Using Purpose-Built Functions And Block Hashes To Enable Small Block And Sub-File Forensics

By

Simson Garfinkel, Alex Nelson, Douglas White, and Vassil Roussev

Presented At

The Digital Forensic Research Conference

DFRWS 2010 USA Portland, OR (Aug 2nd - 4th)

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.

http:/dfrws.org

Using purpose-built functions and block hashes to enable small block and sub-file forensics.

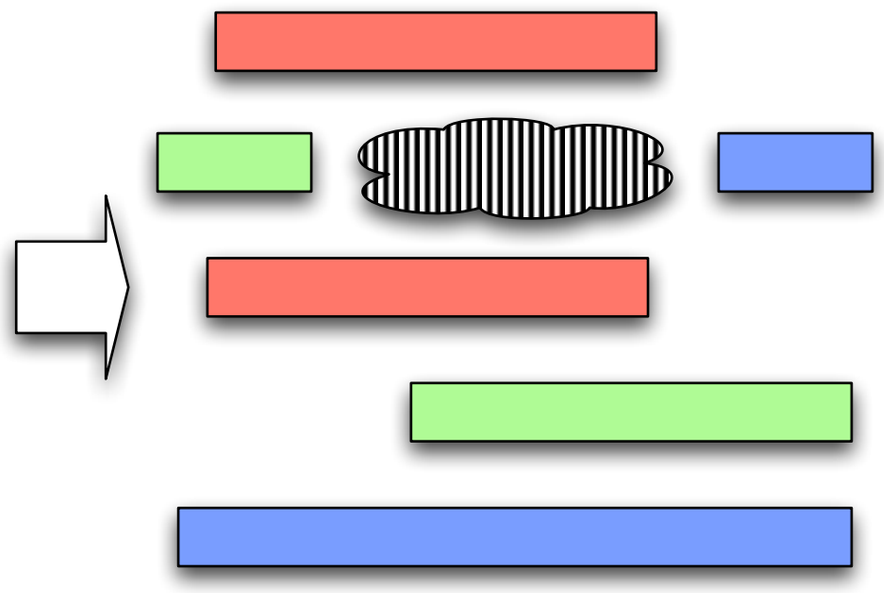
Simson Garfinkel, Vassil Roussev, Alex Nelson and Douglas White

DFRWS 2010

10:15am, August 2, 2010

Today most computer forensics is "file forensics."

1. Extract files from disk images.
2. Extract text from files.
3. Carving unallocated space to find more files.



This is the basic approach of EnCase, FTK, SleuthKit, Scalpel, etc.

File forensics has *significant limitations*.

The file system may not be recoverable.

- Logical damage / partial overwriting
- Media failure
- File system not supported by forensic tool.

There may be insufficient time to read the entire file system.

- The tree structure of file systems makes processing in parallel hard.
- Fragmented MFT or other structures may significantly slow ingest.

File contents may be encrypted.

Small block forensics: analysis below the files. Instead of processing file-by-file, process block-by-block.

The file system may not be recoverable.

- *No problem! No need to recover the file system.*

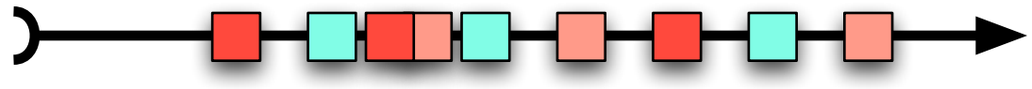
There may be insufficient time to read the entire file system.

- *No problem! Just sample as necessary.*

File contents may be encrypted.

- *Files that are passed around remain unchanged.*

You may be working with packets on a network.



This paper introduces an approach for performing small block forensics. Critical techniques include:

- *block hash calculations; bulk data analysis*

Outline of the paper

Introduction: Why Small Block Forensics are Interesting

Prior Work

Distinct Block Recognition

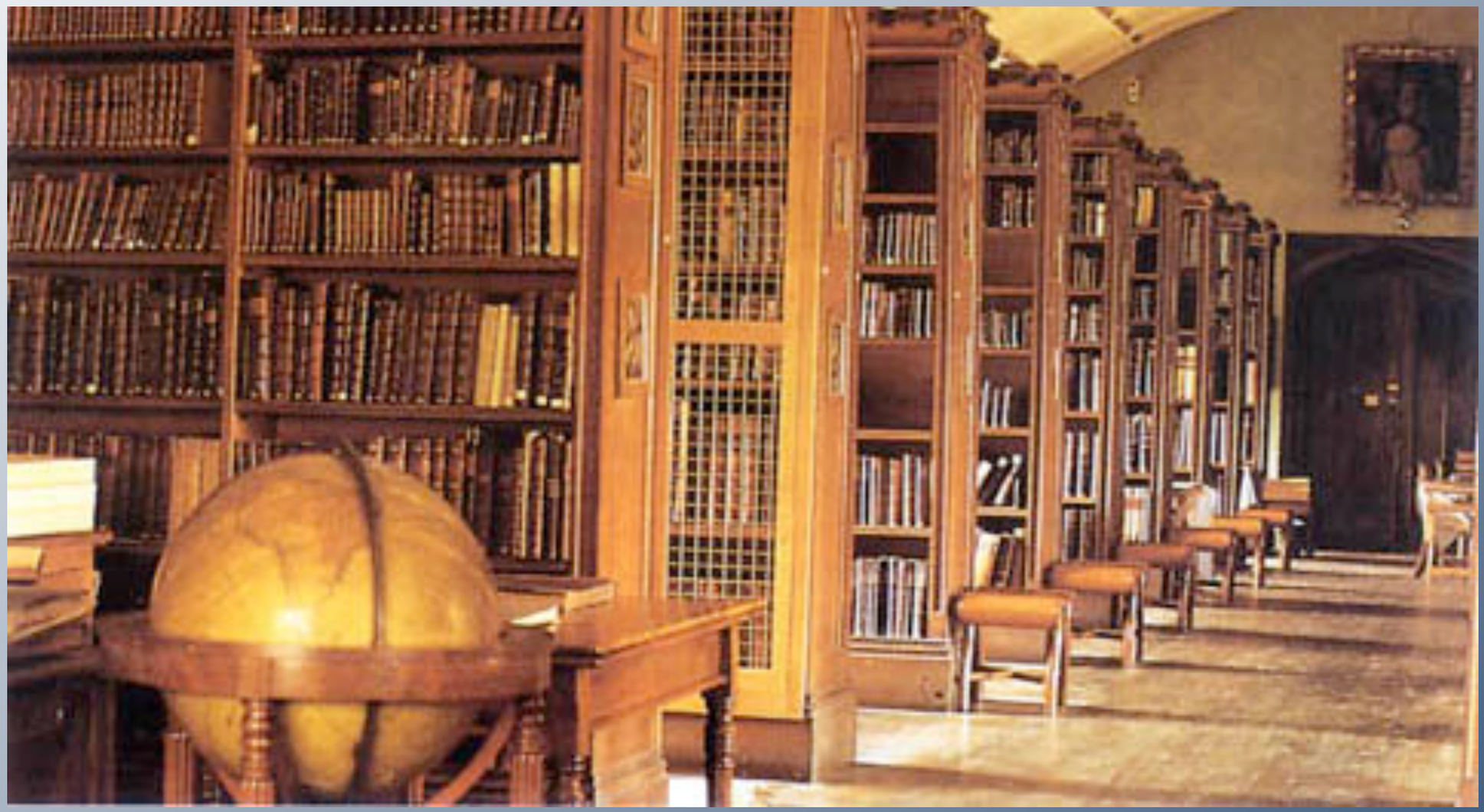
- A definition of "distinct" and the distinct block hypothesis
- Block, Sector and File Alignment — what block size should we use?
- Hash-based carving with **frag_find**
- Statistical Sector Sampling to detect the presence of contraband data.

Fragment Type Discrimination

- Why "discrimination" and not "recognition" or "identification"
- Three discriminators: JPEG, MPEG, and Huffman.
- Statistical Sector Sampling to determine the "forensic inventory."

Lessons Learned

Conclusions



Prior Work

Prior work 1/4

File identification:

- Unix "file" command (libmagic)
- DROID (National archives of UK)
- TrID (Marco Pontello)
- File Investigator TOOLS (Forensic Innovations)
- Outside In (Oracle)

Statistical File fragment classification (bigrams & n-grams)

- McDaniel; Calhoun & Coles; Karresand & Shahmehri; Li, Wang, Herzog; Moody & Erbacher; Veenman;
- and others

Distinguishing random from encrypted data:

- Speirs patent application (20070116267, May 2007)

Prior work 2/4: Hash-Based Carving

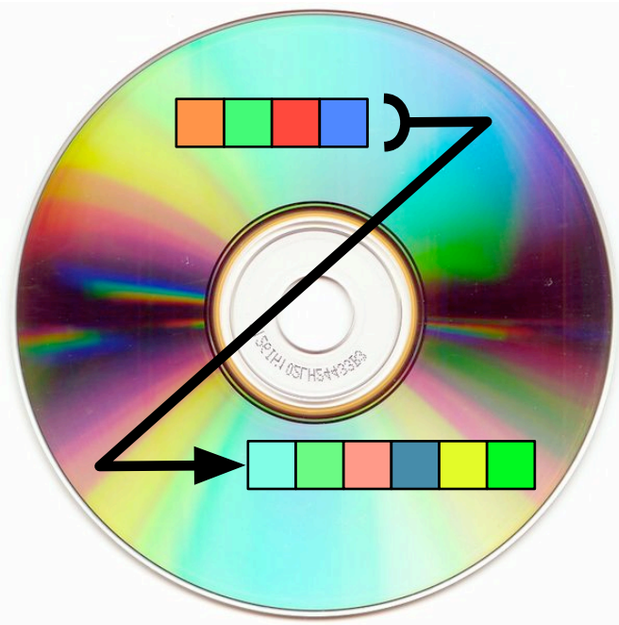
Big idea: recognize the presence and location of files based on the hash codes of individual blocks or sectors.

- Garfinkel DFRWS 2006 Carving Challenge. "The MD5 trick"
- Collange et al, 2008, "An empirical analysis of disk sector hashes for data carving"

Requires hashing every sector on the drive.

- MD5 is really fast.

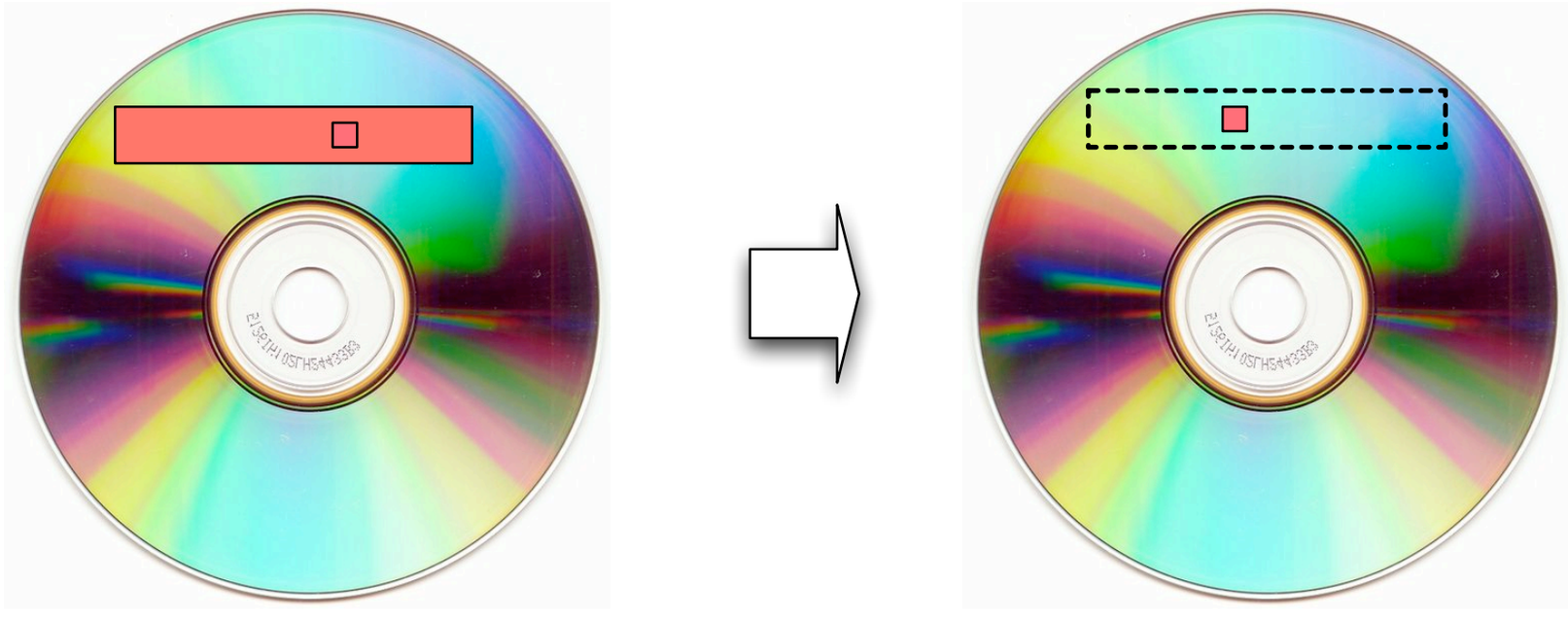
Master file: 



Prior work 3/4: Shared Sectors

Cross Drive Analysis (Garfinkel 2006)

- A sector from a file found on a second drive may indicate that the entire file was once present.



Disk #1 **Disk #2**

Prior work 4/5: 10 years of fragment identification…
… mostly n-gram analysis

Standard approach:

- Get samples of different file types
- Train a classifier (typically k-nearest-neighbor or Support Vector Machines)
- Test classifier on "unknown data"

Examples:

- 2001 — McDaniel — "Automatic File Type Detection Algorithm"
  - —*header, footer & byte frequency (unigram) analysis (headers work best)*
- 2005 — LiWei-Jen et. al — "Fileprints"
  - —*unigram analysis*
- 2006 — Haggerty & Taylor — "FORSIGS"
  - —*n-gram analysis*
- 2007 — Calhoun — "Predicting the Type of File Fragments"
  - —*statistics of unigrams*

  - —*http://www.forensicswiki.org/wiki/File_Format_Identification*

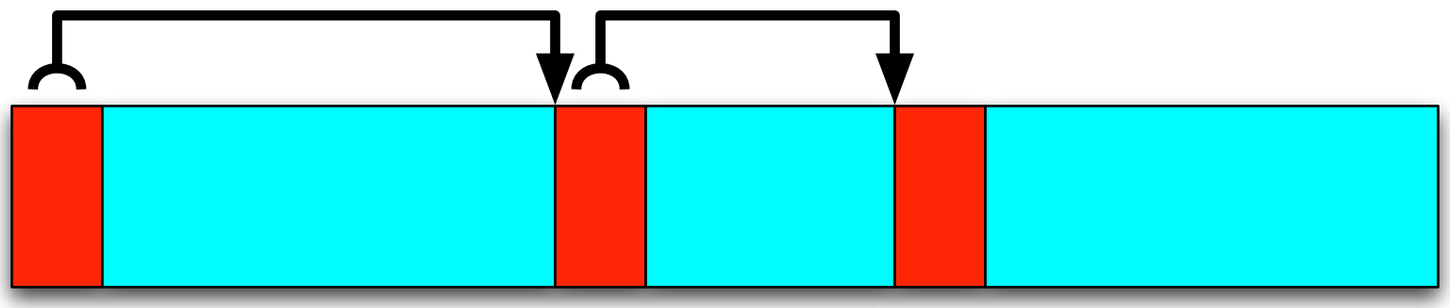
Prior work 4/4: MP3 Validation using Frames

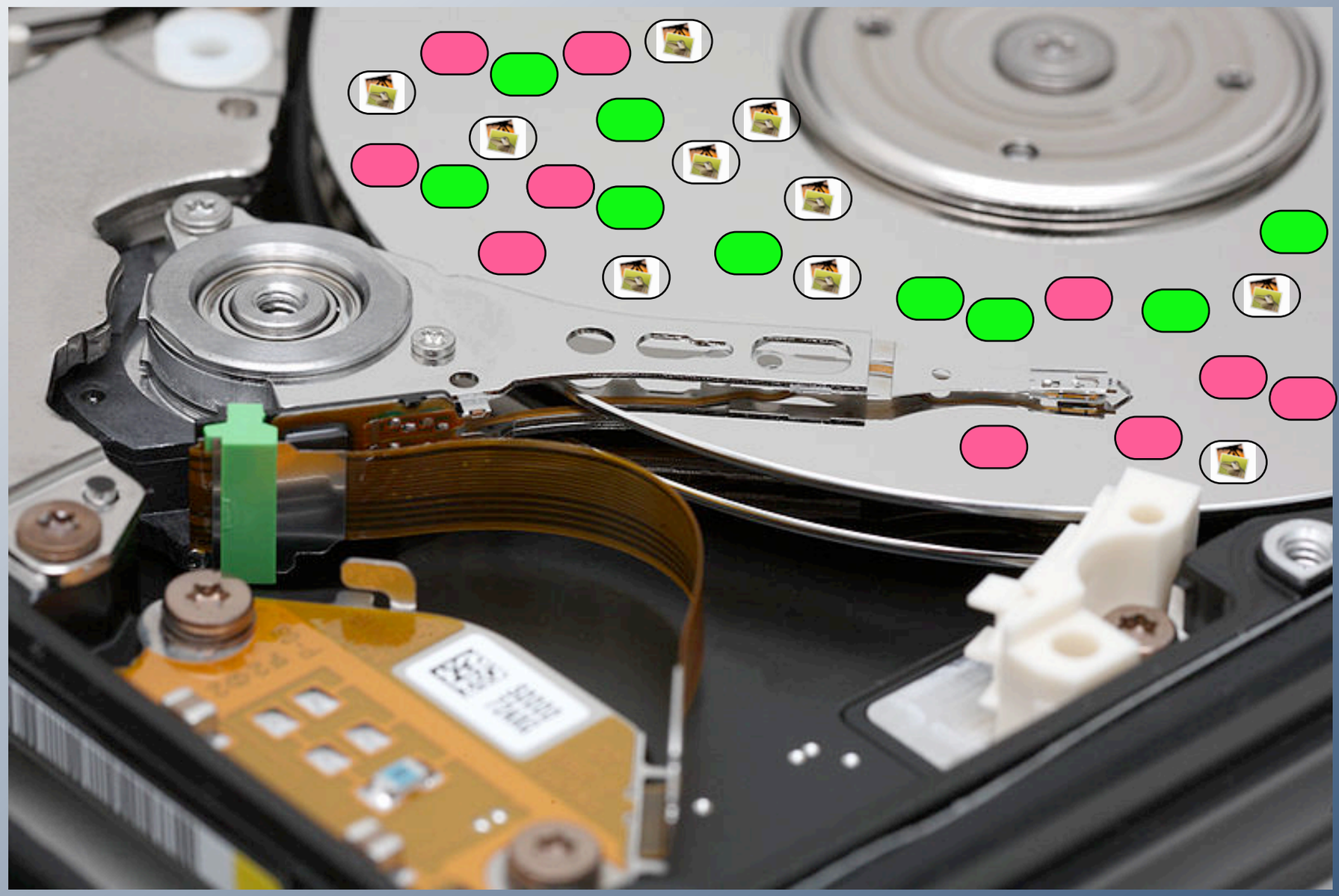
Big idea: Identify MP3 files based on chained headers

- Al-Dahir *et. al*, DFRWS 2007 Carving Challenge, **mp3scalpel**

Sliding Window Algorithm:

- Look for MP3 Sync Bits
  - —*Sanity check location BUF[n], sanity-check all MP3 header fields.*
  - —*Calculate location of next frame header*
  - —*Recurse if valid*
- If not valid, advance window and try again.





Part 1:
Distinct Block Recognition

Distinct block: a block of data that does not arise by chance more than once.

Consider a disk sector with 512 bytes.

- There are $2^{512x8} \approx 10^{1,233}$ different sectors.
- A randomized sector is likely to be "distinct."
  - —*e.g. encryption keys, high-entropy data, etc.*

Distinct Block Hypothesis #1:

- If a block of data from a file is distinct, then a copy of that block found on a data storage device is evidence that the file was once present.

Distinct Block Hypothesis #2:

- If a file is known to have been manufactured using some high-entropy process, and if the blocks of that file are shown to be distinct throughout a large and representative corpus, then those blocks can be treated as if they are distinct.

What kinds of files are likely to have distinct blocks?

A bock from a JPEG image should be distinct.

208 distinct 4096-byte block hashes

"You cannot step twice into the same river."

—*Heraclitus*

"You cannot step twice into the same sunny day."

—*Distinct Block Hypothesis*

… Unless the image is all black.

Probably no distinct blocks.

In fact, many JPEGs seem to contain distinct blocks.

Experimentally, we see that most JPEGs have distinct blocks…

208 distinct 4096-byte block hashes

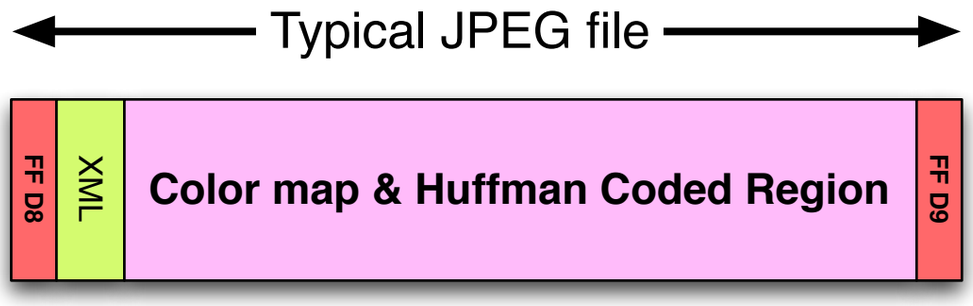
∩

204 distinct 4096-byte block hashes

=

0 block hashes in common

Even with JPEG headers, XML, and Color Maps:



Other kinds of files likely have distinct blocks as well.

Files with high entropy:

- Multimedia files (Video)
- Encrypted files.
- Files with *original* writing.
- Files with just a few characters "randomly" distributed
  - *There are 10^{33} ($^{512!}/_{500!}$) different sectors with 500 NULLs and 12 ASCII spaces!*

What kinds of files won't have distinct blocks?

- Those that are filled with a constant character.
- Simple patterns (`00 FF 00 FF 00 FF`)

Modern file systems align files on sector boundaries.

Place a file with distinct blocks on a disk.

- Distinct disk blocks => Distinct disk sectors.

208 distinct 4096-byte block hashes

So finding a distinct block on a disk is evidence that the file was present.

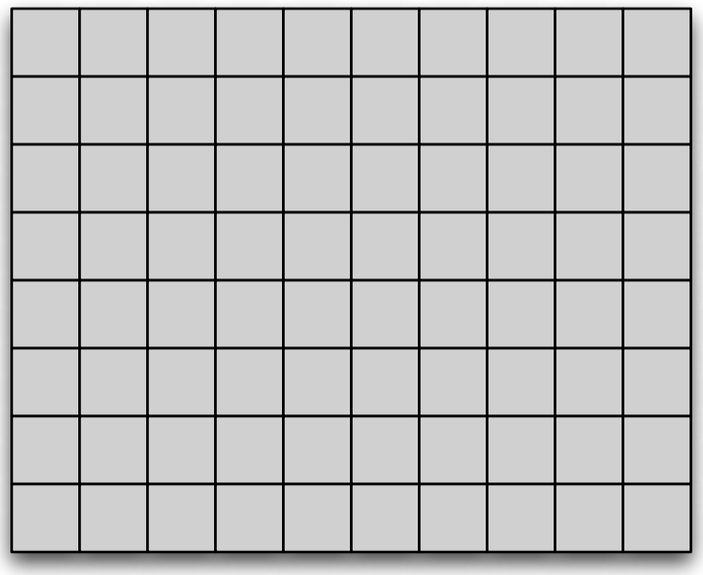
- *(Distinct Block Hypothesis #1:*
  - *If a block of data from a file is distinct, then a copy of that block found on a data storage device is evidence that the file was once present.)*

Hash-based carving

Input:

- 1 or more *Master Files*

- A disk image



Algorithm:

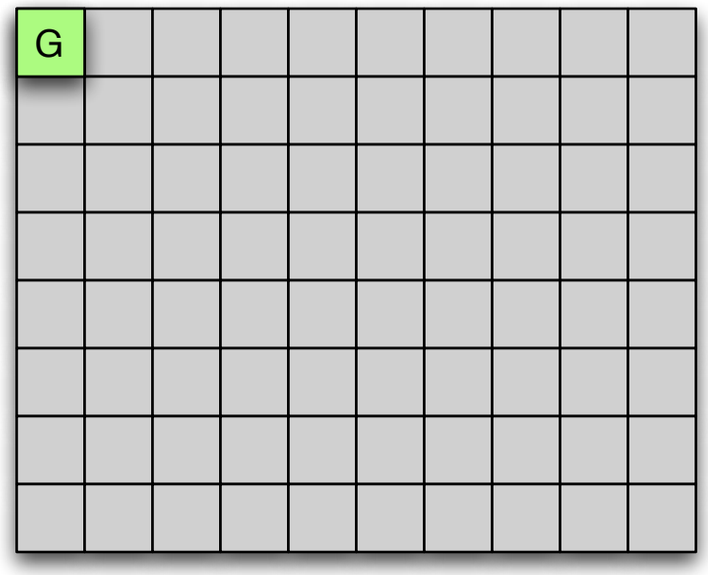
- Hash each sector of each master file.
  - *Store hashes in a map[].*
- Hash each sector of the disk image.
  - *Check each sector hash against the map[]*
  - *If a sector hash matches multiple files, choose the file that creates the longer run.*

Hash-based carving

Input:

- 1 or more *Master Files*

- A disk image



Algorithm:

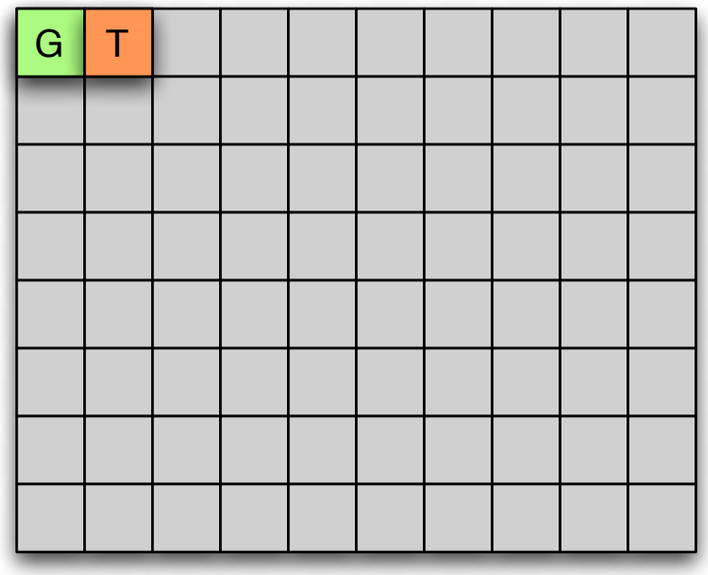
- Hash each sector of each master file.
  - *Store hashes in a map[].*
- Hash each sector of the disk image.
  - *Check each sector hash against the map[]*
  - *If a sector hash matches multiple files, choose the file that creates the longer run.*

Hash-based carving

Input:

- 1 or more *Master Files*

- A disk image



Algorithm:

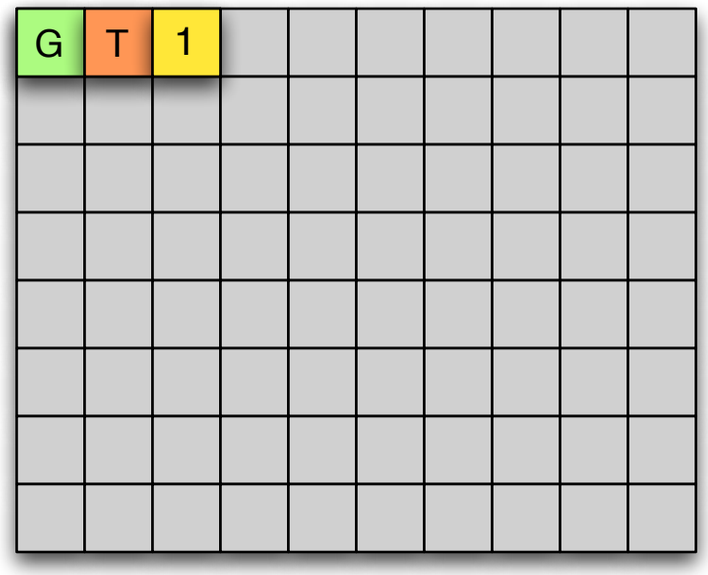
- Hash each sector of each master file.
  - *Store hashes in a map[].*
- Hash each sector of the disk image.
  - *Check each sector hash against the map[]*
  - *If a sector hash matches multiple files, choose the file that creates the longer run.*

Hash-based carving

Input:

- 1 or more *Master Files*

- A disk image



Algorithm:

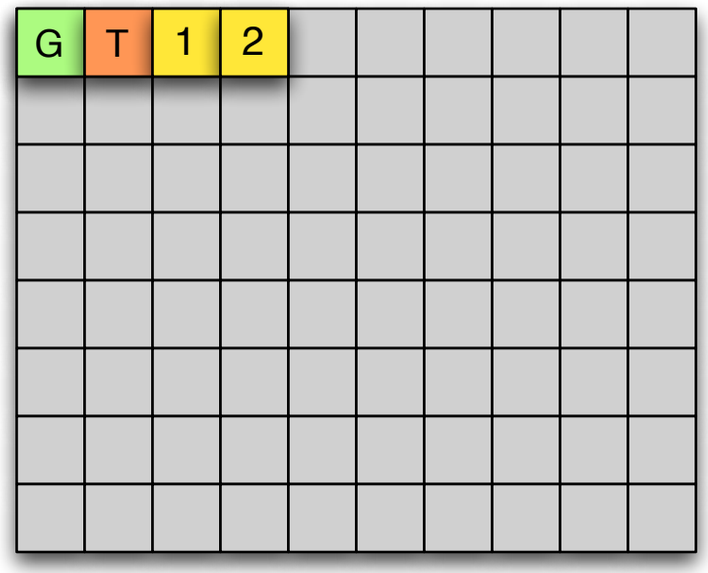
- Hash each sector of each master file.
  - *Store hashes in a map[].*
- Hash each sector of the disk image.
  - *Check each sector hash against the map[]*
  - *If a sector hash matches multiple files, choose the file that creates the longer run.*

Hash-based carving

Input:

- 1 or more *Master Files*

- A disk image



Algorithm:

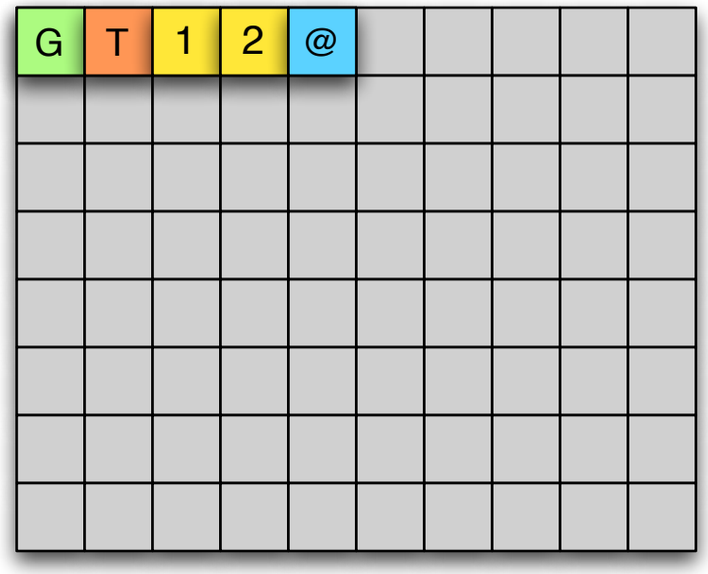
- Hash each sector of each master file.
  - *Store hashes in a map[].*
- Hash each sector of the disk image.
  - *Check each sector hash against the map[]*
  - *If a sector hash matches multiple files, choose the file that creates the longer run.*

Hash-based carving

Input:

- 1 or more *Master Files*

- A disk image



Algorithm:

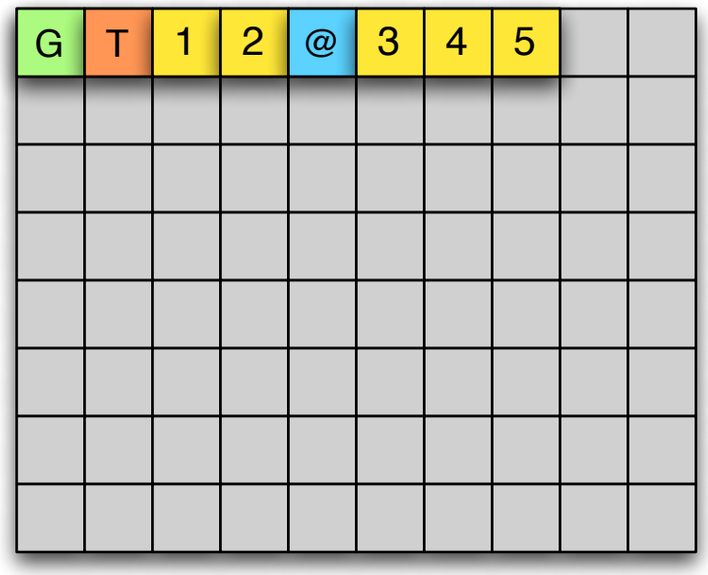
- Hash each sector of each master file.
  - *Store hashes in a map[].*
- Hash each sector of the disk image.
  - *Check each sector hash against the map[]*
  - *If a sector hash matches multiple files, choose the file that creates the longer run.*

Hash-based carving

Input:

- 1 or more *Master Files*

- A disk image



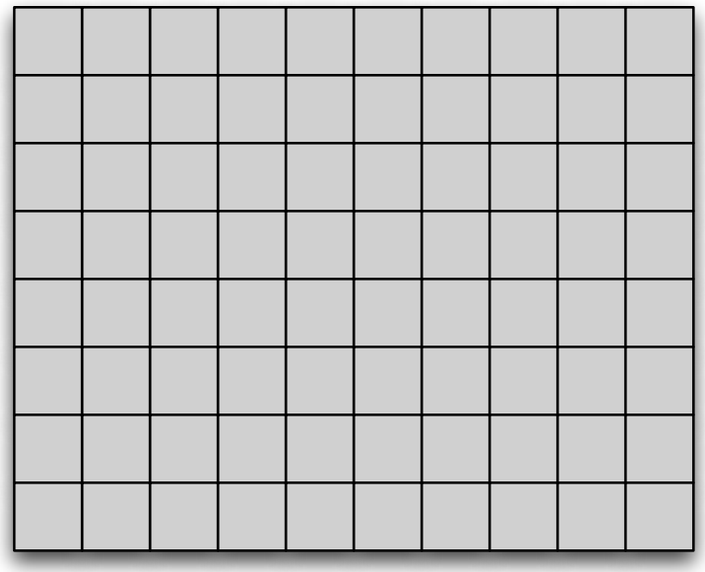
Algorithm:

- Hash each sector of each master file.
  - *Store hashes in a map[].*
- Hash each sector of the disk image.
  - *Check each sector hash against the map[]*
  - *If a sector hash matches multiple files, choose the file that creates the longer run.*

frag_find is a high-performance hash-based carver.

Implementation:

- C++
- Pre-filtering with NPS Bloom package.
  - *All sector hashes are put in a Bloom Filter*
  - *Block size = Sector Size = 512 bytes*



Output in Digital Forensics XML:

```
<fileobject>
  <filename>DCIM/100CANON/IMG_0001.JPG</filename>
  <byte_runs>
    <run file_offset='0' fs_offset='55808' img_offset='81920' len='855935'/>
      <hashdigest type='MD5'>b83137bd4ba4b56ed856be8a8e2dc141</hashdigest>
      <hashdigest type='SHA1'>03eaa4a5678542039c29a5ccf12b3d71ae96cbd2</hashdigest>
  </byte_runs>
</fileobject>
```

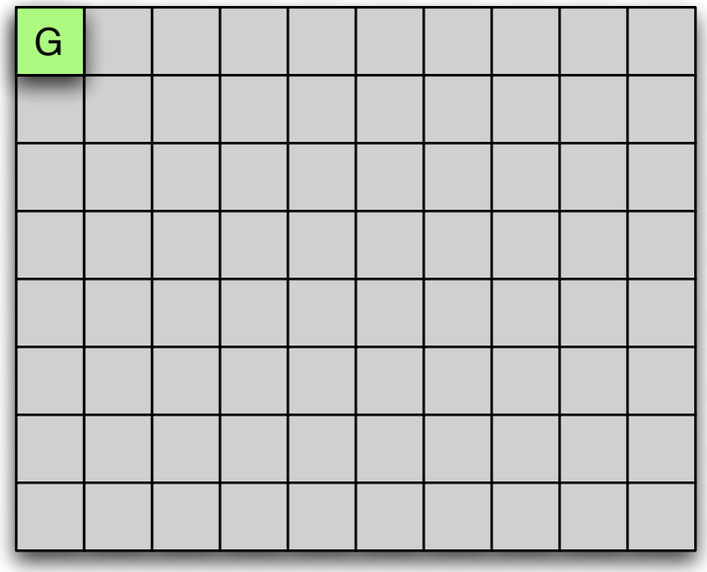
Uses:

- Exfiltration of sensitive documents;
- Data Loss Detection; etc.
- Download from http://afflib.org/

frag_find is a high-performance hash-based carver.

Implementation:

- C++
- Pre-filtering with NPS Bloom package.
  - *All sector hashes are put in a Bloom Filter*
  - *Block size = Sector Size = 512 bytes*



Output in Digital Forensics XML:

```
<fileobject>
  <filename>DCIM/100CANON/IMG_0001.JPG</filename>
  <byte_runs>
    <run file_offset='0' fs_offset='55808' img_offset='81920' len='855935'/>
      <hashdigest type='MD5'>b83137bd4ba4b56ed856be8a8e2dc141</hashdigest>
      <hashdigest type='SHA1'>03eaa4a5678542039c29a5ccf12b3d71ae96cbd2</hashdigest>
  </byte_runs>
</fileobject>
```

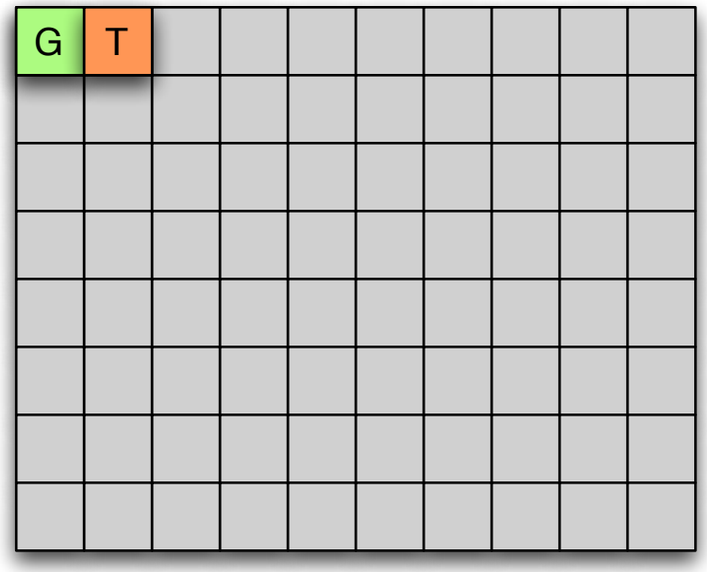
Uses:

- Exfiltration of sensitive documents;
- Data Loss Detection; etc.
- Download from http://afflib.org/

frag_find is a high-performance hash-based carver.

Implementation:

- C++
- Pre-filtering with NPS Bloom package.
  - *All sector hashes are put in a Bloom Filter*
  - *Block size = Sector Size = 512 bytes*



Output in Digital Forensics XML:

```
<fileobject>
  <filename>DCIM/100CANON/IMG_0001.JPG</filename>
  <byte_runs>
    <run file_offset='0' fs_offset='55808' img_offset='81920' len='855935'/>
      <hashdigest type='MD5'>b83137bd4ba4b56ed856be8a8e2dc141</hashdigest>
      <hashdigest type='SHA1'>03eaa4a5678542039c29a5ccf12b3d71ae96cbd2</hashdigest>
  </byte_runs>
</fileobject>
```

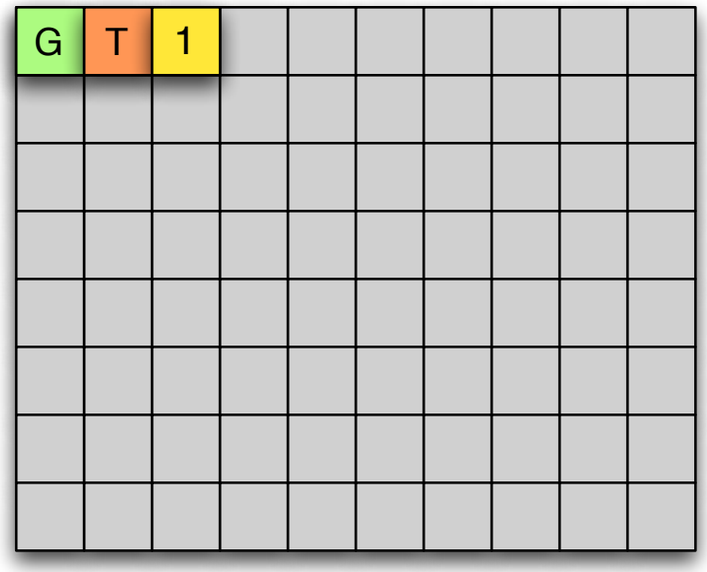
Uses:

- Exfiltration of sensitive documents;
- Data Loss Detection; etc.
- Download from http://afflib.org/

frag_find is a high-performance hash-based carver.

Implementation:

- C++
- Pre-filtering with NPS Bloom package.
  - *All sector hashes are put in a Bloom Filter*
  - *Block size = Sector Size = 512 bytes*



Output in Digital Forensics XML:

```
<fileobject>
  <filename>DCIM/100CANON/IMG_0001.JPG</filename>
  <byte_runs>
    <run file_offset='0' fs_offset='55808' img_offset='81920' len='855935'/>
      <hashdigest type='MD5'>b83137bd4ba4b56ed856be8a8e2dc141</hashdigest>
      <hashdigest type='SHA1'>03eaa4a5678542039c29a5ccf12b3d71ae96cbd2</hashdigest>
  </byte_runs>
</fileobject>
```

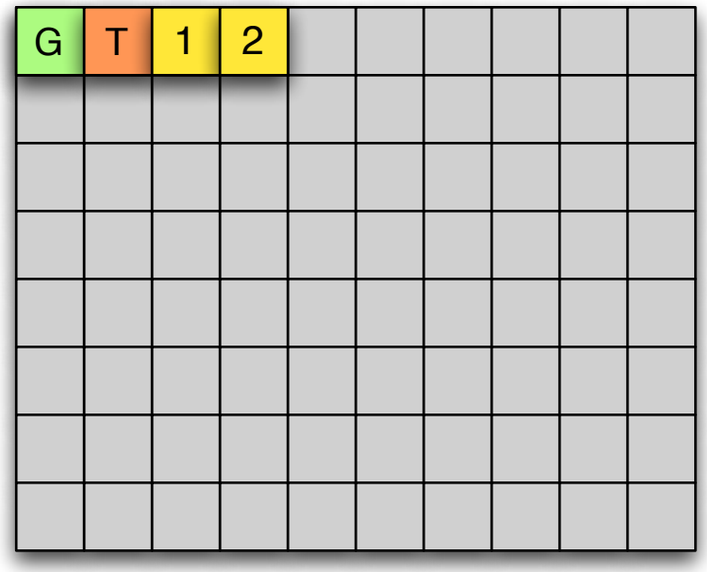
Uses:

- Exfiltration of sensitive documents;
- Data Loss Detection; etc.
- Download from http://afflib.org/

frag_find is a high-performance hash-based carver.

Implementation:

- C++
- Pre-filtering with NPS Bloom package.
  - *All sector hashes are put in a Bloom Filter*
  - *Block size = Sector Size = 512 bytes*



Output in Digital Forensics XML:

```
<fileobject>
  <filename>DCIM/100CANON/IMG_0001.JPG</filename>
  <byte_runs>
    <run file_offset='0' fs_offset='55808' img_offset='81920' len='855935'/>
      <hashdigest type='MD5'>b83137bd4ba4b56ed856be8a8e2dc141</hashdigest>
      <hashdigest type='SHA1'>03eaa4a5678542039c29a5ccf12b3d71ae96cbd2</hashdigest>
  </byte_runs>
</fileobject>
```

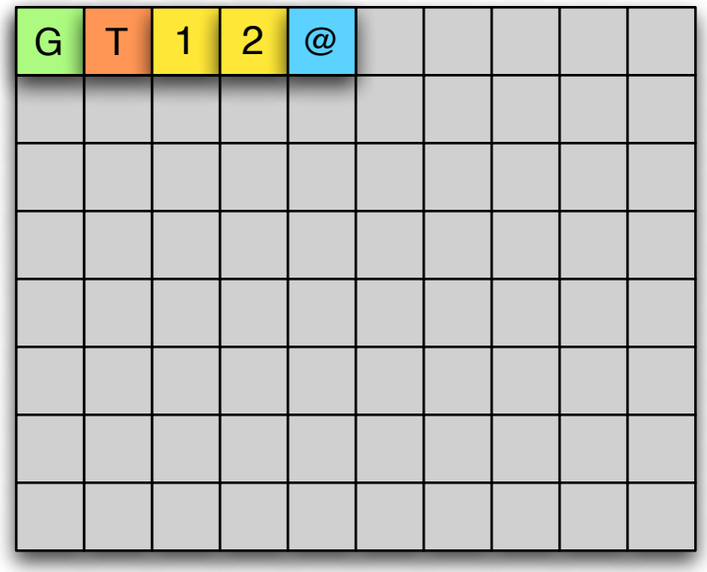
Uses:

- Exfiltration of sensitive documents;
- Data Loss Detection; etc.
- Download from http://afflib.org/

frag_find is a high-performance hash-based carver.

Implementation:

- C++
- Pre-filtering with NPS Bloom package.
  - *All sector hashes are put in a Bloom Filter*
  - *Block size = Sector Size = 512 bytes*



Output in Digital Forensics XML:

```
<fileobject>
  <filename>DCIM/100CANON/IMG_0001.JPG</filename>
  <byte_runs>
    <run file_offset='0' fs_offset='55808' img_offset='81920' len='855935'/>
      <hashdigest type='MD5'>b83137bd4ba4b56ed856be8a8e2dc141</hashdigest>
      <hashdigest type='SHA1'>03eaa4a5678542039c29a5ccf12b3d71ae96cbd2</hashdigest>
  </byte_runs>
</fileobject>
```

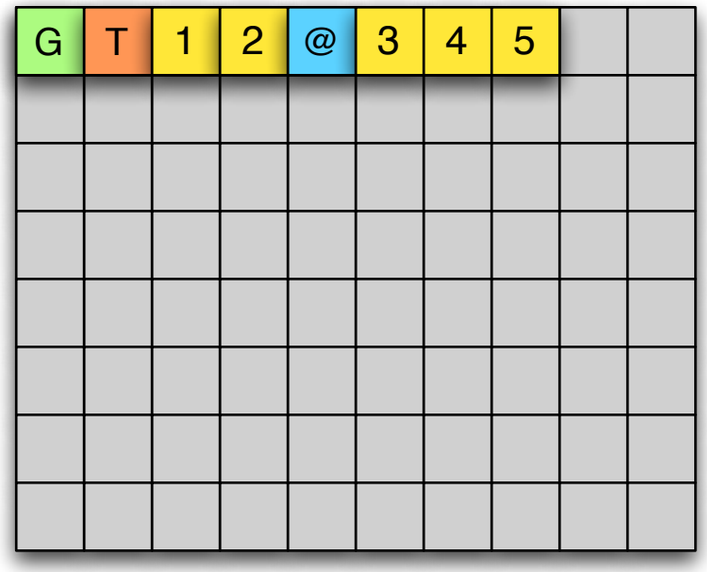
Uses:

- Exfiltration of sensitive documents;
- Data Loss Detection; etc.
- Download from http://afflib.org/

frag_find is a high-performance hash-based carver.

Implementation:

- C++
- Pre-filtering with NPS Bloom package.
  - *All sector hashes are put in a Bloom Filter*
  - *Block size = Sector Size = 512 bytes*



Output in Digital Forensics XML:

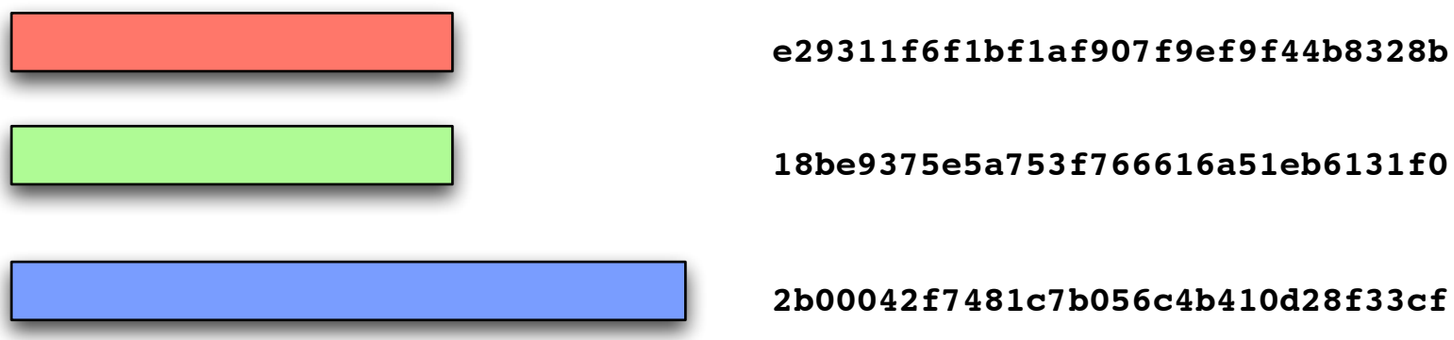
```
<fileobject>
  <filename>DCIM/100CANON/IMG_0001.JPG</filename>
  <byte_runs>
    <run file_offset='0' fs_offset='55808' img_offset='81920' len='855935'/>
      <hashdigest type='MD5'>b83137bd4ba4b56ed856be8a8e2dc141</hashdigest>
      <hashdigest type='SHA1'>03eaa4a5678542039c29a5ccf12b3d71ae96cbd2</hashdigest>
  </byte_runs>
</fileobject>
```

Uses:

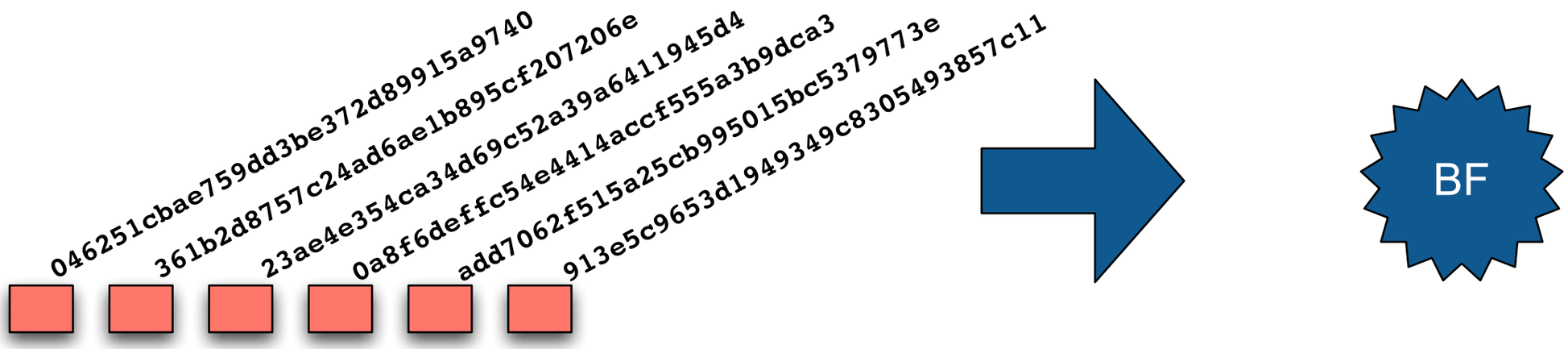
- Exfiltration of sensitive documents;
- Data Loss Detection; etc.
- Download from http://afflib.org/

Distinct Block Recognition can be used to find objectionable material.

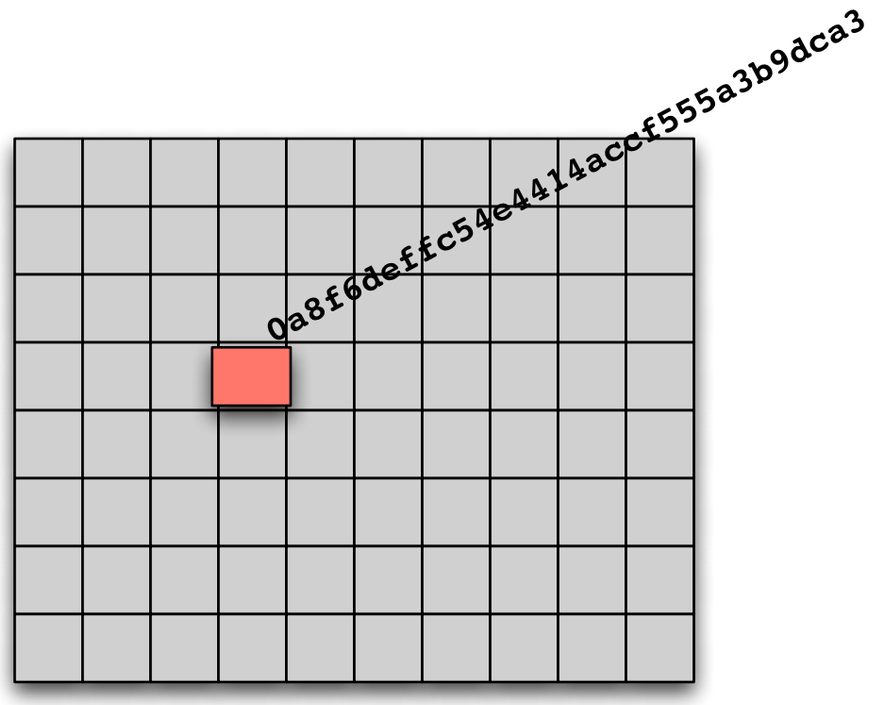
Currently objectionable materials are detected with hash sets.



With the block-based approach, each file is broken into blocks, and each block hash is put into a Bloom Filter:

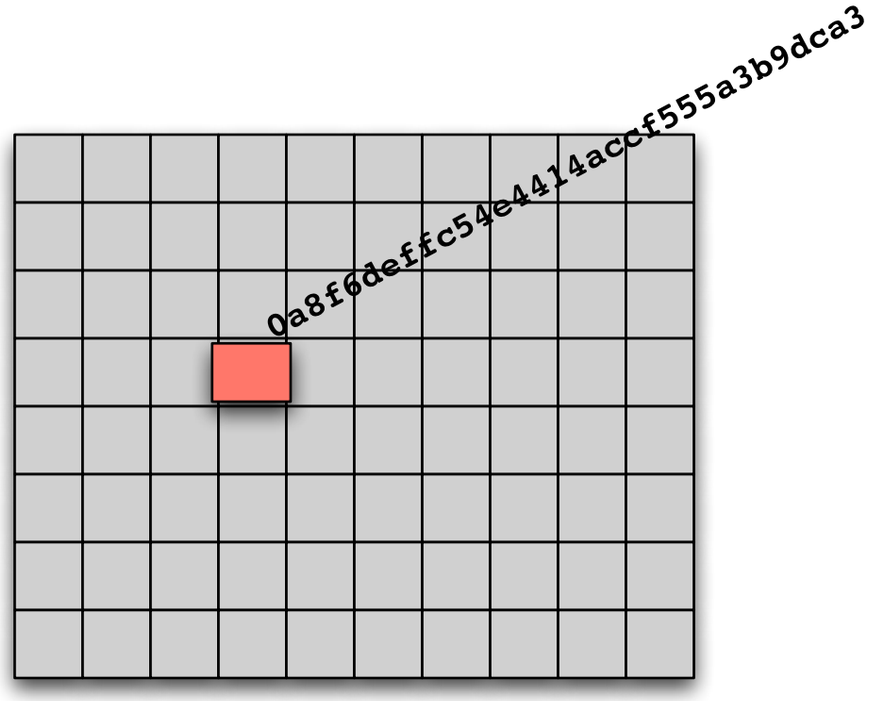


If a sector of an objectionable file is found on a drive...



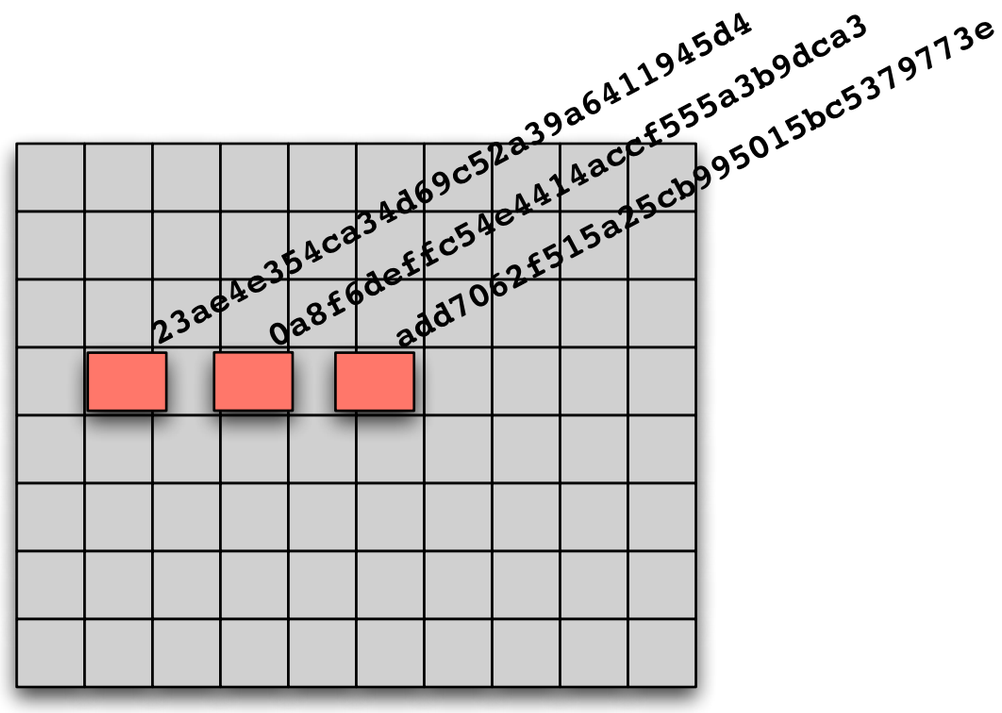
If a sector of an objectionable file is found on a drive...

Then either the entire file was once present…



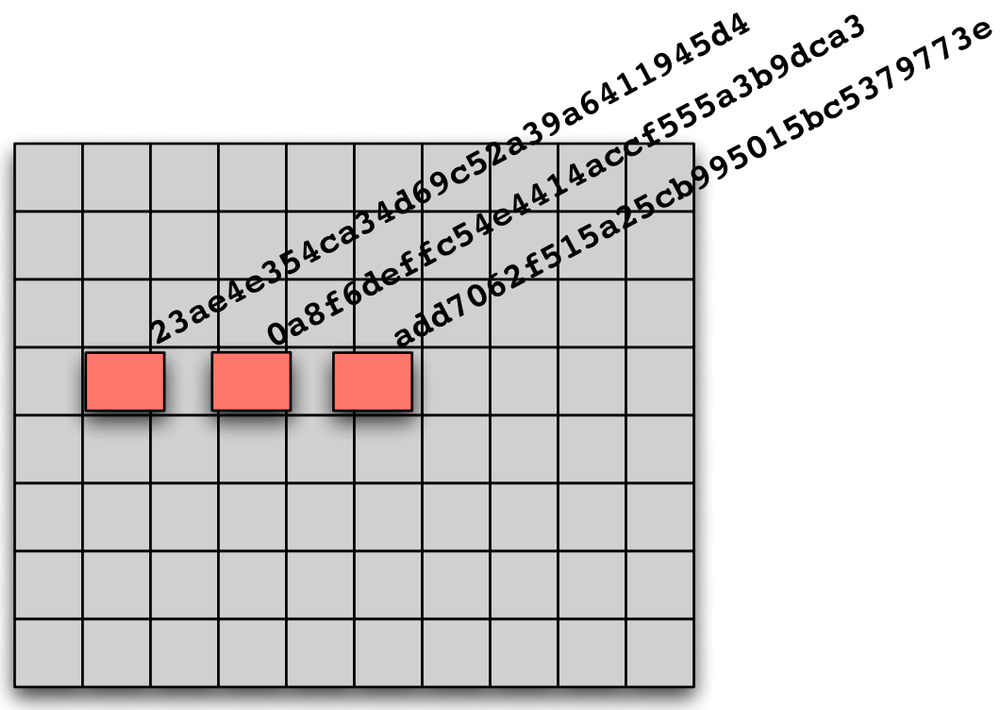
If a sector of an objectionable file is found on a drive...

Then either the entire file was once present…



If a sector of an objectionable file is found on a drive...

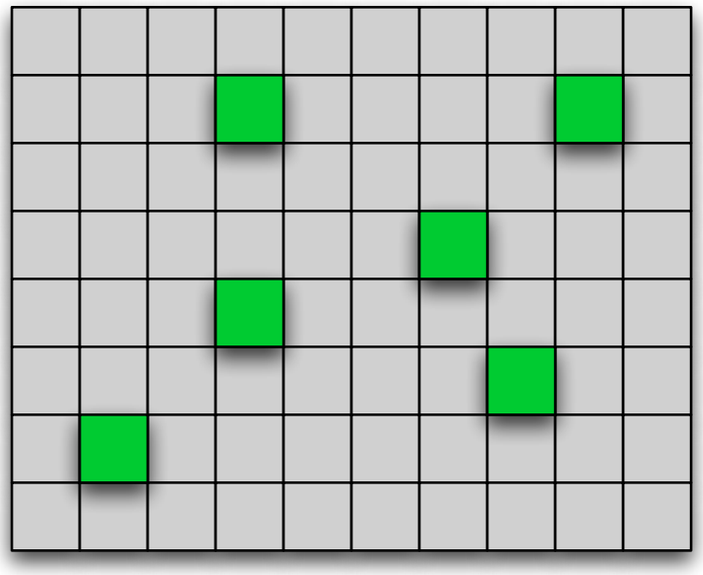
Then either the entire file was once present…



… or else the sector really isn't distinct.

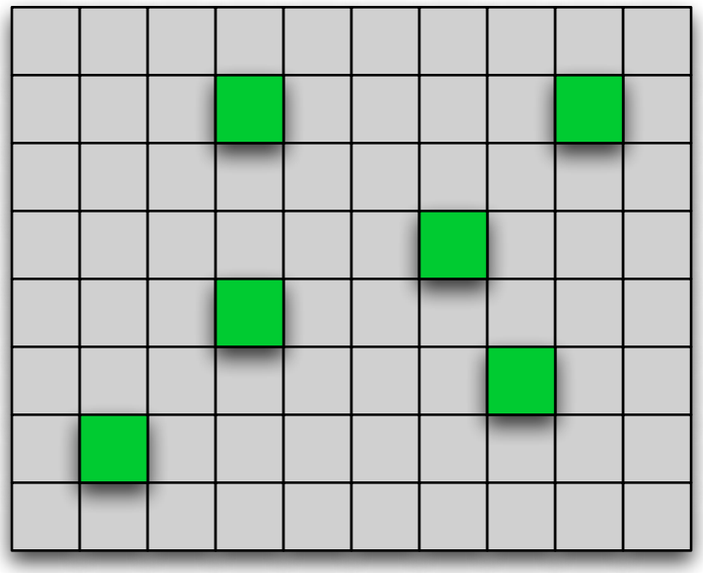
Random sampling can rapidly find the presence of objectionable material on a large storage device.

1TB drive = 2 billion 512-byte sectors.



Random sampling can rapidly find the presence of objectionable material on a large storage device.

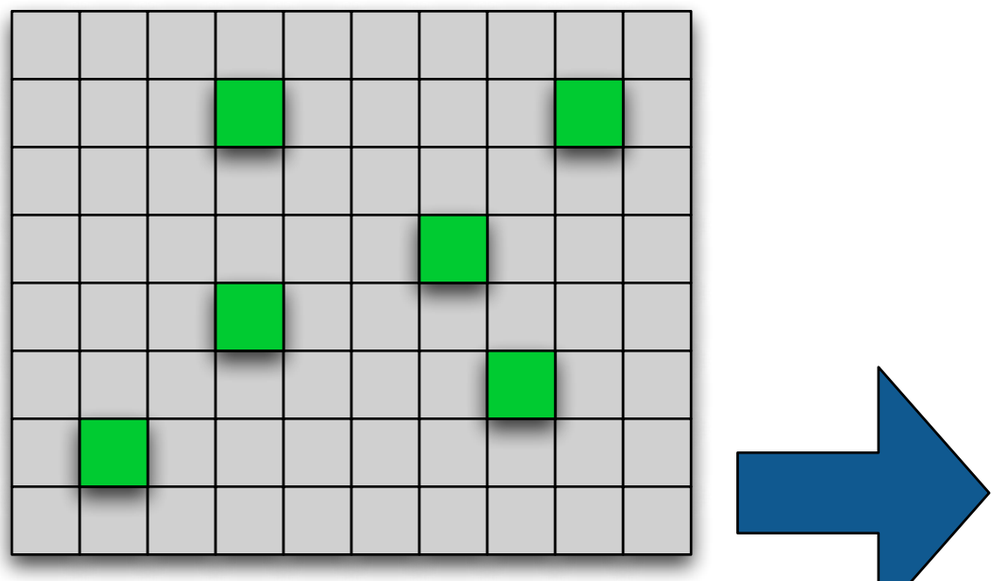
1TB drive = 2 billion 512-byte sectors.



We can pick random sectors, hash them, and probe the Bloom Filter.

Random sampling can rapidly find the presence of objectionable material on a large storage device.

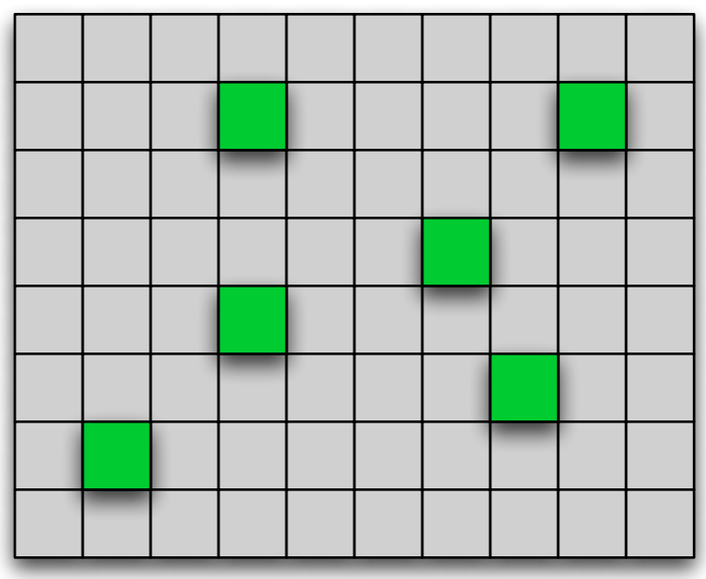
1TB drive = 2 billion 512-byte sectors.

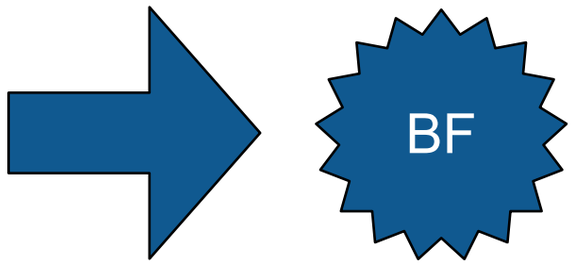


We can pick random sectors, hash them, and probe the Bloom Filter.

Random sampling can rapidly find the presence of objectionable material on a large storage device.

1TB drive = 2 billion 512-byte sectors.

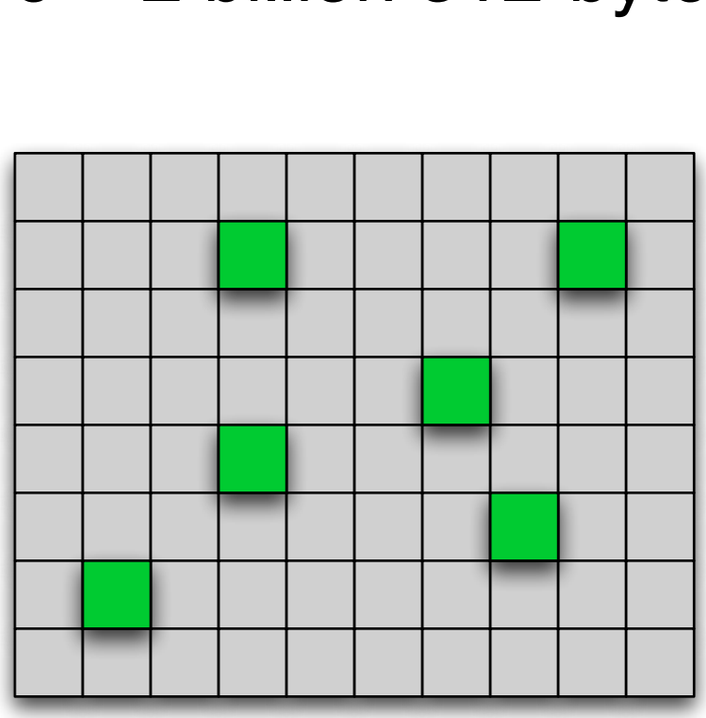


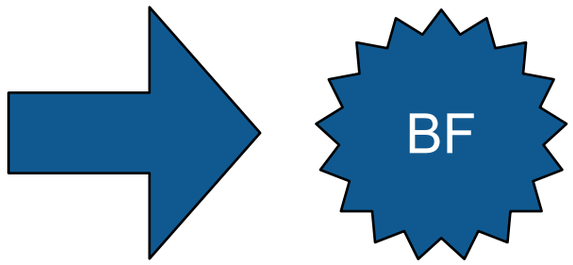


We can pick random sectors, hash them, and probe the Bloom Filter.

Random sampling can rapidly find the presence of objectionable material on a large storage device.

1TB drive = 2 billion 512-byte sectors.



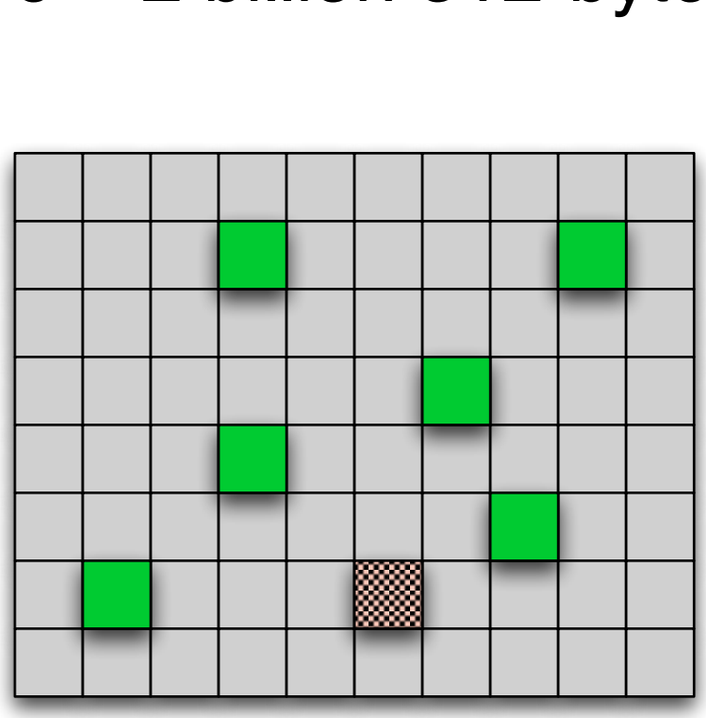


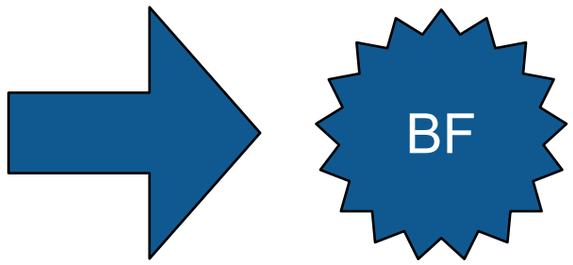
We can pick random sectors, hash them, and probe the Bloom Filter.

Finding a match indicates the presence of objectionable material.

Random sampling can rapidly find the presence of objectionable material on a large storage device.

1TB drive = 2 billion 512-byte sectors.





We can pick random sectors, hash them, and probe the Bloom Filter.

Finding a match indicates the presence of objectionable material.

Advantages of block recognition with Bloom Filters

Speed & Size:

- Can store billions of sector hashes in a 4GB object.
- False positive rate can be made very small.

Security:

- File corpus can't be reverse-engineered from BF
- BF can be encrypted for further security.

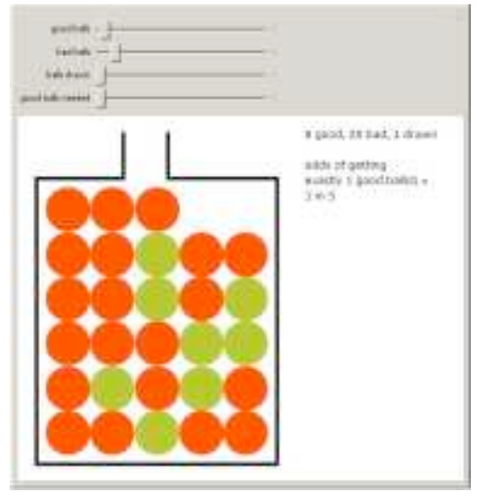
False positive rate:

- $m = 2^{32}$ $k = 4$ n=80 million $p < .0000266$ (512MiB BF)

The odds of finding the objectionable content depends on the amount of content and the # of sampled sectors.

Sectors on disk: 2,000,000,000 (1TB)

Sectors with bad content: 200,000 (100 MB)



Chose one sector. Odds of missing the data:

- (2,000,000,000 - 200,000) / (2,000,000,000) = 0.9999
- You are *very likely* to miss one of 200,000 sectors if you pick just one.

Chose two sectors. Odds of missing the data on both tries:

- 0.999 * (1,999,999,999-200,000) / (1,999,999,999) = .9998
- You are still *very likely* to miss one of 200,000 sectors if you pick two…
- … but a little less likely

Increasing # of samples *decreases* the odds of missing the data.

- The "Urn Problem" from statistics.

The more sectors picked, the less likely you are to miss *all* of the sectors that have objectionable content.

$$p = \prod_{i=1}^{n} \frac{((N-(i-1))-M)}{(N-(i-1))} \qquad (1)$$

Sampled sectors	Probability of not finding data
1	0.99999
100	0.99900
1000	0.99005
10,000	0.90484
100,000	0.36787
200,000	0.13532
300,000	0.04978
400,000	0.01831
500,000	0.00673

Table 1: Probability of not finding any of 10MB of data on a 1TB hard drive for a given number of randomly sampled sectors. Smaller probabilities indicate higher accuracy.

Non-null data		Probability of not finding data
Sectors	Bytes	with 10,000 sampled sectors
20,000	10 MB	0.90484
100,000	50 MB	0.60652
200,000	100 MB	0.36786
300,000	150 MB	0.22310
400,000	200 MB	0.13531
500,000	250 MB	0.08206
600,000	300 MB	0.04976
700,000	350 MB	0.03018
1,000,000	500 MB	0.00673

Table 2: Probability of not finding various amounts of data when sampling 10,000 disk sectors randomly. Smaller probabilities indicate higher accuracy.

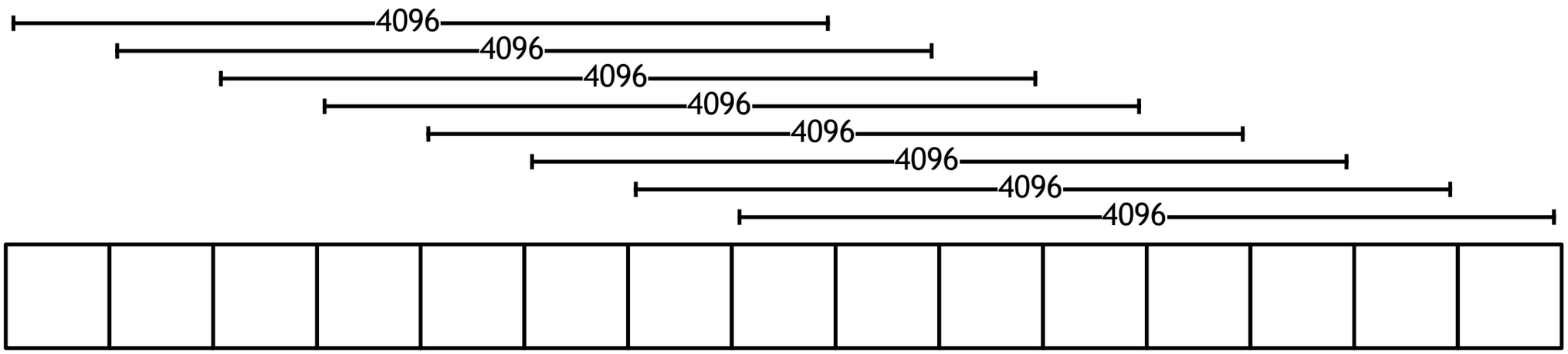
Increase efficiency with larger block size.

We use 4096-byte blocks instead of 512-byte sectors.

- Bloom Filter utilization is ⅛; we can hold 8x the number of hashes!
- Takes the same amount of time to read 4096 bytes as to read 512 bytes
- Most file fragments are larger than 4096 bytes.

But file systems do not align on 4096-byte boundaries!

- We read 15 512-byte blocks.
- Then we compute 8 different 4096-byte block hashes.
- Each one is checked in the Bloom Filter



(We can read 64K and trade off I/O speed for CPU speed.)

With this approach, we can scan a 1TB hard drive for 100MB of objectionable material in 2-5 minutes.

		
Minutes	208	5
Max Data Read	1 TB	24 GB

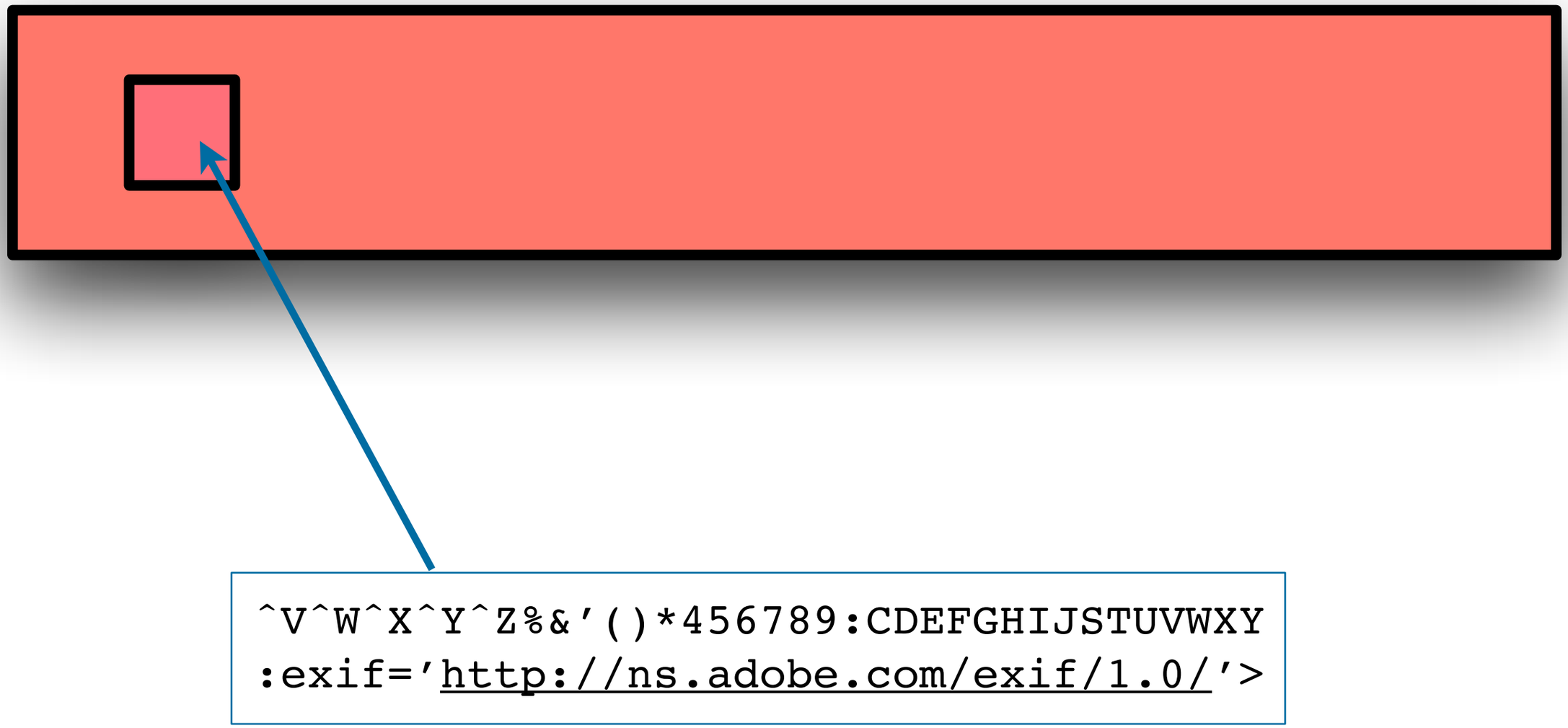
We lower the chance of missing the data to $p<0.001$

```
^V^W^X^Y^Z%&'()*456789:CDEFGHIJSTUVWXY
:exif='http://ns.adobe.com/exif/1.0/'>
```

```
New York, September 2008^M\223Security
Metrics: What can you test?\224, Veri
fy 2007 International Software Testing
Conference, Arlington, Virginia, Octo
ber 2007.^M\223Attacks and Countermeas
```

Part 2:
Fragment Type Discrimination

File fragment identification is a well-studied problem.



This fragment from a file is highly suggestive of a JPEG.

Prior academic work has stressed machine learning.

Algorithm:

- Collect *training* data.
- Extract a feature. (Typically byte-frequency distribution or n-grams.)
- Build a machine learning classifier. (KNN, SVN, etc.)
- Apply classifier to previously unseen *test data.*

Much of this work had problems:

- Many machine learning schemes were actually header/footer recognition.
  - *Well-known n-grams in headers dominated results.*
  - *Some techniques grew less accurate as analyzed more of a file!*
- Container File Complexity:
  - *Doesn't make sense to distinguish PDF from JPEG (if PDFs contain JPEGs.)*

We introduce three advances to this problem.

#1 — Rephrase problem as "discrimination," not recognition.

- Each discriminator reports likely presence or absence of a file type in [BUF]
- Thus, a fragment can be *both* JPEG and ZIP

#2 — Purpose-built functions.

- Develop specific discriminators for specific file types.
- Tune the features with *grid search*.

We've created three discriminators:

- JPEG discriminator
- MP3 discriminator
- Huffman-Coded Discriminator

JPEGs:
Most FFs are followed by 00 due to “byte stuffing.”

Terminal — emacs — 70×27

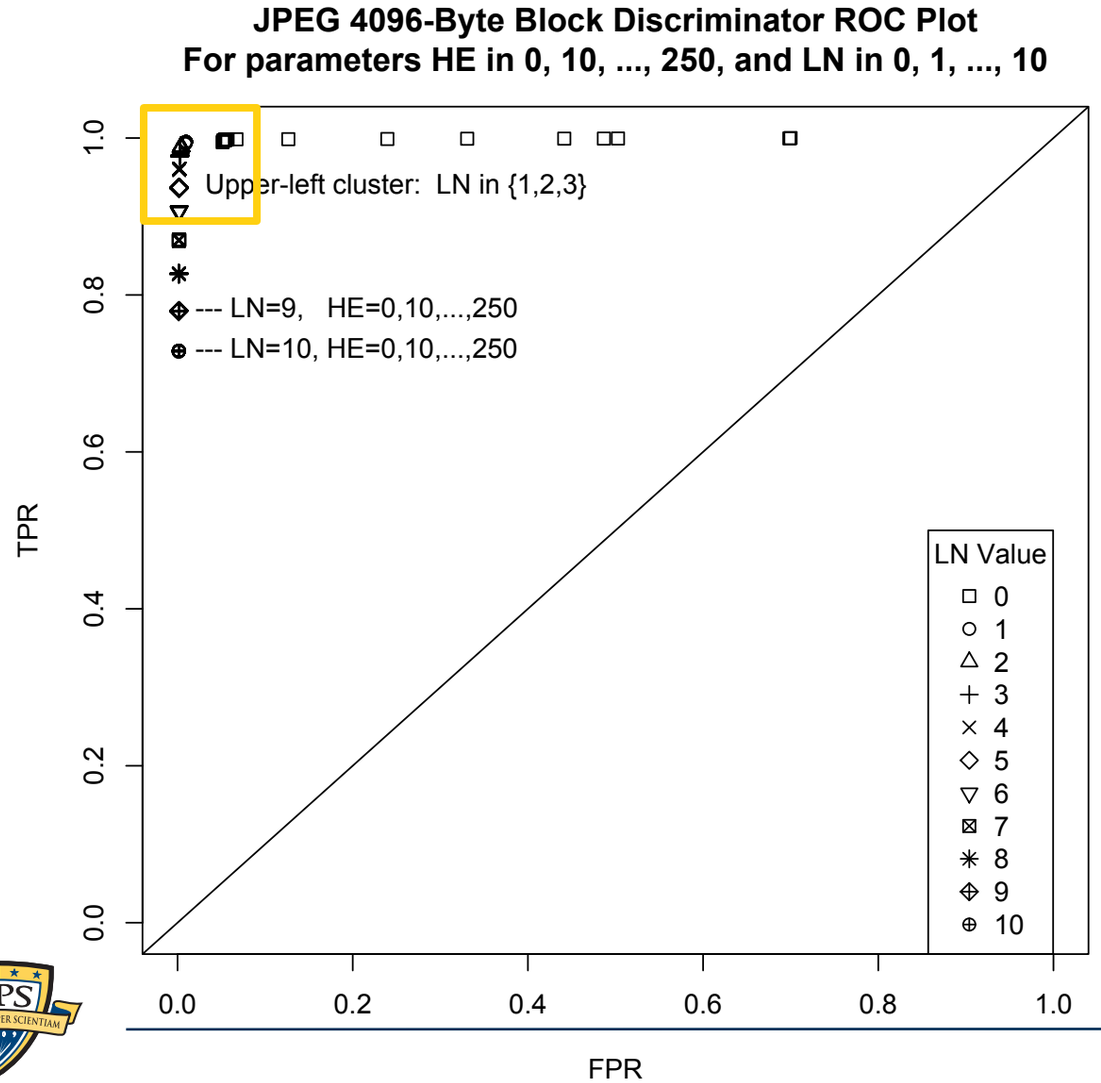
```
87654321  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789abcdef
00006a20: 6b4c cd62 54a0 b214 52ff 0074 ba4f 4622  kL.bT...R..t.OF"
00006a30: d1bf bf4c 67c4 aa2a 4a91 036f f3b3 7ddc  ...Lg..*J..o..}.
00006a40: 98d5 f078 7f28 d327 340d a2f2 c916 da4f  ...x.(.'4......O
00006a50: aefa 0cbc e9a6 a580 4b20 952c 17d2 7a09  ........K .,..z.
00006a60: 377b 097c 7395 b7e4 c661 730c 447f 9b5a  7{.|s....as.D..Z
00006a70: 7675 e9d1 e14a 81a8 26a2 2948 93bc 4749  vu...J..&.)H..GI
00006a80: 94fd 8d3f fce2 4a13 e529 2b64 8f31 b961  ...?..J..)+d.1.a
00006a90: 368b 827f 677e 7a64 9a62 60f9 9826 c4e0  6...g~zd.b`..&..
00006aa0: b65e bfa9 97fc 5aa9 6a94 626a 602e 4ac7  .^....Z.j.bj`.J.
00006ab0: 9cb1 0311 3d9d 3e33 e941 482e caf2 8676  ....=.>3.AH....v
00006ac0: 240d 43ae ce27 a39e 98d3 f14a 6a23 116a  $.C..'.....Jj#.j
00006ad0: af80 dffc 1867 58be 0eaa a9a9 b29f 3331  .....gX.......31
00006ae0: 20b1 9da6 46d3 eb6d 4846 774c 1870 4c98   ...F..mHFwL.pL.
00006af0: 60fd 0f7d 8382 2f04 e2a9 e314 d982 5947  `..}../.......YG
00006b00: 11ef bef1 7df3 9c6a f0ab 289d 2d99 b6fb  ....}..j..(.-...
00006b10: ff00 9b6d a903 35aa 8b3c 8014 9240 6006  ...m..5..<...@`.
00006b20: cece 5c3b 9f4d af7f 8934 44d8 bd10 4044  ..\;.M...4D...@D
00006b30: 0124 bd6e b80d 61ff 001d 388c 8b74 aaef  .$.n..a...8..t..
00006b40: 32f9 3010 c487 a6fa 681a 4a23 4a8a 5441  2.0.....h.J#J.TA
00006b50: 5b00 3e19 7762 443b 1376 07a1 96c6 5553  [.>.wbD;.v....US
00006b60: 4bbc 285a 7e57 393d e521 e8ce b48a c99a  K.(Z~W9=.!......
00006b70: 69aa 9129 bdab 0361 ba5b 6c36 418d 3e85  i..)...a.[l6A.>.
00006b80: 2c2b 5fc4 55c2 162e 0a60 1209 2144 5887  ,+_.U....`..!DX.
00006b90: 20a4 3055 81c3 a566 799d 84b2 1493 28ac   .0U...fy.....(.
-:---F1  iStock Privacy.jpg    8% L1714  (Hexl)----8:37PM-------------
Mark saved where search started
```

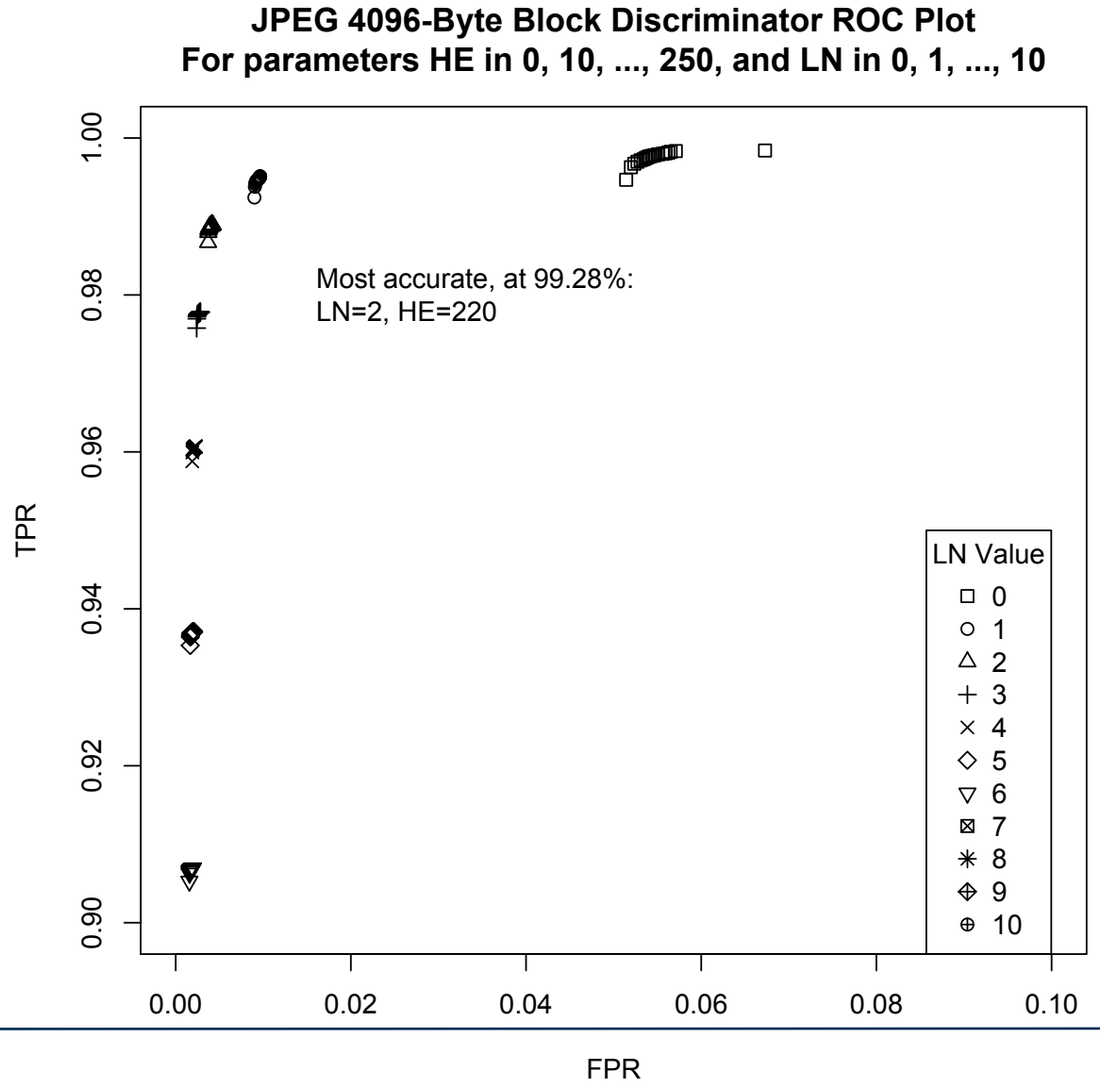
Our JPEG discriminator counts the number of FF00s.

Two tunable parameters:

- High Entropy (HE) - The minimum number of distinct byte values in the 4096-byte buffer.
- Low FF00 N-grams (LN) - The minimum number of <FF><00> byte pairs

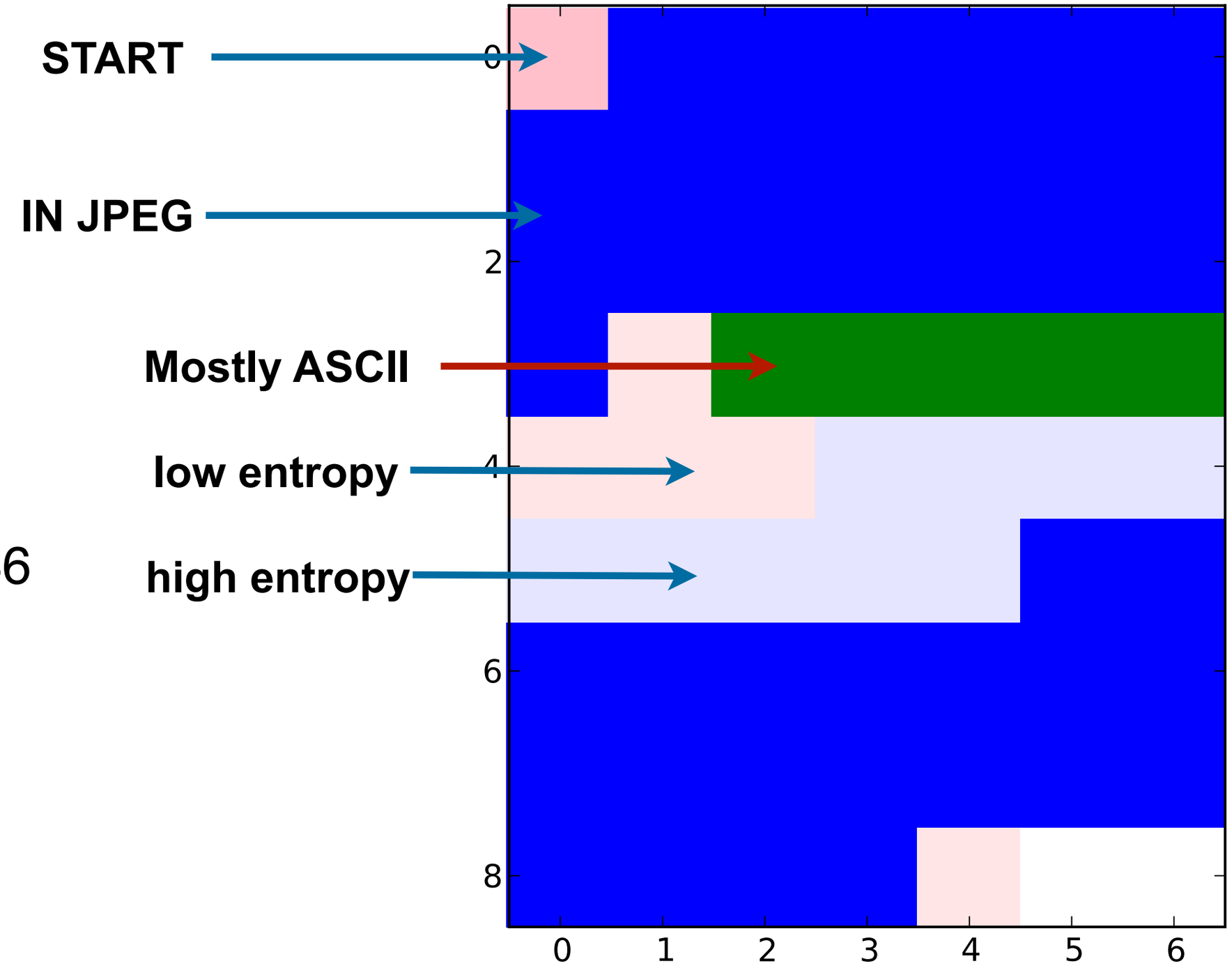
We perform a grid search with a variety of possible values.





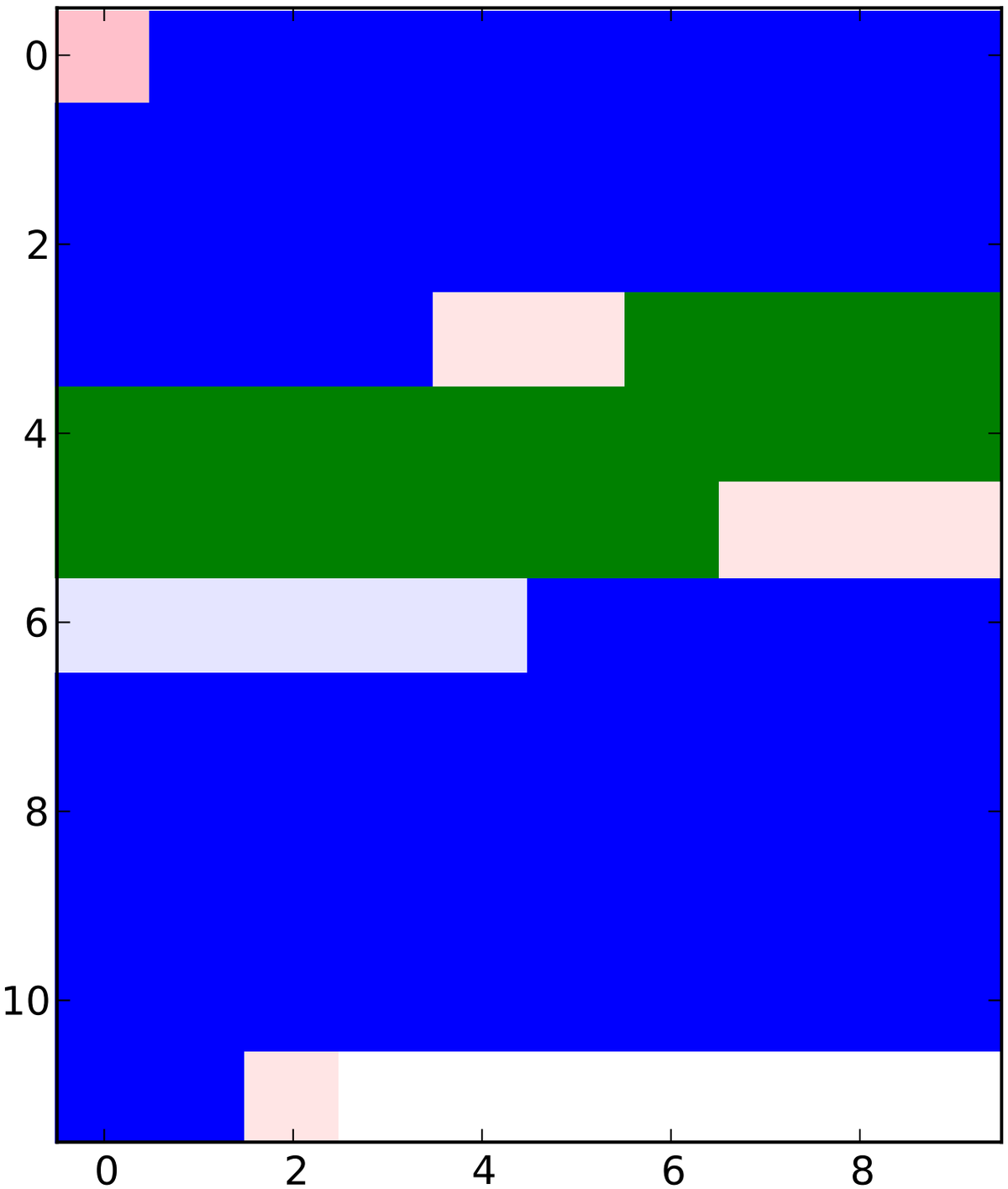
These maps of JPEG blocks show the accuracy.
000109.jpg

Bytes: 31,046



000897.jpg

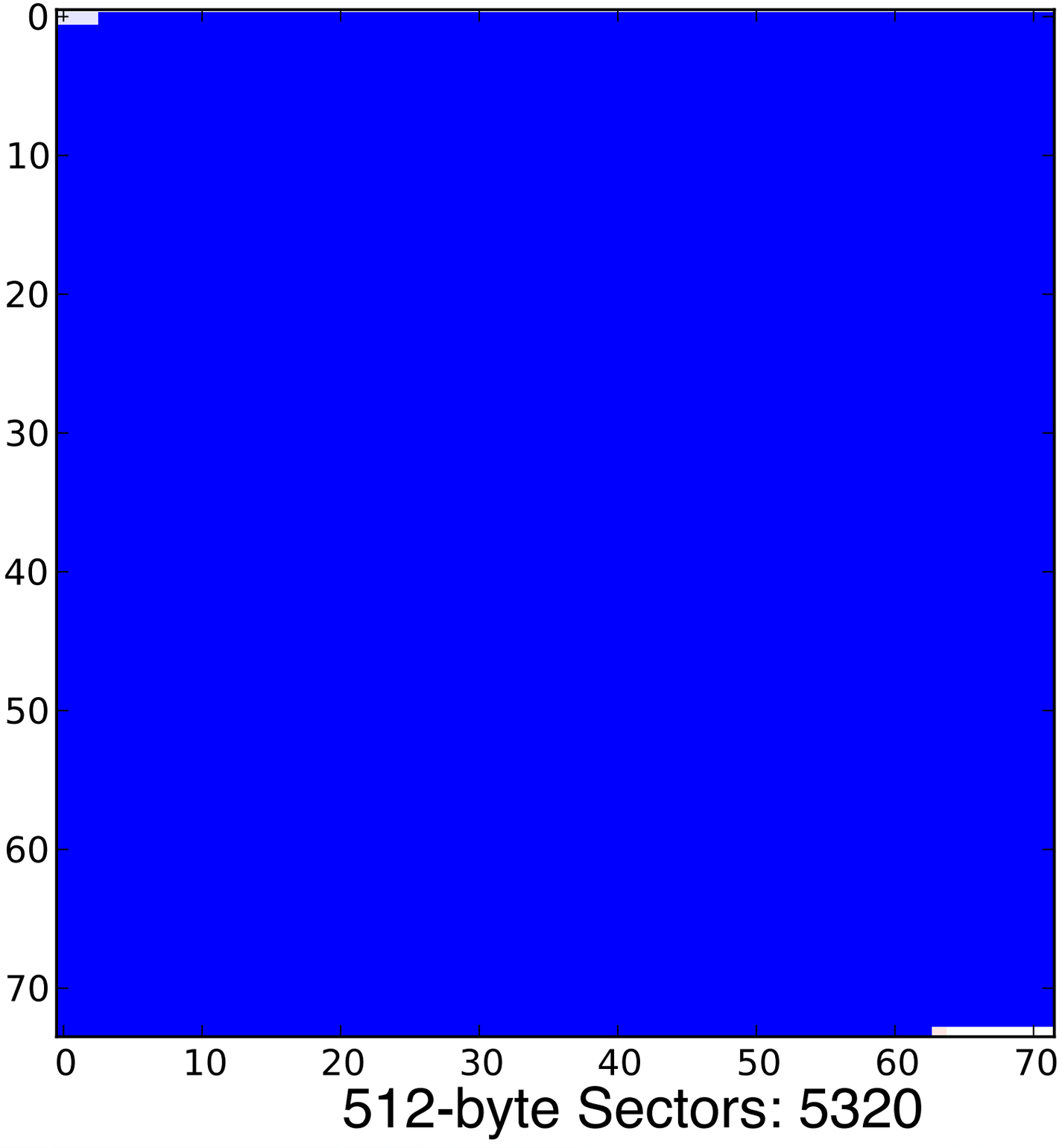
Bytes: 57,596



512-byte Sectors: 113

000888.pdf

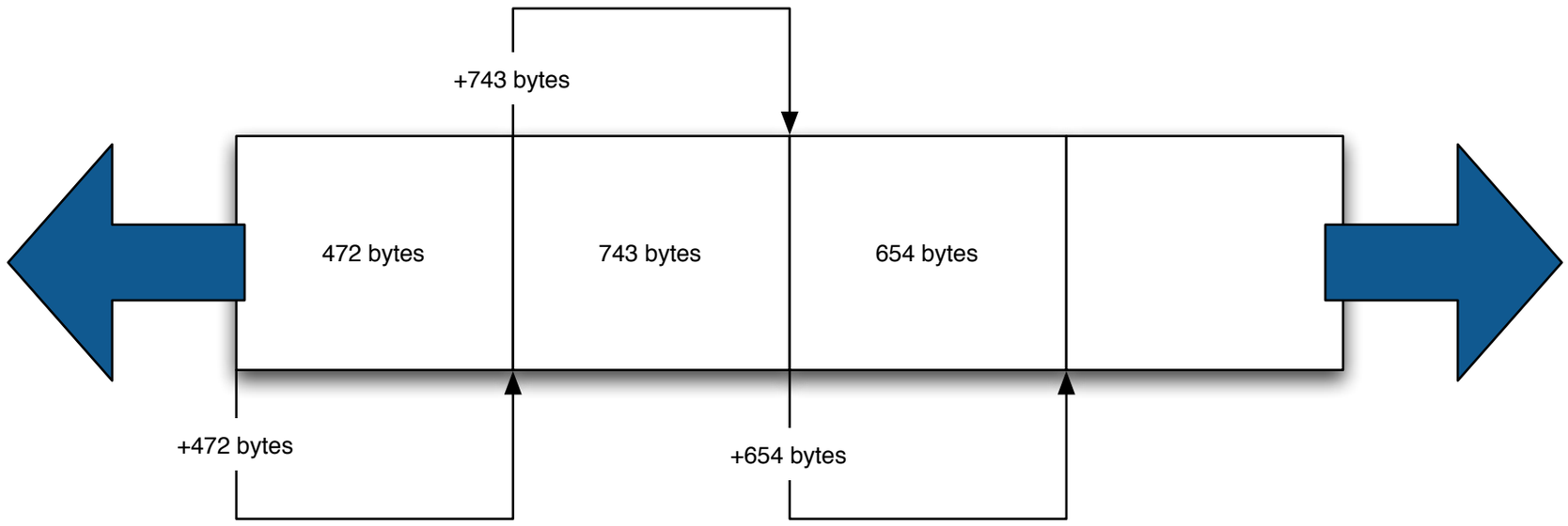
Bytes: 2,723,425



The MPEG classifier uses the frame chaining approach.

Each frame has a header and a length.

Find a header, read the length, look for the next header.



Our MP3 discriminator:

- Frame header starts with a string of 11 sync bits
- Sanity-check bit rate, sample rate and padding flag.
- FrameSize = 144 x BitRate / (SampleRate + Padding)
- Skip to next Frame and repeat.
- Chain Length (CL) = 4 produced 99.56% accuracy with 4K buffer.

The Huffman-Encoding detector is based on autocorrelation.

Huffman-coding is a variable-length bit-level code.

- Symbols may be any number of bits.
- More frequent symbols are shorter.
- Hard to distinguish from random data.

Hypothesis:

- Common symbols will occasionally line up in successive bytes.

"alan" = `01011001 01000100`

- If we perform an *autocorrelation*, common symbols will self-align more often than by chance, producing more 0s:

```
  01011001
⊕ 01000100
==========
  00011101
```

- With random (or encrypted) data, autocorrelation should not significantly change the statistics.

Char	Freq	Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000

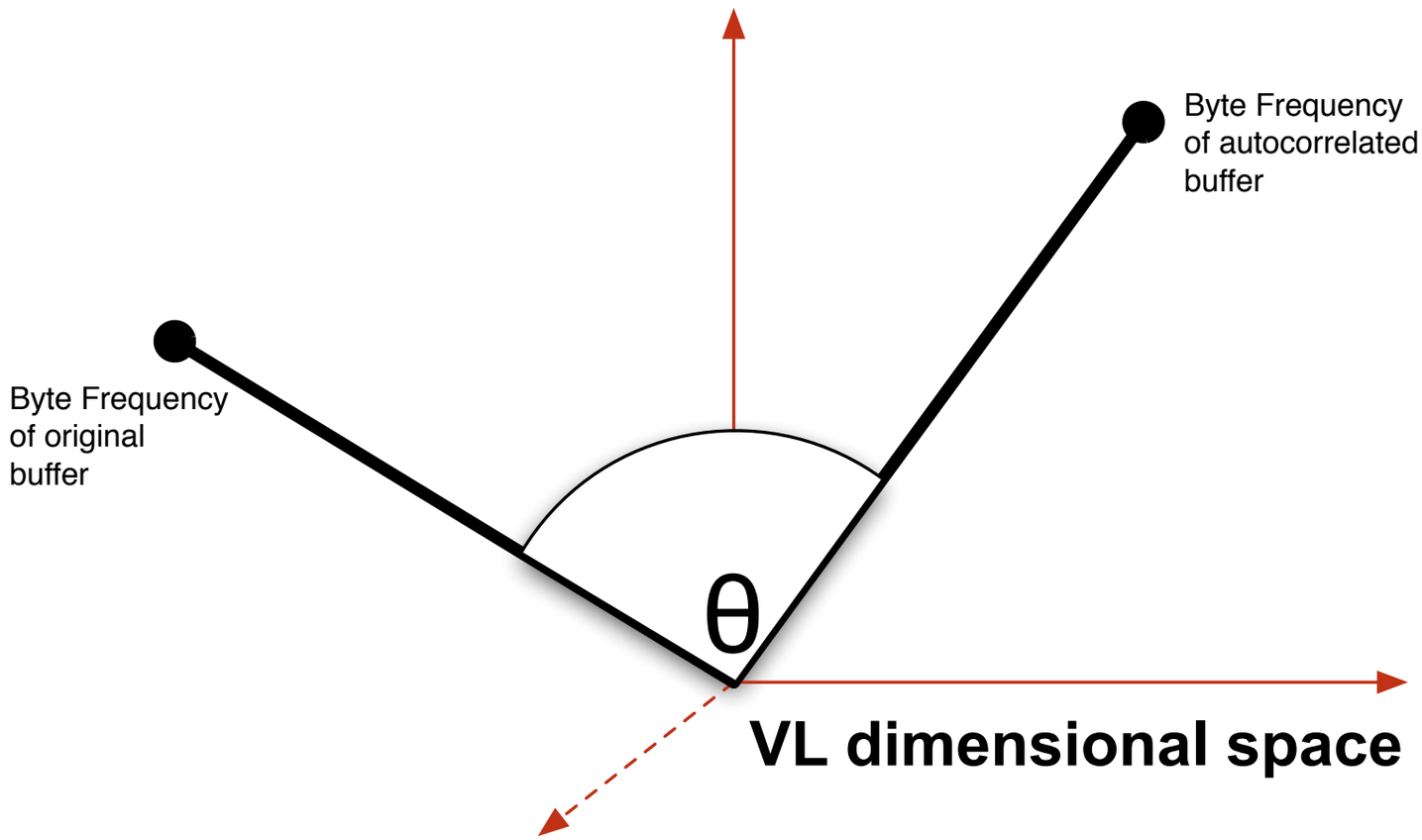
Our approach computes the cosine similarity of the byte-frequency distribution in multi-dimensional space

Two tunable parameters:

- VL - Vector Length - The number of dimensions to consider (this is VL=3)
- MCV - Minimum Cosine Value - if cos(θ) < MCV, data is deemed to be Huffman.

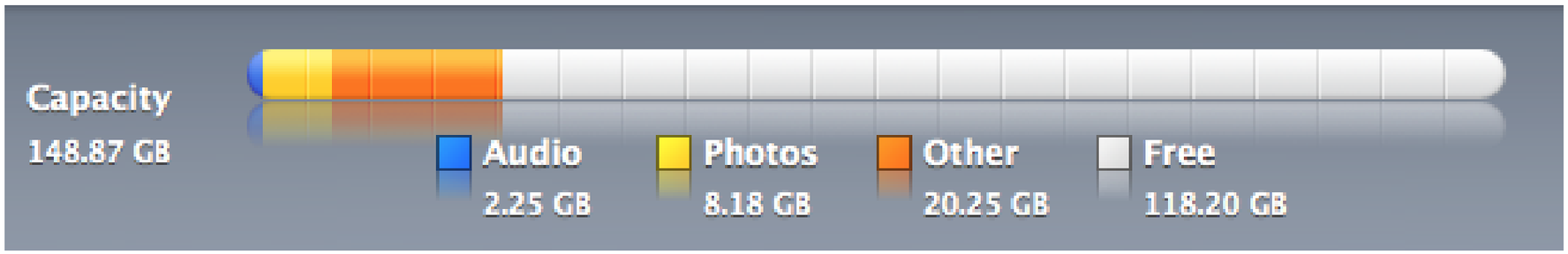
Best Results:

- 16KiB-block discriminator:
- 66.6% accurate,
- TPR 48.0%,
- FPR 0.450%.
- VL=250,
- MCV=0.9996391245556134.



Combine random sampling with sector discrimination to obtain the forensic contents of a storage device.

Our numbers from sampling are similar to those reported by iTunes.



Audio Data reported by iTunes:	2.25 GiB	2.42 GB	
MP3 files reported by file system:		2.39 GB	
Estimated MP3 usage with random sampling :		2.49 GB	10,000 random samples
		2.71 GB	5,000 random samples

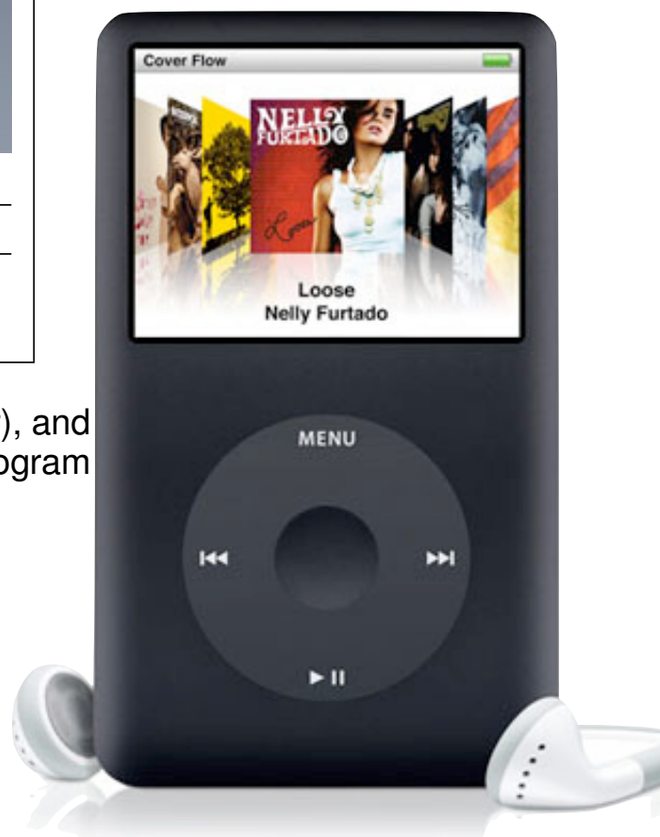


Figure 1: Usage of a 160GB iPod reported by iTunes 8.2.1 (6) (top), as reported by the file system (bottom center), and as computing with random sampling (bottom right). Note that iTunes usage actually in GiB, even though the program displays the “GB” label.

We could accurately determine:

- Amount of free space
- Amount of JPEG
- Amount of MPEG

http://www.uscg.mil/history/articles/LessonsLearnedHomePage.asp

Lessons Learned, Conclusions & Future Work

Lessons Learned

frag_find

- C++ implementation was 3x faster than Java implementation.
- Probably because OpenSSL is 3x faster than Java Crypto
- STL map[] class worked quite well.

Databases:

- Bloom Filters work well to pre-filter database queries.
- SQL servers can be easily parallelized using "prefix routing"

Hash codes:

- Use binary representations whenever possible.
- base64 coding would make sense.


NPS
PRAESTANTIA PER SCIENTIAM


NPS
PRAESTANTIA PER SCIENTIAM
1909