On Efficiency of Artifact Lookup Strategies in Digital Forensics

L. Liebler, P. Schmitt, **H. Baier**, F. Breitinger

Hochschule Darmstadt, CRISP, da/sec Security Group

2019-04-25

Agenda

Data overload

Source: www.spiegel.de

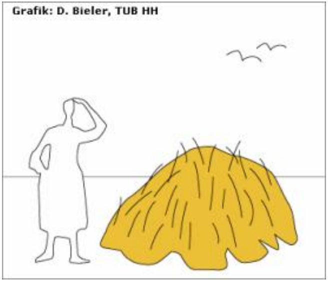
Source: Eric Gaba, Wikimedia; CC-BY-SA

1 TiB digital text equals (approximately):

- 220 million printed pages: 1 page = 5000 characters.
- 1 million kg paper: printed one-sided.

Motivation

Finding relevant artifacts resembles ...



Source: tu-harburg.de

Source: beepworld.de

Digital forensic experts need automated filtering to
reduce the haystack or
increase the needle.

General process pipeline: approximate matching

1. Construction phase of data set (e.g., a blacklist) using approximate matching:
   - Extract blocks / features
   - Hash them
   - Insert hashed block into 'database'
   - Sorting difficult due to fuzzy nature of input
2. Lookup phase:
   - Extract blocks / features from seized device
   - Hash them
   - Comparison against the 'database'

We focus on alternative 'database' approaches to solve the database lookup problem.

Use Case / Goals

1. Use case: find efficient (i.e. fast) strategies to detect known digital traces, e.g., in the context of
   - white- and blacklisting scenarios in forensic use cases
   - carving
   - within large corpora (memory-, lookup-efficient)
2. General goal: discuss, reassess and extend three widespread lookup strategies
3. Further goals:
   - deduplication (i.e., remove common blocks)
   - adding and deleting items

Candidate preselection

Preselection of three 'database' approaches and corresponding lookup strategies suitable for storing hash-based fragments:

- *hashdb*: Hash-based carving due to Garfinkel et al. [GM15], part of the `bulk_extractor`
- *hbft*: Hierarchical Bloom filter trees originally due to Breitinger et al. [BRB14]
- *fhmap*: flat hash map, presented by Malte Skarupke at C++Now in 2018

[GM15] S. Garfinkel, M. McCarrin, Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb, Digital Investigation 14 (2015), pp. 95-105

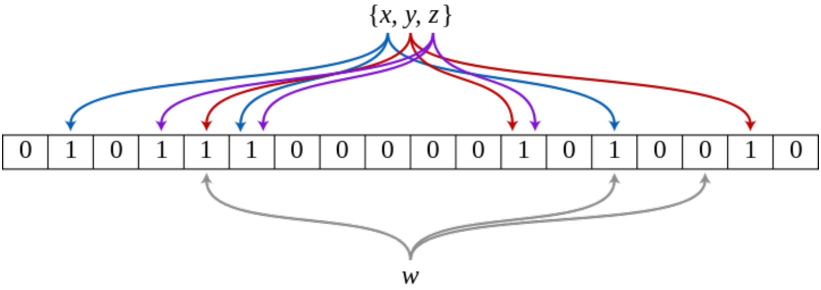
[BRB14] F. Breitinger, C. Rathgeb, H. Baier, An efficient similarity digests database lookup a logarithmic divide and conquer approach, Journal of Digital Forensics, Security and Law (Special Issue: Proceedings of 6th International Conference on Digital Forensics & Cyber Crime, ICDF2C14) 9(2) (2015), pp. 155-166

hashdb: main features

- Uses lightning memory mapped database structure (LMDB)
- Handles large data sets (1 million files in [GM15])
- Read-optimised (read-only transactions operate in parallel)
- Built-in deduplication (common block / multi hit prevention)
- Adding and deleting items is possible
- Uses fixed sliding window for block building

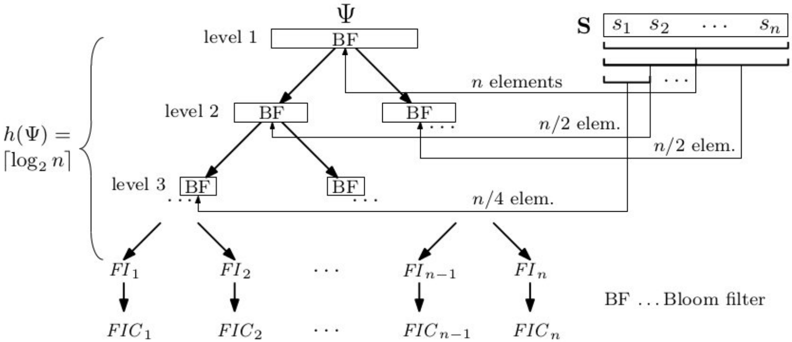
Bloom filter (Burton Howard Bloom in 1970)

- Very space-efficient + probabilistic data structure
- Array with the size of m bits ($m = 18$ in the following sample Bloom filter)



Source: https://commons.wikimedia.org/wiki/User:David_Eppstein

Hierarchical Bloom filter tree (hbft): concept

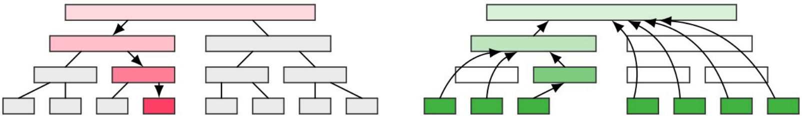


mrsh-hbft proof-of-concept by Lillis et al. [LBS17]

[LBS17] D. Lillis, F. Breitinger, M. Scanlon, Expediting mrsh-v2 approximate matching with hierarchical bloom filter trees, ICDF2C17, (2017), pp. 144-157

hbft: main features

- Lookup complexity of $O(\log(n))$
- **False positive rate** of a bloom filter is influenced by three parameters:
  1. Size of the filter m
  2. Number of n inserted elements of a set $S = \{s_1, ...s_n\}$
  3. Number of used hash functions k
- Deletion of elements hardly possible



flat hash map (fhmap): main features

- Fast hash table (actually the author claims that the implementation features the fastest lookups until now): lookup complexity of $O(1)$
- Robin Hood hashing according to [CLM85]: ensures that most of the elements are close to their ideal entry in the table by rearrangement
- No false positives

[CLM85] P. Celis, P.-A. Larson, J. I. Munro, Robin hood hashing, 26th Annual Symposium on Foundations of Computer Science, IEEE (1985), pp. 281-288

Requirements / Capabilities

Multi hit handling

- Identical blocks of different files (e.g., file header structures, statically linked libraries)
- Often no value to an analyst (block is not characteristic for a given artifact)
- Needs to be filtered out (during construction or lookup phase)
- Keep multi hits which only appear within one file

Summary capability analysis

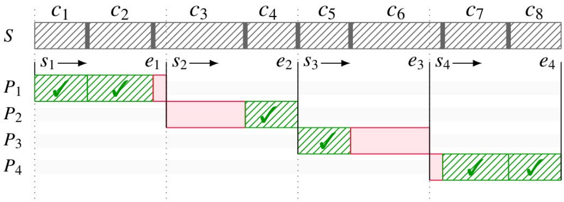
A direct comparison is hard as capabilities differ → re-implementation of several features needed

	hashdb	hbft	fhmap
Storing Technique	LMDB	Bloom filter tree	Hash table
Block Building	Fixed sliding window	**Fixed size*** / rolling hash	**Fixed size*** / **rolling hash***
Block Hashing	MD5	FNV-256	FNV-1
Multithreading	All phases	**Block building***	**Block building***
Multihit Handling	✓	*	*
Add / Remove Hashes	✓/ ✓	**Partially / ˆ**	✓/ ✓
Prefilter	"Hash Store"	Root Bloom filter	✗
False Positives	✗	✓	✗
Storing Type	Single-level storage	Primary storage	Primary storage
Not limited to RAM	✓	✗	✗
Persistent Database	✓	✓	*

Overview of implemented extensions

- Multi hit prevention hbft:
  - Tree-filter based
  - Global-filter based
  - Evaluation
- Multi hit prevention fhmap
- Parallelisation of block building

Parallelisation of block building

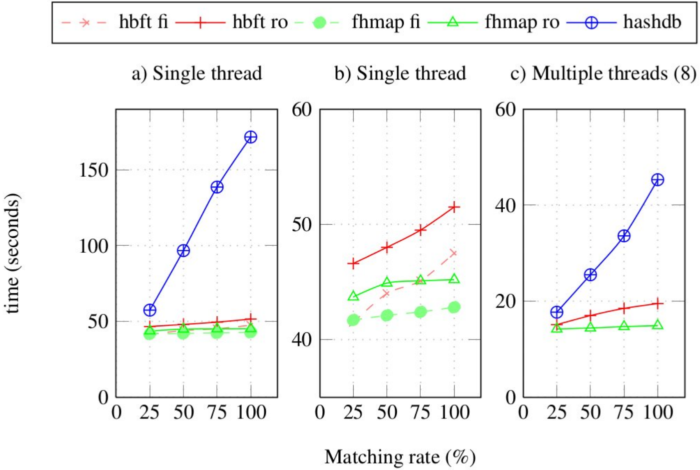


	Singlethread	Multithread (8 Threads)
Real	43.82 s	13.59 s
CPU	35.87 s	49.25 s

Overview of evaluated aspects

- Memory consumption
- Run time of construction phase:
  - Single threaded
  - Multiple threaded
- Run time of deduplication:
  - Single threaded
  - Multiple threaded
- Run time of lookup phase (depending on matching rate)

Lookup evaluation



Overall evaluation

	hashdb	hbft	fhmap
Multithreading	++	0	0
Add Hashes	++	-	++
Remove Hashes	++	- -	++
Limited to RAM	++	-	-
Transactions	++	-	-
Persistent Database	++	+	+
Prefilter	+	+	0
False Positives	+	-	+
Memory Usage	-	+	+
Build Phase (Single)	-	++	++
Build Phase (Multiple)	+	++	++
Deduplication Phase (Single)	-	-	+
Deduplication Phase (Multiple)	-	-	+
Lookup Phase (Single)	-	++	++
Lookup Phase (Multiple)	0	++	++

Conclusion

Conclusion

- fhmap outperforms both hbft and hashdb for our use case
- Extending hbft is hard without loosing its advantages
- fhmap integrated into the memory carving engine

Contact

- harald.baier@h-da.de / lorenz.liebler@h-da.de / FBreitinger@newhaven.edu
- Interested in internship at CRISP?