*

Australian Federal Police, High Tech Crime Operations, 203 Wharf St., Spring Hill, Brisbane 4001, Australia

Keywords:
Digital forensics
Image
Hard disk Imaging
Digital Evidence Management
Distributed Storage
Distributed Forensic Analysis
Forensic File Format
Evidence Archiving
Cryptography
Forensic Integrity

ABSTRACT

Forensic analysis requires the acquisition and management of many different types of evidence, including individual disk drives, RAID sets, network packets, memory images, and extracted files. Often the same evidence is reviewed by several different tools or examiners in different locations. We propose a backwards-compatible redesign of the Advanced Forensic Format—an open, extensible file format for storing and sharing of evidence, arbitrary case related information and analysis results among different tools. The new specification, termed AFF4, is designed to be simple to implement, built upon the well supported ZIP file format specification. Furthermore, the AFF4 implementation has downward comparability with existing AFF files.

1. Introduction

Storing and managing digital evidence is becoming increasingly more difficult, as the volume and size of digital evidence increases. Evidence sources have also evolved to include data other than disk images, such as memory images, network images and regular files. Preserving such digital evidence is an important part of most digital investigations (Carrier and Spafford, 2004), and managing the evidence in a distributed organization is now emerging as a critical requirement.

This paper presents a framework for managing and storing digital evidence. We first examine existing evidence management file formats and outline their strengths and limitations. We then explain how the proposed Advanced Forensics Format (AFF4) framework extends these efforts into a universal evidence management system. The detailed description of the AFF4 proposal is then followed by concrete real world use cases.

1.1. *Prior work*

In recent years there has been a steady and growing interest in the actual file formats and containers used to store digital evidence. Early practitioners created exact bit-for-bit copies (commonly referred to as ''dd images''). More recently, proprietary software systems for making and authenticating ''images'' of digital evidence have become common (e.g. B.S. NTI Forensics Source, 2008; Ilook investigator, 2008; Guidance Software, Inc., 2007). PyFlag (Cohen, 2008a) introduced a ''seekable gzip'' format that allowed disk images to be stored in a form that was compressed but allowed random access to evidence data necessary for forensic analysis.

* *Corresponding author*. Tel.: +61 732221361.
E-mail address: scudette@gmail.com (M. Cohen).

1742-2876/$ – see front matter © 2009 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2009.06.010

The Expert Witness Forensic (EWF) file format was originally developed for Encase (Guidance Software, Inc., 2007), but then adopted by other vendors (Kloet et al., 2008). The EWF file format similarly compresses the image into 32 kb chunks which are stored back to back in groupings inside the file. The format employs tables of relative indexes to the compressed chunks to improve random access efficiency. EWF volumes have a maximum size limit of 2 Gb and therefore usually split an image across many files. EWF provides for a small number of predefined metadata fields to be stored within the file format.

The Advanced Forensic Format (AFF) expanded on this idea with a forensic file format that allowed both data and arbitrary metadata to be stored in a single digital archive (Garfinkel et al., 2006).

Both the AFF and EWF file formats are designed to store a single image, and any metadata that implicitly refers to that image such as sector size and acquisition date. Unlike EWF, AFF employed a system to store arbitrary name/value pairs for metadata, using the same system for both user-specified metadata and for system metadata, such as sector size and device serial number. For example, `Aimage`, the AFF hard disk acquisition tool, not only stores the image, but additionally stores a description of the tool itself, the version of AFFLIB used to create the image, the computer on which the image was made, the operator of the tool, the user supplied parameters supplied to the tool.

Schatz proposed a Sealed Digital Evidence Bags architecture, facilitating composition of evidence and arbitrary evidence related information, through a simple data model and globally unique referencing scheme (Schatz and Clark, 2006).

1.2. *This paper*

An important advance of this work is the introduction of storage transformation functions to the forensic storage container. Prior works simply focused on forensically sound storage of bit-streams, leaving the necessary activities of translating low level storage into higher level abstractions at the aggregate block (i.e. RAID), volume, and filesystem layers in the domain of analysis tools, as transiently constructed artifacts. In contrast AFF4 has mechanisms for describing transformation in a flexible and concise way, allowing users to view multiple transformations of the same data with little additional storage cost. This mechanism is an important enabler for inter-operable forensic tools. For example, carved files may be described in terms of their block allocation sequences from an image, rather than requiring the carved file to be copied again.

This paper extends previous work on the Advanced Forensic Format (AFF) by taking many of the concepts developed and designing a new specification and toolset. The AFF4 format is a complete redesign of the architecture. The new architecture is capable of storing multiple heterogeneous data types that might arise in a modern digital investigation, including data from multiple data storage devices, new data types (including network packets and memory images), extracted logical evidence, and forensic workflow. The AFF4 format extends the format to make it the basis of a global distributed evidence management system.

We call the new system AFF4, and use the phrase AFF1 to refer to the legacy system developed by Garfinkel et al.[1] The publicly released AFF4 implementation, is able to read existing AFF files.

2. The need for an improved forensic format

AFF1's flexibility came from a data model of forensic data and metadata stored as arbitrary name/value pairs called *segments*. For example, the first 16 MB of a disk image is stored in a segment called page0, the second 16 MB in a segment called page1, *etc.* Because of this flexibility, it was relatively easy to extend AFF1 to support encryption, digital signatures, and the storage of new kinds of metadata such as chain-of-custody information (Garfinkel, 2009).

2.1. *AFF limitations*

We observed a number of practical problems in the underlying AFF1 standard and Garfinkel's AFFLIB implementation:

- While AFF1's design stores a single disk image in each evidence file, modern digital investigations typically involve many seized computers or pieces of media.
- The data model of AFF1 enabled storing metadata related to the contained image as (property, value) pairs. This data model does not, however, support expressing arbitrary information about more than one entity.
- AFF1 has no provision for storing memory images or intercepted network packets.
- AFF1 has no provisions for storing extracted files that is analogous to the EnCase "Logical Evidence File" (L01) format, or for linking evidence to web pages.
- AFF1's encryption system leaks information about the contents of an evidence file because segment names are not encrypted.
- AFF1's default compression page size of 16 MB can impose significant overhead when accessing NTFS Master File Tables (MFT), as these structures tend to be highly fragmented on systems that have seen significant use.
- Although the AFF1 specification calls for a "table of contents" similar to the Zip (Katz, 2007) "central directory" that is stored at the end of AFF files, Garfinkel never implemented this directory in the publicly released AFF1 implementation, AFFLIB. As a result, every header of every segment in an AFF file needs to be read when a file is opened. In practice this can take up to 10–30 s the first time a large AFF file is opened.
- AFF1's bit-level specification is essentially a simple container file specification. Given that there are other container file specifications that are much more widely supported with both developer and end-user tools, it seemed reasonable to migrate AFF from its home-grown format to one of the existing standards.

[1] Although Garfinkel never changed the AFF bit-level specification, Garfinkel released AFFLIB implementations with major version numbers 1, 2 and 3. We therefore call our system AFF4 to avoid confusion.

2.2. Global distributed evidence management

While AFF1 was designed for use on a single machine that could both image evidence and perform analysis, many modern practitioners work in distributed environments in which imaging and analysis takes place in multiple locations and is performed by multiple individuals.

Global distributed evidence management requires more than simply tracking the movement of disk images: it requires approaches for sharing evidence to multiple disconnected evidence, allowing offline work, and then seamlessly recombining the work products of the analysts in a third security domain.

Managing evidence in a globally distributed system requires the use of globally unique identifiers to ensure no name collisions can occur with disconnected locations. AFF1 assigns each piece of evidence a unique 128-bit identifier called a GID but did not make it clear when this identifier should be changed and when it should remain the same.

Consider the typical usage scenario depicted in Fig. 1, of a volume containing a disk image. This volume is distributed to two independent analysts, Alice and Bob. Alice may find and extract individual files, while Bob may correlate information in the evidence file with other data that is available on departmental servers. Although in some environments Alice and Bob may be able to work on a shared file that is located on a server, in other environments there will not be sufficient connectivity. Instead, each analyst will be required to store the information in their own evidence file; these files will then be recombined at a later point in time.

In this case they can each create a new volume which extends the original volume and save their analysis on this new volume. Now they only need to share this new volume with other analysts who also have a copy of the old volume to interchange their findings.

This is made possible because each volume is independent of one another, but is still viewed as part of a bigger evidence set.

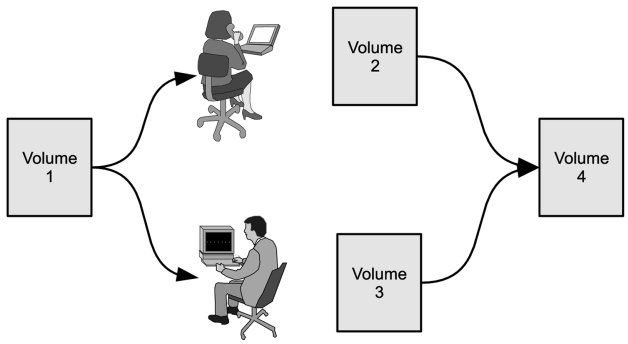

Fig. 1 – A typical usage scenario. Both Alice and Bob receive an AFF volume but work independently. Rather than modifying the volume, they each create their own local volumes and save their results into those files. They can now exchange the smaller new volumes and effectively merge their results into the same AFF set when they are finished.

3. Introducing AFF4

This section discusses the AFF4 terminology and architecture. The AFF4 design is object oriented, in that a few generic objects are presented with externally accessible behavior. We discuss a number of implementations of these high level concepts and show how these can be put together in common usage cases.

- *An AFF Object* is the basic building block of our file format. AFF Objects have a globally unique name (URN) as described in (Sollins and Masinter, 1994; Fielding, 1995, Hoffman et al., 1998). The name is defined within the aff4 namespace, and is made unique by use of a unique identifier generated as per RFC4122 (Leach et al., 2005).
- *A Relation* is a factual statement which is used either to describe a relationship between two AFF Objects, or to describe some property of an object. The relation comprises of a tuple of (Subject, Attribute, Value). All metadata is reduced to this tuple notation.
- *An Evidence Volume* is a type of AFF Object which is responsible for providing storage to AFF segments. Volumes must provide a mechanism for storing and retrieving segments by their URN. We discuss two volume implementations below, namely the *Zip64 based volume* and the *Directory* based volume.
- *A stream* is an AFF Object which provides the ability to seek and read random data. Stream objects implement abstracted storage, but must provide clients the stream like interface. For example, we discuss the *Image stream* used to store large images, the *Map stream* used to create transformations and the *Encrypted stream* used to provide encryption.
- *A segment* is a single unit of data written to a volume. AFF4 segments have a *segment name* provided by their URN, a *segment timestamp* in GMT, and the *segment contents*. Segments are suitable for storing small quantities of data, and still present a stream interface.
- *A Reference* is a way of referencing objects by use of a Uniform Resource Identifier (URI). The URI can be another AFF Object URN or may be a more general Uniform Resource Locator (URL), such as for example a HTTP or FTP object. This innovation allows objects in one volume to refer to objects in different volumes, facilitating data fusion and cross referencing.
- *The Resolver* is a central data store which collects and resolves attributes for the different AFF Objects. The Resolver has universal visibility of objects from all volumes, and therefore guides implementations in resolving external references.

4. Metadata and the universal resolver

Management of evidence requires an effective identification, with practitioners currently employing acquisition time metadata such as case identifiers and description fields in the EWF file format; file and directory naming schemes, and labeling of evidence container hard drives. Evidence may also be referenced by external means in an inconsistent way. For example, in an investigator's case note a disk image may be referred to

by the name of the suspect (e.g. Joe's hard disk), the case number or dates.

Such individuation schemes may be problematic when automatically managing evidence. For example, at acquisition time a suitably unique individuator may not be selected. If that occurred, at analysis time evidence container files may need to be renamed to avoid name collisions.

The AFF4 design adopts a scheme of globally unique identifiers for identifying and referring to all evidence. We define an AFF4 specific URN scheme, which we call the AFF4 URN. URN's of this scheme use the namespace (Sollins and Masinter, 1994) "*aff4*" and therefore begin with the string "*urn:aff4*". AFF4 URNs are then be made unique by use of a unique identifier generated as per RFC4122 (Leach et al., 2005). For example, an AFF4 URN might be *urn:aff4:bcc02ea5-eeb3-40ce-90cf-7315daf2505e*.

The AFF4 model treats metadata as an abstract concept which may exist independently from the data itself. We term *metadata* to be a set of statements about objects, written in tuple notations (Subject, Attribute, Value), where *Subject* is the URN of the object the statement is made about. An *Attribute* can be any kind of value or relationship, such as the sector size of a device, a device capacity, or the name of the person who performed an imaging operation. A *Value* is the value of the attribute, which is either another URN, or some textual value. Using this system we are able to store arbitrary attributes about any object in the AFF4 universe. Additionally, as these statements are universally scoped, they may be stored anywhere.

The AFF4 design extends beyond the management of a single volume, stream or image to a universal system for managing data of many types. This necessarily means that a single running instance is generally unable to have visibility of the entire AFF4 universe. For example, if a volume is opened which contains a Map Stream targeting a stream stored in a different volume, it is not generally possible to tell where that volume is actually stored.

To provide this global visibility of metadata we define a central metadata management entity, named the *Universal Resolver*. The Universal Resolver contains all the metadata about the AFF4 universe, that is to say it is able to resolve queries for any attribute about any URN in the universe.

Although the resolver has complete visibility of all attributes, it is still useful to store metadata within the volume itself, particularly data pertaining to the volume itself. If we did not store the metadata within the volume itself, then the volume would not be accessible to implementations which do not have this metadata.

To this end we define a way for serializing metadata statements (or tuples) into a standard format which implementations can load into their respective resolvers when parsing the volume. Relations can be stored in segments having a URN ending with "*properties*". The AFF4 implementation loads these segments automatically into the Universal Resolver.

Relations are stored within the properties segment one per line, with the subject URN (encoded according to RFC1737), followed by whitespace and the attribute name. This is then followed by the equal sign and the UTF8 encoding of the value. An example properties file for an Image Stream is shown in Listing 1.

It is important to stress that the properties file is simply a serialization of statements into volume segments. The statements may exist without being stored in a volume (for example, being stored on an external SQL server). Alternatively, these statements may be stored in some other way inside or outside the volume (e.g. SQLite database files).

When the volume is loaded, the AFF4 implementation automatically loads any properties files and populates its Universal Resolver with the information visible to it. AFF4 provides a mechanism to use an external resolver as well—for example, we have implemented a resolver that stores Attributes in a MySQL database to provide for a persistent Universal Resolver that shares information between different instances on the same network.

Although the Universal Resolver should be thought of as a truly universal entity, the library provides a local resolver which is available to the running instance. As the library explores different volumes, relations are added to the local resolver. This means that the AFF4 library does not necessarily need to have an ideal Universal Resolver, but can approximate this by use of a local resolver. The local resolver can be primed in advance by the user, by loading various volumes which may be needed to resolve internal references.

Each URN within the AFF4 universe must have an "`aff4:type`" attribute to denote the type of the Object. Objects may also have a the "`aff4:interface`" attribute to denote what kind of interface they present (e.g. stream or volume).

5. Volumes

The volume object is responsible for providing storage for segments. Segments are stored and retrieved using their URNs. We describe two different implementations of volume objects, namely the *Directory Volume* and the *ZipFile Volume*. It is possible to convert from one implementation to another easily, without affecting any external references.

It is important to emphasize that Volumes are merely containers which provide storage for segments. There is no restriction of which segments can be stored by any particular volume. For example, the segments which make up a single Image stream may be stored in a number of volumes (splitting the image in some way among them). Similarly, the segments representing a number of streams may be stored in the same volume.

5.1. *Directory volumes*

The Directory Volume is the simplest type of volume. It simply stores different segments based on their URNs in a single directory. Since some filesystems are unable to represent URNs accurately (e.g. Windows has many limitations on the types of characters allowed for a filename), the Directory Volume encodes URNs according to RFC1738 (Berners-Lee et al., 1994); non-printable characters are escaped with a % followed by the ASCII ordinal of the character.

The Directory Volume uses the *aff4:stored* attribute to provide a base URL. The URL for each segment is then constructed by appending the escaped segment URN to the base URL. Note that there is no restriction on what type of URL this

Listing 1

Example properties files for several AFF4 objects (URNs are shortened for illustration).

```
Directory Volume:
   urn:aff4:f901be8e-d4b2 aff4:stored = http://../case1/
   urn:aff4:f901be8e-d4b2 aff4:type = directory
ZipFile Volume:
   urn:aff4:98a6dad6-4918 aff4:stored = file:///file.zip
   urn:aff4:98a6dad6-4918 aff4:type = zip
Image Stream:
   urn:aff4:83a3d6db-85d5 aff4:
   stored = urn:aff4:f901be8e-d4b2
   urn:aff4:83a3d6db-85d5 aff4:chunks_in_segment = 256
   urn:aff4:83a3d6db-85d5 aff4:chunk_size = 32 k
   urn:aff4:83a3d6db-85d5 aff4:type = image
   urn:aff4:83a3d6db-85d5 aff4:size = 5242880
Map Stream:
   urn:aff4:ed8f1e7a-94aa aff4:target_period = 3
   urn:aff4:ed8f1e7a-94aa aff4:image_period = 6
   urn:aff4:ed8f1e7a-94aa aff4:blocksize = 64 k
   urn:aff4:ed8f1e7a-94aa aff4:
   stored = urn:aff4:83a3d6db-85d5
   urn:aff4:ed8f1e7a-94aa aff4:type = map
   urn:aff4:ed8f1e7a-94aa aff4:size = 0xA00000
Link Object:
   map aff4:target = urn:aff4:ed8f1e7a-94aa
   map aff4:type = link
Identity Object:
   urn:aff4:identity/41:13 aff4:common_name = /C = US/ST = CA/L = SanFrancisco/O = Fort-Funston/CN = client1/
   emailAddress = me@myhost.mydomain
   urn:aff4:identity/41:13 aff4:type = identity
   urn:aff4:identity/41:13 aff4:statement = 00000000
   urn:aff4:identity/41:13 aff4:x509 = urn:aff4:identity/41:13/cert.pem
```

can be, so it may be a location on a filesystem (e.g. *file:///some/directory/*) or a location on a HTTP server (e.g. *http://intranet.server/some/path*). In this way its possible to move the entire volume from a filesystem to a web server transparently.

The Directory Volume stores its own URN in a special segment named "__URN__" at the base of the directory.

5.2. *Zip64 volumes*

For AFF4, we have changed the default volume container file format to Zip64 (Katz, 2007). There are many reasons for this decision:

- There is already wide support for the Zip and Zip64 formats. By migrating to these formats, we can take advantage of the rich number of user and developer tools already available. The volume may be inspected using any number of commercial or open source zip application (e.g. Windows Explorer natively supports Zip files as can be seen in Fig. 3, Zip64 is supported natively by Java, Python and PERL).
- Zip64 libraries are readily available making proprietary implementations of interfaces to the AFF4 volume format simple to write. For example, a simple python program to dump out an Image stream (Section 6.1) is illustrated in Listing 2.

Fig. 2 shows the basic structure of a Zip archive. As can be seen, the archive consists of a *Central Directory* (CD) located at the end of the archive. The CD is a list of pointers to individual *File header* structures located within the body of the archive. Headers are then followed by the file data, after it has been compressed by the appropriate compression method (as specified in the header). Each archived file is optionally followed by a *Data Descriptor* describing the length and CRC of the archived file. Using the data descriptor field allows implementations to write archives without needing to seek in the output file. This allows Zip files to be written to pipes for example, sending an image over the network using netcat or ssh. AFF4 always uses the data description header to ensure volumes are written continuously without needing to seek in the output file.

Listing 2

Sample Python code to dump out an Image Stream. As can be seen the chunk index segment is used to slice the data segment into chunks. The chunks are decompressed and written to the output file.

```
volume = zipfile.ZipFile(INPUT_FILE)
outfd = open(OUTPUT_FILE,'w')
count = 0
while 1:
   idx_segment = volume.read(STREAM+''/%08d.idx''%count)
   bevy = volume.read(STREAM+''/%08d''count)
   indexes = struct.unpack(''<'' + ''L'' *
   (len(idx_segment)/4), idx_segment)
for i in range(len(indexes)-1):
   chunk = bevy[indexes[i]:indexes[i+1]]
   outfd.write(zlib.decompress(chunk))
count += 1
```

It is important to note that AFF4 only requires that the volume be capable of storing multiple named segments of data. Although our AFF4 implementation uses the Zip64 file format as an underlying storage mechanism, our system also supports legacy AFF1 volumes as well as Expert Witness Evidence files (Kloet et al., 2008).

We ignore Zip64's built-in support for splitting archives into multiple Zip files. Instead, our implementation treats each volume as a complete and stand-alone Zip file. The AFF4 implementation then considers the segments contained within as belonging to the universal collection. This provides the ability to split a stream across volumes automatically, as different segments within the same stream may be stored in different volumes.

Zip64 also defines encryption and authentication extensions. We do not use them due to the restrictions imposed on their use and because they lack the functionality that is important for a forensic user. Instead, we use AFF4's digital signature facilities for integrity and non-repudiation, and we introduce a new stream based encryption scheme for ensuring data privacy (Section 6.4).

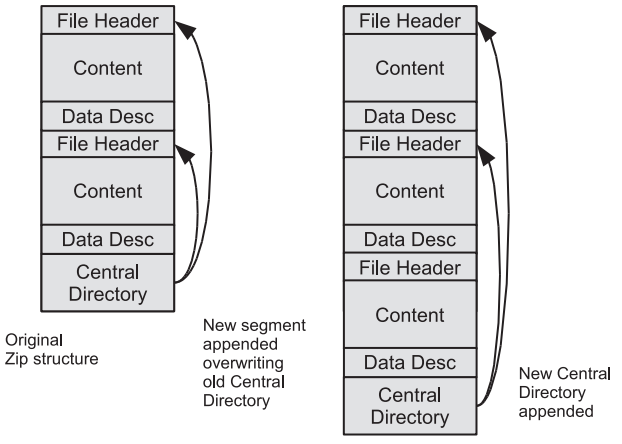

Fig. 2 – The basic structure of a Zip archive. Also shown is how new archive members are added to an existing Zip File. The Central Directory is overwritten by the new member, and a new Central Directory is written on the end.

Although there are numerous Zip implementations available today, we have created our own implementation. There are many reasons to develop our own Zip64 implementation for AFF4:

- The commonly available Zip implementations written in C do not implement the Zip64 extensions. These extensions are required to support Evidence Volumes larger than 2 GB.
- Simple Zip implementations might rescan the Central Directory for each segment request. Since in practice there can be a large number of segments in a volume, it is advisable to have a Zip64 implementation that is optimized to storing thousands (or even hundreds of thousands) of segments in an efficient data structure. In fact our implementation uses the Universal Resolver itself to store the parsed central directory information, which means that in most cases we do not even need to scan the Central Directory at all.
- While the Zip specification duplicates data found in the Central Directory entry in each File Header (such as filename, size, CRC etc), many implementations that we have examined only populate this information in one of these places. In the interest of robustness, we wanted to ensure that data stored in both locations would be populated to allow recovery of at least *some* evidence that might exist in damaged volumes. If the central directory is lost, it is possible to scan through the volume, and locate all the Zip64 file headers. Then it is possible to repair and reconstruct the central directory.
- Our implementation supports simultaneous access by multiple readers and writers. Since our system requires all metadata to be shared through the Universal Resolver, this lends itself to providing Universal Locking on a per Object basis. So for example, if one process wants to add a new segment into a Zip volume, they can lock it via the Resolver, add the segment and unlock the volume object in the

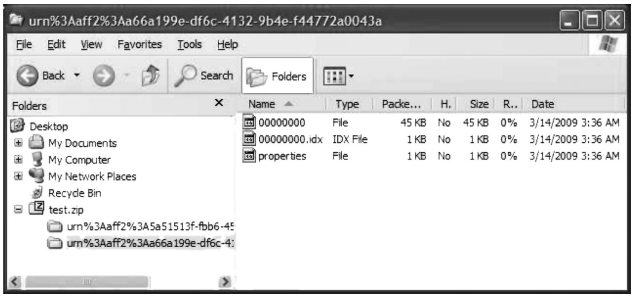

Fig. 3 – An Image stream browsed from Windows Explorer. Basic access to the evidence volume can be made using familiar tools improving transparency.

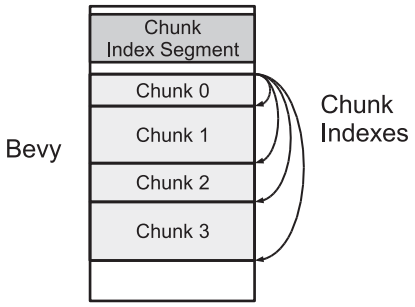

Fig. 4 – The structure of Image Stream Bevies. Each bevy is a collection of compressed chunks stored back to back. Relative chunk offsets are stored in the chunk index segment.

resolver, stopping concurrent access by other programs, even on different machines.

6. Streams

The Stream system provides random access to an abstract representation of a body of data. Our implementation allows the segments in a stream to be operated on as if they were a single file by supporting the traditional POSIX-like functionality of `open()`, `seek()`, `write()`, and `read()`. All streams also have a "`size`" attribute to denote the last byte addressable within the stream. This is required in order to support the POSIX *whence* attribute which may require seeking from the end of the stream.

The following sections describe a number of types of streams. It is important to note that clients of our implementation do not care how a particular stream is implemented. Streams are opened by their URNs, and the library itself ensures they provide the Stream interface. So for example, users do not care if a stream is a Map Stream or an Image Stream—the interface provided is the same.

6.1. *The image stream*

The AFF4 *Image Stream* stores a single read-only forensic data set. For example, this stream might contains a hard disk image, a memory image or a network capture (in PCAP format). Image streams have an `aff4:type` attribute of image.

Storage for the data is done by using multiple data segments stored on various volumes. Data segment URNs are derived by appending an 8 digit, zero padded decimal integer representation of an incrementing id to the stream URN (e.g. "urn:aff4:83a3d6db-85d5/00000032"). Each data segment is called a *bevy* and stores a number of compressed chunks back to back.

The chunk index segment is a segment containing a list of relative offsets to the beginning of each chunk within the bevy. The chunk index segment URN is derived by appending the bevy URN with ".idx". This is illustrated in Fig. 4.

Image streams specify the `chunk_size` attribute, as the number of image bytes each chunk contains (chunk size defaults to 32 kb). Also specified is the `chunks_per_segment` attribute which specifies how many chunks are stored in each bevy. Each chunk is compressed individually using the zlib compress algorithm. This general structure of storing chunks within larger segments is similar to the technique used by the Expert Witness file format (EWF) used by EnCase (Keightley, 2003) and implemented by the open source libewf (Kloet et al., 2008) package. This improvement from AFF1's 16 MB segment size results in a better match between requested size and the minimum size required for decompression. Less data is needed to be decompressed unnecessarily where reading small sectors randomly, leading to vast performance improvements.

6.2. *The map stream*

Linear transformations of data are commonplace in forensic analysis. For example, a file is often simply a collection of bytes drawn from an image, while a TCP/IP stream is simply a collection of payloads from selected network packets. Sometimes the same data may be viewed in a number of ways—for example a Virtual Address Space is a mapping of the Physical Address Space through a page table transformation (Tanenbaum, 2008). Zero storage carving (2006) is a way of specifying carved files in terms of a sequence of blocks taken from the image; Cohen extended this concept to an arbitrary mapping function (Cohen, 2008b, 2007) which can be used to describe arbitrary mappings of carved files within a single image.

In this work we extend the mapping function concept to allow a single map to draw data from arbitrary streams (called *targets*). This transform is implemented via the *Map stream*.

The mapping function is described in a segment named by appending "/map" to the stream URN. The segment data consists of a series of lines, each containing a stream offset, a target offset and a target URN. Offsets are encoded using decimal notation.

Denoting the stream offset by x, and the target offset by y, the Map specifies a set of points (X_i, Y_i, T_i). Read requests for a byte at a mapped stream offset x can then be satisfied by reading a byte from target T_i at offset y given by:

$$y = (x - X_i) + Y_i \quad \forall x \in [X_i, X_{i+1}) \tag{1}$$

For example, consider the following map:

To read this stream we satisfy read requests of offsets between 0 and 4095 in the stream from offset 0 to offset

4095 in *urn:aff4:83a3d6db-85d5*. Requests for bytes between 4096 and 8191 are fetched from *urn:aff4:f901be8e-d4b2* from offset 10000. Finally bytes after 8192 (until the specified size of the stream) are fetched from offset 5000 in *urn:aff4:83a3d6db-85d5*.

In order to efficiently express periodic maps such as those found in RAID arrays, the Map stream may be provided with two optional parameters: a *target_period* (T_p), and *stream_period* (S_p). If specified, the above relation becomes:

$$p := floor\left(\frac{x}{S_p}\right)$$
$$x' := mod(x, S_p)$$
$$y := (x' - X_i) + Y_i + p \times T_p$$

Where *mod* is the modulus function and *floor* signifies integer division. For example consider Listing 3, which corresponds to a 3 disk RAID-5 array.

6.3. The HTTP stream

Arguably the most ubiquitous protocol for information sharing is the HTTP protocol (Fielding et al., 1999). The protocol features mature authentication and auditing and is fast and easy to set up with numerous web server implementations available on the market. The HTTP protocol is also designed to operate across a wide range of network architectures and is therefore more deployable than traditional file sharing protocols.

For these reasons it is desirable to allow the HTTP protocol to be used in facilitating the sharing of evidence files between investigators. Luckily, the HTTP protocol fits naturally within the URN based scheme adopted by AFF4, since the HTTP Universal Resource Locator (URL) scheme is a subset of the URN scheme.

For this reason, URLs may be used interchangeably with a URN within the AFF4 universe. For example, the *aff4:stored* attribute of a volume may be specified as a URL (e.g. *http://intranet/123453/*). AFF4 provides transparent support for HTTP and FTP URLs by means of the Curl HTTP library (Various, 2009). The HTTP Stream, therefore satisfies read requests by making HTTP requests to the web server. We use the *Content-Range* HTTP header to request exactly the byte range the client is interested in. This allows efficient network transport as we do not need to download unnecessary data, we just request those chunks the client application requires.

Our implementation also enables direct writing to a HTTP URL using the WebDav extensions to HTTP (Goland et al., 1999). The HTTP stream also supports the File Transfer Protocol (FTP) and HTTPS (Secure Sockets Layer—SSL) protocols transparently, as provided by the Curl library.

6.4. Encrypted streams

Encryption is an important property in an evidence file format. In particular, multiple streams may be present in the file set, and often different access levels are desired. For example, for evidence set containing both network captures and disk images it may be desirable to limit access to streams based on legal authorizations, even though the same set is distributed to a number of people.

```
0,0,urn:aff4:83a3d6db-85d5
4096,10000,urn:aff4:f901be8e-d4b2
8192,5000,urn:aff4:83a3d6db-85d5
```

Although the Zip64 standard specifies encryption, it is not suitable for our purposes since it encrypts each segment separately, and does not specify a sufficiently flexible scheme (e.g. support for PKI or PGP keys). Segment based encryption may lead to information leakage when segments are compressed, as the uncompressed size of the segment may be deduced.

AFF4 therefore introduces a new encryption scheme, the Encrypted Stream. The Encrypted Stream provides transparent encryption and decryption onto a single target stream. The target stream actually stores the encrypted data, and read requests from the stream are satisfied by decrypting the relevant data from this backing stream. The encrypted stream itself does not store any data at all—all data is stored on its target stream.

The Encrypted Stream may contain any data at all, including disk images, network captures or memory images. It is useful however, to store an entire AFF4 volume within the Encrypted stream. This provides block level encryption for the contained AFF4 volume (which might contain arbitrary streams). This approach is illustrated in Fig. 5.

The result is that a number of AFF4 volumes are used as *Container Volumes* to provide storage for Encrypted Streams. The main *Embedded Volume*, which actually contains data is stored within the Encrypted Stream, effectively distributed throughout the container volumes. Note that the outer Volume may contain several Encrypted Streams and therefore contain multiple AFF4 Encrypted Volumes. Container Volumes may contain non-encrypted streams as well, and may implement different encryption schemes and keys for each Encrypted stream. This effectively allows arbitrary access policies to be implemented as only volumes which can be accessed can be read.

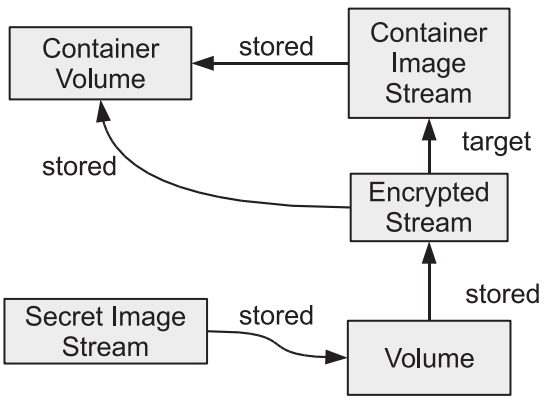

Fig. 5 – Embedding an encrypted AFF4 volume within an Encrypted Stream. The container volume contains an encrypted stream backed by an image stream which is also stored in the container. Once the encrypted stream is opened, the volume stored on its image stream is accessible. Now it is possible to see the secret image stream stored within the volume.

6.5. The link object

Although the URN of a stream names it unambiguously in the AFF4 universe it is difficult to use and communicate due to its random nature. Most investigators would prefer to use a shorter name which might well represent the image better in their minds (e.g. a case name or warrant number).

A Link object has a *aff4:target* attribute. When the Link object is opened, the object named by this attribute is returned. This allows images with complex names to be referred to via short, meaningful names. In practice both Image Streams and Link Objects are automatically created by imaging tools, so users can always refer to the Image stream via the simplified Link name.

7. Identity object

AFF4 defines a *Statement* as a collection of relations, or (subject, attribute, value) tuples. Listing 4 illustrates a collection of relations encoded in the standard AFF4 notation (SHA254 hashes are base64 encoded).

The statement expresses a set of attributes of other AFF4 objects, and in particular the attribute of SHA256 hash is expressed (but other attributes may also be expressed).

Digital signatures have been used in previous forensic file formats (such as AFF1) to provide authentication and non-repudiation of forensic evidence. In essence, a when a person signs an object they are vouching for its authenticity. Similarly, when a person signs a *Statement*, they are vouching for its authenticity. This concept is similar to the Bill of Material (BOM) from AFF1.

An AFF4 Identity object represents an entity, currently described by way of an X509 certificate. The URN of an identity object is the certificate's fingerprint, and is therefore unique to the certificate. Identity objects contain *aff4:statement* attributes which refer to AFF4 streams containing statements. The identity object also contains a copy of the certificate used to sign the statements.

Listing 3

A Map stream that corresponds to a 3 disk RAID-5 array. The targets are URNs for the respective disks. Note that map coordinates are given in multiples of block size.

```
aff4:block_size = 64 k
aff4:stream_period = 6
aff4:target_period = 3
0,0,disk1
1,0,disk0
2,1,disk2
3,1,disk1
4,2,disk0
5,2,disk2
```

To verify the signatures, the AFF4 library loads the stored certificate, then checks the signature for each statement. If a statement is verified (i.e. deemed as correct according to the identity), the relations within it are checked. Note that it is possible for multiple identities to sign the same data.

8. Usage scenarios

In this section we describe how AFF4 may be used in various situations. Since the AFF4 framework implements a distributed evidence management system, we demonstrate its use by a fictitious multinational corporation with offices in Los Angeles and New York. Each office has its own computer forensics lab and is connected via a WAN.

8.1. Using distributed evidence

An investigation is conducted by the New York team. The case relates to a hard disk Image stream stored inside a volume, in turn stored on the NY evidence server at URL http://ny.wan/evidence1.aff4. The team requires an analyst (Bob) in LA to assist with their analysis. The LA analyst types[2]: This command causes the local AFF4 implementation to:

1) Contact the universal resolver asking where *"NY case 1"* is stored.
2) The universal resolver replies that it is a symbolic link to a stream called *"urn:aff4:1234"* stored within the volume *"urn:aff4:9876"*. Further queries reveal that the volume is located at *http://ny.wan/evidence1.aff4*.
3) The local AFF4 library then directly accesses the volume at the given URL. Note that the entire volume is not copied, instead specific chunks are retrieved on an as needed basis.

The overall effect is that the user in LA is able to directly access the disk image specified using a friendly name, and stored at a remote location easily.

8.2. Load redistribution

In the previous scenario, Bob becomes involved in this case, and wishes to download the entire image locally to *http://la.wan/evidence1.aff4*. The Universal Resolver now has two possible locations for the same volume URN, since there are two copies in existence. Based on predetermined distance metrics, the resolver directs requests from Bob to the LA copy, while Alice is redirected to the NY copy. This load redistribution can be used for optimal management of evidence storage in a transparent way. Analysts are not aware of where the evidence is physically stored, and it appears as though all evidence is always available.

If Alice's local NY copy is now lost, Alice's local AFF4 library will fail to open the NY URL, and will automatically fall back to the copy stored in LA. This will require access across the WAN,

[2] Fls is the file listing command which is part of the Sleuthkit.

Listing 4

An Example Statement (URNs and hashes are shortened for illustration).

```
urn:aff4:34a62f06/00000 aff4:sha256 = +Xf4i....7rPCgo =
urn:aff4:34a62f06/00000.idx aff4:sha256 = ptV7xOK6....C7R6Xs =
urn:aff4:34a62f06/properties aff4:sha256 = yoZ....YMtk =
urn:aff4:34a62f06 aff4:sha256 = udajC5C...BVii7psU =
```

which will be slower, but provides a kind of distributed fail over capability.

8.3. *Remote imaging*

The NY IT security team has just responded to an incident on one of their servers. Alice, the responding officer, wishes to image the server. She types:

This command requests an image be created directly on the evidence server (it will be uploaded using WebDav). The image is signed using Alice's certificate and key (which might need to be unlocked). Note that Alice does not need any hardware to obtain the image as it is done over the network—she therefore can respond rapidly.

Bob is an analyst in LA which specializes in filesystem analysis. As soon as the acquisition is complete, the image is available for Bob to examine. Bob does not have permissions to create volumes on the NY evidence server, so he types:

This creates a new volume on the LA server which contains a set of Map streams referring to the original evidence. The new volume is near zero cost but refers to the original image (which is still stored in NY).

8.4. *Rapidly converting a set of DD images*

Many hardware devices are available to acquire hard disks in the field. These often produce a set of uncompressed images split at a certain size. It is possible to construct a Map Stream which seamlessly reassembles the logical image from all the individual disk images. The map stream may be kept in its own volume, or appended to one (or all) of the image fragments.

Similarly, each component can be compressed independently into its own stream. A single map stream can then be produced to combine all the component streams into a single logical stream. This approach can take advantage of multiple systems to actually do the compression in parallel as each component is compressed independently.

```
fls -i aff4 ''NY case 1''
```

8.5. *Acquisition of RAID disks*

Often disks in a system are grouped into RAID devices, commonly RAID-5 or RAID-0. Previously, if disks were acquired independently, they would need to be analyzed using a tool which was able to reassemble RAID devices.

With the AFF4 format, each of the disks can be acquired as a separate Image Stream. Finally a tool such as PyFlag (Cohen, 2008c) may be used to deduce the RAID map, which can be appended to the AFF4 file as a Map Stream. This Map Stream can then be opened by any tool to get a logical view of the RAID, without the tool needing to have explicit support for RAID reassembling. This approach enables parallel acquisition of RAID drives, a feature long desired to handle the vast quantities of data presented by RAID.

8.6. *Cryptographic management of evidence*

An AFF4 archive may hold multiple encrypted volumes, each in its own Encrypted Stream. Each of those streams is encrypted using a different master key, and therefore can have different passphrases, and can be assigned to different users by encrypting the master key with different X509 certificates. It is also possible for users to create non-encrypted volumes within the AFF4 volume.

This can be used to enforce access controls in line with current legislative requirements. For example, within the same investigation different material is often obtained under different warrants (e.g. wiretap authorizations are different from search warrants). Therefore, different investigators and analysts need different access to the different streams. However, the analysts may still store the results of their analysis in an un-encrypted form, or assign others permissions to decrypt their analysis results, without providing access to the underlying data.

This can be used in sharing meta data (e.g. Map Streams of files of interest) between analysts, without needing to provide access to the underlying data.

```
aff4imager -i -o http://ny.wan/evidence2.aff4 \
   -k http://ny.wan/alice.key \
   -c http://ny.wan/alice.crt/dev/sda
```

```
fsbuilder -o http://la.wan/evidence3.aff4 \
   http://ny.wan/evidence2.aff4
```

8.7. *Logical file acquisition*

Alice is responding to an incident on a critical corporate server. Since the system cannot be taken down for forensic imaging, Alice must resort to acquiring discrete files instead. Alice is unable to install and run any acquisition software on the server due to policy restrictions.

It is still advantageous in this case to bring discrete files into the AFF4 evidence universe by acquiring each evidence file using a unique URN. As explained in Section 3, segments are AFF4 stream objects which are implemented by storing the data in Zip archive members. It follows, therefore, that a regular zip file containing files is also a valid AFF4 volume.

So a logical image of files, can be created by any regular Zip compression program in the field. Once brought into the lab these volumes are given a volume URN and imported into the Universal Resolver to provide access to all the files within the archive. At this stage digital signatures can also be added for each logical file.

Alice uses windows explorer to obtain a Zip file of the files of interest. After taking the archive back to the lab, she then signs the files, and adds a volume URN, making the Zip file a fully compliant AFF4 volume.

9. Conclusion and future work

This paper describes a significant enhancement to the Advanced Forensic Format (AFF1). AFF4, extends beyond a file format to describe a universal framework for evidence management, offering significant new features such as the ability to store multiple kinds of evidence from multiple devices in a single archive, and an improved separation between the underlying storage mechanism and forensic software that makes use of evidence stored using AFF. This improved system allows a single archive of evidence to be used in a plethora of modalities, including in a single evidence file, multiple evidence files stored on multiple workstations, and evidence stored in a relational database or object management system—all without making changes to forensic software.

We have developed an open source reference implementation, but the AFF4 framework is simple enough for competing implementations. We hope this simplicity enhances AFF4's acceptance and adoption as a standard evidence management platform.

REFERENCES

B.S. NTI Forensics Source. Safeback bit stream backup software [Online]. Available: http://www.forensics-intl.com/safeback.html; 2008.

Berners-Lee T, Masinter L, McCahill M. RFC 1738: uniform resource locators (URL), Dec. 1994, updated by RFC1808, RFC2368 [23][24]. Status: PROPOSED STANDARD.

Carrier BD, Spafford EH. ''An event-based digital forensic investigation framework'', in digital forensics research workshop [Online]. Available: http://www.digital-evidence.org/papers/dfrws_event.pdf; 2004.

Cohen M. Advanced carving techniques. Digital Investigation 2007;4(3–4):119–28.

Cohen MI. Pyflag: an advanced network forensic framework [Online]. Available:. In: Proceedings of the 2008 digital forensics research workshop. DFRWS http://www.pyflag.org; Aug. 2008.

Cohen MI. Advanced jpeg carving. In: e-Forensics '08: proceedings of the 1st international conference on forensic applications and techniques in telecommunications, information, and multimedia and workshop. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering); 2008. p. 1–6.

Cohen M. How to load a raid set into pyflag [Online]. Available: http://www.pyflag.net/cgi-bin/moin.cgi/RaidHowTo; 2008.

Fielding R, RFC 1808: Relative uniform resource locators, Jun. 1995, updates RFC1738 [13]. Updated by RFC2368 [24]. Status: PROPOSED STANDARD.

Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L. RFC 2616: hypertext transfer protocol – HTTP/1.1 [Online]. Available: http://www.w3.org/Protocols/rfc2616/rfc2616.html; June 1999.

Garfinkel SL. Providing cryptographic security and evidentiary chain-of-custody with the advanced forensic format, library, and tools. The International Journal of Digital Crime and Forensics 2009;1.

Garfinkel SL, 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.

Goland Y, Faizi EWA, Carter S, Jensen D. RFC 2518: HTTP extensions for distributed authoring – WEBDAV [Online]. Available: http://www.ietf.org/rfc/rfc2518.txt; February 1999.

Guidance Software, Inc.. EnCase forensic [Online]. Available: http://www.guidancesoftware.com/products/ef_index.asp; 2007.

Hoffman P, Masinter L, Zawinski J, RFC 2368: the mailto URL scheme, Jul. 1998, updates RFC1738, RFC1808 [13], [23]. Status: PROPOSED STANDARD.

Ilook investigator [Online]. Available: http://ilook-forensics.org; 2008.

Katz P. APPNOTE.TXT — .ZIP file format specification [Online]. Available:. PKWare, Inc., Tech. Rep. http://www.pkware.com/documents/casestudies/APPNOTE.TXT; Sep. 28 2007.

Keightley R. EnCase version 3.0 manual revision 3.18 [Online]. Available: http://www.guidancesoftware.com; 2003.

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

Leach P, Mealling M, Salz R. RFC 4122: a universally unique identifier (UUID) URN namespace. standards Track. [Online]. Available: http://www.ietf.org/rfc/rfc4122.txt; Jul. 2005.

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

Sollins K, Masinter L, RFC 1737: Functional requirements for uniform resource names, Dec. 1994, status: INFORMATIONAL.

Tanenbaum AS. Modern operating systems. Prentice Hall; 2008.

Various. libcurl – the multiprotocol file transfer library [Online]. Available: http://curl.haxx.se/libcurl; March 2009.

Zero storage carving; Nov. 2006.

Michael Cohen is a data specialist for the Australian Federal Police in Brisbane, Australia. Michael received his Ph.D. from the Australian National University in 2001 in the field of Semiconductor Physics, but has been working in the field of

Digital Forensics and Information Security since. His research interests include digital forensics, network forensics and programming, especially in Python

Simson L. Garfinkel is an Associate Professor at the Naval Postgraduate School in Monterey, California, and an associate of the School of Engineering and Applied Sciences at Harvard University. His research interests include computer forensics, the emerging field of usability and security, personal information management, privacy, information policy and terrorism. This article does not necessarily represent the view of the US Government or the US Department of Defense.

Bradley Schatz is an adjunct Associate Professor at the Queensland University of Technology (QUT) and the director of computer forensics firm, Schatz Forensic. Dr. Schatz divides his time between providing forensic services primarily to the legal sector and researching and educating in the area of computer forensics. His research focus is in the areas of volatile memory, evidence scalability, and information integration. Bradley received a bachelor's degree in Computer Science from the University of Queensland in 1995 and a Ph.D. in Computer Forensics from QUT in 2007. His early years in computing were spent practicing software engineering and network security.