Hash-Based Carving: Searching media for complete files and fragments with sector hashing and hashdb

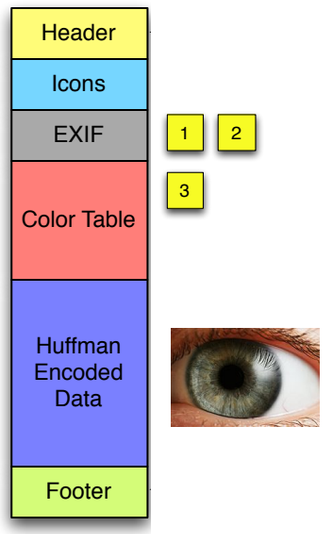
Simson L. Garfinkel
National Institute of Standards and Technology

Michael McCarrin
[Bruce Allen]
Naval Postgraduate School

Tuesday, August 11, 2015
Session 4: Computational Forensics
2:15pm

In this talk, we present hash-based carving. (Finally!)

Hash based carving: the big idea



Work to date

Our toolchain: hashdb, bulk_extractor, report_identified_runs.py

Initial experience running this at scale

0000107.jpg:

Hash-based carving: The Big Idea

Every file has a unique hash

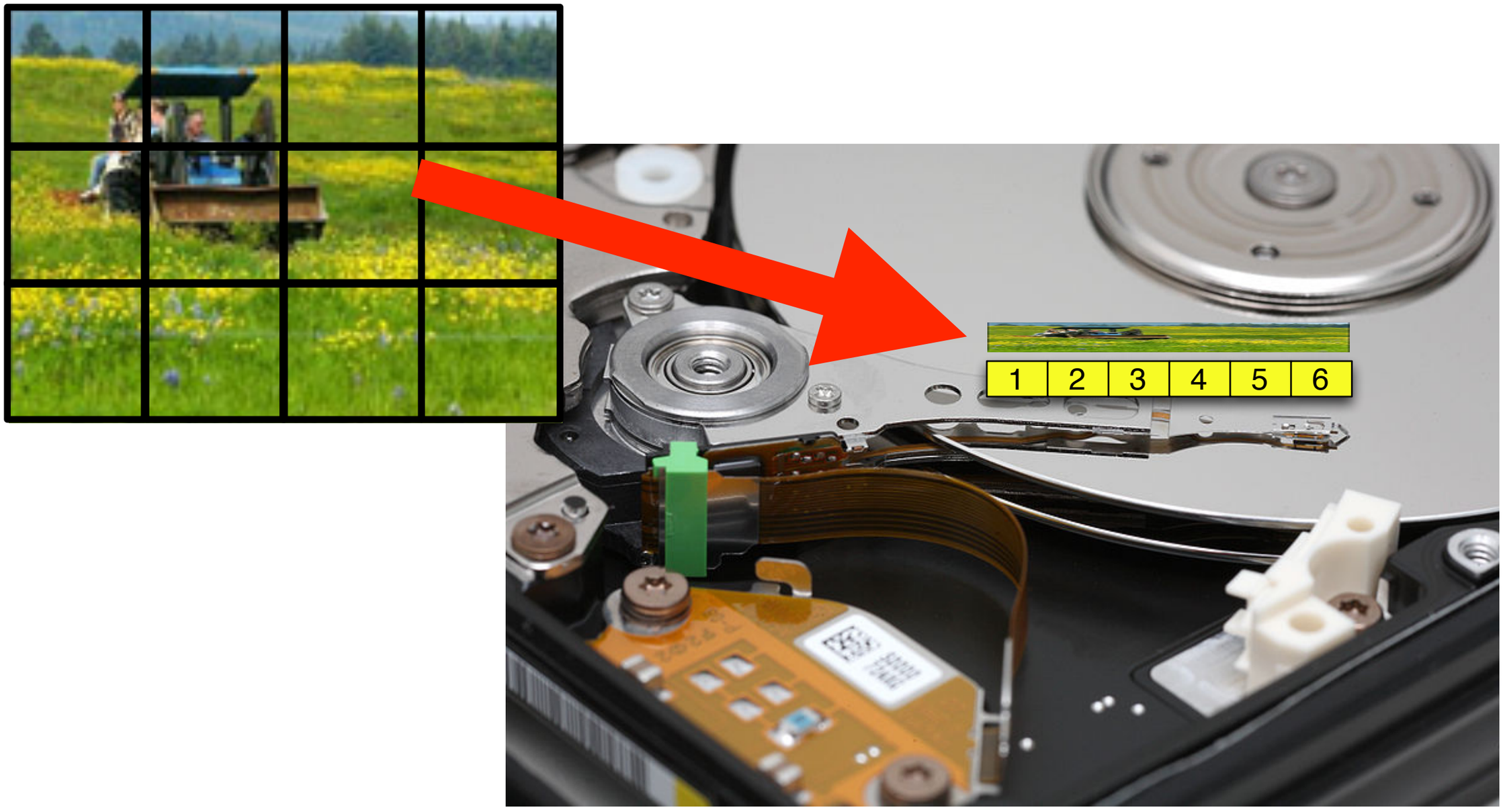
41,572 bytes

`c996fe19c45bc19961d2301f47cabaa6`

Every file can be viewed as a sequence of blocks.

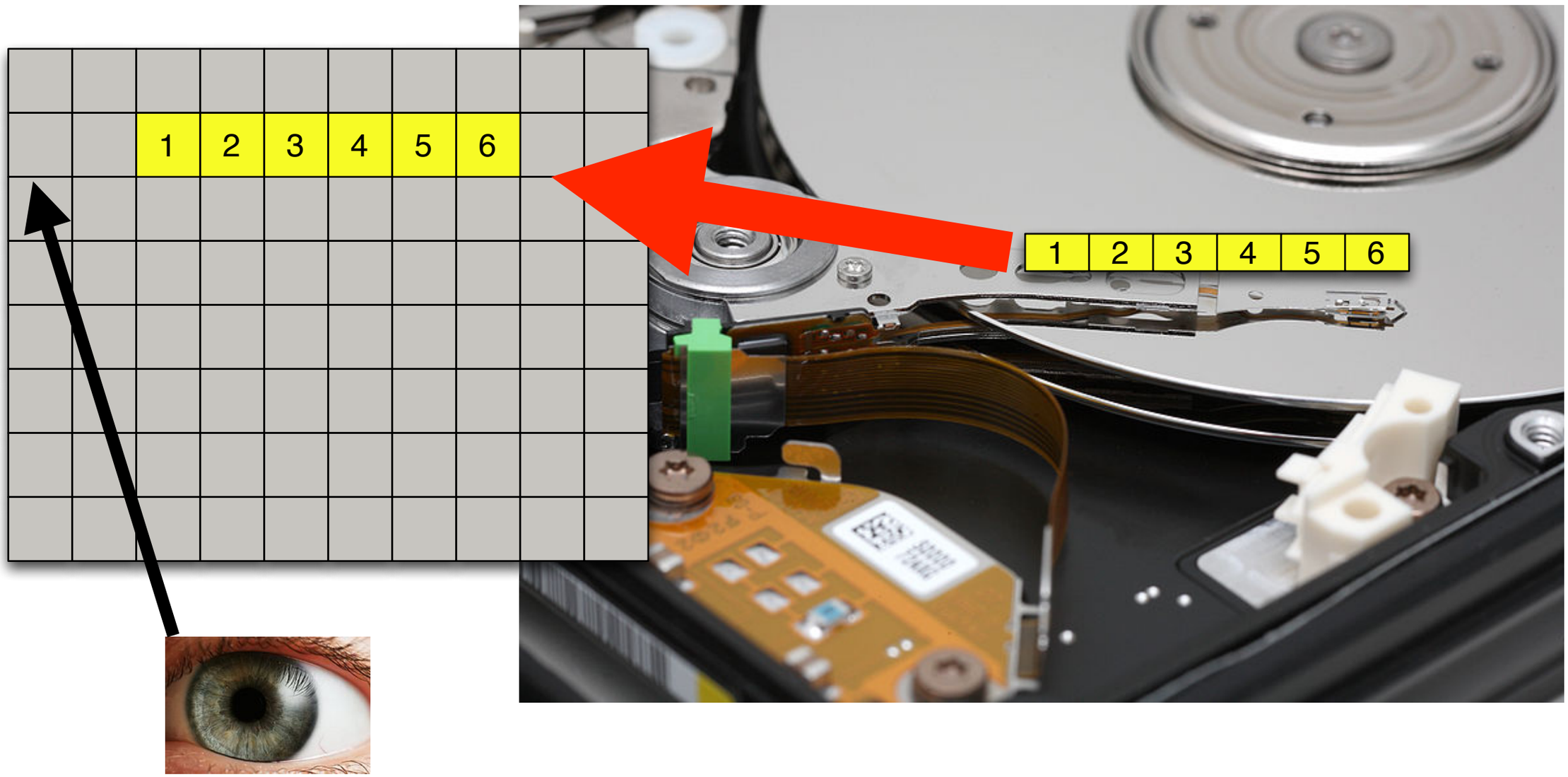
41,572 bytes ÷ 512 bytes/block = 81 blocks + 100 bytes
= 82 blocks
(w/ zero padding)

When a file is stored on a drive, the file's *blocks* are stored in disk *sectors*.

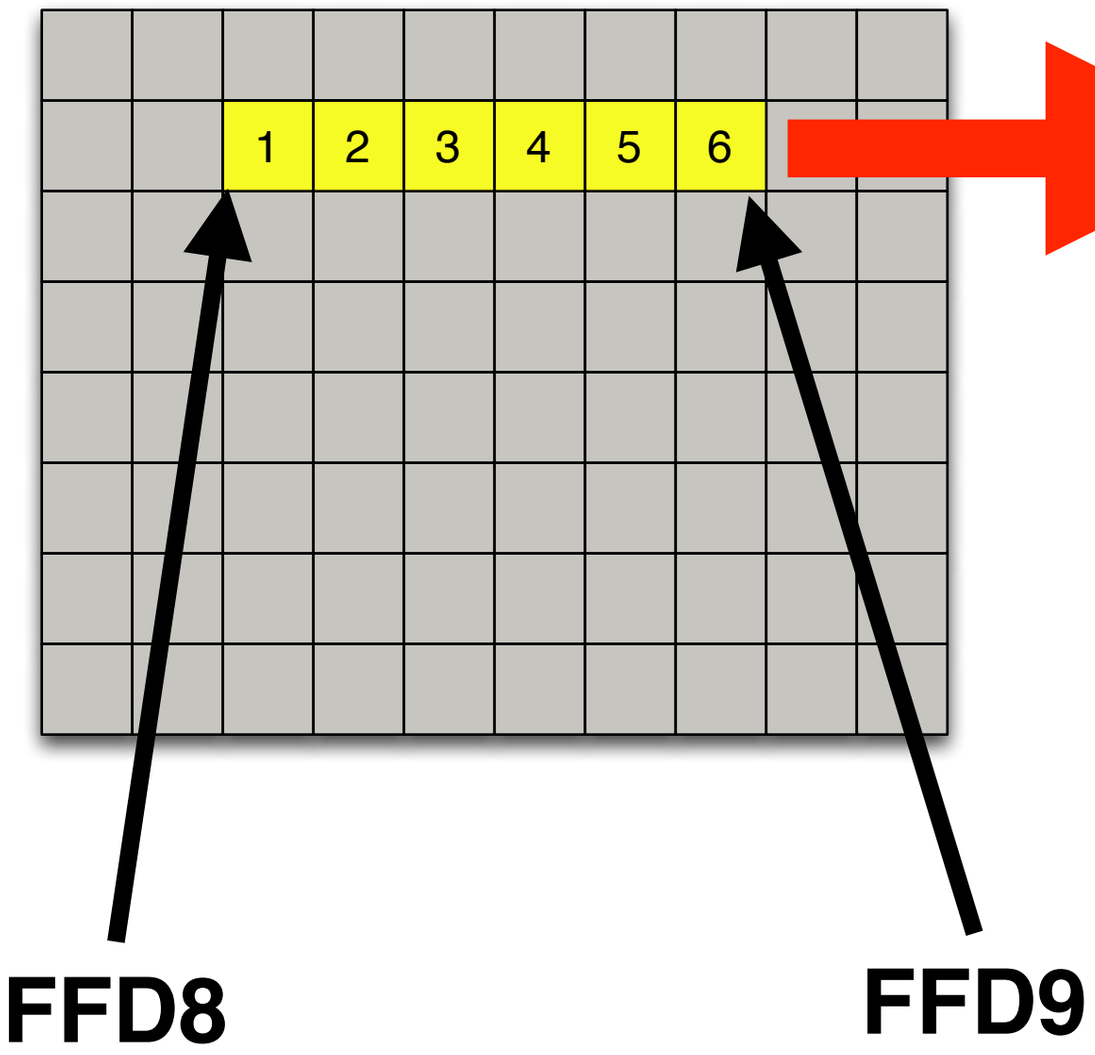


All modern file systems align files (> 4KiB) on sector boundaries.

Traditional carving examines the sectors on the drive.



Any sequence of sectors with a specific header and footer are stored in a file.



Conventional watch lists are based on file hashes.

Software alerts if the hash matches the watch list.

`c996fe19c45bc19961d2301f47cabaa6`

Stolen images

Tractor	`c996fe19c45bc19961d2301f47cabaa6`
Cow	`029bab60cfdc5685b8e6334a35df42bc`
Dog	`ee59a3677ef302bb3c0b816e00063559`
Cat	`253fbeb0834b45e382af5862376a1778`

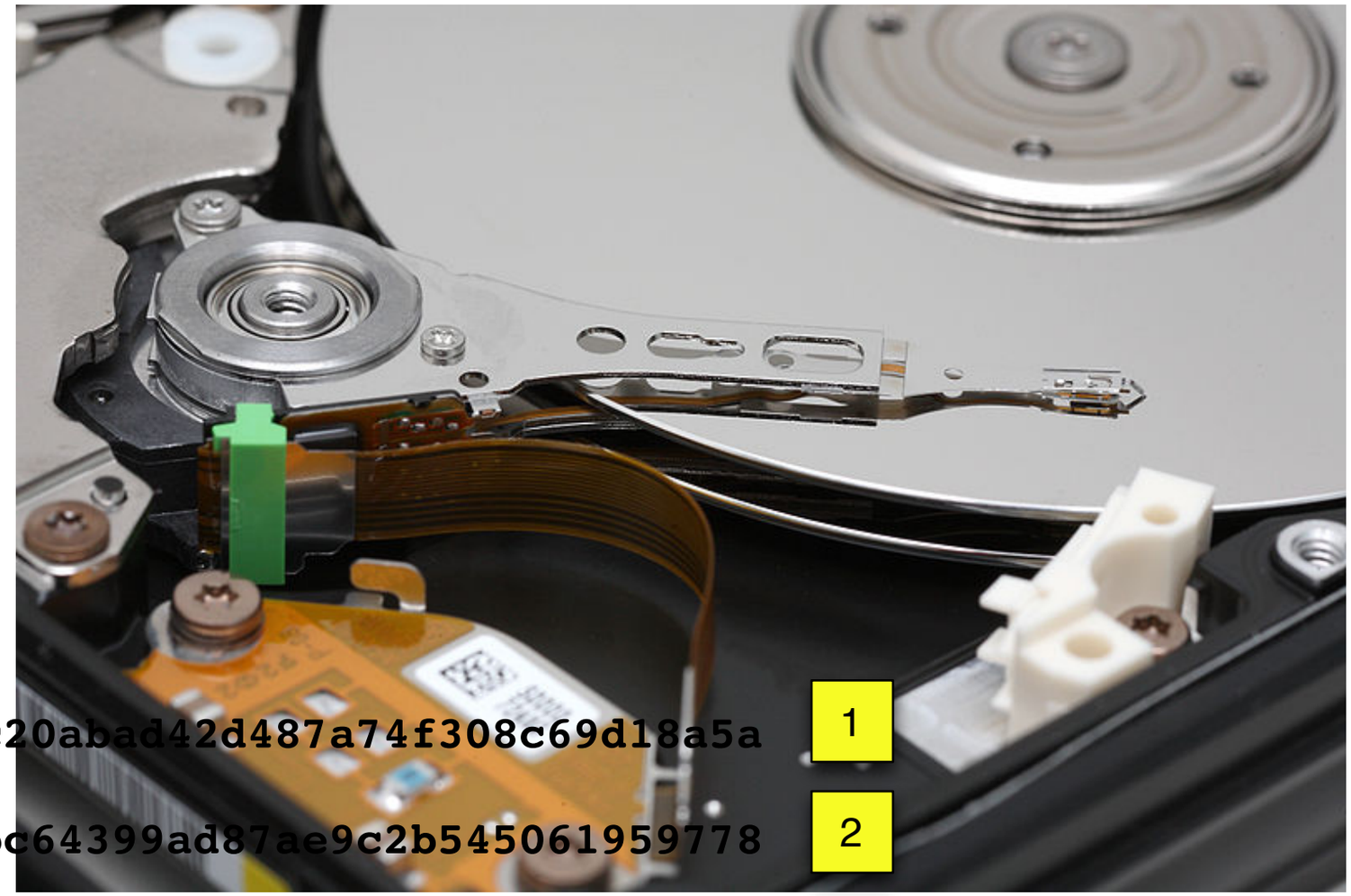
Hash-based carving applies hashing to each file block.

dc0c20abad42d[illegible]7a74f30[illegible]c69d18a5a 1

9e7bc64399ad87ae9c2b54[illegible]061959778 2

6e7f3577b100f9ec7fae18438fd5b047 3

Hash-based carving applies hashing to each file block.

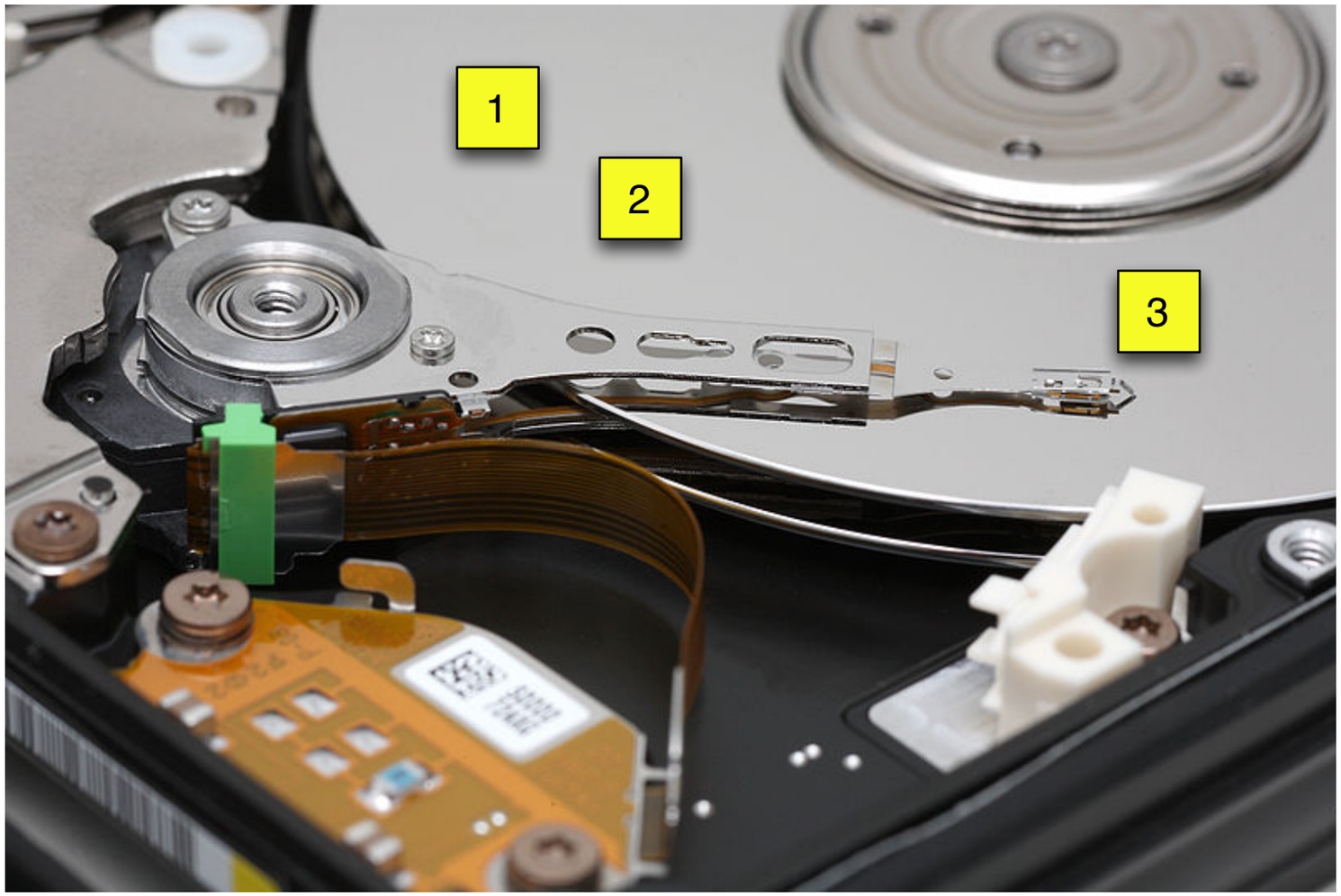


1	dc0c20abad42d487a74f308c69d18a5a	1
2	9e7bc64399ad87ae9c2b545061959778	2
3	6e7f3577b100f9ec7fae18438fd5b047	3

In theory, hash-based carving lets us find file *fragments*.

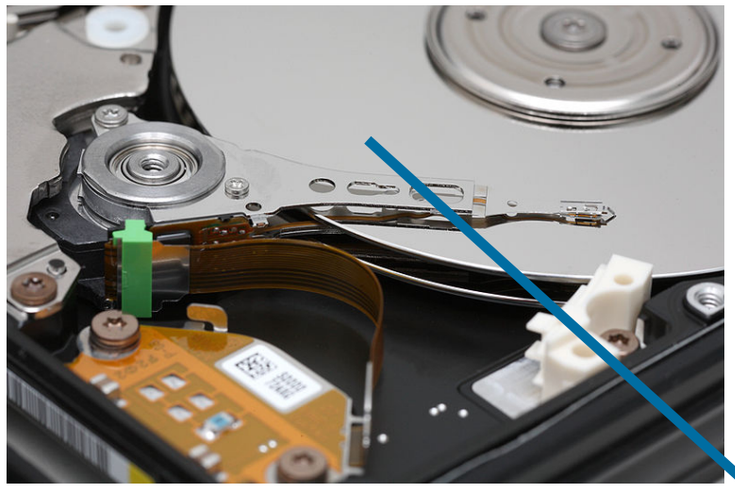
Fragments might come from:

- Fragmented files
- Files deleted and partially overwritten



Hash-based carving should be simple.

1. Hash every sector of the drive



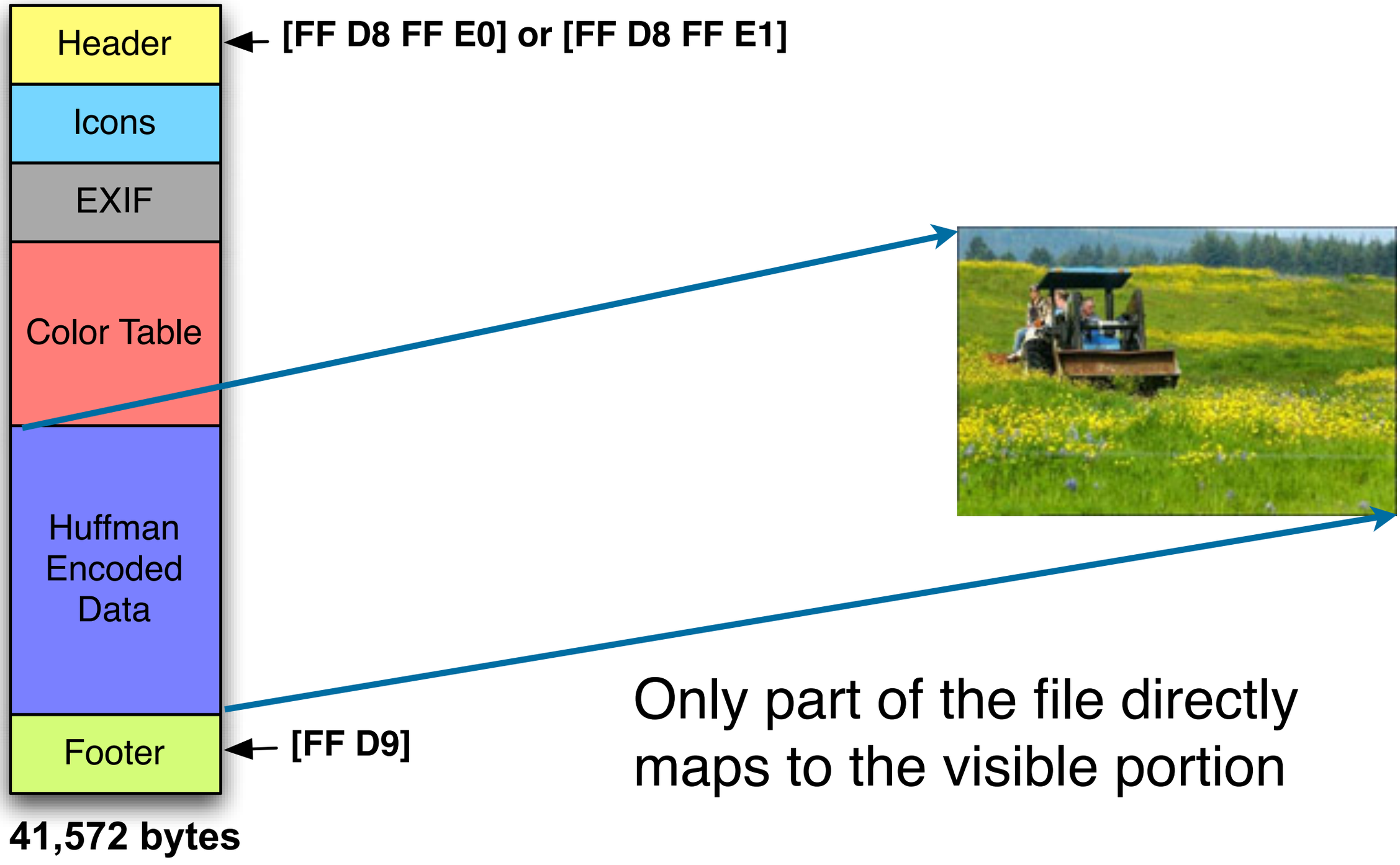
2. Hash every sector of the target files

3. Look for matches

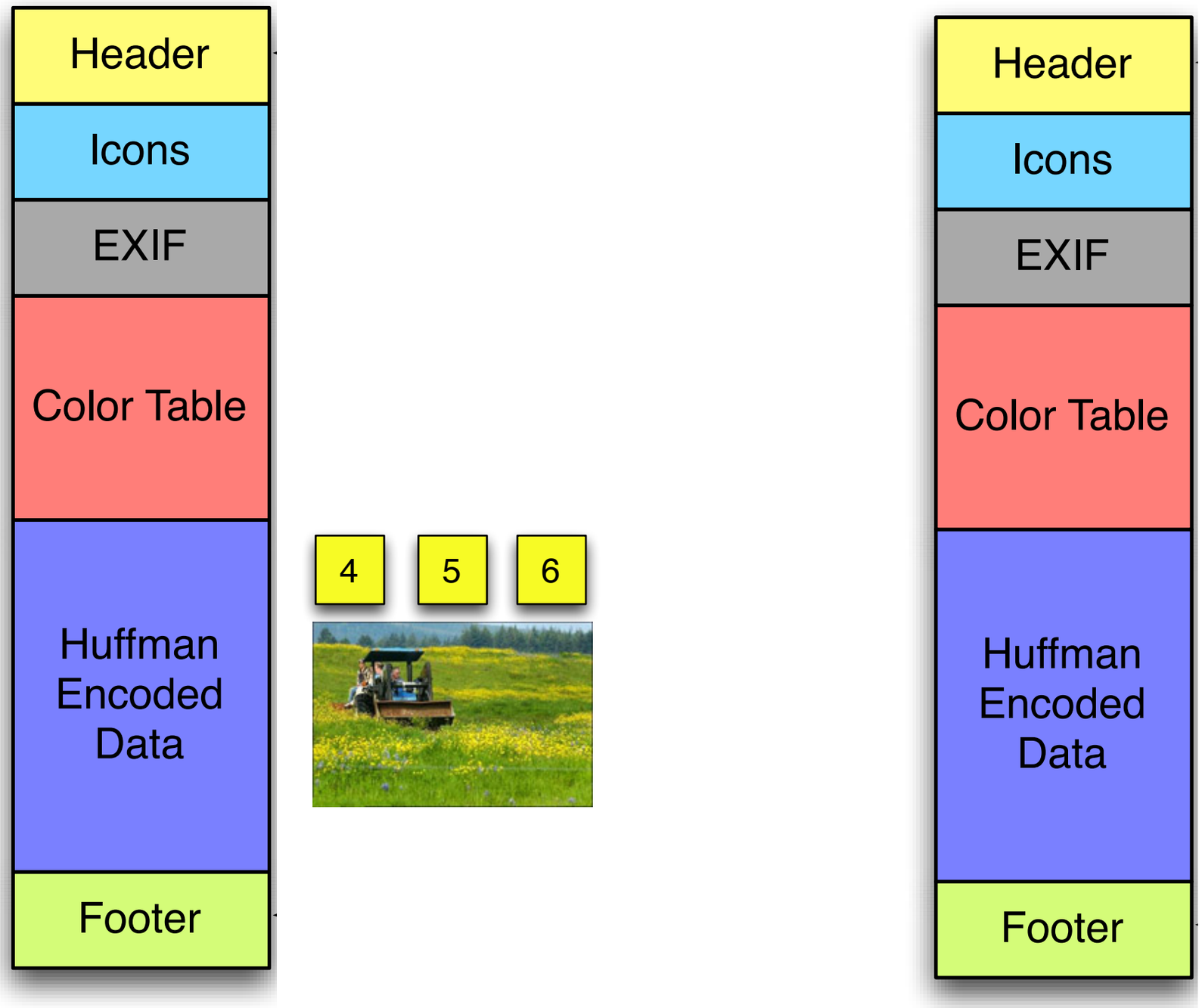
Block #	Byte Range	MD5*(block(N))
0	0- 511	**dc0c20abad42d487a74f308c69d18a5a**
1	512-1023	**9e7bc64399ad87ae9c2b545061959778**
2	1024-1535	**6e7f3577b100f9ec7fae18438fd5b047**
3	1536-2047	**4594899684d0565789ae9f364885e303**
4	...	

Using distinct sectors in media sampling and full media analysis to detect presence of documents from a corpus, Kristina Foster, NPS Master's Thesis, 2012

Of course, files have internal structure.

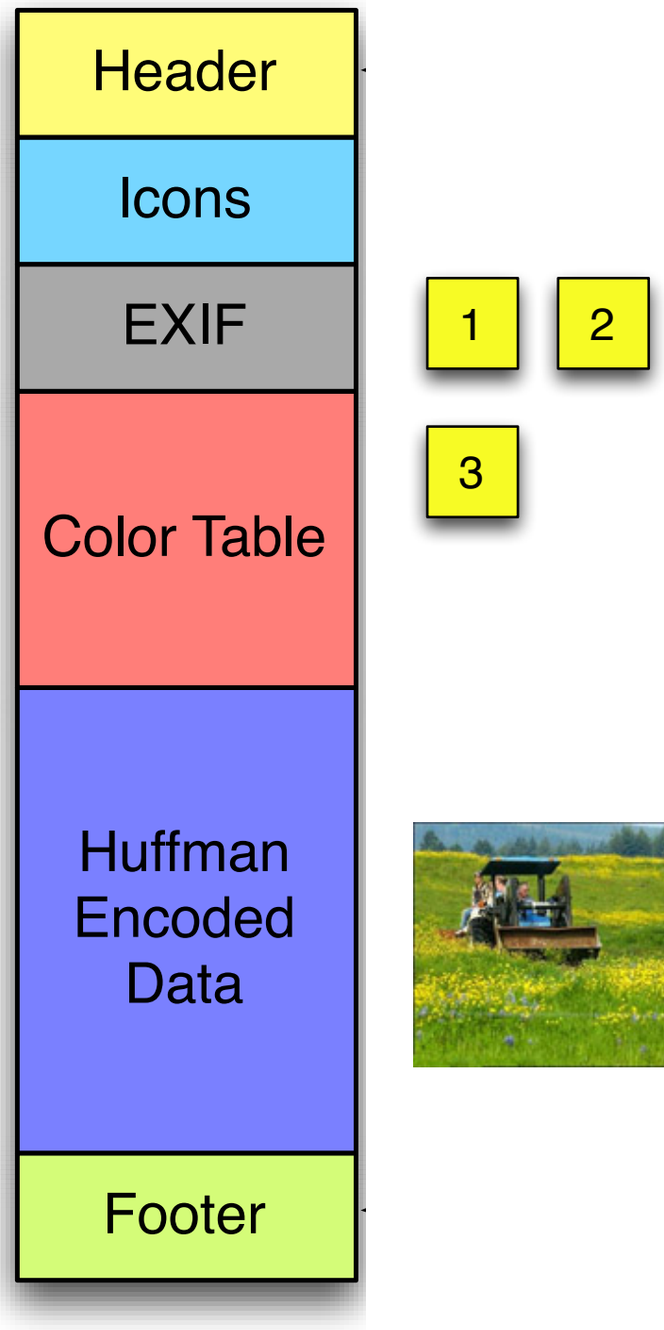


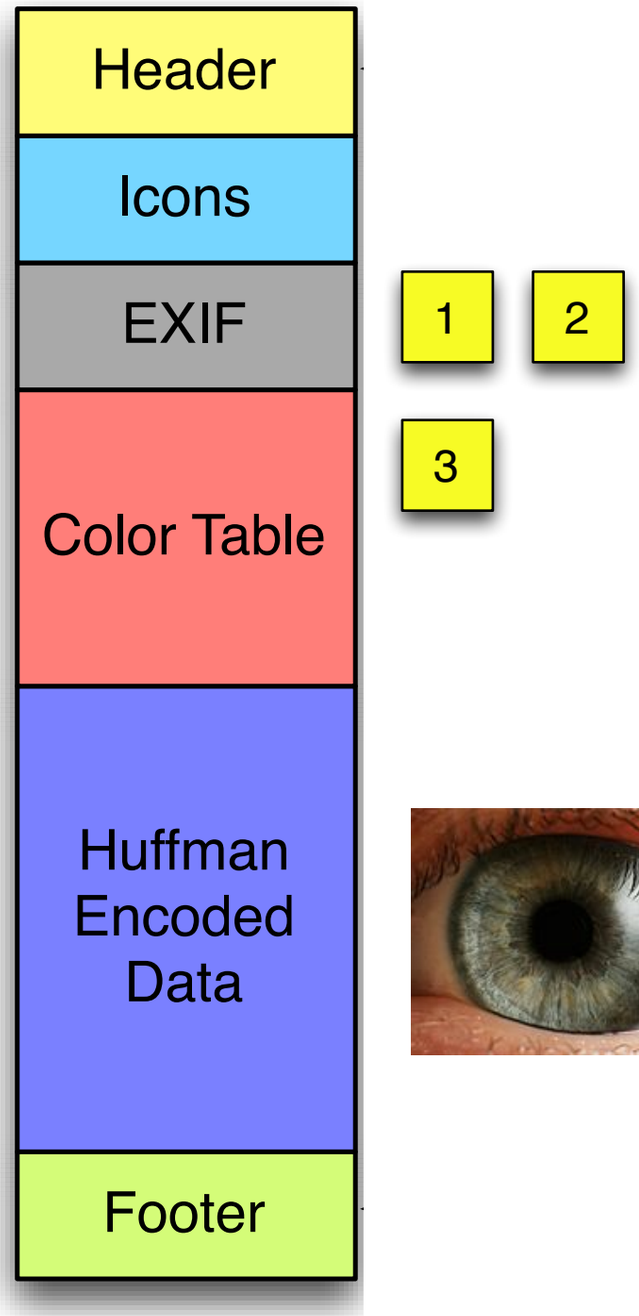
Some blocks are likely to be distinct for each file



Different files will have different Huffman encoded areas.

Other blocks might occur in more than one file.

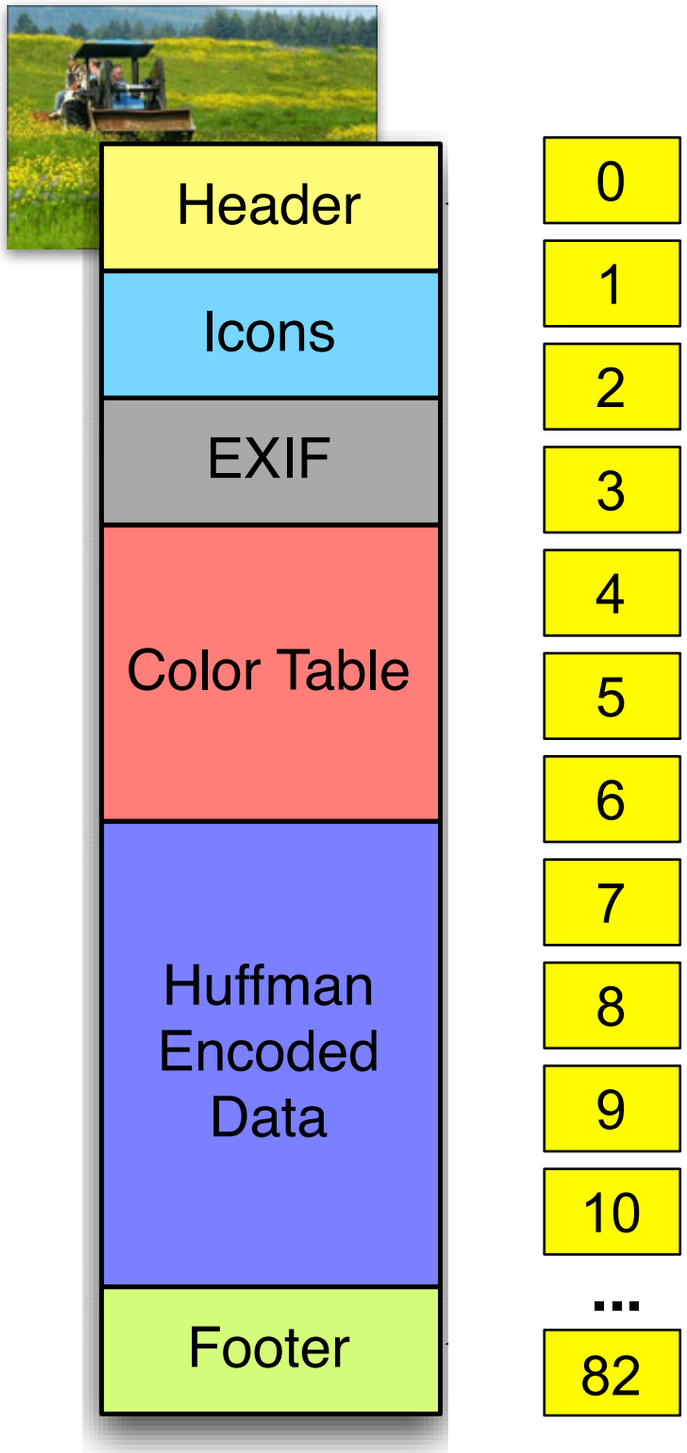




EXIF and color table are generated by the camera.

Likely the same.

Consider the 82 blocks for this 41K JPEG.



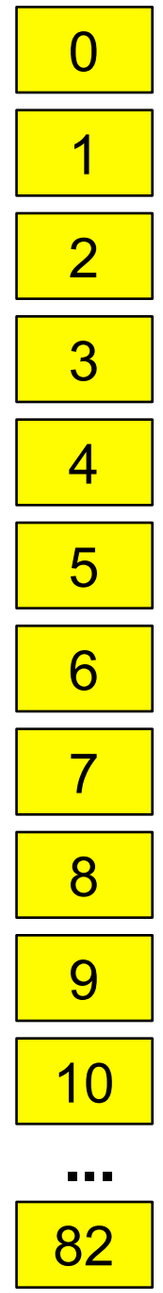
Block #	MD5(Block(N))
0	`dc0c20abad42d487a74f308c69d18a5a`
1	`9e7bc64399ad87ae9c2b545061959778`
2	`6e7f3577b100f9ec7fae18438fd5b047`
3	`4594899684d0565789ae9f364885e303`
. . .	. . .

We searched for these block hashes in a corpus of 4 million files.

≃ 1 million in GOVDOCS1 collection

= 109,282 JPEGs (including 000107.jpg)

≃ 3 million samples of Windows malware

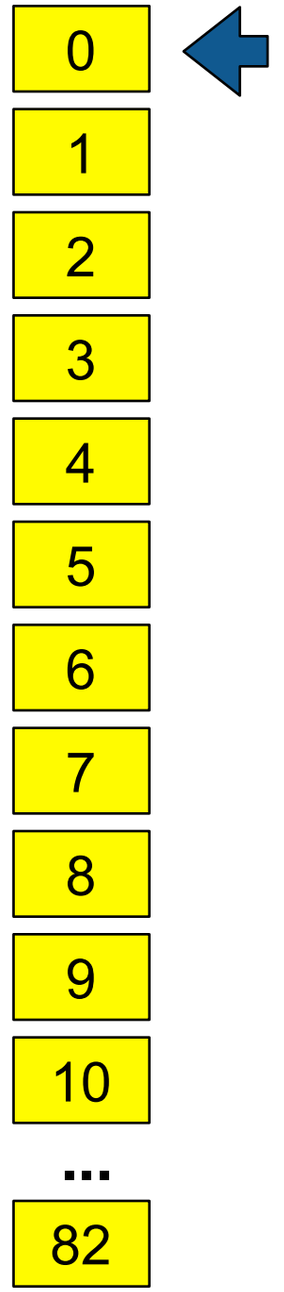


Results:

- Most of the block hashes in **000107.jpg** do not appear elsewhere in corpus.
- Some of the block hashes appeared in other JPEGs.
- None of the block hashes appeared in files that were not JPEGs

The beginning of the file is distinct in GOVDOCS

hash	location	count
dc0c20abad42d487a74f308c69d18a5a	offset 0-511	1
9e7bc64399ad87ae9c2b545061959778	offset 512-1023	1
6e7f3577b100f9ec7fae18438fd5b047	offset 1024-1535	1
4594899684d0565789ae9f364885e303	offset 1536-2047	1
4d21b27ceec5618f94d7b62ad3861e9a	offset 2048-2559	1
03b6a13453624f649bbf3e9cd83c48ae	offset 2560-3071	1
c996fe19c45bc19961d2301f47cabaa6	offset 3072-3583	1
0691baa904933c9946bbda69c019be5f	offset 3584-4095	1
1bd9960a3560b9420d6331c1f4d95fec	offset 4096-4607	1
52ef8fe0a800c9410bb7a303abe35e64	offset 4608-5119	1
b8d5c7c29da4188a4dcaa09e057d25ca	offset 5120-5631	1
3d7679a976b91c6eb8acd1bfa3414f96	offset 5632-6143	1
8649f180275e0b63253e7ee0e8fa4c1d	offset 6144-6655	1
60ebc8acb8467045e9dcbe207f61a6c2	offset 6656-7167	1
440c1c1318186ac0e42b2977779514a1	offset 7168-7679	1
72686172f8c865231e2b30b2829e3dd9	offset 7680-8191	1
fdff55c618d434416717e5ed45cb407e	offset 8192-8703	1
fcd89d71b5f728ba550a7bc017ea8ff1	offset 8704-9215	1
2d733e47c5500d91cc896f99504e0a38	offset 9216-9727	1
2152fdde0e0a62d2e10b4fecc369e4c6	offset 9728-10239	1
692527fa35782db8592486343 6d45d7f	offset 10240-10751	1
76dbb9b469273d0e0e467a55728b7883	offset 10752-11263	1



The middle of 000107.JPG appears elsewhere...

hash	location	count	
9df886fdfa6934cc7dcf10c04be3464a	offset 14848-15359	1	
95399e7ecc7ba1b38243069bdd5c263a	offset 15360-15871	1	0
ef1ffcdc11162ecdfedd2dde644ec8f2	offset 15872-16383	1	
7eb35c161e91b215e2a1d20c32f4477e	offset 16384-16895	1	1
38f9b6f045db235a14b49c3fe7b1cec3	offset 16896-17407	1	
edceba3444b5551179c791ee3ec627a5	offset 17408-17919	1	2
6bc8ed0ce3d49dc238774a2bdeb7eca7	offset 17920-18431	1	
5070e4021866a547aa37e5609e401268	offset 18432-18943	14	3
13d33222848d5b25e26aefb87dbdf294	offset 18944-19455	9198	
0dfcde85c648d20aed68068cc7b57c25	offset 19456-19967	9076	4
756f0bbe70642700aafb2557bf2c5649	offset 19968-20479	9118	
c2c29016d3005f7a1df247168d34e673	offset 20480-20991	9237	5
42ff3d72b2b25f880be21fac46608cc9	offset 20992-21503	9708	
b943cd0ea25e354d4ac22b886045650d	offset 21504-22015	9615	6
a003ec2c4145b0bc871118842b74f385	offset 22016-22527	9564	
1168c351f57aad14de135736c06665ea	offset 22528-23039	7	7
51a50e6148d13111669218dc40940ce5	offset 23040-23551	83	
365b122f53075cb76b39ca1366418ff9	offset 23552-24063	83	8
9ad9660e7c812e2568aaf063a1be7d05	offset 24064-24575	84	
67bd01c2878172e2853f0aef341563dc	offset 24576-25087	84	9
fc3e47d734d658559d1624c8b1cbf2c1	offset 25088-25599	84	
cb9aef5b7f32e2a983e67af38ce8ff87	offset 25600-26111	1	10
			...
			82

Block 37 was found in 9198 other files.
The sector is filled with blank lines 100 characters long...

```
13d33222848d5b25e26aefb87dbdf294   offset 18944-19455      9198

$ dd if=000107.jpg skip=18944 count=512 bs=1|xxd
0000000: 2020 2020 2020 2020 2020 2020 2020 2020
0000010: 2020 2020 2020 2020 2020 2020 0a20 2020                .
0000020: 2020 2020 2020 2020 2020 2020 2020 2020
0000030: 2020 2020 2020 2020 2020 2020 2020 2020
0000040: 2020 2020 2020 2020 2020 2020 2020 2020
0000050: 2020 2020 2020 2020 2020 2020 2020 2020
0000060: 2020 2020 2020 2020 2020 2020 2020 2020
0000070: 2020 2020 2020 2020 2020 2020 2020 2020
0000080: 200a 2020 2020 2020 2020 2020 2020 2020    .
0000090: 2020 2020 2020 2020 2020 2020 2020 2020
00000a0: 2020 2020 2020 2020 2020 2020 2020 2020
00000b0: 2020 2020 2020 2020 2020 2020 2020 2020
00000c0: 2020 2020 2020 2020 2020 2020 2020 2020
00000d0: 2020 2020 2020 2020 2020 2020 2020 2020
00000e0: 2020 2020 2020 0a20 2020 2020 2020 2020                .
00000f0: 2020 2020 2020 2020 2020 2020 2020 2020
```

This pattern comes from the “whitespace padding” of the XMP section.

- The whitespace can start on any byte offset, making collisions likely but not common

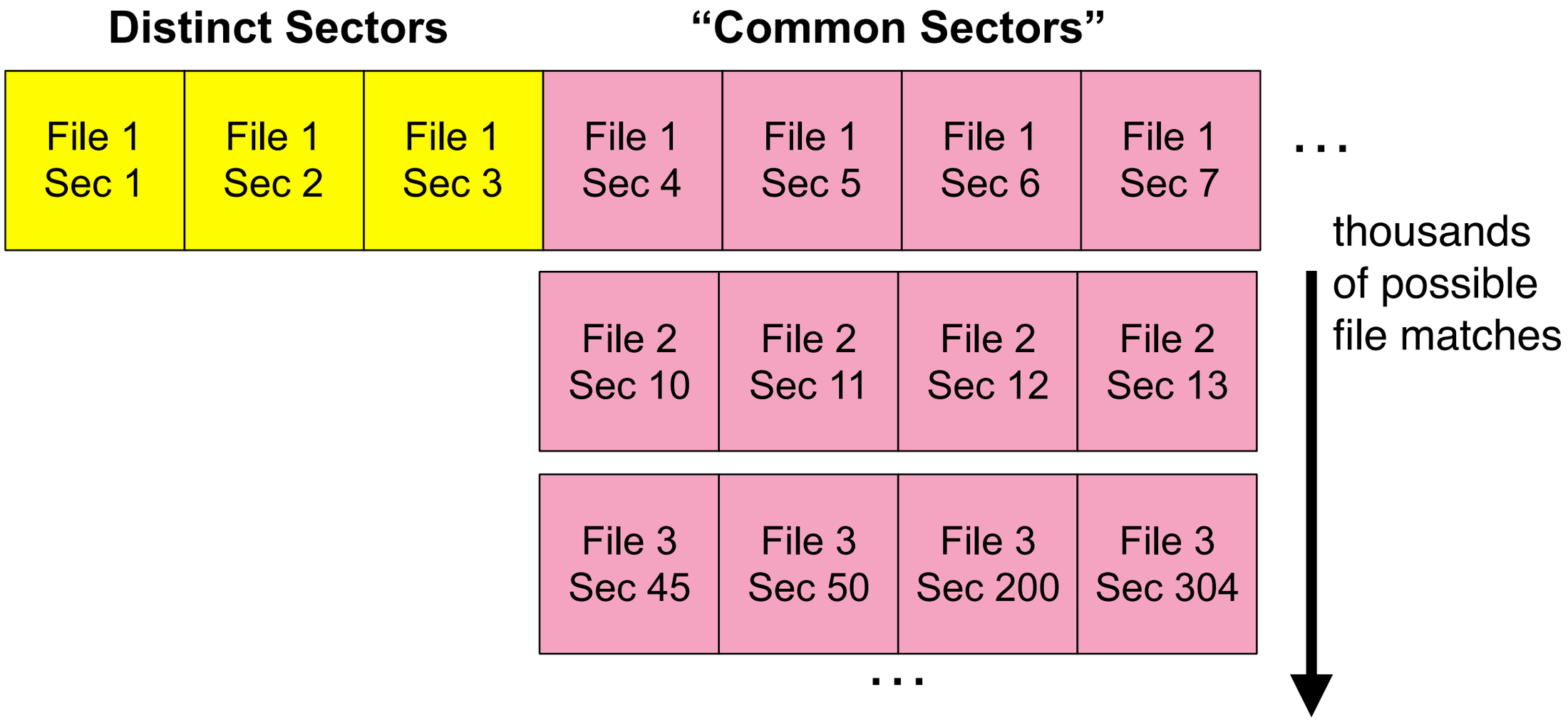
Block 45 was found in 83 other files.
It appears to contain EXIF metadata

```
51a50e6148d13111669218dc40940ce5   offset 23040-23551  83

$ dd if=000107.jpg skip=23040 count=512 bs=1|xxd
0000000: 3936 362d 322e 3100 0000 0000 0000 0000  966-2.1.........
0000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000030: 0000 0000 0000 0000 0058 595a 2000 0000  .........XYZ ...
0000040: 0000 00f3 5100 0100 0000 0116 cc58 595a  ....Q........XYZ
0000050: 2000 0000 0000 0000 0000 0000 0000 0000   ...............
0000060: 0058 595a 2000 0000 0000 006f a200 0038  .XYZ ......o...8
0000070: f500 0003 9058 595a 2000 0000 0000 0062  .....XYZ ......b
0000080: 9900 00b7 8500 0018 da58 595a 2000 0000  .........XYZ ...
0000090: 0000 0024 a000 000f 8400 00b6 cf64 6573  ...$.........des
00000a0: 6300 0000 0000 0000 1649 4543 2068 7474  c........IEC htt
00000b0: 703a 2f2f 7777 772e 6965 632e 6368 0000  p://www.iec.ch..
00000c0: 0000 0000 0000 0000 0016 4945 4320 6874  ..........IEC ht
00000d0: 7470 3a2f 2f77 7777 2e69 6563 2e63 6800  tp://www.iec.ch.
00000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000100: 0000 0000 0000 0000 0000 0000 0064 6573  .............des
0000110: 6300 0000 0000 0000 2e49 4543 2036 3139  c........IEC 619
0000120: 3636 2d32 2e31 2044 6566 6175 6c74 2052  66-2.1 Default R
```

Block 48 was found in 84 collisions files.
It appears to contain part of a JPEG color table...

```
67bd01c2878172e2853f0aef341563dc   offset 24576-25087        84

$ dd if=000107.jpg skip=24576 count=512 bs=1 |xxd
0000000: 7a27 ab27 dc28 0d28 3f28 7128 a228 d429  z'.'.(.(?(q(.(.)
0000010: 0629 3829 6b29 9d29 d02a 022a 352a 682a  .)8)k).).*.*5*h*
0000020: 9b2a cf2b 022b 362b 692b 9d2b d12c 052c  .*.+.+6+i+.+.,.,
0000030: 392c 6e2c a22c d72d 0c2d 412d 762d ab2d  9,n,.,.-.-A-v-.-
0000040: e12e 162e 4c2e 822e b72e ee2f 242f 5a2f  ....L....../$/Z/
0000050: 912f c72f fe30 3530 6c30 a430 db31 1231  ././.050l0.0.1.1
0000060: 4a31 8231 ba31 f232 2a32 6332 9b32 d433  J1.1.1.2*2c2.2.3
0000070: 0d33 4633 7f33 b833 f134 2b34 6534 9e34  .3F3.3.3.4+4e4.4
0000080: d835 1335 4d35 8735 c235 fd36 3736 7236  .5.5M5.5.5.676r6
0000090: ae36 e937 2437 6037 9c37 d738 1438 5038  .6.7$7`7.7.8.8P8
00000a0: 8c38 c839 0539 4239 7f39 bc39 f93a 363a  .8.9.9B9.9.9.:6:
00000b0: 743a b23a ef3b 2d3b 6b3b aa3b e83c 273c  t:.:.;-;k;.;.<'<
00000c0: 653c a43c e33d 223d 613d a13d e03e 203e  e<.<.="=a=.=.> >
00000d0: 603e a03e e03f 213f 613f a23f e240 2340  `>.>.?!?a?.?.@#@
00000e0: 6440 a640 e741 2941 6a41 ac41 ee42 3042  d@.@.A)AjA.A.B0B
00000f0: 7242 b542 f743 3a43 7d43 c044 0344 4744  rB.B.C:C}C.D.DGD
0000100: 8a44 ce45 1245 5545 9a45 de46 2246 6746  .D.E.EUE.E.F"FgF
0000110: ab46 f047 3547 7b47 c048 0548 4b48 9148  .F.G5G{G.H.HKH.H
0000120: d749 1d49 6349 a949 f04a 374a 7d4a c44b  .I.IcI.I.J7J}J.K
```

Non-distinct sectors complicate file reassembly.



Hash-based carving: the vision and our contribution.

We hope to use hash-based carving to:

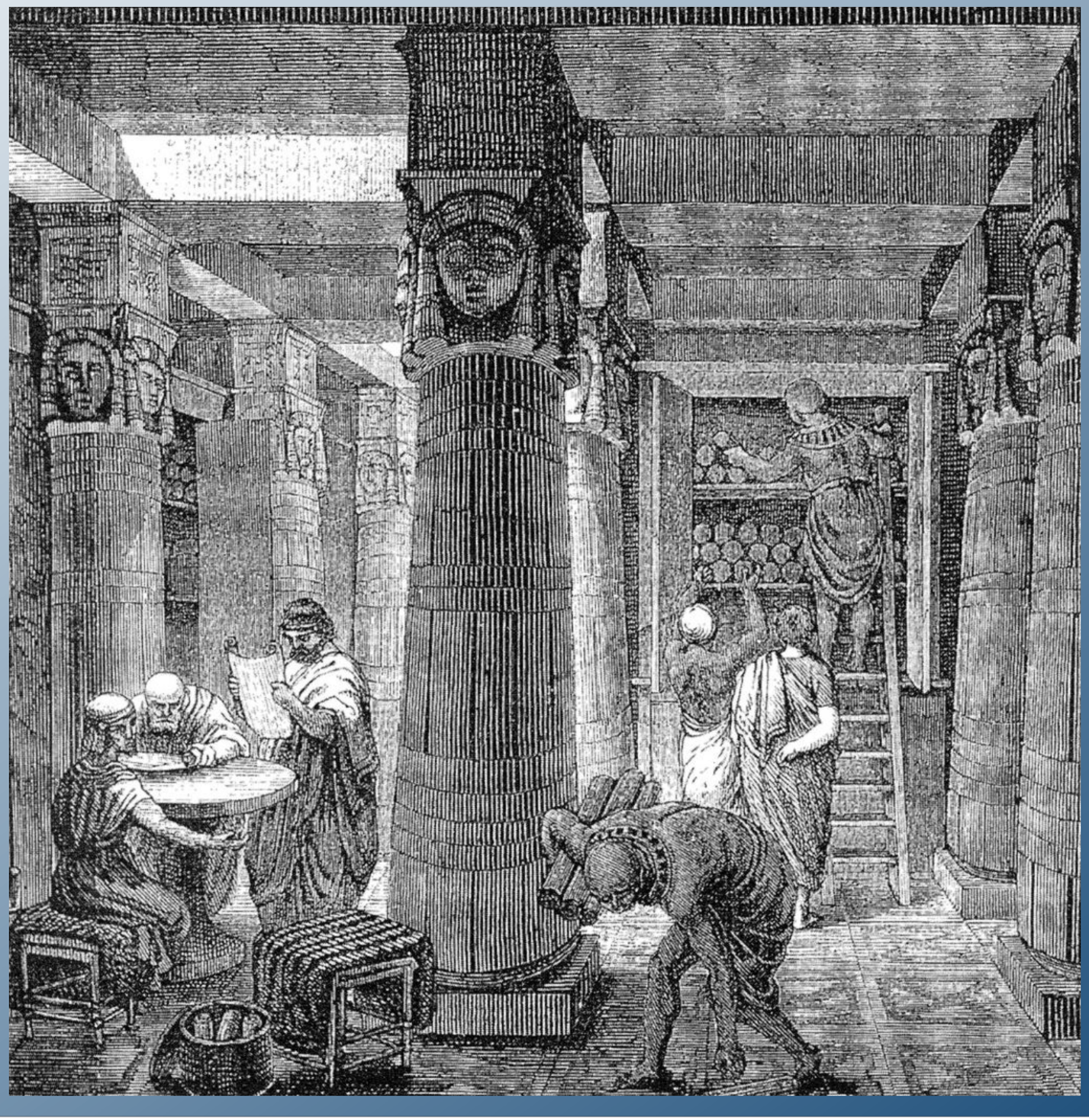
- Look for “known content” on new media.
  - —*E.g., CP videos on suspect drives.*
- Search with a target database of 4GB files (1 billion 4KiB blocks)
- Process media at I/O speed

Previous work thought that the hash database was the challenge!

- —*Requires approximately 100,000 database lookups per second w/ 4K sectors.*
- —*We created hashdb and a bulk_extractor plug-in to scan media.*
- —*We discovered that having the list of hash hits wasn’t enough!*

Our contributions:

- We present our real-world experiences of hash-based carving on a non-trivial problem
- We present an algorithm for reassembling matched blocks into files.



Prior work

There has been work on hash-based carving since 2006

DFRWS 2006 Carving Challenge

- Extracted text from the challenge; searched the Internet for the text
- Downloaded MSWORD files from the Internet
- Block-hashed the target files, looked sector hash matches, copied runs into files.
  - —*Note: target database size = 1 file!*

Dundass et al. (2008), Collange et al (2009a, b)

- Introduced the term “hash-based carving”
- Surveyed algorithms; explored GPUs for hashing on 4-byte boundaries

Foster (2012)

- Looked for occurrence of common blocks in GOVDOCS
- Identified patterns in some common blocks (EXIF structures, PDF structures, etc.)

Taguchi (2013)

- Concluded that a 64KiB read size was optimal when performing sector hashing.

There are two existing hash-based carving systems

frag_find — Garfinkel (2009)

- C++ script, distributed with NPS Bloom Filter package.
- Hashes target files, keeps hash database in RAM.

File Block Hash Map Analysis (FBHMA) — Key (2013)

- EnScript for EnCase

Both programs will work with only a few target files.

Terminology and theory

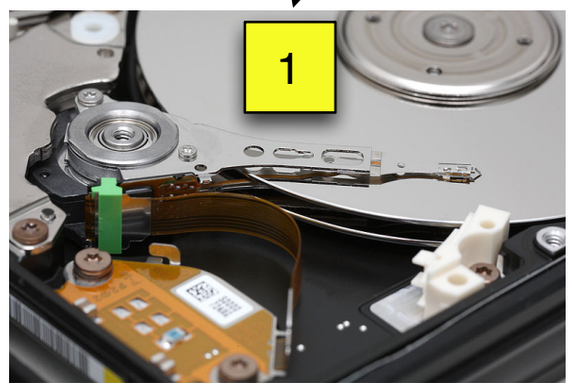
A consistent terminology is important.

hash-based carving: recognizing a *target file* on a piece of *searched media* by hashing same-sized blocks of data from both the file and the media and looking for hash matches.

Target file

File block

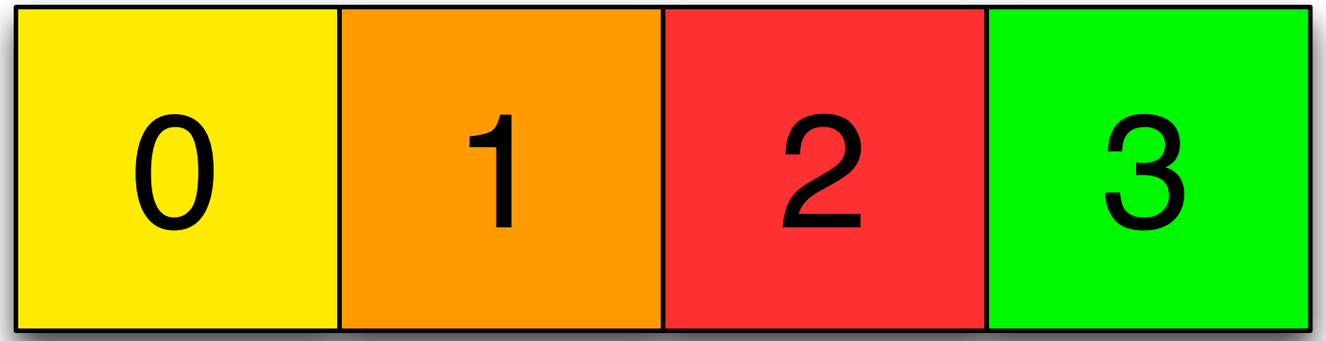
Disk sector



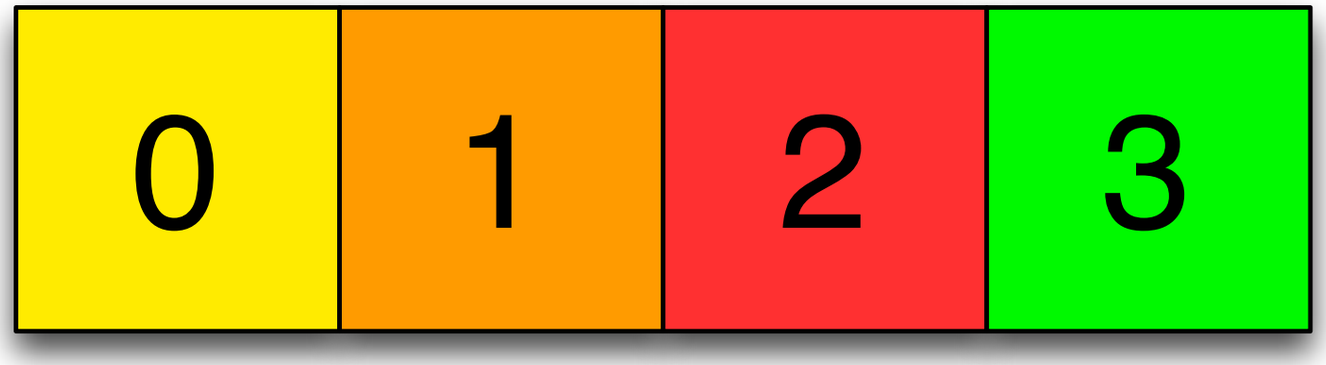
Search media

Blocks and sectors must be the same size.
We use 4KiB blocks

File blocks



Disk sectors

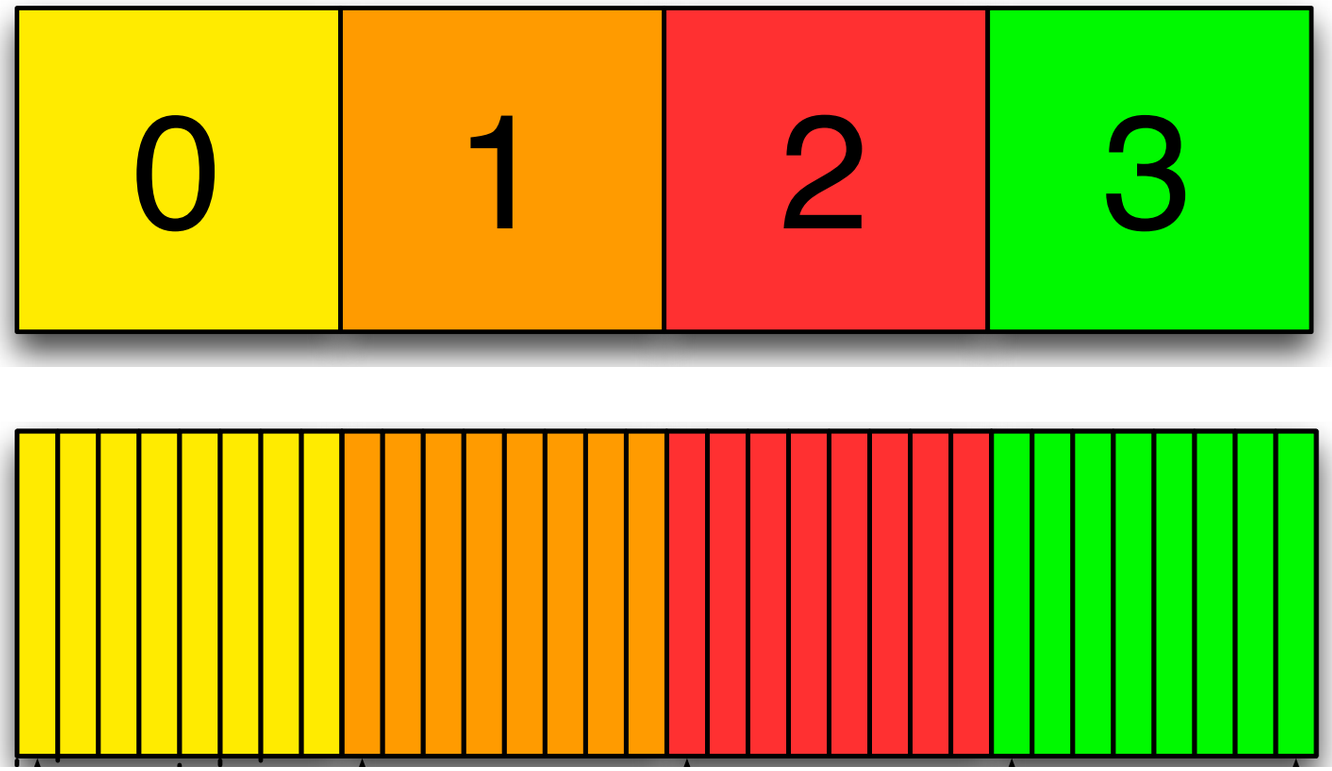


New hard drives will have 4KiB disk sectors.

Most file systems allocate in 4KiB blocks or larger.

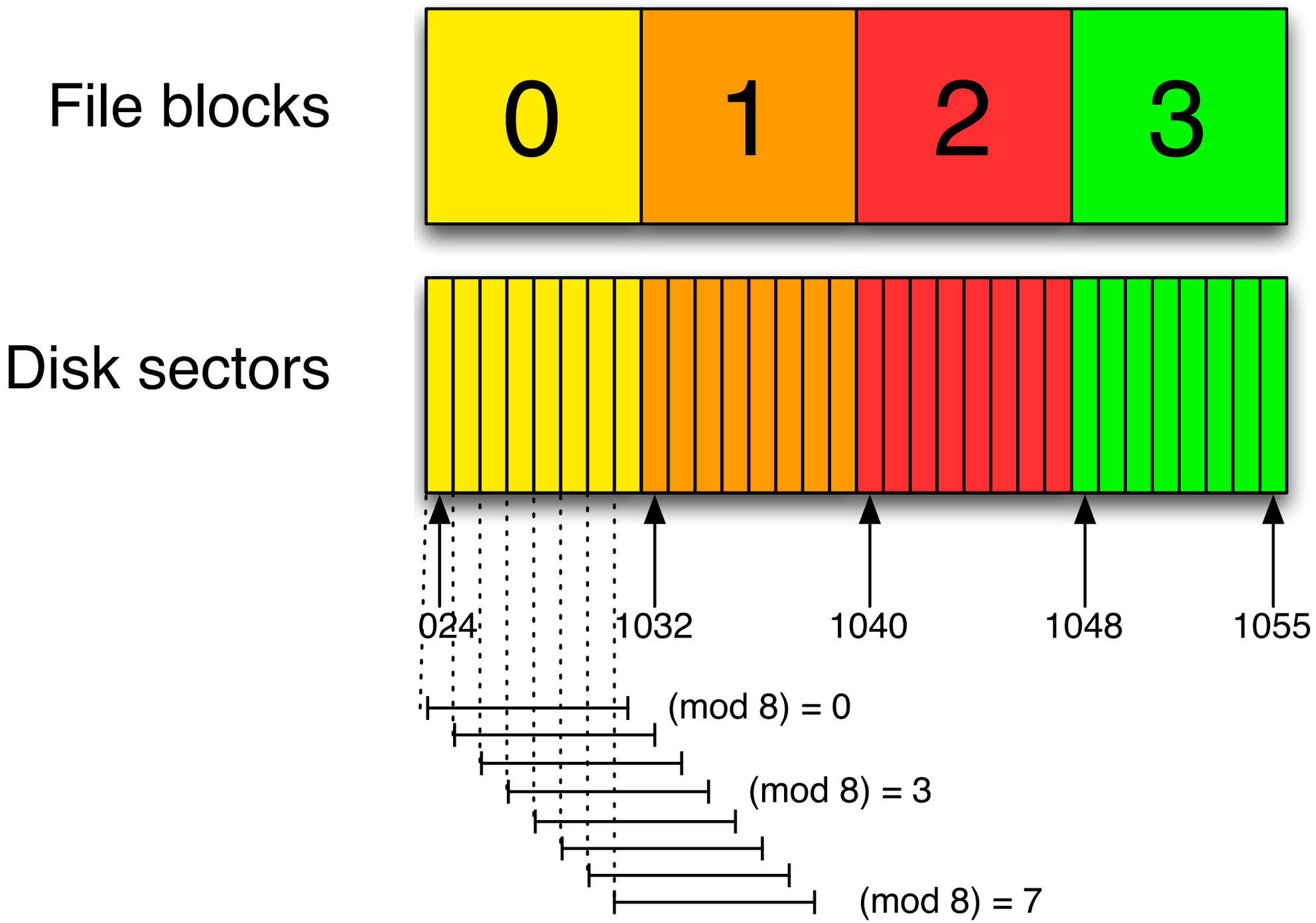
For legacy drives, we hash groups of 8 sectors.

File blocks



Disk sectors

We don’t know the starting sector offset, so we hash with a sliding 8-sector window



Comparing 512-byte and 4096-byte blocks:

512 bytes:

- Advantage: Matches today's disk sector size
- Advantage: No alignment issues. Every file starts on a disk sector
- Disadvantage: Data reduction factor = 16/512 = 1/32 (1 billion hashes = 512GiB)

4096 bytes:

- Advantage: Data reduction factor = 16/4096 = 1/256 (1 billion hashes = 4TiB)
- Disadvantage: Alignment!
  - —*Files are aligned with start-of-partition, not start-of-disk.*
  - —*Partition starts at block 63, 4K cluster size:*

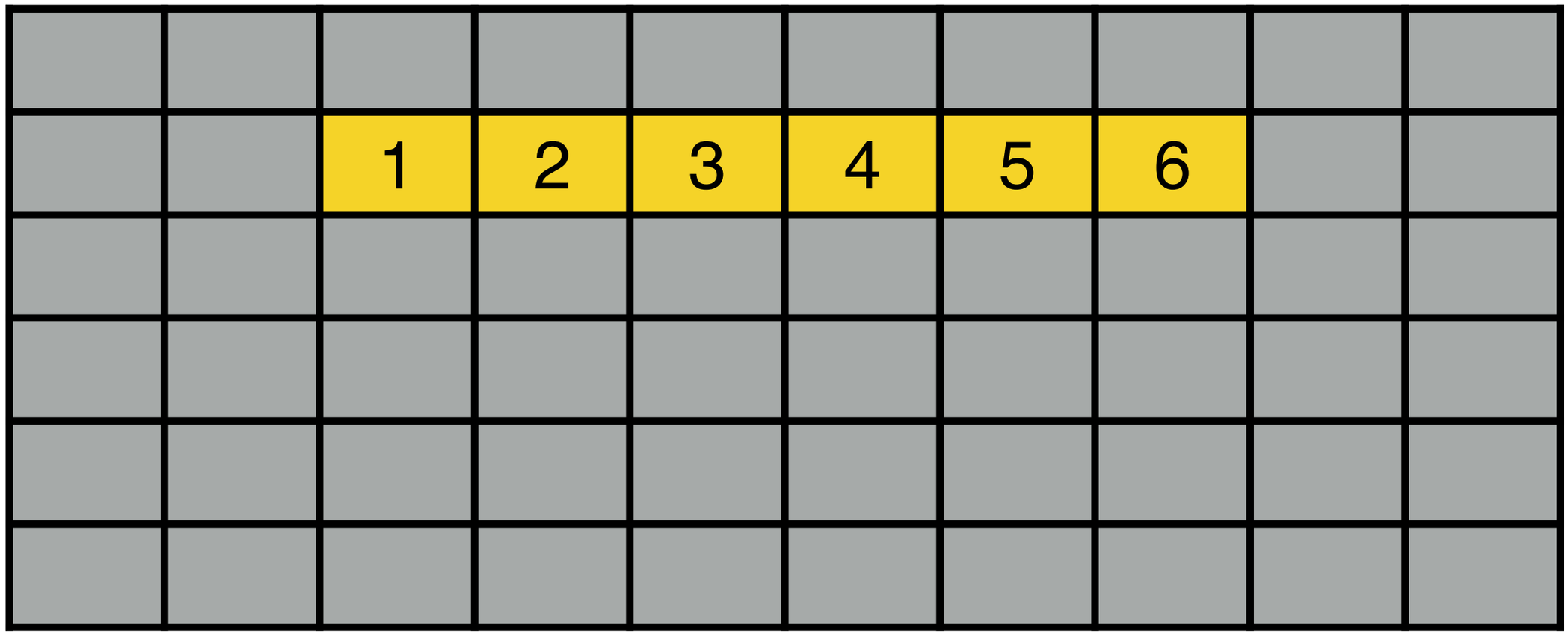
    files start at sector: (63+B*Cluster Size)
    63 (mod 8) = 7 ; we want to hash blocks 7-14, 15-22, 23-30 …
  - —*Partition starts at block 1024, 4K cluster size:*

    files start at sector: (1024+B*Cluster Size)
    1024 (mod 8) = 0 ; we want to hash blocks 0-7, 8-15, 16-23

(mod 8) value depends on current & previous file system(s)

Hash matches can result from four scenarios:

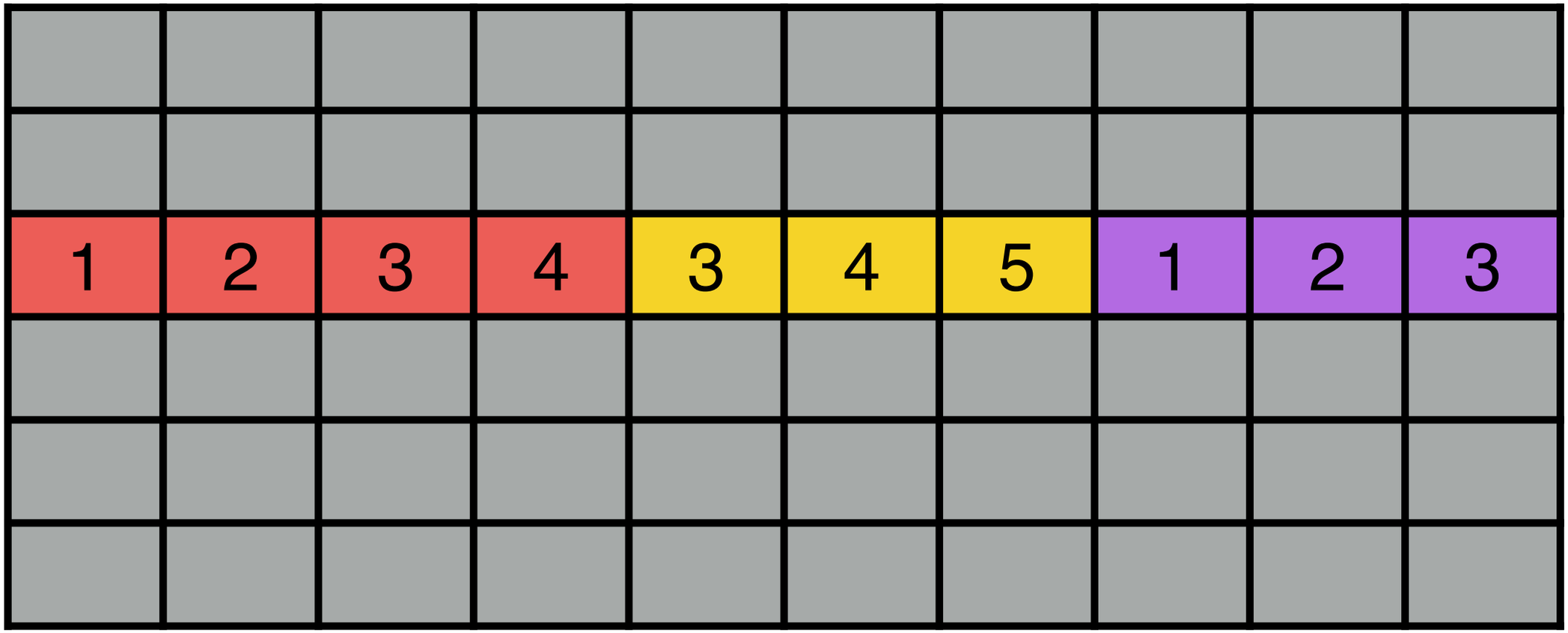
A copy of the file is present on the search media. (1-6)



Hash matches can result from four scenarios:

A copy of the file is present on the search media

A copy present on media, later partially overwritten

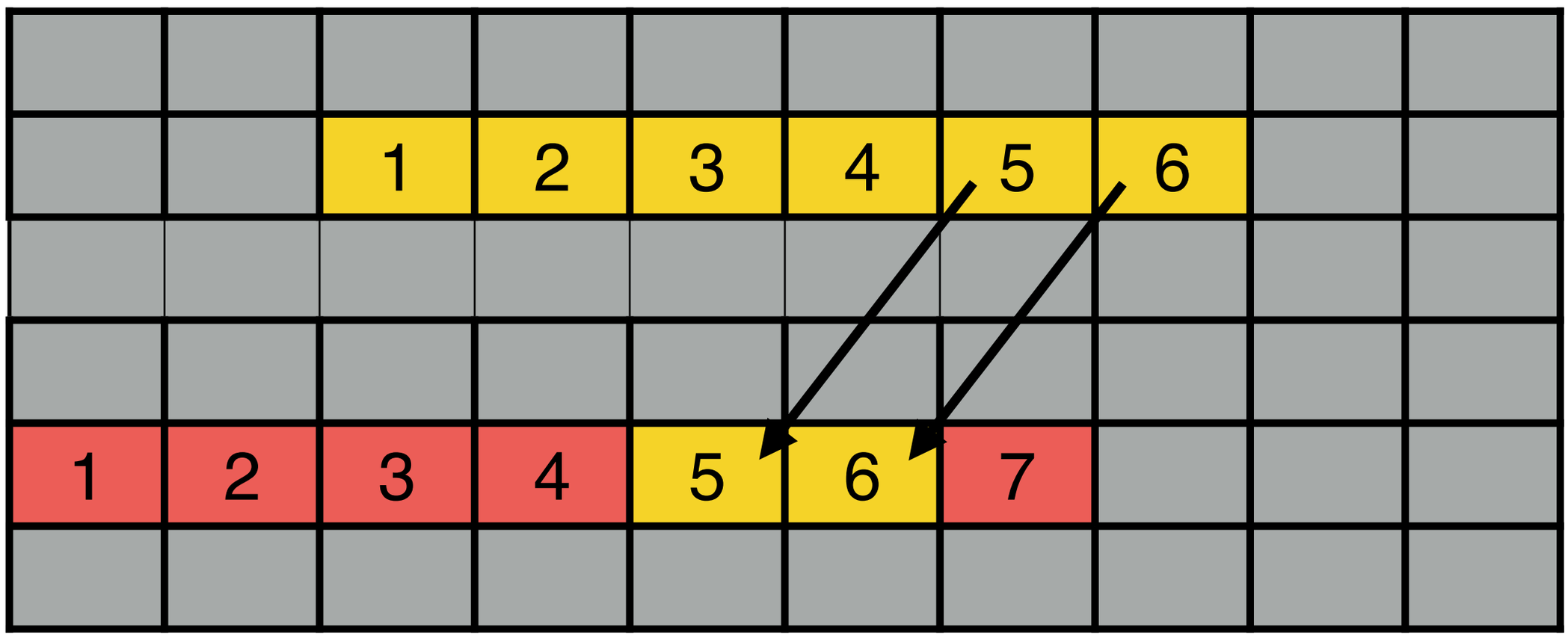


Hash matches can result from four scenarios:

A copy of the file is present on the search media

A copy present on media, later partially overwritten

A new file that has sectors in common with an existing file



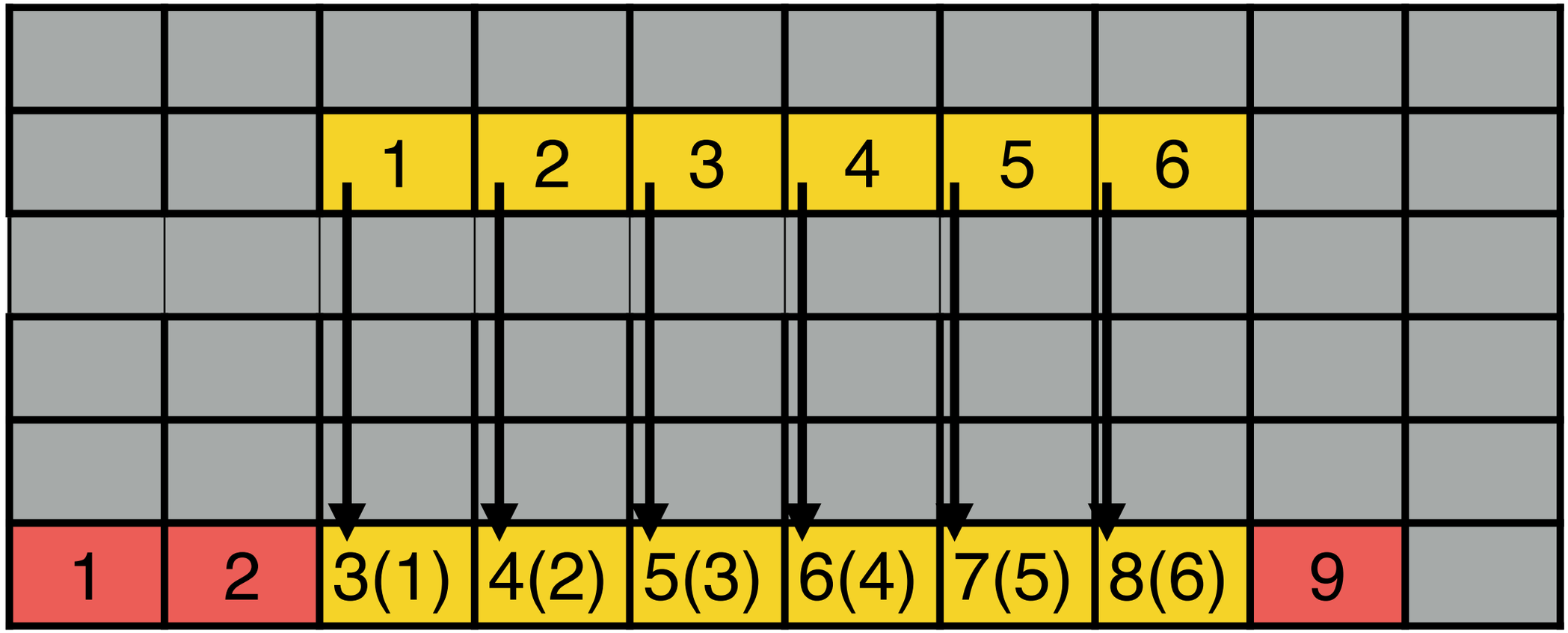
Hash matches can result from four scenarios:

A copy of the file is present on the search media

A copy present on media, later partially overwritten

A new file that has sectors in common with an existing file

A target file embedded in a larger file



We created a four-step hash-based carving process.

Step 1 — Database Building

- Create a database of file block hashes from *target files*.
- Each hash may be in multiple files!
  - *Database: Hash = [(file1,offset1), (file2,offset2), (file3,offset3), …]*

Step 2 — Media Scanning

- Scan the *searched media* by hashing 4KiB of sectors
- Search each hash in database

Step 3 — Candidate Selection

- Determine which files are likely present

Step 4 — Target Assembly

- Produce a map showing how media sectors map to target files.

Our tool chain is based on bulk_extractor and hash_db.

bulk_extractor — Open source tool for media scanning

hash_db — High-speed hash database

- Bloom filter stores hashes for fast “false” lookup.
- For each hash, hashdb stores: {data set, file id, offset}
- For each file, hashdb stores: {name, length}

Two primary functions:

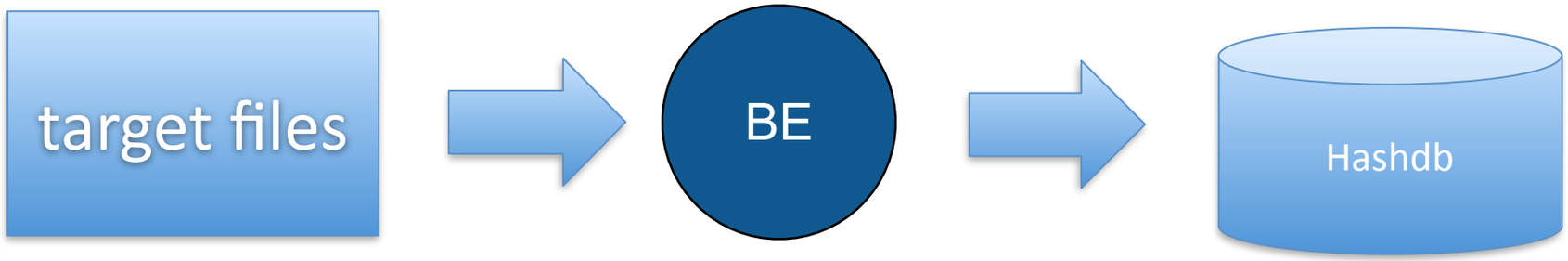
- Import hashes into database
- Look up hashes

Lookup speed ≈ 100,000 hashes/sec on Laptop w/SSD.

Some database management functions:

- Dump, Remove duplicates (hashes with more than one source)

bulk_extractor's "hashdb" scanner hashes blocks. import mode: sector-hash target files and store in DB



scan mode: sector-hash the “search drive” and compare against the sector hash database.

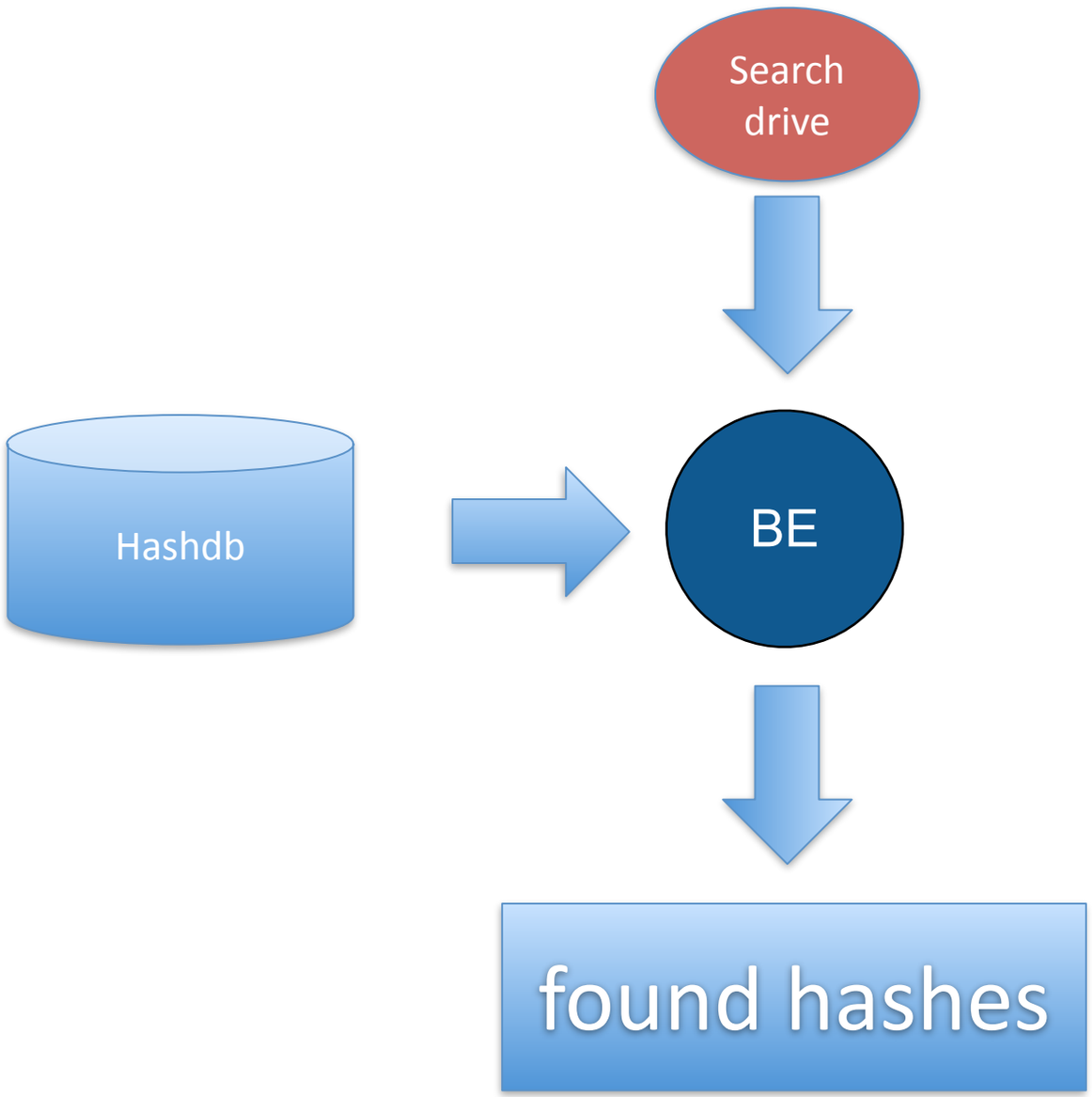
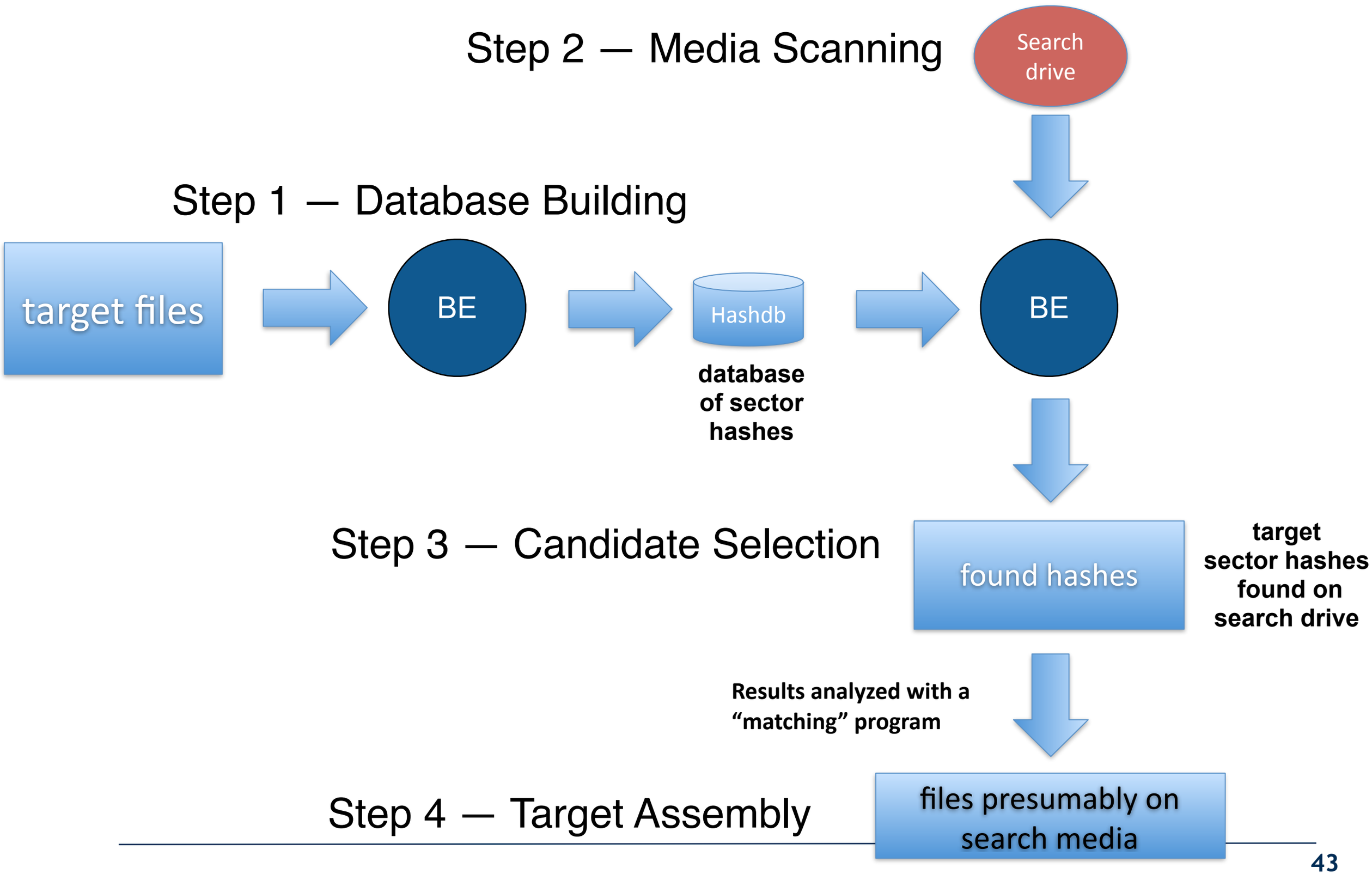


Diagram of hash-based carving



Experimental Setup

We performed a realistic test

Target files:

- "Monterey Kitty" — 82 JPEGs, 2 QT movies, 4 MPEG4 files (201MB in total)
- GOVDOCS1 — ≈1M files downloaded from US Government web sites

Search Media:

- M57-Patents — Scenario of a small business developed by NPS in 2009.
- jo-2009-11-20-oldComputer — disk image of person who had "kitty" materials.
  - *— 13 GB disk image*

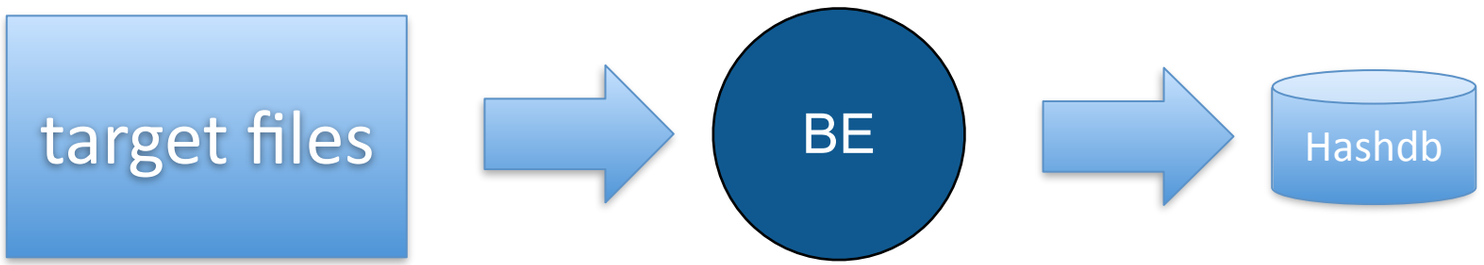
Experiment:

1. Create hashdb database with Monterey Kitty & GOVDOCS1 (kitty+govdocs.hdb)
2. Use database to scan M57-Patents drives
3. Hypothesis:
   1. *If we found a single "distinct" block from a file, that file was on the drive.*

Step 1 — Database Building

Create hashdb database using bulk_extractor

- Monterey Kitty database: 50,206 hashes from 88 different files

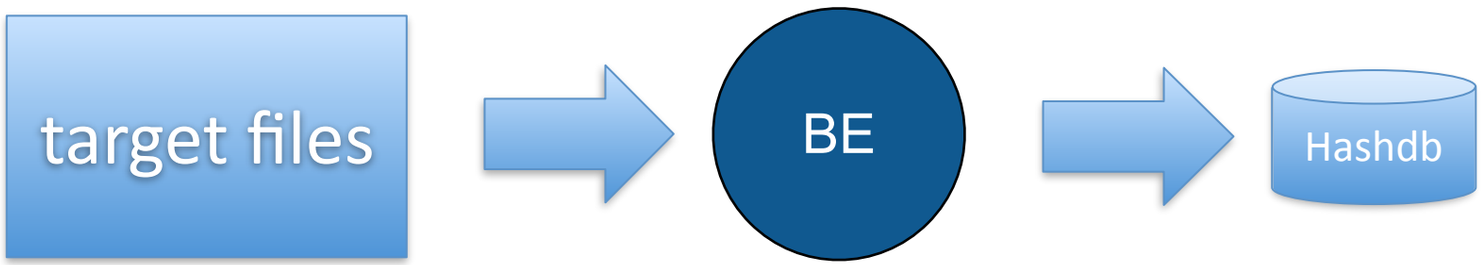


# times in DB	# of hashes
F Singleton	50,206
F 2 x	0
F 3 x	0
…	

Step 1 — Database Building

Create hashdb database using bulk_extractor

- Monterey Kitty database: 50,206 hashes from 88 different files
- GOVDOCS corpus: 119,687,300 hashes from 909,815 files



# times in DB	# of hashes
F Singleton	117,213,026
F 2 x	514,238
F 3 x	60,317
…	
11,434	1 (“null”)

Step 2 — Media Scanning:

Input files: 16GB disk image

- 394 pages (6.3GB) x 32,768 overlapping 4KiB blocks per page.

Scan time: 116 seconds (64-core reference system)

- 111 K lookups/sec

Output — 33,847 matches found:

```
# Feature-Recorder: identified_blocks
# Filename: nps-2009-m57-patents-redacted/jo-2009-11-16.E01

86435328 736d99610d0097be78651ecdae4714bb {"count":39,"flags":"H"}

1231920640 90ccbdf24a74c8c05b94032b4ce1825d {“count":1,"flags":"H"}

1231924736 9403e1cac89e860b93570ac452d232a5 {“count":1}
```

What we found — graphically

M57-Patents drives:

- Found nearly all Kitty files
  - —*Found multiple copies*
  - —*In some cases, found all of a file except the first sector (that's good!)*

Can only be TiggerTheCat.m4v

This produces a simple reassembly algorithm:

- For every file whose hash was found:
  - —*Make a list of the [file block #, disk block #]*
  - —*Sort*
  - —*Report complete runs:*

```
[0, 1024]
[1, 1032]
[2, 1040]
[3, 1056]
[4, 1064]
```

TiggerTheCat.m4v

First complication: the same block might be in two places in a file

M57-Patents drives:

- Found nearly all Kitty files
  - —*Found multiple copies*

  - —*In some cases, found all of a file except the first sector (that's good!)*

This produces a ~~simple~~ more complicated reassembly algorithm:

- For every file whose hash was found:
  - —*Make a list of the [disk block #, {file block #s}]*
  - —*Sort*
  - —*Report complete runs:*

```
[1024, 0]
[1032, {1,4}]
[1040, 2]
[1056, 3]
[1064, {1,4}]
```

We also found distinct blocks from GOVDOCS files on the M57 drive

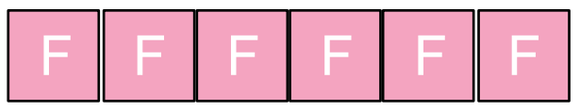
M57-Patents drives:

- Found nearly all Kitty files
  - —*Found multiple copies*

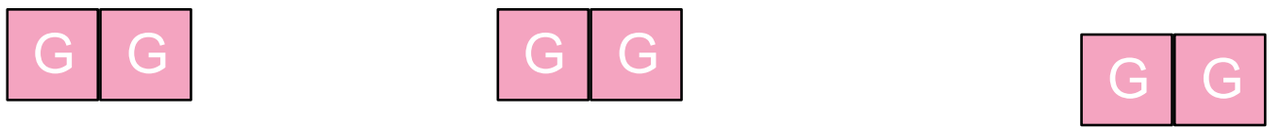
  - —*In some cases, found all of a file except the first sector (that's good!)*

- Distinct GOVDOCS files:
  - —*Found several complete files!* ***These files really were present! (fonts)***
  - —*Found several runs of distinct blocks* ***that were never present!***

  - —*Found many runs of common blocks.*



  - —*Frequently, we find common runs scattered:*

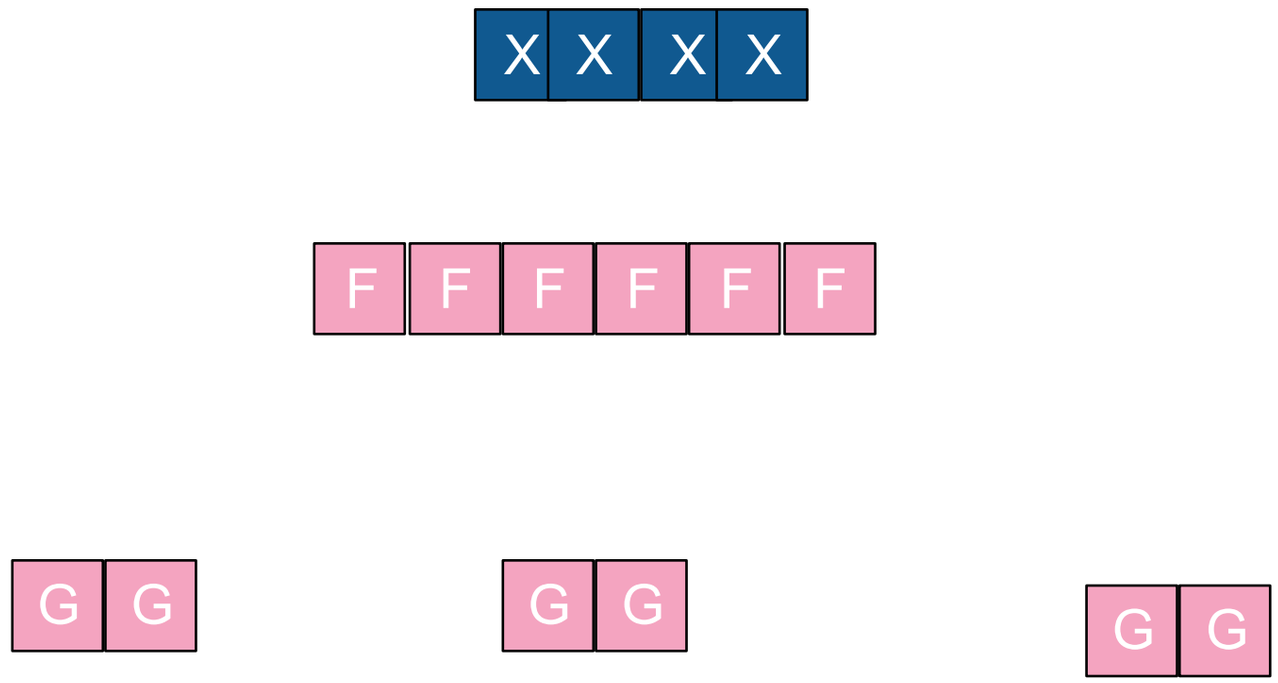


These are non-probative blocks

These blocks match files ***that we know are not present.***

Why did we think they were distinct?

- We had not looked at enough files!



These are non-probative blocks

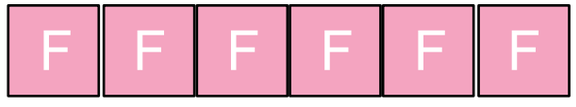
These blocks match files ***that we know are not present.***

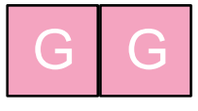
Why did we think they were distinct?

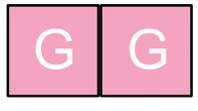
- We had not looked at enough files!

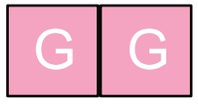
These blocks were similar to the common blocks we had seen in 0000107.jpg:

- Incrementing binary numbers
- Whitespace
- Strange binary structures



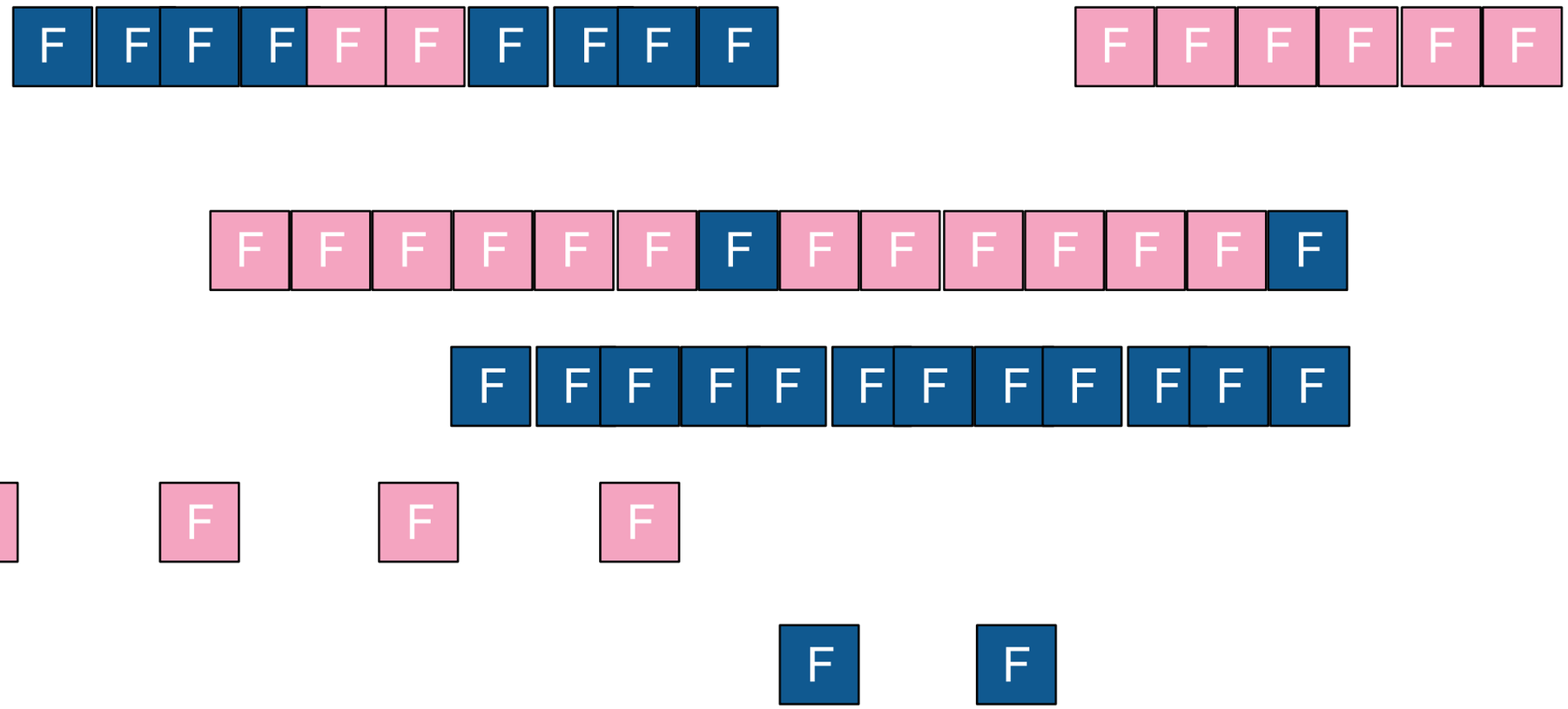






We found many blocks on the media that appeared “distinct” but could not have been.

The blocks must have been “common” — but we had not sampled enough files



Step 3 — Candidate Selection — determining which files might have been on the drive.

Previously we deemed a file present if we found a *distinct* block.

Now we will evaluate each “distinct” block to see if it is probative.

- Some distinct blocks are “distinct” in our data set, but are built from common structures:

Candidate selection requires that we screen out non-probative blocks.

We developed four tests for non-probative blocks.

1. The Ramp Test

- Detect and mark blocks with incrementing 4-byte binary numbers:

```
8102 0000 8202 0000 8302 0000 8402 0000
8502 0000 8602 0000 8702 0000 8802 0000
8902 0000 8a02 0000 8b02 0000 8c02 0000
8d02 0000 8e02 0000 8f02 0000 9002 0000
```

- These typically come from Microsoft Office Sector Allocation Tables.
  - *—They have a strong chance of appearing distinct…*
  - *—but they are algorithmically generated*

We developed four tests for non-probative blocks.

1. The Ramp Test
2. The White Space Test
   - Any sector that is 3/4 white space is non-probative.
   - Screens out whitespace in JPEGs and other files

```
0000000: 2020 2020 2020 2020 2020 2020 2020 2020
0000010: 2020 2020 2020 2020 2020 2020 0a20 2020                .
0000020: 2020 2020 2020 2020 2020 2020 2020 2020
0000030: 2020 2020 2020 2020 2020 2020 2020 2020
0000040: 2020 2020 2020 2020 2020 2020 2020 2020
0000050: 2020 2020 2020 2020 2020 2020 2020 2020
0000060: 2020 2020 2020 2020 2020 2020 2020 2020
0000070: 2020 2020 2020 2020 2020 2020 2020 2020
0000080: 200a 2020 2020 2020 2020 2020 2020 2020    .
0000090: 2020 2020 2020 2020 2020 2020 2020 2020
```

We developed four tests for non-probative blocks.

1. The Ramp Test
2. The White Space Test
3. The 4-byte Histogram Test
   - Suppresses sector if any 4-byte n-gram is present more than 256 times
   - Usually catches white space test as well (but not always)

```
0000 6400 0000 01ff ffff 9c00 0000 0100
0000 6400 0000 01ff ffff 9c00 0000 0200
0000 0000 0000 0100 0000 6400 0000 01ff
ffff 9c00 0000 0100 0000 6400 0000 01ff
ffff 9c00 0000 0100 0000 6400 0000 01ff
ffff 9c00 0000 0100 0000 6400 0000 01ff
ffff 9c00 0000 0100 0000 6400 0000 01ff
ffff 9c00 0000 0100 0000 6400 0000 01ff
```

We developed four tests for non-probative blocks.

1. The Ramp Test
2. The White Space Test
3. The 4-byte Histogram Test
4. The Entropy Test

- Mark as non-probative any block with entropy lower than a threshold
- Possibly use instead of "ad hoc" tests
- Didn't work as well

Evaluating the rules — Blocks that should be eliminated (Distinct blocks that were non-probative.)

Distinct blocks for files that could not be on oldComputer	677
matched by ad hoc rules:	600
matched by entropy rule:	600

Evaluating the rules — Blocks that should not be eliminated (distinct blocks that were matched)

kitty and 4 GOVDOCS files	21,469 blocks
Matched by ad hoc rules:	126
Matched by entropy rule:	149
Matched by both:	78
Matched by either:	197

Matching blocks were metadata, unpopulated arrays, control structures, and hybrid blocks containing a mix of data and long strings of nulls.

Step 4 — Target Assembly

After determining ***candidate files***, we reassemble each target file.

Two algorithms.

HASH_SETS — Reports % of the file present on the search drive

- Fast and efficient
- Doesn’t identify where the files are.

HASH_RUNS — Finds the runs of the file for reassembly

- Identifies individual runs of blocks.
- Allows re-creating original files solely from data on the search drive.

Reassembly algorithm #1: HASH_SETS

Each HASH has 1..N FILES at 1..N Offsets

```
(Hash, File#, Offset#)
```

ALL FILES SEEN = Union(ALL HASHES::ALL FILES)

```
SELECT DISTINCT filenum FROM hashes.
```

For each FILE SEEN:

```
% of file recovered = # File Hashes Recovered for that file
                      / # Hashes in File
```

Problems with this algorithm:

- Doesn't handle a single hash being in the file multiple times
- Doesn't handle all of the hashes recovered being for common blocks.

Reassembly Algorithm #2: HASH_RUNS

For each HASH of each IDENTIFIED FILE, we build an array:

```
[disk block#, {file blocks}, count-in-corpus]
```

- We sort the array by disk block #.
- We look for runs of blocks where the (sector #/8) and block # increment in step.

```
[12496591, {0}, 1]
[12496599, {1}, 1]
[12496607, {2}, 1]
[12496615, {3}, 1]
[12496623, {4}, 1]
[12496631, {5}, 1]
```

A run of a distinct video

```
[5891103, {879}, 3207]
[5891111, {880}, 2440]
[5891119, {881}, 1886]
[5891127, {882}, 1596]
[5891135, {883}, 1397]
[5891143, {884}, 1215]
```

A set of alias blocks

Additional rules in the HASH_RUNS algorithm let us throw out spurious blocks.

Singletons that aren't in runs and/or aren't in order are easy to identify:

```
[19252422, {1595}, 1]  //
[19252558, {2609}, 1]  //
[19263478, {2070}, 1]  //
```

—*Such singletons probably aren't probative.*

In cases with identical blocks in multiple locations, it's easy to verify runs:

```
/govdocs/168/168524.doc mod8=7

[10053327, {119, 180}, 2]
[10053335, {120, 181}, 1]
```

The output is a table of file fragments.

Target File name	Score	Start Sector	Start Block	End Block	Sector (mod 8)	Percent Recovered	Allocated File on Target Media
569152.pdf	3	18433052	373	375	4		n/a
569152.pdf	3	19652860	373	375	4	4%	*/WINDOWS/Fonts/courbd.ttf*
…							
970013.pdf	2	18433380	279	280	4	0.2%	n/a
970013.pdf	2	19653188	279	280	4	0.2%	*/WINDOWS/Fonts/courbi.ttf*
215955.ps	3	18192573	571	573	5	0.2%	*/Program Files/ … Data1.cab*
235835.ps	2	18192598	11503	11504	6	0.02%	*/Program Files/Adobe/Reader 9.0/ … Data1.cab*
MontereyKittyHQ.m4v	6132	18639703	0	6131	7	100%	*/Documents and Settings/ … /MontereyKittyHQ.m4v*
TiggerTheCat.m4v	3059	3532519	0	3058	7	100%	*/Documents and Settings/ … /TiggerTheCat.m4v*
KittyMaterial/Cat.mov	1374	18696759	0	1393	7	100%	*/Documents and Settings/ … /Cat.mov*
466982.csv	4	2932551	0	3	7	0%	*… /Cache/F5433139d01*
466982.csv	8	2833023	4	11	7	1%	*… /Cache/F5433139d01*
466982.csv	16	2800423	12	27	7	2%	*… /Cache/F5433139d01*
466982.csv	36	10062599	28	63	7	4%	*… /Cache/F5433139d01*
…							
DSC00072.JPG	234	14306831	0	233	7	100%	*/Documents and Settings/ … /DSC00072.JPG*
…							
809089.eps	6	11907023	0	5	7	100%	*/Python26/ … /pwrdLogo.eps*
574989.csv	4	3713503	0	3	7	67%	*… /Cache/4787E2CCd01*
574989.csv	2	3713359	4	5	7	33%	*… /Cache/4787E2CCd01*
…							
466749.csv	2	5032175	0	1	7	100%	*… Cache/_CACHE_003_*

In this example, all **bold** filenames are actually on the drive.

Overall performance is quite fast.

Database construction:

- Only have to do this once.
- Hashing is parallelized with bulk_extractor
- Building a 500GB database ≈ 24 hours

Media Scanning:

- bulk_extractor runs at I/O speed with enough cores.
  - —*Hashing is parallelized.*
  - —*Database > 100,000 lookups/second*
- Scanning 20GB disk image took less than 10 minutes

Reassembly:

- Main time is reading number of hashdb hits.
- Reassembly time proportional to sort of mostly-sorted array.
  - —*≈30 seconds for 20GB test disk.*

Conclusions

Engineering decisions and lessons learned

512-byte vs. 4KiB:

- We thought we could determine in advance the (mod 8) of a drive.
- But a single drive might have two partitions with different (mod 8) values.
- Within a .DOC file, embedded pictures will be on 512-byte offsets.
- Our solution gives best of both worlds:
  - —*8x extra hashing is not significant on multi-core system (we use MD5)*
  - —*Database is hashed with 4K chunks*
  - —*Probative block selection prevents false positives*

C++ vs. Python:

- Large data operations are (still) best done in C++ (e.g. bulk_extractor, hashdb)
- Python was better for algorithm development.

Flat files vs. databases:

- Regret not leveraging bulk_extractor's SQLite support.

In summary

We've been talking about hash-based carving for ≈10 years.

To date, most of the tools would only search for a few target files.

- Previously most researchers thought that finding a few high-entropy blocks from a target file meant that the target file had probably been present.
- We were wrong! Many high-entropy blocks that are infrequent but not distinct.

We developed a system that can search for a million files, 500GB of target data.

- High-performance implementation with publicly available tools.
- All you need is a large database of target content.

Available on github today

- https://github.com/simsong/bulk_extractor — current bulk_extractor with HBC.
  `python/report_identified_runs.py`
- https://github.com/simsong/hashdb — the hashdb library and command line tool

Questions?