Hash based disk imaging with AFF4

Dr Bradley Schatz

Adjunct Associate Professor, Queensland University of Technology

Director, Schatz Forensic Pty Ltd

* Dr Michael Cohen

Australian Federal Police

QUT isi Information Security Institute

Motivation

Current container formats only address storage and representation of a fraction of case related information

- **Data content:** *Single Streams,* Multiple streams (HPA, DCO), Hierarchical data relationships (Logical imaging), Addressing windows (RAM holes, bad sectors), Addressing schemes (Block size, CHS/LBA), SMART status
- **Physical characteristics:** Make, Model, Serial number, Interface (SATA, etc)
- **Context:** Environment the hard drive existed in, Case related information
- **Behaviour:** Error codes related to bad sectors

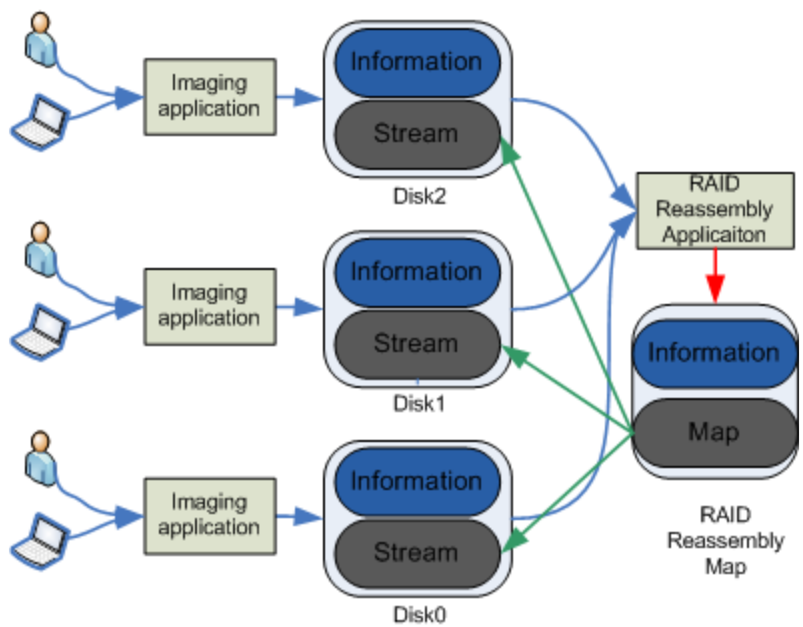
- **Efficiency**: *Storage space minimising, Random access performance, IO Bandwidth*
- **Authentication** : *Cryptographic signing, Hash storage*
- **Privacy** : Encryption, Redaction
- **Resilience**: Tolerance of underlying storage medium failure

Current tool interoperability is hampered by current container formats

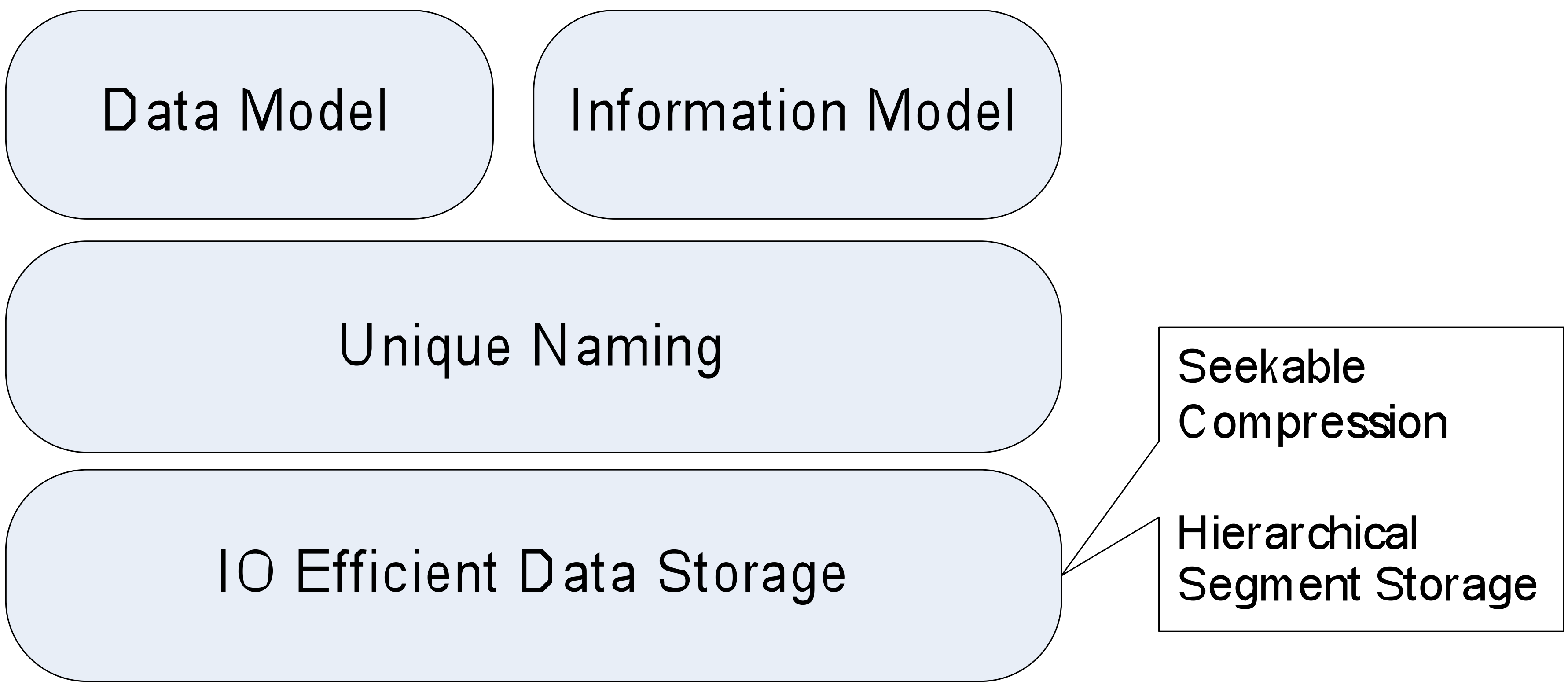
- Too much copying
- Not enough accessible information

Foundations

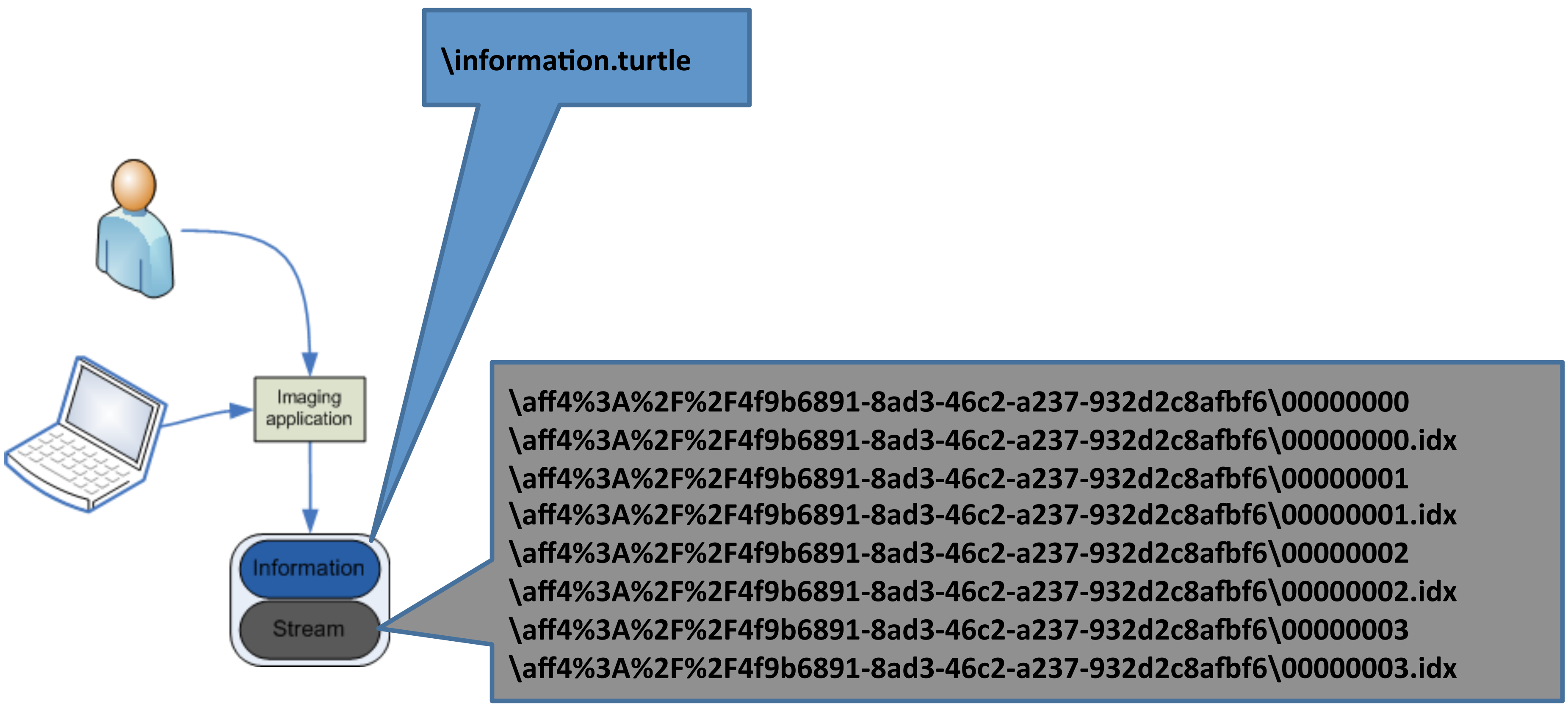
AFF4 is a new container format for storing digital evidence



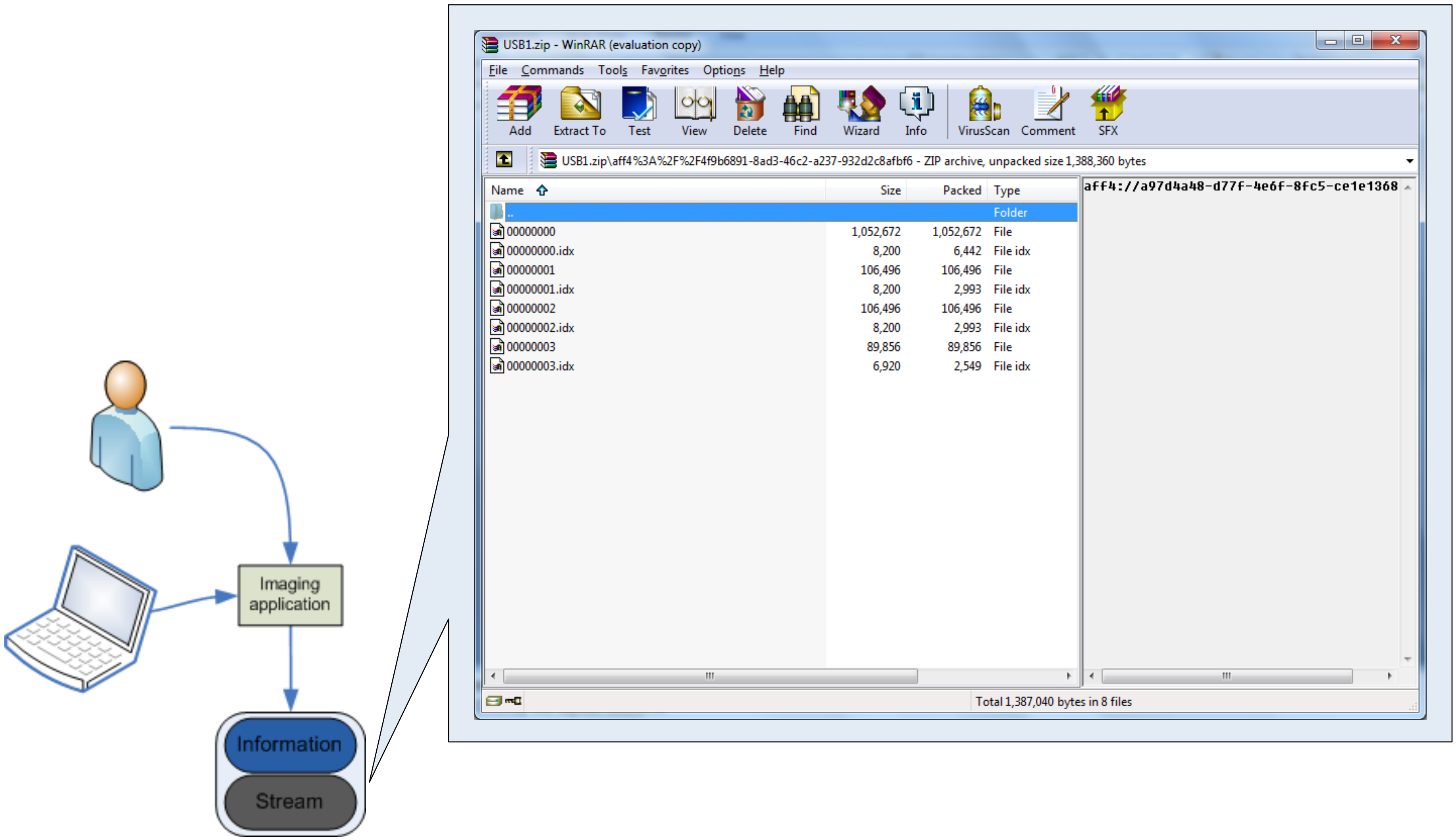
The AFF4 Data Storage Model



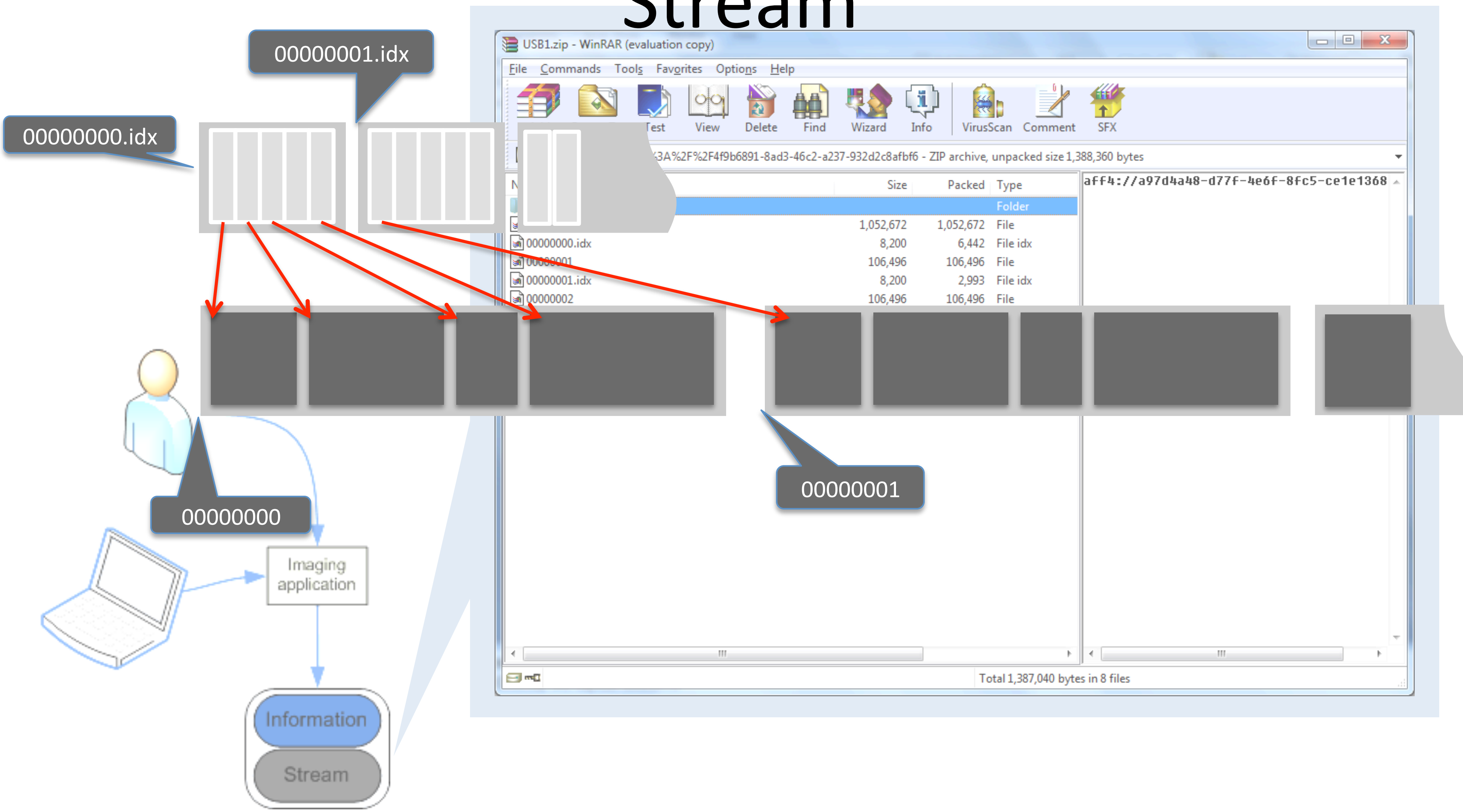
data and information are stored within a hierarchical storage container



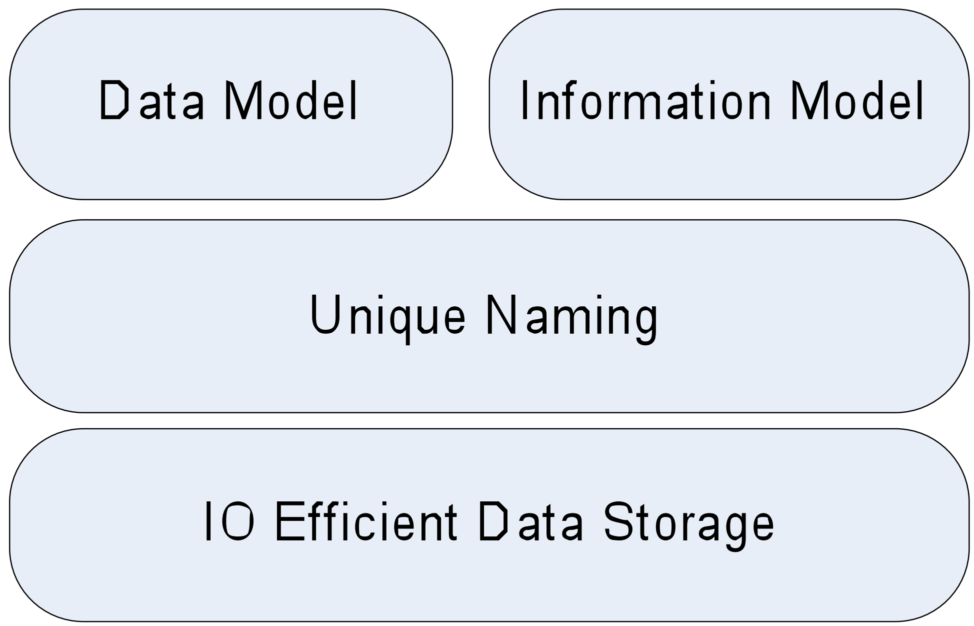
that storage container is a zip file



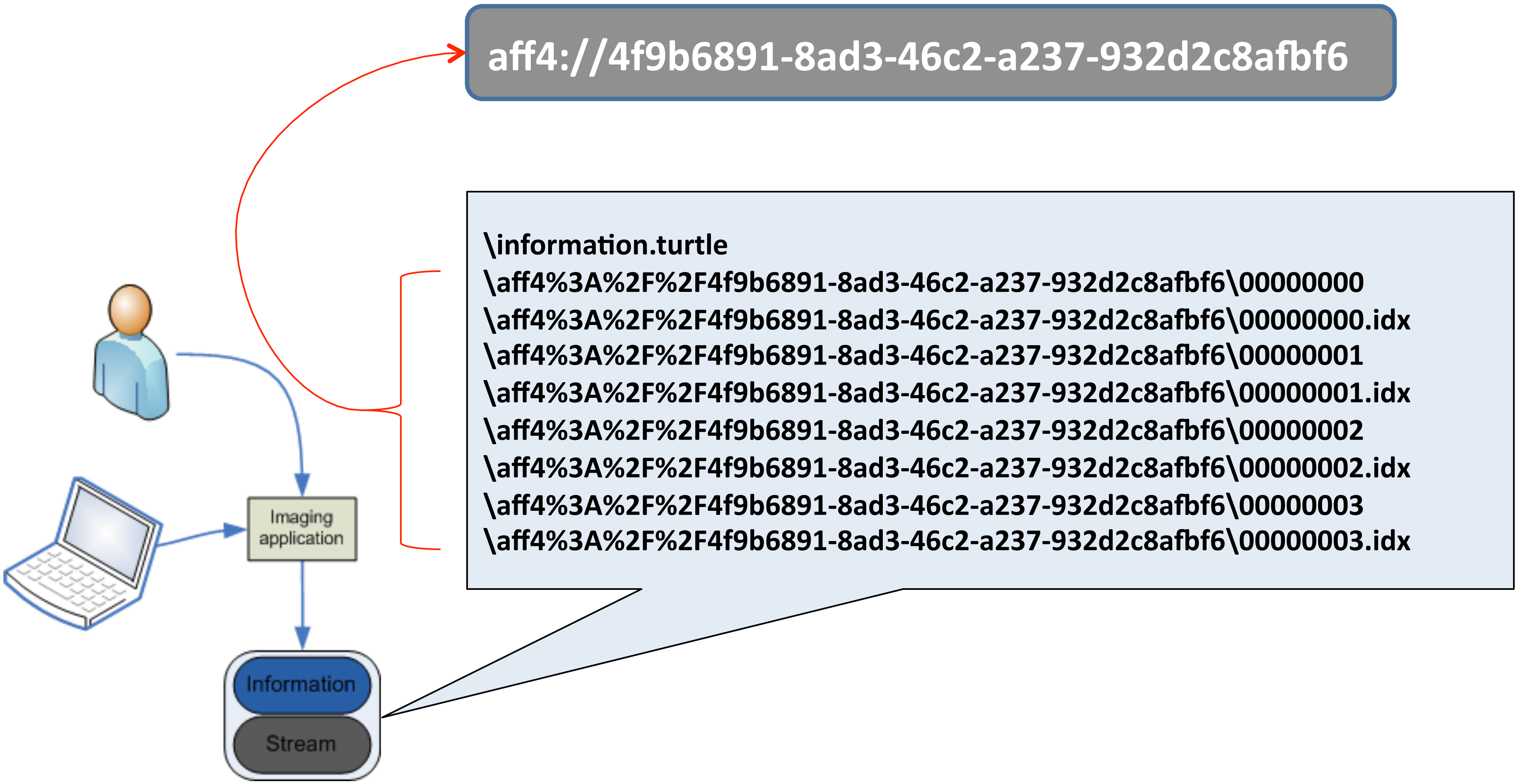
bitwise data is stored in an efficient seekable compressed form: the Stream



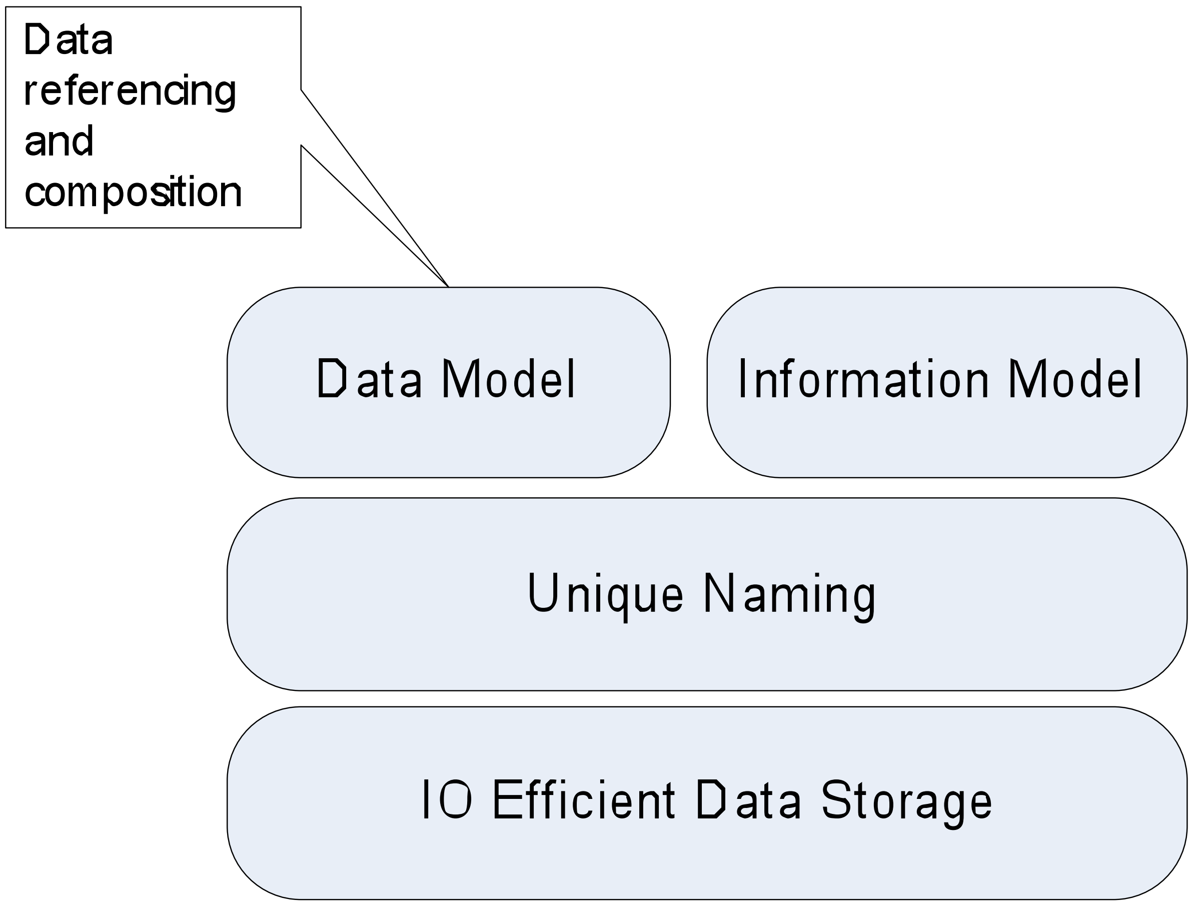
The AFF4 Naming Approach



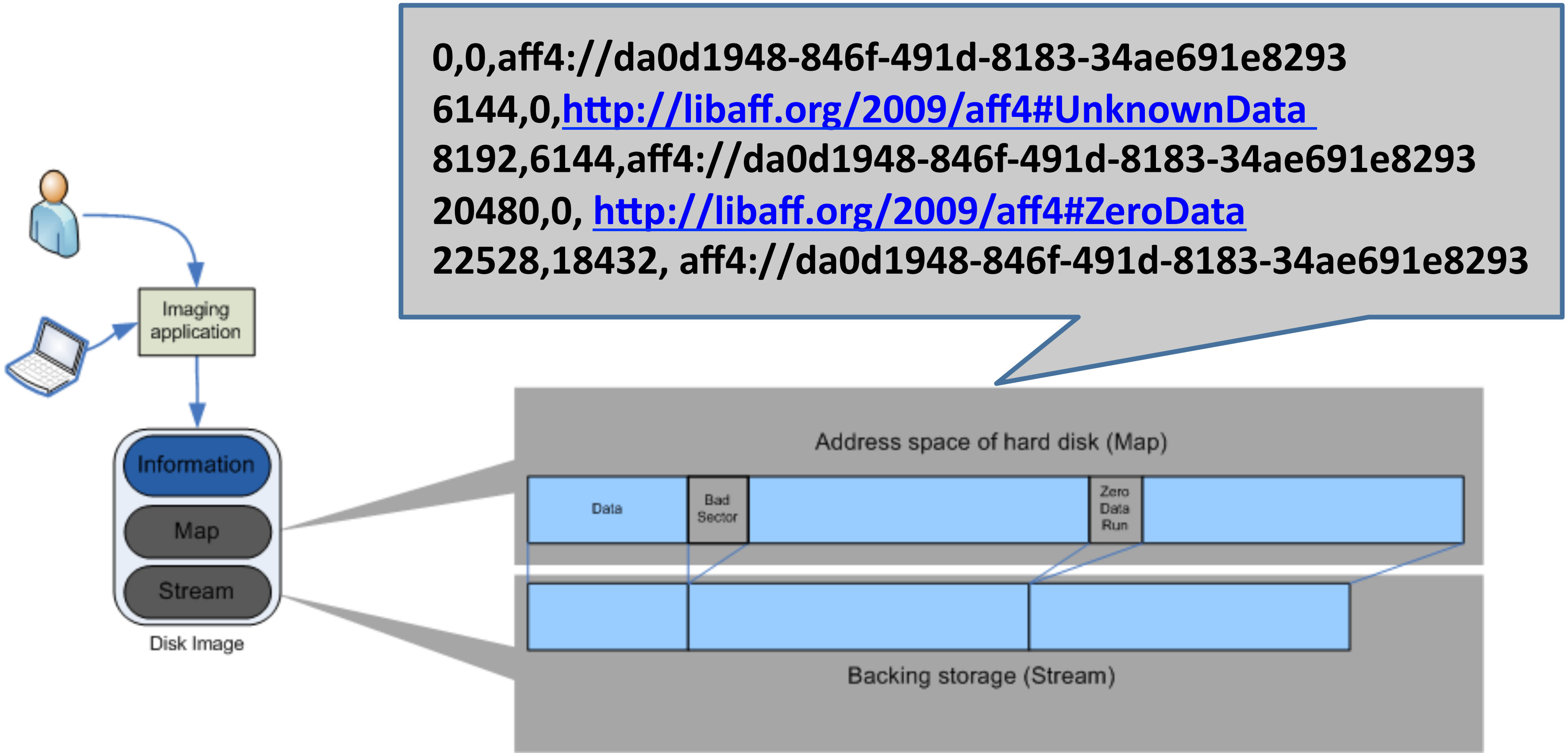
data and information are represented by surrogates, which are identified by a globally unique name



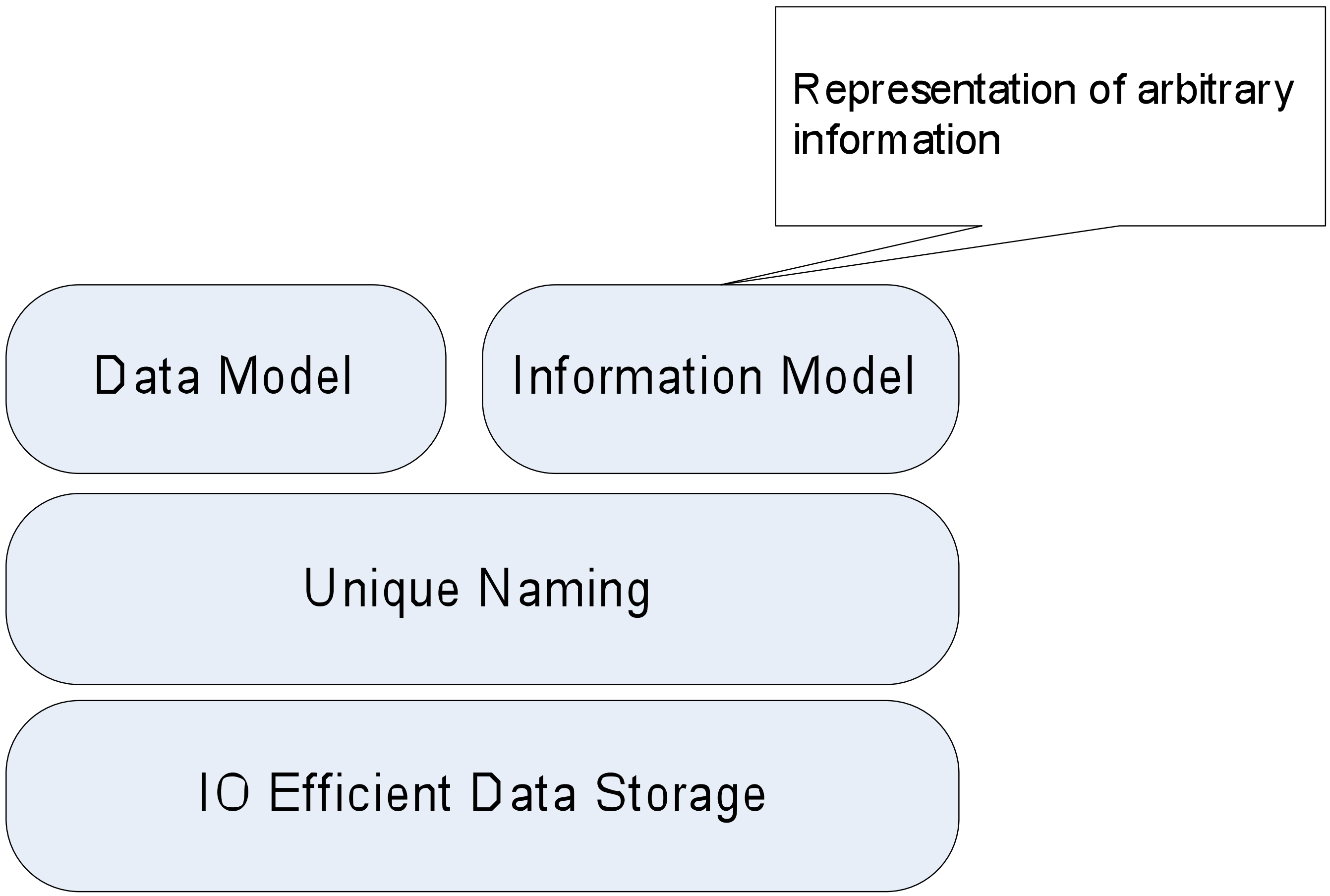
The AFF4 Data Model



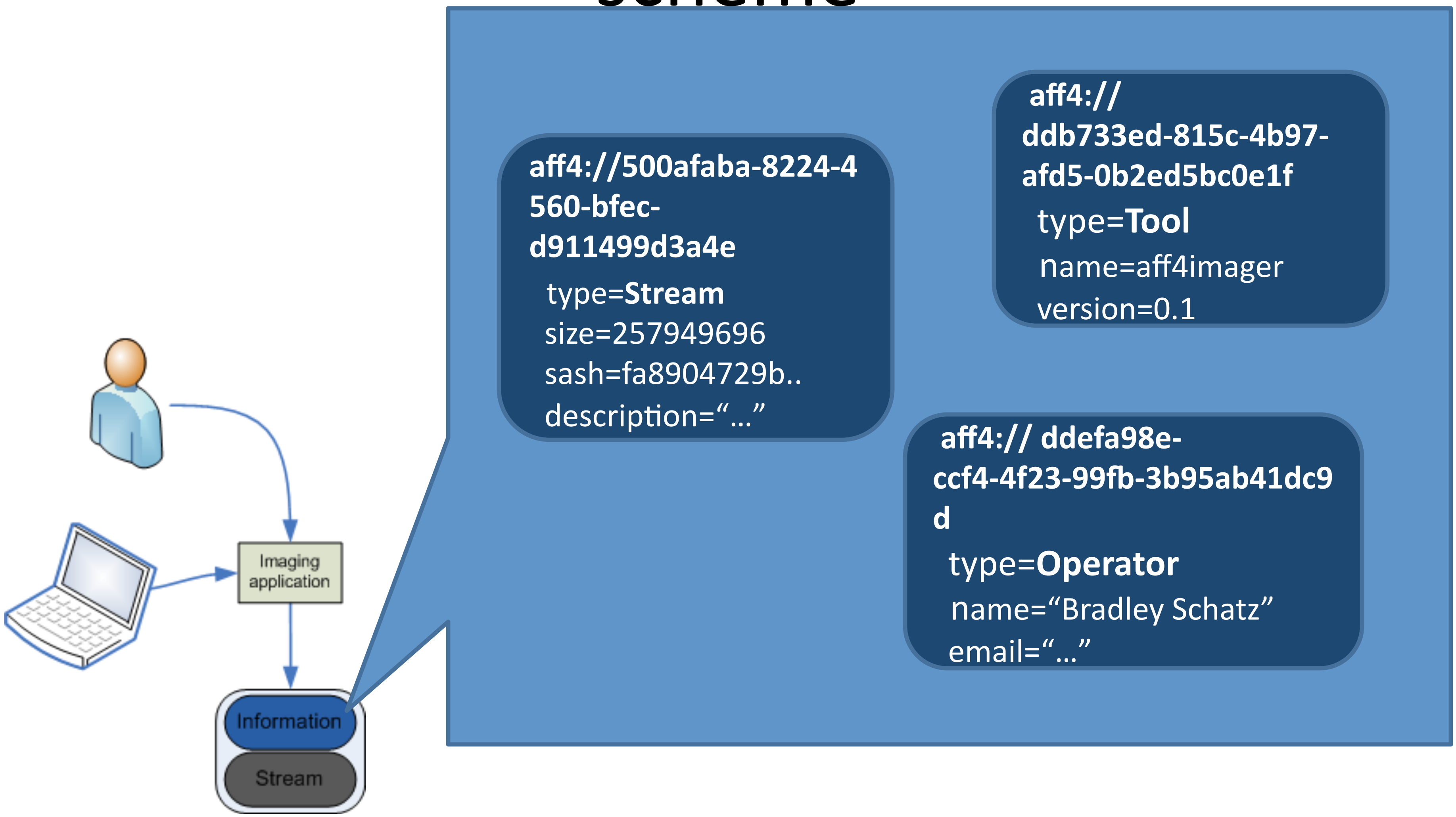
Maps enable describing discontiguous byte ranges



The AFF4 Information Model



“object” instances are uniquely identified by a globally unique naming scheme

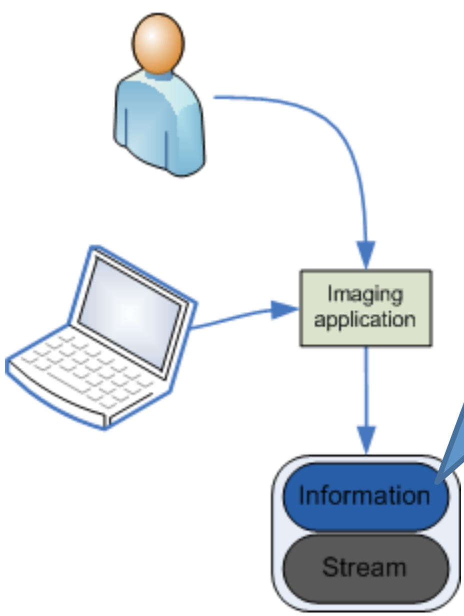


instances are serialized as RDF triples

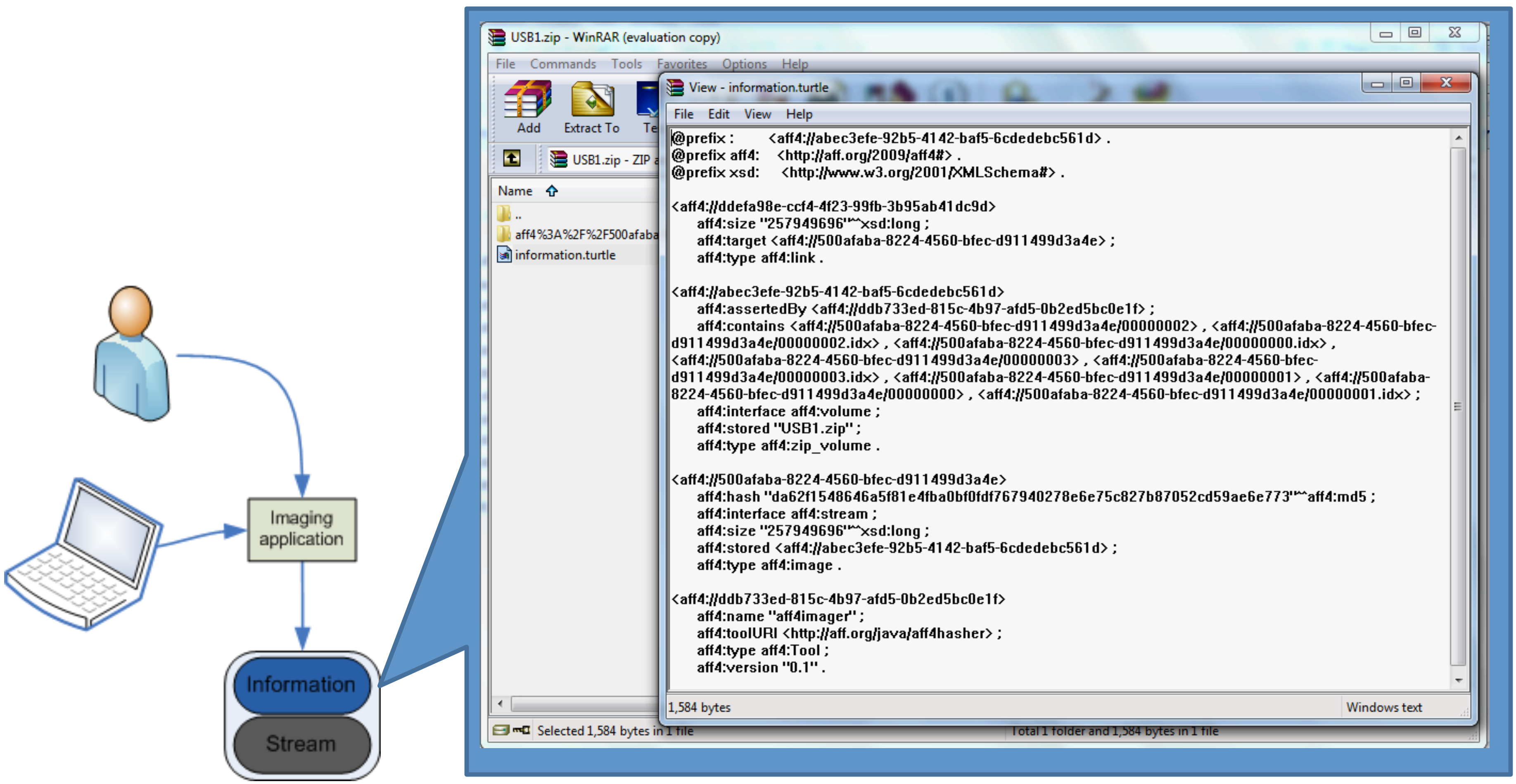
aff4://500afaba-8224-4560-bfec-d911499d3a4e type Stream
aff4://500afaba-8224-4560-bfec-d911499d3a4e size “257949696”^^xsd:long
aff4://500afaba-8224-4560-bfec-d911499d3a4e hash “fa8904729b…”^^aff4:md5
aff4://500afaba-8224-4560-bfec-d911499d3a4e description “320GB WD HDD”

aff4://ddb733ed-815c-4b97-afd5-0b2ed5bc0e1f type Tool
aff4://ddb733ed-815c-4b97-afd5-0b2ed5bc0e1f name “aff4Imager”
aff4://ddb733ed-815c-4b97-afd5-0b2ed5bc0e1f version “0.1”

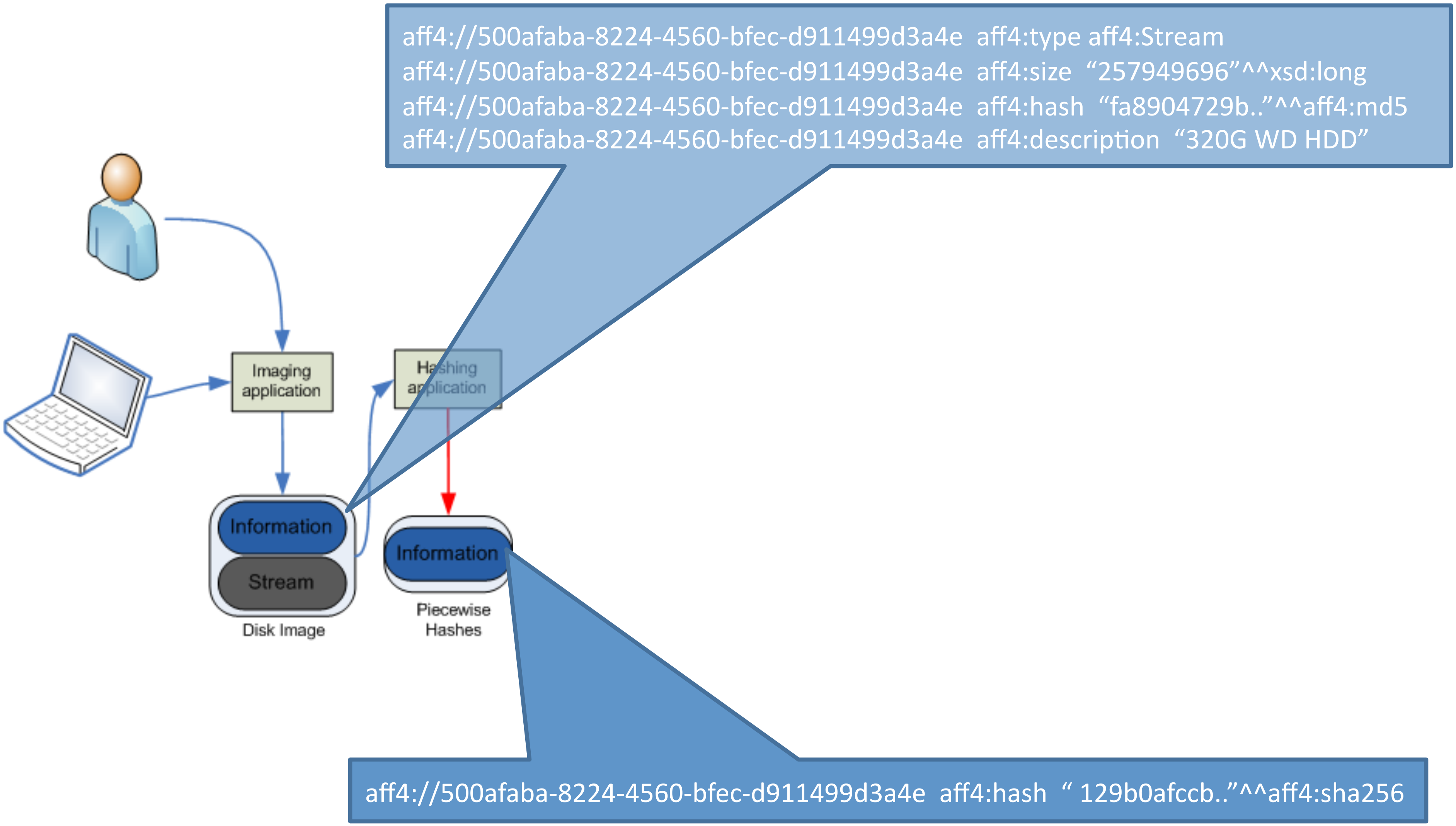
aff4://abec3efe-92b5-4142-baf5-6cdedebc561d type Operator
aff4://abec3efe-92b5-4142-baf5-6cdedebc561d name “Bradley Schatz”
aff4://abec3efe-92b5-4142-baf5-6cdedebc561d email bradley@foo.com



…and are stored within the "information.turtle" file using the Turtle RDF serialization



properties of instances may be defined in any container



Current status

AFF4 Library

- Reference implementation of the AFF4 format
  - It helps us test and evolve the format by using it
  - Flexible – can combine all types of AFF4 objects together
  - Python bindings automatically generated
    - Easy to keep in sync with C library
    - Very Fast
  - Multithreaded
  - Easy to use
  - API still in flux

Libaff4 overview

- Single entry point to library is the Resolver class:
  - Persistent data store
  - Currently implemented using tdb the trivial database
    - Very fast key/value database
  - Object manipulation
    - Opening and creating new AFF4 objects
    - Loading new AFF4 volumes
  - Information model access
    - Resolving, setting and adding predicates and Iterating over values.
  - Miscelaneous Registrations
    - Registering new RDF data types, aff4 object implementations, logging subsystem and a security provider for key management.

Example – Volume creation

Get resolver

```
oracle = pyaff4.Resolver()
```

Create a new volume

```
volume_urn = pyaff4.RDFURN()
volume_urn.set("/tmp/test.zip")
volume = oracle.create(pyaff4.AFF4_ZIP_VOLUME)
oracle.set_value(volume.urn, pyaff4.AFF4_STORED,  volume_urn)
```

Finish the volume
Return to cache

```
volume = volume.finish()
volume_urn = volume.urn
volume.cache_return()
```

Create a new
Stream stored
In the new volume
Finish the stream

```
image_fd = oracle.create(pyaff4.AFF4_IMAGE)
oracle.set_value(image_fd.urn, pyaff4.AFF4_STORED,  volume_urn)

image_fd = image_fd.finish()
```

Copy data
Into the stream

```
fd = open("/bin/ls")
while 1:
   data = fd.read(1000000)
   if not data: break

   image_fd.write(data)
```

Done, close everything

```
image_fd.close()
volume = oracle.open(volume_urn, 'w')
volume.close()
```

Note use of
Open() to reacquire
The volume lock

Typical use pattern

- Creating a new object:
  - Call resolver.create(type) to make a new object
    - It will receive a unique URI
  - Call set() method on it to set various predicates
    - Especially important is the aff4:stored predicate
  - When finished call the finish() method on it to complete the object
  - When done with the object call close() to finalize it.
- Opening an existing object:
  - Call resolver.open(url, mode) to get the object
  - Call the close() method when done.

Thread control and locking

- Since the library is multithreaded:
  - An object opened for writing will be locked to the calling thread.
    - The lock is not recursive so deadlocks can result
    - Library will raise an exception if a deadlock is detected.
  - Threads must release the object by calling the cache_return() method as soon as possible.
  - Threads can reacquire the same object by calling resolver.open() using its URL later.
    - Will block if object is locked by some other thread.
  - No locks for reading
    - multiple simultaneous readers are allowed

Applications

Applications

- We can get some useful effects by combining AFF4 objects together:
  - Carving
    - Carver generates a sequence of maps from the original stream
    - Zero copy carving
  - RAID reassembly
    - Acquire each disk in a RAID as a separate Image stream
    - Build a map which provides access to the logical (Reassembled) view with no copy overheads.

Applications

- Sparse image support:
  - A map stream is placed in front of an Image stream
  - Data is written to the image stream, while the map is adjusted
  - Holes in the map may be represented as bytes taken from the aff4:zero special target
    - Similar to /dev/zero
  - Bad blocks can be represented as taken from the aff4:bad_block target.
    - This can be varied to represent different reasons for missing data

Applications

- TCP/IP stream reassembly
  - A reassembler creates a map which pieces together streams from TCP payloads.
  - Further protocol dissection can be applied to TCP streams
    - HTTP, POP, SMTP etc
  - Zero copy process
  - Maintain Provenance

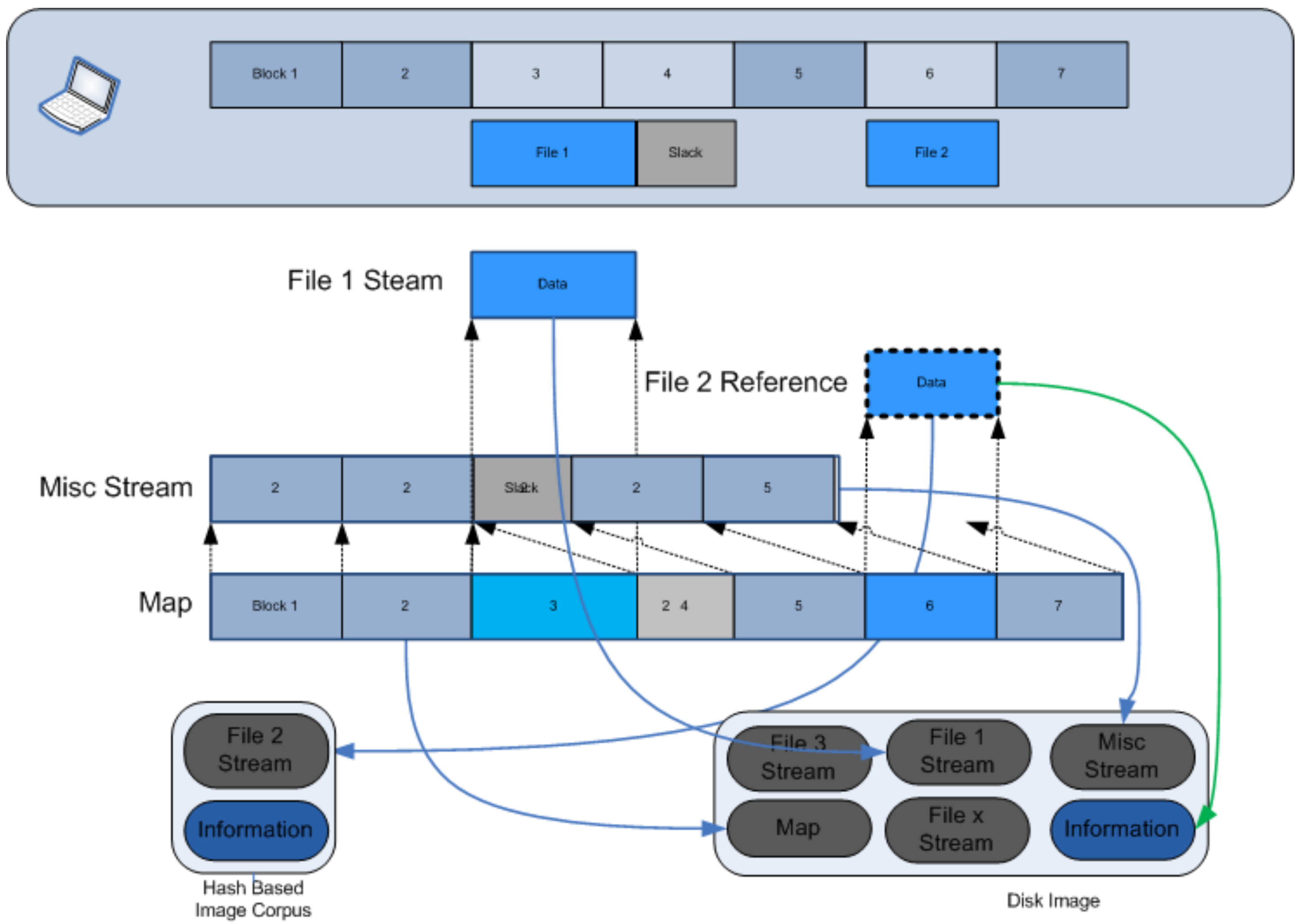
This paper

Hash based imaging

- The concept:
  - Store references to byte streams (hashes) rather than actual bit sequences
  - Potential benefits: storage/bandwidth efficiency
- Previous work: Teleporter
  - Client server protocol for transmission and reconstruction of images
  - Hashes based on file content
  - Soundness with sparse and encrypted files

AFF4 Hash imager

- Break a hard disk image into a set of block runs.
  - e.g. Using filesystem block allocation information.
- Each block run is stored as an image stream
- Block run is addressed by hash
- The disk is represented as a map which targets all the block runs.
- Block runs can be omitted
  - Saving acquisition time and space
  - The same hash within the image (e.g. file copies).
  - The same hash within the corpus


Block 1
2
3
4
5
6
7
File 1
Slack
File 2
File 1 Steam
Data
File 2 Reference
Data
Misc Stream
2
2
Slack
2
5
Map
Block 1
2
3
2 4
5
6
7
File 2 Stream
Information
Hash Based Image Corpus
File 3 Stream
File 1 Stream
Misc Stream
Map
File x Stream
Information
Disk Image

The segmenting algorithm

- Tradeoffs
  - Maximum/minimum block size
  - Compressibility
  - Sequential IO

Example map file

Image offset	Target offset	Length (B)	Target stream
0	0	8,192	aff4://2efe1ec75b170112b354b7e62a701f7929bff265
8,192	0	10,240,000	aff4://cdff2e763ed9b545b201b28780b4df9f45260619
10,248,192	0	10,240,000	aff4://f8a8f6f689a197d85905094b525ed2ea9fa1e30a
20,488,192	0	10,240,000	aff4://c2bfe67cfb2e4cac7edbb3013888abea6a123ac5
30,728,192	0	56,832	aff4://6ea4826afcfb22483ad02716027122b09cd77d6f
30,785,024	0	2,097,152	aff4://3ef9b3329d8b24f28824abc457781ab0a51e23e7
32,882,176	0	2,560	aff4://b49d7f48300701235231f6b6fc3d92a5630f9e70
32,884,736	0	4,096	aff4://01829df4c765b8fa8991537d641916ae368d71a5
32,888,832	0	908	aff4://e78424ed-9398-4c94-bc9f-e5e8684564ff/ ntfs_image.dd/misc
32,889,740	1798	628	aff4://e78424ed-9398-4c94-bc9f-e5e8684564ff/ ntfs_image.dd/misc

Results

Comparison of imaging times and volume size for various techniques

Acquisition Method	Total Image Size (B)	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

Compression is expensive

Acquisition Method	Total Image Size (B)	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

Hash based imaging may result in substantial space savings

Acquisition Method	Total Image Size (B)	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

Hash based imaging currently costs more time when used in absence of corpus

Acquisition Method	Total Image Size (B)	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

Conclusions

Conclusions

- Hash based imaging:
  - A space/bandwidth efficient approach to storing forensic images
  - Competitive with current compressed imaging speeds

Conclusions

- AFF4
  - Container format is rich and expressive
    - We did not discuss encryption and signing support
    - Can express any information using ours or third party defined vocabulary
    - Ideal for metadata interchange (it's basically XML)
  - AFF4 is extensible
    - Can register your own:
      - AFF4 object
      - RDF Datatype
    - Implementations may not implement all features
      - Nice for simple, embedded applications, can implement a very minimalistic AFF4 writer

Future

- AFF4 aware imagers
  - Shift more analytic building blocks to acquisition stage
- Formal information models
- AFF4 aware tools
  - Support modular and composeable tools
  - Integration of arbitrary information

The AFF4 format & API will stabilise to beta shortly

- Parallel implementations
  - C and Java
  - Python binding to C

- Integration with
  - PyFlag
  - Sleuthkit

- For more information
  - http://www.forensicswiki.org/wiki/AFF4

Thank you

Dr Bradley Schatz
b.schatz@qut.edu.au

QUT isi Information Security Institute