AFF4-L: A Scalable Open Logical Evidence Container

By
Dr. Bradley L. Schatz

From the proceedings of
The Digital Forensic Research Conference
DFRWS 2019 USA
Portland, OR (July 15th - 19th)

DFRWS is dedicated to the sharing of knowledge and ideas about digital forensics research. Ever since it organized the first open workshop devoted to digital forensics in 2001, DFRWS continues to bring academics and practitioners together in an informal environment.
As a non-profit, volunteer organization, DFRWS sponsors technical working groups, annual conferences and challenges to help drive the direction of research and development.

https://dfrws.org

Digital Investigation 29 (2019) S143–S149

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2019 USA — Proceedings of the Nineteenth Annual DFRWS USA

AFF4-L: A Scalable Open Logical Evidence Container

Dr Bradley L. Schatz

Schatz Forensic, Brisbane, Australia

ARTICLE INFO

Article history:

Keywords:
Logical image
AFF4
Deduplication

ABSTRACT

With the proliferation of cloud-based evidence and locked down physical storage logical imaging is increasingly necessary in digital forensics. In practice closed formats are commonly used, however they lack extensibility and expressiveness, are poorly defined, and suffer from limited interoperability. This work proposes and implements an open logical imaging format based on the AFF4 evidence container, supporting scalable arbitrary metadata storage and deduplicated logical image storage.
© 2019 The Author(s). Published by Elsevier Ltd on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

The concept of logical imaging has been adopted for a range of purposes in digital forensic practice, ranging from sharing of subsets of evidential files post extraction from traditional physical forensic images, to preservation of file subsets during triage. In the mobile device forensics space (and especially iDevices) logical images are widely used due to physical access to block level storage being locked down. Finally, with triage's focus on rapid identification and preservation of sets of files, logical imaging is widely used.

A logical image is commonly understood to be a collection of bytestream copies of one or more suspect files, associated file metadata (including path), and file integrity information (typically bitstream hashes). In 2018, common logical image formats in use are the Encase L01 and Lx01, Access Data AD1, and Cellebrite's format. These are all vendor specific formats and lack formal specifications. Beyond these, a range of ad-hoc hybrid schemes based on existing archive formats such as tgz, cpio, and zip are also used.

Recent changes to the computing landscape increasingly motivate the usage of logical imaging. These include the following.

1.1. Locked & encrypted physical storage

With first iOS devices, and now the recent generation of Mac computers with the T2 Security Chip and FileVault2 enabled, physical imaging has become is increasingly impractical with the resulting physical images generally unusable due to a lack of access to key material for decryption. File level access to a multi-versioned filesystem is however still available from user space, assuming access.

1.2. Cloud/SaaS evidence

In the cloud Software-as-as-Service (SaaS) environment, physical imaging is in general not feasible. Forensically relevant data sources are regularly byte-oriented streams that can only be accessed by web service calls. Logical imaging matches well with forensic preservation of these sources.

1.3. Streamed software distribution

With the advent of SaaS and regular patching software is increasingly distributed online to endpoints, with traditional “distributions” of software being less easy to define. Examples include Windows Update and games deployed on the Steam platform. Curating collections of such software requires regular snapshots, however employing regular physical disk images for such snapshots is impractical due to the granularity of such images. The space requirements of such snapshots quickly multiply when undertaken regularly, which motivates the usage of deduplicated storage techniques.

1.4. Contribution

This paper proposes an openly specified logical evidence container based on the AFF4 evidence container (Cohen and Garfinkel, 2009), introducing the concept of deduplication into logical imaging. The proposed container provides the same logical imaging primitives as existing approaches, with the advantages of deduplicated content storage, extensibility and expressiveness via

E-mail address: bradley@schatzforensic.com.

https://doi.org/10.1016/j.diin.2019.04.016
1742-2876/© 2019 The Author(s). Published by Elsevier Ltd on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

arbitrary metadata storage, along with a freely available implementation.

The challenges involved in implementing the proposed format are discussed, validation and testing of the implementation described and a scalability challenge identified. This challenge applies not only to the proposed approach, but also to other forensic representation approaches. A solution to the challenge is proposed and evaluated.

2. Related work

2.1. Linked data in the representation of forensic information

The usage of a linked data model for representing forensic related information within and between forensic evidence containers was originally proposed in Secure Digital Evidence Bags (Schatz and Clark, 2006) and subsequently refined in the Advanced Forensic Format v4 (AFF4) (Cohen and Garfinkel, 2009). Both approaches use the Resource Description Framework (RDF) for describing forensically relevant information in a machine and human readable format.

Information is represented in RDF as objects, properties and values of properties (referred to collectively as an *RDF triples*). Relationships between objects are described by naming each object uniquely - for example by using a Universal Resource Locator (URL). Information represented using the RDF model can be expressed in a number of text-based languages, including Turtle and JSON-LD.

The AFF4 is narrowly focused on providing an evidence container for storing arbitrary forensic information, eschewing formal ontology specification by relying on a simple lexicon.

The wider ontological challenge of representing and exchanging forensic and cyber investigation related information has taken up by the CASE/UCO effort (Casey et. al, 2017). Like AFF4, it relies on a linked data model based on RDF, while its concerns are in general orthogonal to those of the AFF4.

2.2. Logical forensic imaging

The Encase Logical Evidence File (LEF), which is also widely known as an L01 and Lx01 have been reverse engineered by Metz (2019) and an implementation in the C language was produced.

Bjelland has made available a Python-based AD1 parser (Bjelland, 2019). Write support is not provided.

The original AFF4 forensic image container proposal contemplated supporting logical imaging, and the PMEM volatile memory imager (Cohen, 2019) pioneered logical acquisition of memory-mapped files (amongst others) associated with physical memory images.

2.3. Deduplicated storage

In the wider Internet, deduplicated data representation (also known as Content Based Addressing) is widely used and relied on in products such as Rsync, BitTorrent, Git, and Dropbox, in filesystems such as ZFS, and in operating systems such as Windows Server.

In the context of digital forensics, Teleporter (Watkins et al., 2009) proposed using the hashes of known files to represent file content for efficient transmission of physical forensic images, and AFF4 Hash Based Imaging (Cohen and Schatz, 2010) proposed using hashes of fixed size data blocks for improving the efficiency of physical forensic imaging.

3. Extensions to the AFF4 for logical imaging

The proposed logical image container formalizes the existing work towards AFF4 logical imaging and builds on the existing abstractions provided by the AFF4 Standard v1.0 (Schatz and Cohen, 2017). The proposal described in this paper differs from the prior work in that it focuses on:

- Efficient storage of file content;
- Enabling human readability and browsability of logical image containers using existing ZIP tools (supporting as much as possible a 1:1 mapping between file appearance in-filesystem and in-container).
- Definition of comprehensive lexicon representing the filesystem tree relationships between acquired logical files, and folders.

3.1. Formalizing resource naming

In the AFF4 architecture, all evidence objects are identified using an Resource Description Format (RDF) compatible resource identifier. The AFF4 initially employed Universal Resource Names (URN's) before shifting to a bespoke Universal Resource Identifiers (URI) scheme beginning with "`aff4://`". This was due to constraints in encoding file paths in way that complied with Universal Resource Name (URN) standards while at the same time working with overly strict RDF parser implementations.

In the intervening years between the initial proposal of the AFF4 and today, the RDF standard has evolved to support internationalization by relying on the Internationalized Resource Identifiers (IRI) standard for resource identifiers.

We propose formalizing the AFF4 resource naming scheme in the following way. We call such an IRI an AFF4 Resource Name (ARN) to distinguish it from more generic identifier schemes. AFF4 Objects named based on GUID's are named using an Internationalized Resource Identifier (IRI) rather than a URI with the existing scheme "`aff4://`".

A GUID based ARN in constructed as follows:

```
AFF4-ARN = "aff4://" object-guid-part [ "/"
host "/" path ]
```

The object GUID part follows the UUID standard. The host and path part may contain any Unicode character that is not forbidden in the IRI specification ("/" is used as a path delimiter). Forbidden printable characters must be percent encoded per the standard URL encoding scheme. The forbidden printable characters are:

<>\^`{|}

A key difference between this new scheme and the old is that Unicode characters from other languages are now allowed, and the path part is more expressive, allowing characters which may take on special roles in a URL (such as = and ?) to be used in the path part with impunity.

3.2. Suspect path to resource name mapping

The proposed approach aims to represent as succinctly as possible the path of suspect files in the ARN of the preserved object. However, native OS paths may contain characters that are illegal in an IRI (and onwards an ARN). For example, HFSPlus will allow any Unicode character, including NUL.

Accordingly, we define an encoding function from suspect file paths to AFF4 Resource Names. We adapt rules generally adopted

Table 1
Suspect path to ARN mappings.

OS Path	AFF4 Resource Name
c:	aff4://e6bae91b-14d231833e18//c:
c:\	aff4://e6bae91b-14d231833e18//c:/
c:\foo	aff4://e6bae91b-14d231833e18//c:/foo
\\bar\c$	aff4://e6bae91b-14d231833e18/bar/c$
\\bar\c$\foo\ ネコ.txt	aff4://e6bae91b-14d231833e18/bar/c$/foo/ネコ.txt
/foo/bar	aff4://e6bae91b-14d231833e18//foo/bar
/foo/some file	aff4://e6bae91b-14d231833e18//foo/some%20file

Table 2
AFF4 Resource name to Zip Segment Name mapping.

AFF4 Resource Name	Zip segment name
aff4://e6bae91b-14d231833e18//c:	/C:
aff4://e6bae91b-14d231833e18//c:/	/C:/
aff4://e6bae91b-14d231833e18//c:/foo	/C:/foo
aff4://e6bae91b-14d231833e18/bar/c$	bar/c$/foo
aff4://e6bae91b-14d231833e18/bar/c$/foo/ネ コ.txt	bar/c$/foo/ネ コ.txt
aff4://e6bae91b-14d231833e18//foo/bar	/foo/bar
aff4://e6bae91b-14d231833e18//foo/some%20file	/foo/some file

for the `file://`URI scheme, with the following rules and exceptions:

- Forward slashes delimit paths.
- Any control, space, percent and forbidden characters are percent encoded.
- Unicode printable characters outside of the ascii range are UTF-8 encoded and case sensitive.
- The host part of a UNC file URL is treated as a regular path component.

Table 1 summarizes valid mappings for a range of suspect path to ARN mappings. The ARN is constructed by taking the Volume ARN (the identifier of the AFF4 volume) and appending the path component per the above.

With reference to Table 1, note that non-UNC paths involve a double slash "//" to distinguish the absence of a host in the path.

It is anticipated that implementers might elect to re-use the existing OS and browser provided "`file://`" encoding functions as a basis for implementing this encoding, however such implementations are a moving target as implementers update behavior (Whatwg, 2017). We intend to build a comprehensive canonical path to ARN mapping table in support of identifying where using such implementations will produce non-compliant encodings, and to produce independent implementations of the encoding function without resorting to the OS provided encoding function.

In addition to the above mapping, we preserve the original file path name using the `aff4:originalFileName` property.

3.3. Storage of logical file bytestreams

The standard abstraction provided by the AFF4 for storing bytestreams is the Image Stream (a type of Compressed Block Stream). This abstraction was developed with a focus on supporting efficient seek-able compressed block storage for physical images. The PMEM work adapted the AFF4 approach to additionally store content as simple Zip Segments (zip files).

We propose a hybrid approach where the bytestream content of acquired logical files are stored using either approach based on their size (depicted in Fig. 1). Large files use an AFF4 Image Stream for efficient seekable compression, and for small files a native Zip Segment is used. We do the latter for reasons of efficiency – the Image Stream requires at least two Zip Segments and an extra layer of indirection for storage. In our prototype implementation we choose to store any bytestreams greater than 1M in size as Image Streams, and smaller as Zip Segments.

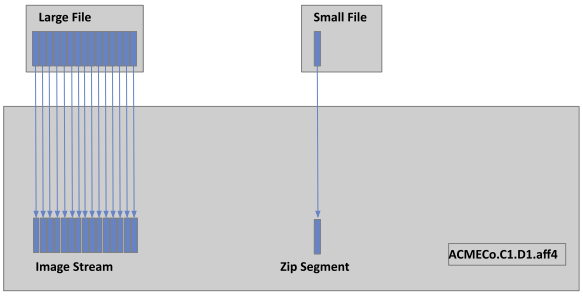

Fig. 1. The bytestream of logical images are stored as regular zip files. For larger images, Image Streams are used.

3.4. ARN to zip file segment mapping

Existing versions of the AFF4 defined a simple mapping rule for mapping resource identifiers to Zip Segments. For resource names that begin with a Volume GUID resource name, such as those shown in Table 2, the Volume resource name is removed, and the remainder is used as the Zip Segment name, after being URL percent encoded.

In order to enhance human viewability and retain Unicode filenames when browsing with standard Zip file browsers[1], we propose refining the encoding rules to the following:

- The Volume identifier part of the ARN is removed.
- The following "/" separator is removed.
- Any percent encoded spaces are converted back to spaces.

The resulting Zip Segment Name can then be used to locate relevant Zip resident data.

3.5. File metadata

File metadata is stored using the standard AFF4 methods as RDF. We define the lexicon properties and classes described in Table 3 to support representing the relevant metadata. We anticipate these properties are only the beginning of those that will be defined (for example the time of deletion may be a candidate).

3.6. Resource enumeration

Logical file consumer applications typically present the container of logical files images as a file system hierarchy. In order to do this, such applications require the ability to enumerate the logical files in the image and to identify the roots of trees of files and folders in the image.

This is supported in the following way. Producers of such images declare an `aff4:LogicalAcquisitionTask` object at the time of acquisition, and define an `aff4:filesystemRoot` property for each root folder or file that is acquired. For example, the RDF from a logical image of the `C:\Windows\System32\` folder and subfolders would include:

[1] For example WinRAR or 7-Zip.

Table 3
New AFF4 lexicon supporting logical imaging.

Lexicon item	Meaning
aff4:originalFileName	The original unencoded file path and name of a logical evidence object
aff4:birthTime	The birth time of a file's content and metadata.
aff4:lastWritten	The last modified time of a file's content.
aff4:recordChanged	The last modified time of a file's filesystem metadata
aff4:lastAccessed	The last access time of a file's content.
aff4:FileImage	Class representing a suspect file
aff4:Folder	Class representing a suspect folder
aff4:child	Property representing the FilesImages contained in a Folder
aff4:LogicalAcquisitionTask	Class representing a logical acquisition activity
aff4:filesystemRoot	Property pointing to a Folder or FileImage which forms the root of an acquisition operation.

```
_:1 rdf:type aff4:LogicalAcquisitionTask
      aff4:filesystemRoot <//c:/Windows/System32>

<//c:/Windows/System32>
      rdf:type aff4:Folder
      aff4:child …
```

Client applications may query the AFF4 RDF for nodes of type `aff4:LogicalAcquisitionTask` and follow links from there.

3.7. Integrity

We employed the standard AFF4 linear bitstream hash property (`aff4:hash`) to store the SHA1 and MD5 hashes of each logical file image. The block-based hashing approach of (Schatz, 2015) is compatible with this approach and may be employed by implementations.

3.8. Bringing it all together

With the prior in place, logical imaging of a small file involves the following steps:

- A new AFF4 volume is created;
  The suspect file path is encoded using the suspect path to ARN Path Fragment encoding rules;
  The ARN representing the file is created from the Volume GUID and the ARN Path Fragment;
- The ARN is given the `rdf:type` of `aff4:FileImage` and `aff4:Image`;
- File metadata is read and recorded against the ARN using the properties in Table 3;
- The ARN is encoded as a Zip Segment Name.
- The file content is stored to the AFF4 volume as a Zip Segment using the Zip Segment Name.
- The file hashes are calculated and stored as the `aff4:hash` property of the ARN.
- The class name `aff4:zip_segment` is added to the `rdf:type` list of the ARN to indicate that it is stored as a Zip Segment.

Appendix B contains an example of RDF representing a small logical file image produced per the above for the ARN `aff4://df2a4a50-84b7-4408-98ce-9cf99b91f070/test_images/AFF4-L/ネコ.txt`. The RDF representing a logical file image named `</test_images/AFF4-L/dream.txt>` is additionally visible.

4. Extensions to the AFF4 for deduplicated logical imaging

We adapt the deduplicated physical image approach of (Cohen and Schatz, 2010) towards logical file imaging, by defining an `aff4:Map` for each logical file image. A segmenting algorithm is applied to split the source file into fixed size chunks, and each chunk is hashed. The chunk hashes are used as unique addresses to identify the byte streams contained in the chunks.

The former approach uses the `sha1` chunk hashes to name Zip Segments for storing the byte streams of the associated chunks. This is inefficient in terms of storage, requiring a Zip File Header and Central Directory entry per unique chunk.

Our proposed approach introduces one layer of indirection. Block Hash ARNs represent the content of a fixed size unique byte stream (a chunk) that has a particular hash. The Map representing the logical file image builds its address space from references to Block Hash ARNs. Block Hash ARNs in turn are associated with a property which points to a byte range in a stream where its corresponding bytes are stored. Under this proposal the chunks are all stored in a single AFF4 Image Stream per acquisition session for efficiency. This structure is depicted in Fig. 2.

The Block Hash ARNs are constructed following the URN standard. The urlsafe base64 (Josefsson, 2006) encoded sha512 hash of the chunk is prefixed with "`aff4:sha512:`". Other hash algorithms may be simply substituted in this scheme in the future.

Using a standard AFF4 Map to refer to the corresponding byte streams in the Image Stream would cost two Zip Segments per Map. Given the potential number of these maps it is desirable to find a more efficient approach.

A new single-entry Map syntax was defined which we call Slice Maps. The syntax is inspired by the array slice syntax in the Python language and allows map storage directly in the RDF the AFF4 volume without the cost of further Zip Segments.

Appendix B shows an example of the RDF of such a Slice Map following the `rdf:dataStream` property. The slice refers to bytes

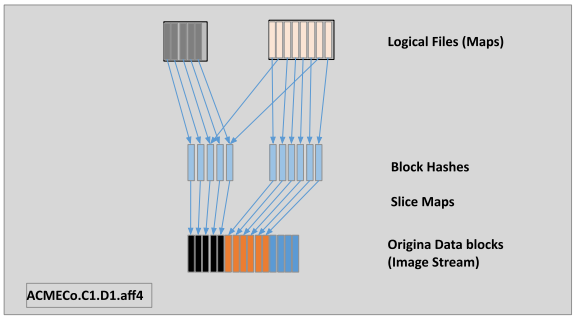

Fig. 2. Logical file image deduplication employs two-level maps and stores unique byte content in an Image Stream.

0 to 0x8000 of the `aff4:ImageStream` named `aff4://32f40158-4abe-48d5-9511-d92cbfa62fa9`. The meaning of the statement can be read as:

"The bytestream with name <`aff4:sha512:E67K3X8M9A_Ba4F6I_F948Cy7n25V2smtLWtAkGpC7ZLW0djC1YTBEpuAA4zcGESafhP--d9_tYUAVav74QcQA==`> and sha512 hash `E67K3X8M9A_Ba4F6I_F948Cy7n25V2smtLWtAkGpC7ZLW0djC1YTBEpuAA4zcGESafhP--d9_tYUAVav74QcQA==` is stored in bytes 0 to 0x8000 of the Image Stream named `aff4://32f40158-4abe-48d5-9511-d92cbfa62fa9`."

Appendix C shows a map entry which refers to this Hash ARN.

Unlike the former hash based disk imaging approach, our chunking algorithm deals with incomplete chunks found at the end portions of files which are not multiples of the chunk size. These are padded with NUL bytes to make a complete chunk.

4.1. Trust or verify

The usage of hashes alone to identify unique bytestreams leads to the possibility of a hash collision. While the usage of the SHA512 algorithm ensures this possibility is highly unlikely, we follow the approach of ZFS (Bronwick, 2009) in enabling hash matches to be either be trusted or verified by byte-by-byte comparison of the original bytestream with a hash matched chunk.

5. Evaluation

The above proposals were implemented in the open source pyAFF4 (AFF4, 2019) library and the implementation validated by creating standard logical and hash-based logical file images. Those images were then verified against the calculated and stored SHA1 and MD5 linear bitstream hashes. The logical images were found to be accurate copies of the original files.

The correct operation of deduplication was verified by creating images of source files with known common blocks and manually inspecting the internal AFF4 structures.

The scalability of the implementation was tested by undertaking deduplicated logical acquisitions of two sets of files. The sets were sourced from the System32 folder of a Windows 10 system and a Windows Server 2012 system.

Referring to Table 4, image A is a logical image of the files from the Server 2012 system, image B is a logical image of the files from the Windows 10 system, and image C is a logical image containing both sets of files. All images were created using deduplication.

The table summarizes the number of logical file images contained in each container, the number of RDF triples consumed in representing the file metadata and AFF4 metadata, and the container size. The column labeled "Initial access latency" records the amount of time it took to open the AFF4 logical image and begin accessing the image contents on the first access, with the value after the slash being the latency on subsequent accesses.

Notable in the results is the length of time it takes to open and access a logical image. For images A and B it is around 30–40s – a considerable time penalty to pay each time the logical image is opened.

We investigated the root cause of this and found it attributable to the cost of parsing the RDF from the logical image. In this case the parser being used was RDFlib (RDFLib, 2019), a pure python RDF parser implementation.

The implementation was then modified to employ a searchable compressed RDF serialization called HDT (Fernández et al., 2013). On load of an image for the first time the AFF4 RDF is read without parsing and an HDT serialization of the RDF is created directly from it and cached. The cached copy is used for all subsequent accesses. In this way, there is a conversion cost the first time the container is opened, but any subsequent accesses are saved that cost: the image contents are immediately accessible. It took around 3–4s each to generate an HDT serialization and index of images A and B, and around 9s for file set C, almost an order of magnitude faster.

6. Discussion

While the above evaluation would benefit from further data points, the findings related to the initial access latency have bearing on the use of RDF as a metadata representation for logical imaging. The initial access latency of the revised HDT approach provides acceptable first-time open performance for file corpuses in the tens of thousands of files. This is arguably sufficient for traditional logical imaging applications such as interchange of evidence extracted from physical images, and likely mobile device imaging. However, for applications such as logical acquisition of all files on a modern Windows system, where files can number in the millions, the initial access latency may be seen as problematic.

The implications of the above have wider bearing on current approaches to representation in forensics. The CASE/UCO effort aims to form a standard specification language for representing and exchanging forensic and cyber investigation related information. Like AFF4, it relies on a linked data model based on RDF, while its concerns are in general orthogonal to those of the AFF4. It's current prototype software implementation, the CASE-Python-API (CASE, 2019) relies on the same RDFlib library for reading and writing RDF, but uses the JSON-LD serialization of RDF for storage.

We converted the RDF metadata for image B (which is stored internally in AFF4 as an RDF language variant called Turtle) into a semantically equivalent JSON-LD serialization. We then timed the initial access latency for loading the JSON-LD serialization and compared it with that for the equivalent AFF4/Turtle. The latency was almost the same (38.7s).

Based on this we anticipate that the CASE effort will face similar concerns in instances where the information represented begins to reach hundreds of thousands of triples.

7. Future work

The scaling issues identified motivate further work in identifying how generalizable the observed issues are across other implementation approaches. Future work is required in evaluating the effect of ontology design approach on the resulting dataset size,

Table 4
Scalability testing results.

Img	Count Files	RDF Triples	Container size (GB)	Initial access latency RDFlib (s)	Initial access latency HDT (s)
A	19,463	228,287	1.5	33/33	3.8/0
B	21,835	236,235	1.9	39/39	4.4/0
C	41,298	461,220	3.4	67/67	8.9/0

as is work on avoiding or frontloading the cost of consuming such information. HDT appears to be a promising direction, however more efficient parsers may be another, as may different data storage approaches.

The two-level map proposal described in this paper may not be ideal for all circumstances. An alternative approach may adopt a one-level approach, constructing the file maps by direct reference to the byte ranges in the chunk storage stream. This may provide more efficient map storage and reads, with less efficient identification of storage corruption at the block level. The approach proposed was chosen primarily because it provides loose coupling between the file maps and the stored blocks, which we anticipate will be allow for repository and lifecycle level concerns such as garbage collection of unreferenced chunks. Future work will consider such alternate approaches.

The chunking algorithm used is not in parity with the current state of the art. Newer approaches employing Content-Defined Chunking (CDC) yield higher storage efficiency for a tradeoff in CPU overhead. Future work will explore the role of CDC approaches in comparison with the current chunking approach.

The proposal exploits the case sensitivity and multilingual support of ZIP and IRI approaches to support an evidence container that, as much as possible, continues to preserve the original file name using regular tools. The approach does present challenges in applying the approach to AFF4 directory-based volumes, which use a simple filesystem layout rather than a ZIP container. Different ARN to Segment mapping rules will be required to cope with variances between filesystems such as case-insensitivity and legal characters.

Finally, we anticipate that future work might include accelerating hash lookup using bloom filters and evaluating the operational effectiveness of AFF4 deduplication by testing against a variety of corpuses.

8. Conclusion

This paper proposes an openly specified logical evidence container based on the AFF4 evidence format, introducing the concept of deduplication into logical imaging. The proposed container provides the same logical imaging primitives as existing approaches, with the advantages of deduplicated content storage, improved extensibility and expressiveness, and possessing a freely available implementation, downloadable at https://github.com/aff4/pyaff4.

Acknowledgements

The author thanks NIST for their partial funding of this work, Michael Cohen for his critique of the revised AFF4 segment naming scheme, and the DFRWS reviewers for their helpful feedback.

Appendix A

```
@base <aff4://df2a4a50-84b7-4408-98ce-9cf99b91f070/> .
@prefix aff4: <http://aff4.org/Schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

</test_images/AFF4-L/dream.txt> a aff4:FileImage,
        aff4:Image,
        aff4:ImageStream ;
    aff4:birthTime "2018-09-17T13:42:20+10:00"^^xsd:datetime ;
    aff4:chunkSize 32768 ;
    aff4:chunksInSegment 1024 ;
    aff4:compressionMethod <http://code.google.com/p/snappy/> ;
    aff4:hash "75d83773f8d431a3ca91bfb8859e486d"^^aff4:MD5,
        "9ae1b46bead70c322eef7ac8bc36a8ea2055595c"^^aff4:SHA1 ;
    aff4:lastAccessed "2018-09-30T11:18:27+10:00"^^xsd:datetime ;
    aff4:lastWritten "2018-09-17T13:42:20+10:00"^^xsd:datetime ;
    aff4:originalFileName "./test_images/AFF4-L/dream.txt"^^xsd:string ;
    aff4:recordChanged "2018-09-17T13:42:20+10:00"^^xsd:datetime ;
    aff4:size 8688 ;
    aff4:stored <> .

</test_images/AFF4-L/ネコ.txt> a aff4:FileImage,
        aff4:Image,
        aff4:zip_segment ;
    aff4:birthTime "2018-09-18T15:49:51+10:00"^^xsd:datetime ;
    aff4:hash "d3b07384d113edec49eaa6238ad5ff00"^^aff4:MD5,
        "f1d2d2f924e986ac86fdf7b36c94bcdf32beec15"^^aff4:SHA1 ;
    aff4:lastAccessed "2018-09-30T11:18:34+10:00"^^xsd:datetime ;
    aff4:lastWritten "2018-09-18T15:49:51+10:00"^^xsd:datetime ;
    aff4:originalFileName "./test_images/AFF4-L/ネコ.txt"^^xsd:string ;
    aff4:recordChanged "2018-09-18T15:49:51+10:00"^^xsd:datetime ;
    aff4:size 4 ;
    aff4:stored <> .
```

Appendix B. Slice map example

```
<aff4:sha512:E67K3X8M9A_Ba4F6I_F948Cy7n25V2smtLWtAkGpC7ZLW0djC1YTBEpuAA4zcGESafhP--d9_tYUAVav74QcQA==>
        aff4:dataStream <aff4://32f40158-4abe-48d5-9511-d92cbfa62fa9[0x0:0x8000]> .
```

Appendix C. dream.txt map entry

```
[0x0,0x21F0] -> <aff4:sha512:E67K3X8M9A_Ba4F6I_F948Cy7n25V2smtLWtAkGpC7ZLW0djC1YTBEpuAA4zcGESafhP--
d9_tYUAVav74QcQA==>
```

References

AFF4, 2019. The Python implementation of the AFF4 standard. https://github.com/aff4/pyaff4. Feb 2019.

Bjelland, Petter, 2019. Pyad1. https://github.com/pcbje/pyad1. Feb 2019.

Bronwick, Jeff, 2009. ZFS deduplication. https://blogs.oracle.com/bonwick/zfs-deduplication-v2. (Accessed February 2019).

CASE, 2019. https://github.com/ucoProject/CASE-Python-API. Feb 2019.

Casey, et al., 2017. Advancing coordinated cyber-investigations and tool interoperability using a community developed specification language. Digit. Invest. https://doi.org/10.1016/j.diin.2017.08.002.

Cohen, Michael, 2019. PMEM. Feb 2019. https://github.com/Velocidex/c-aff4.

Cohen, Michael, Schatz, Bradley, 2010. Hash based disk imaging using AFF4. Digit. Invest. https://doi.org/10.1016/j.diin.2010.05.015.

Cohen, Garfinkel, Schatz, 2009. Extending the advanced forensic format to accommodate multiple data sources, logical evidence, arbitrary information and forensic workflow. Digit. Invest. https://doi.org/10.1016/j.diin.2009.06.010.

Fernández, Javier, et al., 2013. Binary RDF Representation for Publication and Exchange (HDT). Web Semantics: Science, Services and Agents on the World Wide Web, Elsevier.

Josefsson, S., 2006. The Base16, Base32, and Base63 data encodings. Feb 2019. https://tools.ietf.org/html/rfc4648.

Metz, Joachim, 2019. https://github.com/libyal/libewf/blob/master/documentation/Expert%20Witness%20Compression%20Format%20(EWF).asciidoc2. Feb 2019.

RDFLib, 2019. https://github.com/RDFLib/rdflib. Feb 2019.

Schatz, Bradley, 2015. Wirespeed: extending the AFF4 container format for scalable acquisition and live analysis. Digit. Invest. https://doi.org/10.1016/j.diin.2015.05.016.

Schatz, Bradley, Cohen, Michael, 2017. AFF4 Standard v1.0. https://github.com/aff4/Standard/blob/master/AFF4StandardSpecification-v1.0.pdf. Feb 2019.

Schatz, Clark, 2006. An Open Architecture for Digital Evidence Integration. In: AusCERT Asia Pacific Information Technology Security Conference : Refereed R&D Stream. Gold Coast, Queensland, pp. 21–26.

Watkins, K., McWhorte, M., Long, J., Hill, B., 2009. Teleporter: an analytically and forensically sound duplicate transfer system. Digit. Invest. 6, S43e7.

Whatwg, January 2017. URL living standard. https://url.spec.whatwg.org/.