md5bloom: Forensic Filesystem Hashing Revisited

By

Vassil Roussev, Timothy Bourg, Yixin Chen, Golden Richard

Presented At

The Digital Forensic Research Conference

DFRWS 2006 USA Lafayette, IN (Aug 14th - 16th)

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

The 6th Annual Digital Forensic Research Workshop (DFRWS 2006)
Aug 14-16, Lafayette, IN

`md5bloom`: Forensic Filesystem Hashing Revisited

Vassil Roussev
Yixin Chen
Tim Bourg
Golden G. Richard III

Department of Computer Science
University of New Orleans
{vassil,yixin,tbourg,golden}@cs.uno.edu

One-slide Hashing Refresher

- Hashing:
  - Mapping of a large/infinite domain (e.g., all strings) to a small/finite one (e.g., 128-bit strings)
- Properties:
  - Different hashes → different objects
  - Identical hashes (strongly) suggest identical objects
    - Chance collisions (false positives) are possible
    - Good hash functions minimize such probability
    - For **cryptographic** ones it should be virtually zero

Forensic Use of Hashes

- Basic use
  - Assume that identical hashes imply identical objects
  - Compare *before* and *after* hashes
- Scenarios
  - Verify integrity of evidence
  - Automatically identify known content during an investigation
- Hashing pros:
  - Fast, generic, reliable
- Hashing cons:
  - Fragile!

Open Issues

- Evidence validation works just fine, thank you!
- How about content identification?
  - Example scenario:
    - Simple HDD correlation: 500GB vs. 500GB (vs. 500GB …)
    - Related Q: what summaries should we keep of old cases?
  - Approach #1: use NSRL
    - ~ 50,000,000 ***file*** hashes—a **drop** in the bucket!
    - Static reference DB vs. dynamic application installations
  - Approach #2: generate custom hashes
    - File hashes ➔ non-sequential I/O ➔ not scalable!
      ➔ will not identify versions of known files

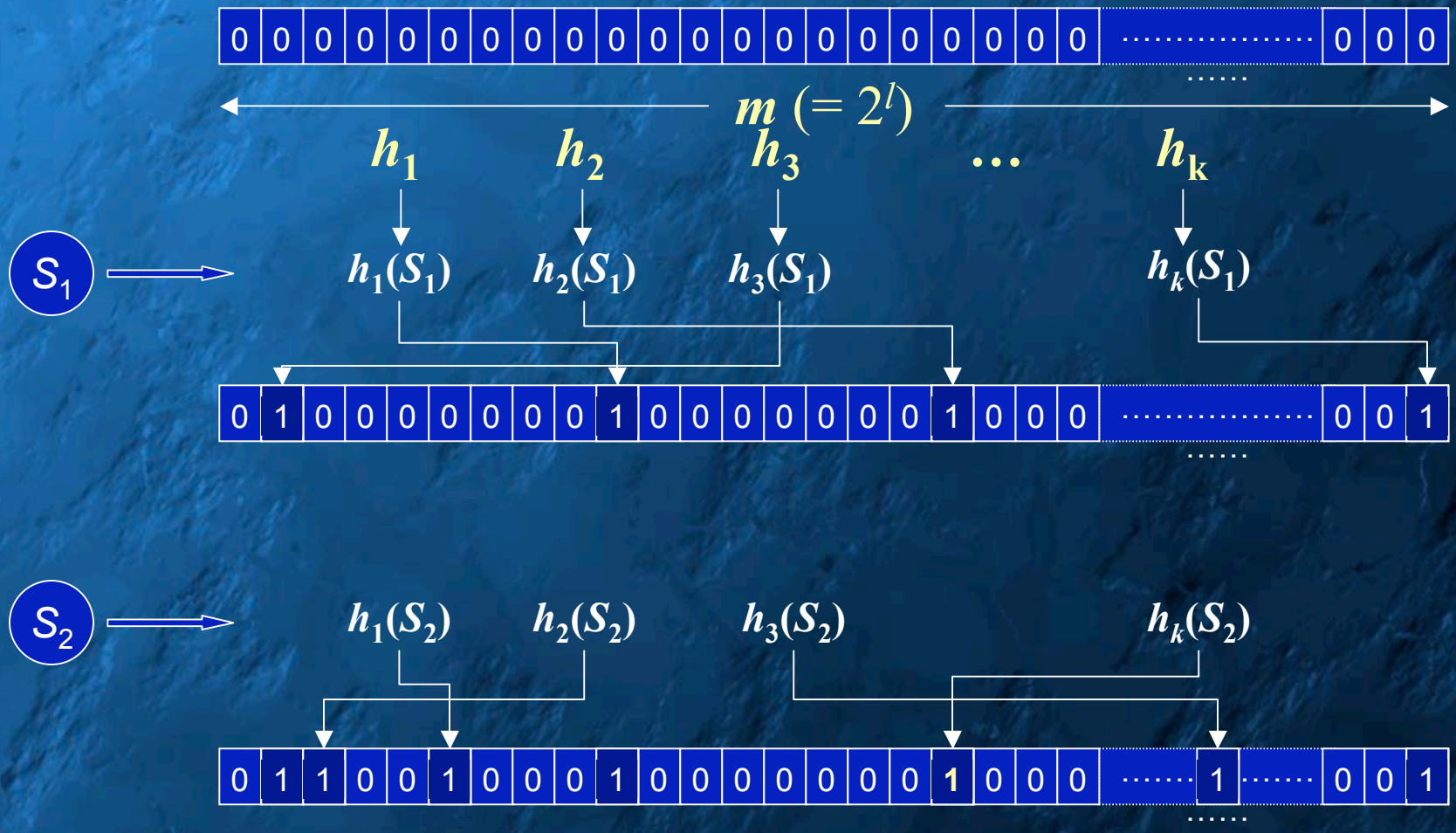
Better Content Identification

- Basic ideas:
  - Finer-grain hashing (e.g. block-level)
    - Linear I/O access—optimal performance
    - Showstopper: RAM
      - Block-level MD5s for 512GB == 16GB
      - ➔ need ~32GB RAM hash table for efficient access
      - RAM & HDD capacities grow at approx the same rate
    - ➔ Need better in-memory representation
  - Structure-based hashing of composite objects
    - Hash logical sub-components (e.g. functions in a library)
    - ➔ Need a unified 'hash bag' representation & comparison rules

Bloom Filters as Hash Bags

- Bloom filter:
  - A simple, space-efficient probabilistic set representation.
    - Operations: insert, lookup, (delete)
      - No false negatives
      - False positives possible (the price of efficiency)
  - Allows quantified trade off between space efficiency and error rate (false positives)
- Numerous variations and uses:
  - Counting, hierarchical, compressed, …
  - Dictionaries, network ID, distributed caching …
  - All uses based on membership queries
  - ➔ no direct comparisons of filters

Bloom Filter Basics



Using MD5s in a *Bloom* Filter

- For all practical purposes, MD5 (SHA-xxx, etc.) is a perfect hash function
  - We can use *any* subset of bits ('subhash) as an independent hash value
  - Predicted FP rates have actually been observed
  - E.g.: $l = 16$ ($m = 2^{16}$)

False Positive (Error) Rates

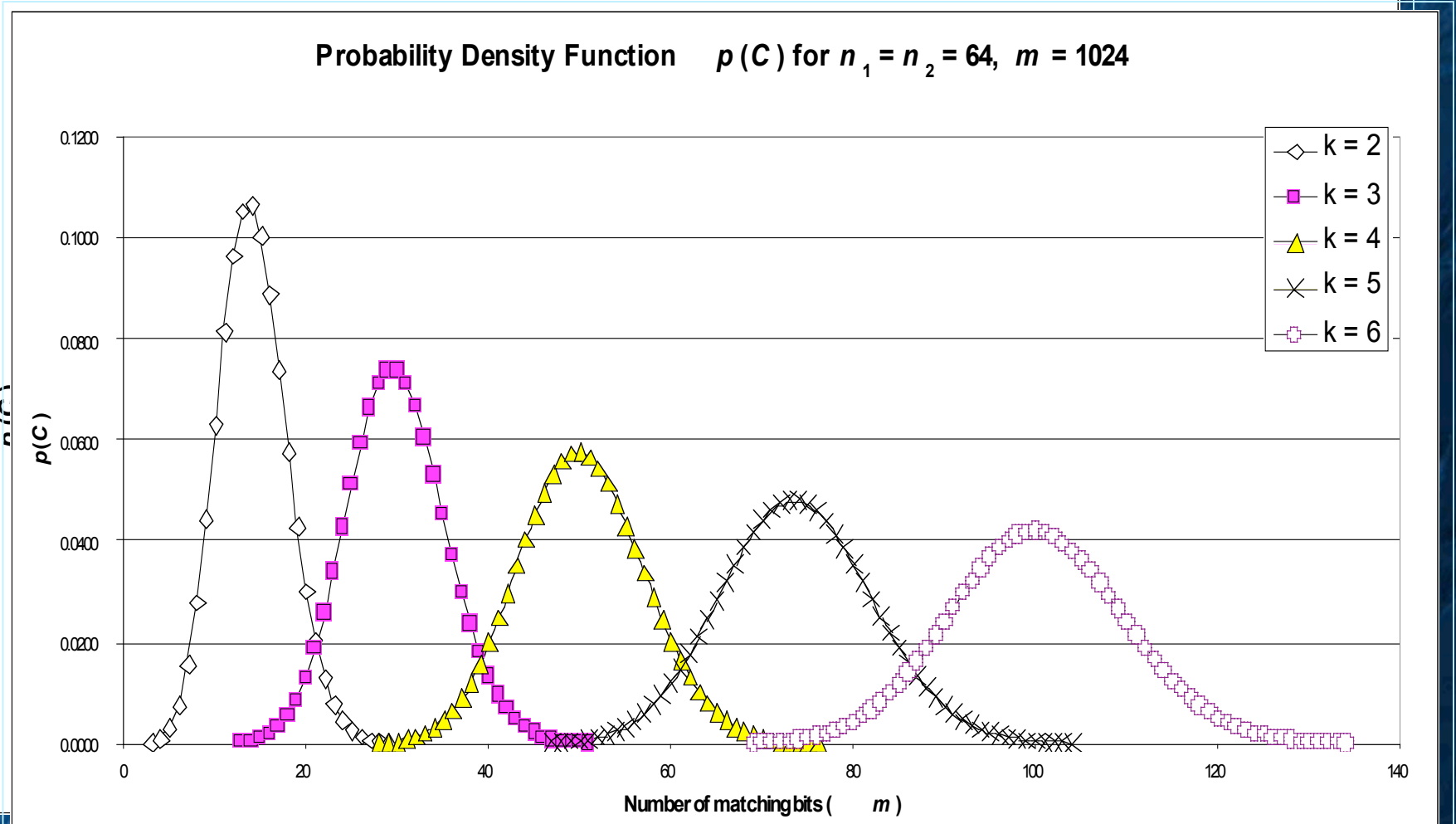
m = 1024		*k*					
		2	4	6	8	12	16
m / *n*	16	0.0138	0.0024	0.0009	0.0006	0.0005	0.0007
	14	0.0177	0.0038	0.0018	0.0013	0.0013	0.0022
	12	0.0236	0.0065	0.0037	0.0032	0.0041	0.0075
	10	0.0329	0.0118	0.0085	0.0085	0.0136	0.0272
	8	0.0490	0.0240	0.0216	0.0255	0.0484	0.0979
	4	0.1549	0.1598	0.2201	0.3128	0.5423	0.7444

- ***n*** = number of elements inserted
- *m*/*n* = bits per element

Bloom Filter Comparison

- Q: How to compare two filters?
- A: Count the number of matching bits
  - ***m***, ***k***, and hash functions fixed, ***n*** may vary
- Q: How to interpret the results?
- A: Estimate the probability that the observed number of matching bits happened by chance (*p*-value)
  - I.e., lower is better
  - E.g., $p = 0.01$ ⇔ there is 1% chance for two random filters with the given parameters to have the observed number of matching bits
  - Matching bits ⇔ inverse of Hamming distance
- Formulas—see paper

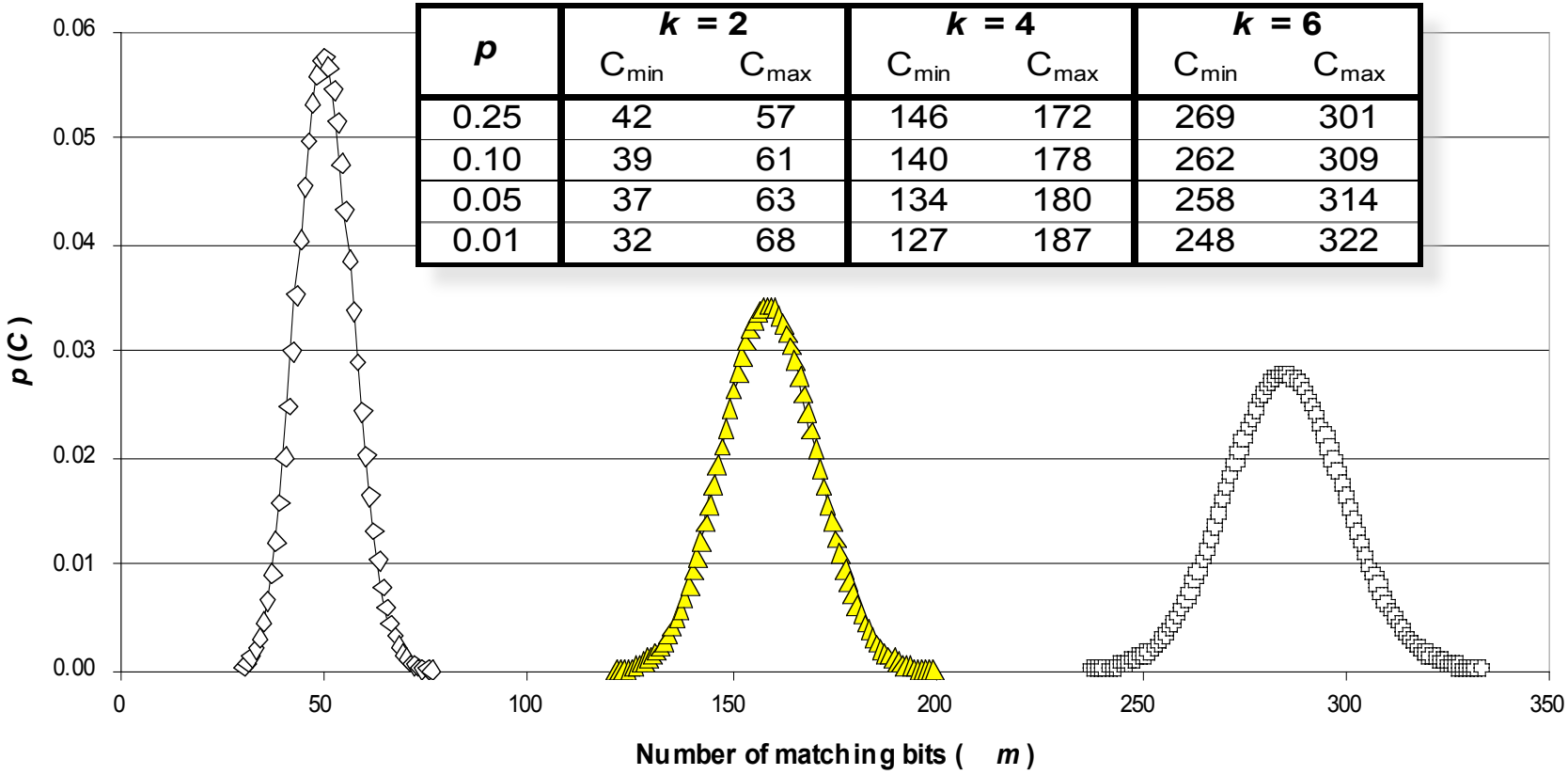
Interpretation: Example *PDFs*



Filter Comparison Interpretation

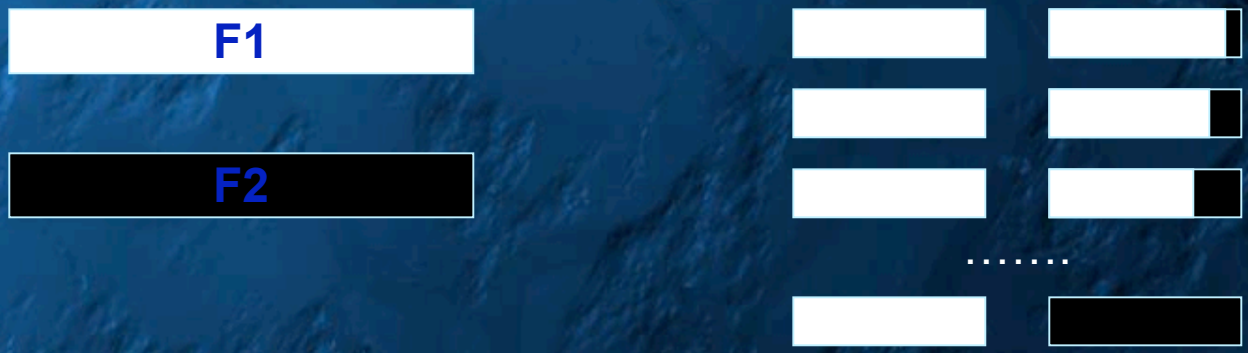
Probability Density Function $p(C)$ **for** $n_1 = n_2 = 128$, $m = 1024$

p	$k = 2$		$k = 4$		$k = 6$	
	C_{min}	C_{max}	C_{min}	C_{max}	C_{min}	C_{max}
0.25	42	57	146	172	269	301
0.10	39	61	140	178	262	309
0.05	37	63	134	180	258	314
0.01	32	68	127	187	248	322

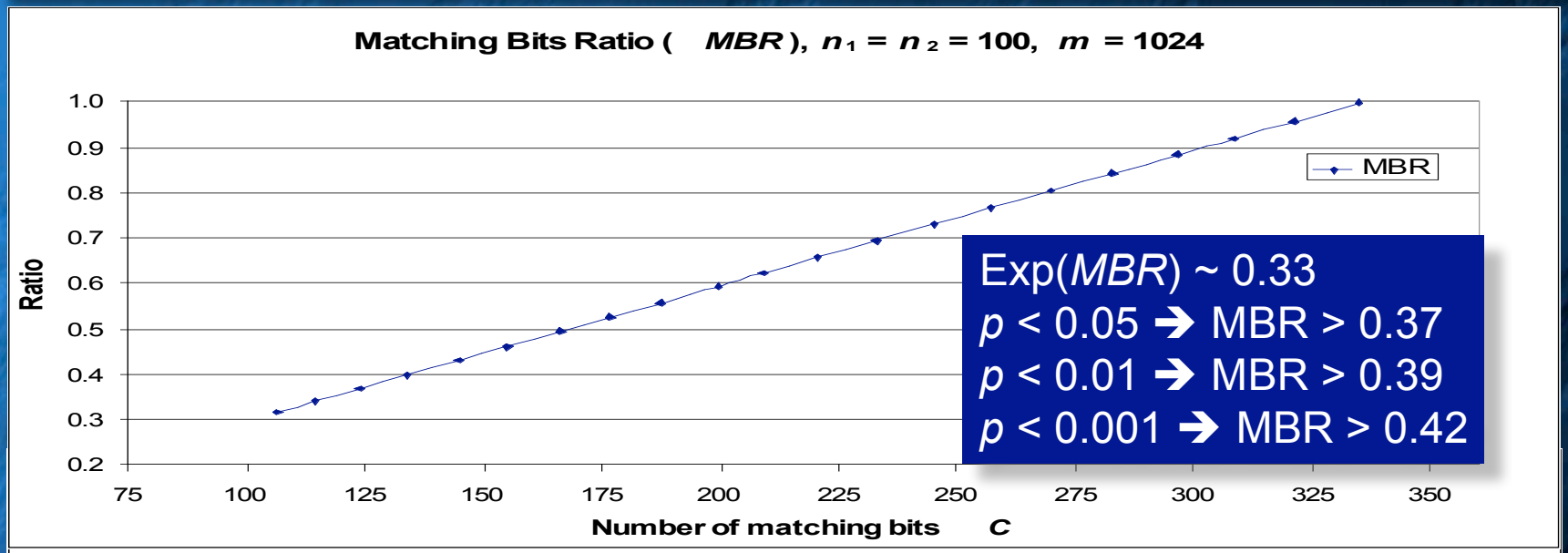


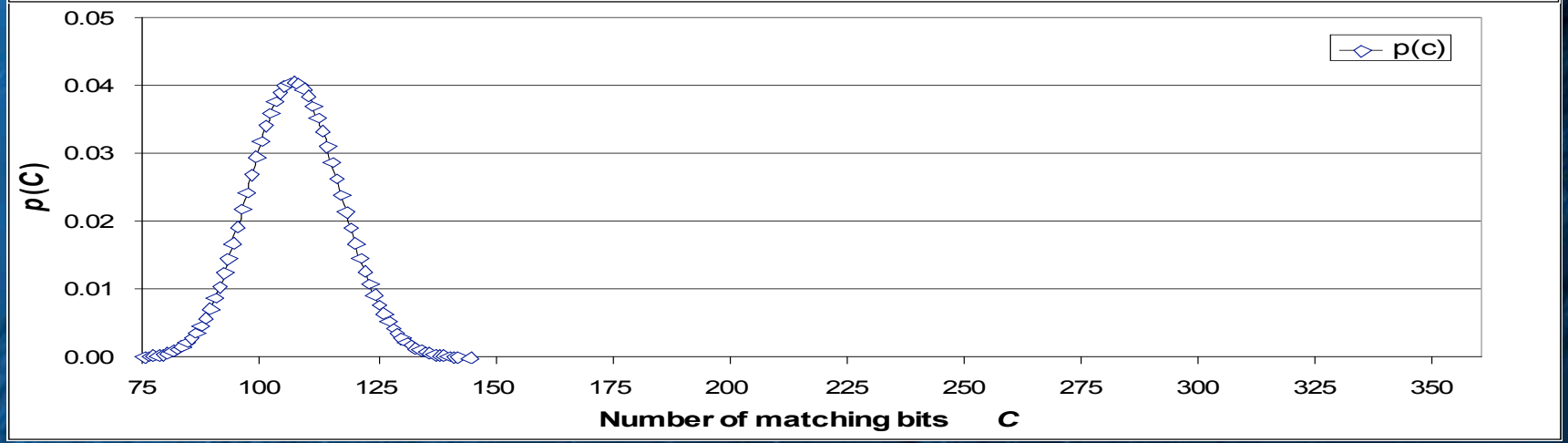
Filter Comparison Interpretation (2)

- Q: Can we relate the fraction of matching bits to the *p*-value?
- A: Demonstration experiment
  - Generate two random filters of MD5 hashes—100 each
  - Compare one filter w/ controlled mix of the two:
    - F_1 vs. 100x/0, F_1 vs. 95/5, F_1 vs. 90/10, …, F_1 vs. 0/100 (F_2)



MBR vs. PDF

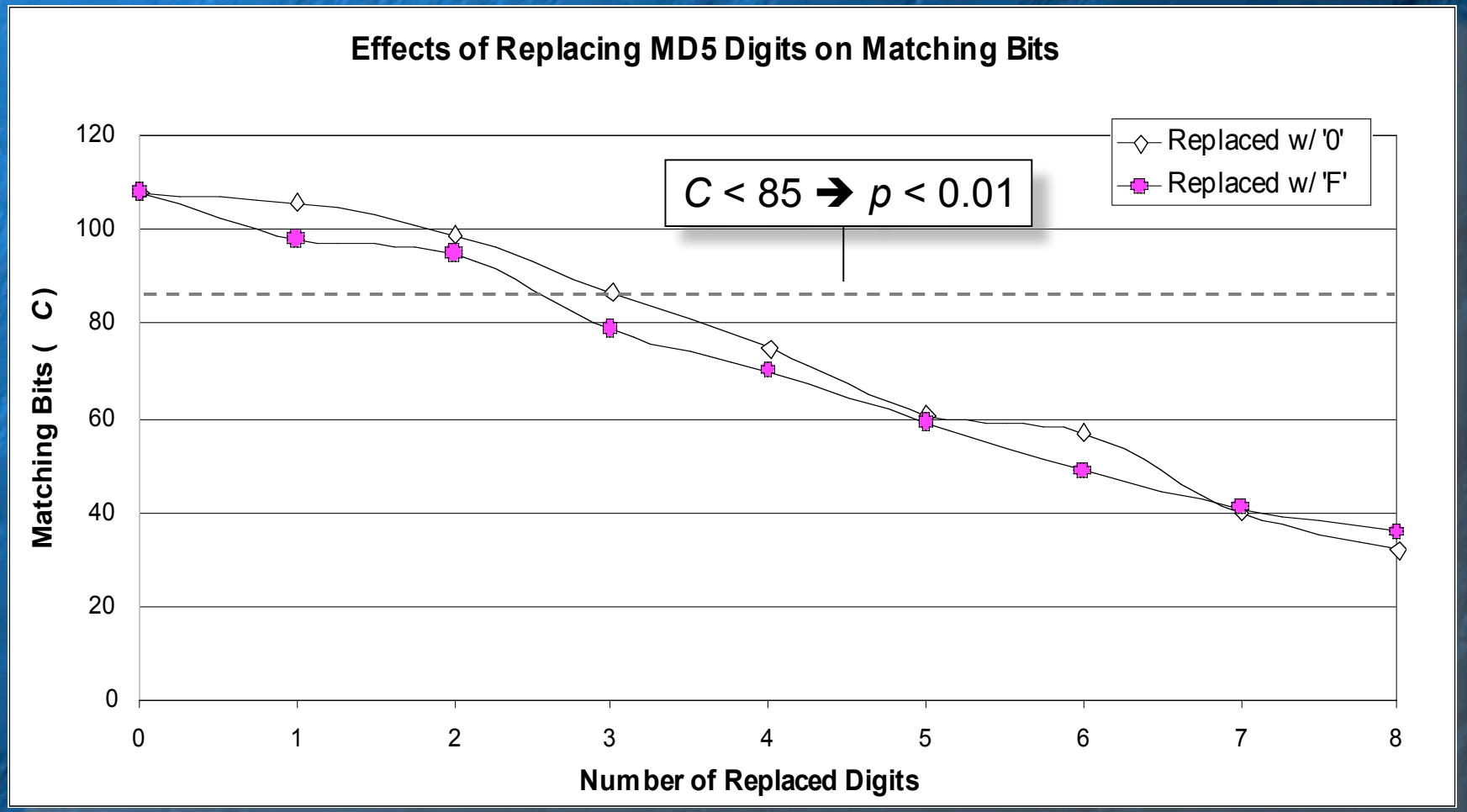

Matching Bits Ratio (MBR), $n_1 = n_2 = 100$, $m = 1024$
Ratio
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
MBR
75
100
125
150
175
200
225
250
275
300
325
350
Number of matching bits C
Exp(MBR) ~ 0.33
p < 0.05 ➔ MBR > 0.37
p < 0.01 ➔ MBR > 0.39
p < 0.001 ➔ MBR > 0.42


p(C)
0.05
0.04
0.03
0.02
0.01
0.00
p(c)
75
100
125
150
175
200
225
250
275
300
325
350
Number of matching bits C

Hash Set Tampering

- Q1: Is it possible for two hash sets to have too few matching bits?
  - I.e., *C* is to the left of the bell curve
- Q2: If so, what does it mean?
- A: Based on hashing function properties, it is extremely (and quantifiably) unlikely to achieve this by chance or by tampering with the original objects.
  - ➔ Any tampering must have happened afterwards
  - E.g., the reference set has been made deliberately different

Tampering Example: MD5 Digit Replacement



Detecting Object Versioning

Name	Reference	Functions
liboskit_fsnamespace.a	fs1	54
liboskit_fsnamespace_r.a	fs2	54
liboskit_posix.a	posix1	103
liboskit_posix_r.a	posix2	127
libpng10.a	png1	303
libpng12.a	png2	329

Detecting Object Versioning (2)

	fs	fs	posix	posix	png
m	256	512	512	1024	1024
n_1	54	54	103	103	303
n_2	54	54	127	127	329
m / n	4.74	9.48	4.45	8.90	3.24
$S_1 \cap S_2$	19	19	53	53	239
C	101	98	218	230	657
p	< 0.015	< 0.001	< 0.001	< 0.001	< 0.01

$k = 4$

Detecting Object Versioning (3)

	fs	posix	png
m / n	4.74	4.45	3.24
$S_1 \cap S_2$	35%	46%	76%
p	< 0.015	< 0.001	< 0.01

- Observations
  - Detecting object versions works very well
  - Depending on the level of overlap, even filters with very low bits per element ratio work reliably
  - In contrast, using similar filters for individual membership queries would be pointless:

$m = 1024$		k					
		2	4	6	8	12	16
m / n	4	0.1549	0.1598	0.2201	0.3128	0.5423	0.7444

Implementation: `md5bloom`

- Goals
  - Develop an open-source solution
  - Efficient, flexible, extendable, reliable
  - Compatible with existing practices
  - Both a standalone utility and a library to be used in other tools
- Design
  - Back-end (*bloom server*): filter management
    - Create, insert, lookup, store/load
    - Hash function independent
  - Front-end (*bloom client*): hash/query management
- Usage scenarios:
  - Same address space
  - Single client/server
  - Multiple client/server

`md5bloom`: Quick Overview

- Functions:
  - Creation
    - Byte stream
    - Existing MD5 hashes
    - Client hashes (over the network)
  - Query lookup
    - Byte stream
    - Existing MD5 hashes
  - Filter comparison

`md5bloom`: Server Examples

- Creation:

  `md5bloom -genstream 10 4 8 512 -daemon 2024`

  - Create filter: $m = 2^{10}$, $k = 4$, $m/n \leq 8$
  - Hash `stdin` every 512 bytes
  - Listen for queries on port 2024

  `md5bloom -genmd5 10 4 8 [-daemon] 2024`

  - Create filter: $m = 2^{10}$, $k = 4$, $m/n \leq 8$
  - Read (hexadecimal) hashes (one per line) from `stdin`
  - No `daemon` ➔ send to `stdout`, else keep in memory

  `md5bloom -genclient 10 4 8 [-daemon] 2024`

  - Create filter: $m = 2^{10}$, $k = 4$, $m/n \leq 8$
  - No `daemon` ➔ send to `stdout`, else keep in memory

`md5bloom`: Client Examples

- Creation
  - **`md5bloom -clientstream host 2024 512`**
    - Connect to server at **`host:2024`**
    - Hash `stdin` every 512 bytes and send to server
  - **`md5bloom -genmd5 host 2024`**
    - Connect to server at **`host:2024`**
    - Read hashes from `stdin` and send to server
- Queries
  - **`md5bloom -querystream host 2024 512`**
    - Hash `stdin` every 512 bytes and query the server
  - **`md5bloom -querymd5 host 2024`**
    - Read hashes from `stdin` and query the server
    - Hash `stdin` every 512 bytes and query the server
- Comparison
  - **`md5bloom -diff <file_1> <file_2>`**

Conclusions

- Bloom filters can be a valuable tool in filesystem forensics to
  - Alleviate scalability problems
  - Provide a framework for finer-grain content identification
- Developed a probabilistic framework that treats filters as first-class objects
  - Direct comparison
  - Reliable interpretation of results
- Provided simulation-based verification of theoretical results
- Developed a practical tool—*md5blooom*—that can be used as a standalone utility, or a component of a system
- Initial testing results
  - Confirm the predicted results
  - Provide new insights into the use and performance of *Bloom* filters

Future Work

- Transition *md5bloom* from alpha to beta
  - Stable code
  - Stable client/server protocol & storage format
  - Competitive performance
- Large-scale testing
  - Various file types
  - HDD-to-HDD correlation
- Develop front-end extensions for various variations of *Bloom* filters
- Explore the use of
  - Chained multi-filter comparisons
  - Recursive filters

Thank You!

Questions?