4-2019

On Efficiency of Artifact Lookup Strategies in Digital Forensics

Lorenz Liebler
CRISP, Center for Research in Security and Privacy

Patrick Schmitt
Technische Universität Darmstadt

Harald Baier
Hochschule Darmstadt

Frank Breitinger
University of New Haven, fbreitinger@newhaven.edu

Publisher Citation

Liebler, L., Schmitt, P., Baier, H., & Breitinger, F. (2019). On efficiency of artifact lookup strategies in digital forensics. Digital Investigation, 28, S116-S125.

Digital Investigation 28 (2019) S116–S125

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

On efficiency of artifact lookup strategies in digital forensics

Lorenz Liebler [a, b, *], Patrick Schmitt [c], Harald Baier [a, b], Frank Breitinger [d]

[a] da/sec Biometrics and Internet Security Research Group, Hochschule Darmstadt, Darmstadt, Germany
[b] CRISP, Center for Research in Security and Privacy, Darmstadt, Germany
[c] Secure Software Engineering Group, Technische Universität Darmstadt, Darmstadt, Germany
[d] Cyber Forensics Research and Education Group (UNHcFREG), University of New Haven, New Haven, USA

ARTICLE INFO

Article history:

Keywords:
Database lookup problem
Artifact lookup
Approximate matching
Carving

ABSTRACT

In recent years different strategies have been proposed to handle the problem of ever-growing digital forensic databases. One concept to deal with this data overload is data reduction, which essentially means to separate the wheat from the chaff, e.g., to filter in forensically relevant data. A prominent technique in the context of data reduction are hash-based solutions. Data reduction is achieved because hash values (of possibly large data input) are much smaller than the original input. Today's approaches of storing hash-based data fragments reach from large scale multithreaded databases to simple Bloom filter representations. One main focus was put on the field of approximate matching, where sorting is a problem due to the fuzzy nature of the approximate hashes. A crucial step during digital forensic analysis is to achieve fast query times during lookup (e.g., against a blacklist), especially in the scope of small or ordinary resource availability. However, a comparison of different database and lookup approaches is considerably hard, as most techniques partially differ in considered use-case and integrated features, respectively. In this work we discuss, reassess and extend three widespread lookup strategies suitable for storing hash-based fragments: (1) Hashdatabase for hash-based carving (*hashdb*), (2) hierarchical Bloom filter trees (*hbft*) and (3) flat hash maps (*fhmap*). We outline the capabilities of the different approaches, integrate new extensions, discuss possible features and perform a detailed evaluation with a special focus on runtime efficiency. Our results reveal major advantages for *fhmap* in case of runtime performance and applicability. *hbft* showed a comparable runtime efficiency in case of lookups, but *hbft* suffers from pitfalls with respect to extensibility and maintenance. Finally, *hashdb* performs worst in case of a single core environment in all evaluation scenarios. However, *hashdb* is the only candidate which offers full parallelization capabilities, transactional features, and a Single-level storage.

Introduction

Approximate matching (a.k.a. fuzzy hashing or similarity hashing) is a common concept across the digital forensic community to do known file/block identification in order to cope with the large amounts of data. However, due to the fuzzy nature of approximate hashes current approaches suffer from the *Database Lookup Problem* (Breitinger et al., 2014a). This problem is based on the decision, if a given fingerprint is member of the reference dataset. The general database lookup problem is of complexity $O(n)$ in terms of the number of queries and hence exponential. To address this problem, different techniques have been discussed such as multiple Bloom filters, single large Bloom filters, Cuckoo filters or Hierarchical Bloom filter trees (Harichandran et al., 2016; Lillis et al., 2017).

Besides the complexity, an investigator has to deal with common blocks which makes the identification of the correct match hard (Garfinkel and McCarrin, 2015). The extraction and correct assignment of a specific data fragment is of crucial importance, i.e., those identified chunks allow the inference of the original source (e.g., a potentially malicious file or a media file). Specificially, different files of the same type or application often share a non-negligible amount of *common blocks*, e.g., file structure elements in the file header. This leads to *multihits*, hence, those blocks or chunks are not suitable for a unique identification of a specific file. To avoid this problem, lookup strategies should consider additional

* Corresponding author. da/sec Biometrics and Internet Security Research Group, Hochschule Darmstadt, Darmstadt, Germany.
E-mail address: lorenz.liebler@h-da.de (L. Liebler).

https://doi.org/10.1016/j.diin.2019.01.020

mechanisms to handle common blocks, e.g., by integrating functions of filtration or deduplication. Those requirements influence the applicability of a specific lookup strategy. The consideration of common blocks influences the results of higher level analysis, too (e.g., approximate matching used for the task of identifying similar binaries or detecting shared libraries (Liebler and Breitinger, 2018; Pagani et al., 2018)).

In this work we discuss, reassess and extend three widespread lookup strategies suitable for storing hash-based fragments which have been proposed and are currently utilized in the field of digital forensics:

1 *hashdb:* In 2015 Garfinkel and McCarrin (2015) introduced *Hash-based carving*, "a technique for detecting the presence of specific target files on digital media by evaluating the hashes of individual data blocks, rather than the hashes of entire files". Common blocks were identified as a problem and have to be handled or filtered out, as they are not suitable for identifying a specific file. To handle the sheer amount of digital artifacts and to perform fast and efficient queries, the authors utilized a so called *hashdb*. The approach was integrated into the bulk_extractor forensic tool. Both implementations have been made publicly available.[1]
2 *hbft:* In the scope of approximate matching, probabilistic data structures have been proposed to reduce the amount of needed memory for storing relevant artifacts. Approaches to store artifacts comprise multiple Bloom filters (Breitinger and Baier, 2012), single Bloom filters 8 or more exotic Cuckoo filters (Fan et al., 2014; Gupta and Breitinger, 2015). One major problem of probabilistic data structures is the fact of losing the ability to actually identify a file. In 2014 Breitinger et al. (2014b) provided a theoretical concept of structured Bloom filter trees for identifying a file. In 2017, a more detailed discussion and concrete implementation was provided by Lillis et al. (2017). The approach is based on "the well-known divide and conquer paradigm and builds a Bloom filter-based tree data structure in order to enable an efficient lookup of similarity digests". This leads to Hierarchical Bloom filter trees (*hbft*).
3 *fhmap:* Recently Malte Skarupke presented a fast hash table called *flat_hash_map*[2] (*fhmap*). The author claims that the implementation features the fastest lookups until now. A hash table features a constant lookup complexity of $O(1)$ given a good hash function. The database implementation provides an interface for accessing the hash table itself, however, it does not feature any image slicing, chunk extraction or hashing. Thus, in order to utilize and evaluate *fhmap* in our context, it has to be extended by additional concepts to extract data fragments comparable to Hash-based carving or fuzzy hashing.

Considering our depicted candidates and the overall goal of reassessing those in terms of capabilities and performance, the goals of this paper are as follows:

1. Assess the aforementened proposed techniques for the task of fast artifact handling (i.e., *hbft*, *fhmap* and *hashdb*). Identify the **capabilities** of those techniques and the possible handling of *common blocks.*
2. We inspect the feasibility and discuss concepts to integrate the missing feature of multihit prevention (**filtration** of *common blocks*) similar to *hashdb*, into *hbft* or *fhmap*.
3. Discuss possible extensions of existing techniques in order to be able to compare the approaches.
4. Assess how the different approaches compete with respect to **runtime performance** and resource usage.

Our result is that *fhmap* is best in case of runtime performance and applicability. *hbft* showed a comparable runtime efficiency in case of lookups, but *hbft* suffers from pitfalls with respect to extensibility and maintenance. Finally, *hashdb* performs worst in case of a single core environment in all evaluation scenarios, however, it is the only candidate which offers full parallelization capabilities and transactional features.

The remainder of this work is structured as follows: In Section 'Candiate and features analysis' we give a short introduction in our considered use case. In addition, an overview of the depicted evaluation candidates and their already integrated features is given. In Section 'Extensions to hbft and fhmap' we describe our proposed extensions and our performed evaluation of those. We give a detailed performance evaluation and discuss advantages and disadvantages of the different techniques in Section 'Evaluation'. Finally, we conclude this work.

Candidates and feature analysis

As all of the mentioned approaches strongly differ (either in their original use case, or in their supported capabilities) we outline the motivation behind our decision of the depicted candidates. Therefore, we first describe the conditions of application and introduce the forensic use case which formalizes additional requirements and mandatory features (see Section 'Use case and requirements'). Afterwards, we explain our three candidates of choice in Section 'Depicted Candidates'. Beside the required features of our considered use case, we need to discuss the already present features and capabilities of the different approaches. Thus, we outline the existing features and capabilities for each candidate in Section 'Feature Analysis'.

Use case and requirements

In this work we address the problem of querying digital artifacts out of a large corpus of relevant digital artifacts. Sample applications are carving or approximate matching. Both applications suffer from the *Database Lookup Problem*, i.e., how to link an extracted artifact within a forensic investigation to a corresponding source of a forensic corpus efficiently (i.e., in terms of required storage capacity, required memory capacity or lookup performance). Beside those, our digital forensic scenarios bare additional pitfalls and challenges.

We consider the extraction of chunks (i.e., substrings) out of a raw bytestream, without the definition of any extraction process. A major challenge of matching an artifact to a source are occurring *multihits*, i.e., one chunk is linked to multiple files. This was first mentioned by Foster (2012). A multihit is also called a *common block* or a *non-probative block*, too. For instance Microsoft Office documents such as Excel or Word documents share common byte blocks across different files (Garfinkel and McCarrin, 2015). Similar problems occur during the examination of executable binaries which have been statically linked (e.g., share a large amount of common code or data). Summarized, multi matches are a challenge for identifying an unknown fragment with full confidence. In addition, storing multihits also increases memory requirements and decreases lookup performance.

[1] https://github.com/simsong/(last accessed 2018-10-23).
[2] You Can Do Better than std::unordered_map: New and Recent Improvements to Hash Table Performance presented by Malte Skarupke at C++Now in 2018; https://probablydance.com/2018/05/28/a-new-fast-hash-table-in-response-to-googles-new-fast-hash-table/(last accessed 2018-10-23).

Multihits can either be identified during the *construction phase* of the database (e.g., by deduplication or filtration) or during the *lookup phase*. By filtration of common blocks during a construction phase, the overall database load gets reduced and the lookup speed is increased as only unique hits are considered. Two different strategies of multihit prevention during the construction phase were proposed. First, as introduced by Garfinkel and McCarrin (2015) *rules* are defined to filter out known blocks with a high occurrence (and thus low identification probability of an individual file). Such an approach requires extensive pre-analysis of the input set and its given structures. A second approach is the filtration of common blocks during construction by the additional integration of a deduplication step. Beside *hashdb*, none of our candidates provide deduplication or multihit prevention techniques so far. We refer to Garfinkel and McCarrin (2015) for further details and solely focus on the utilized database in the following subsection.

Features of adding and deleting artifacts have the major benefit of not needing to re-generate the complete database everytime a new artifact needs to be included. While deleting inputs may be less frequent, adding new items to an existing storage scheme seems obvious and indispensable. While *fhmap* and *hashdb* support adding and deleting hashes from their scheme, this feature is not yet available in the current prototype of *hbft*. In detail, adding new elements to a *hbft* is possible, however, the tree needs to be re-generated as soon as a critical point of unacceptable false positives is reached. The definition of buckets also limits the capabilities to add further files to the database. Loosing the capabilities of deleting elements out of a binary Bloom filter is the main reason for making features of deletion impossible to realize. Summarized, adding and deleting hashes from a database is a mandatory or optional feature, depending on the specific use case.

Depicted candidates

In this section we present three widespread lookup strategies suitable for storing hash-based fragments: (1) Hashdatabase for hash-based carving (*hashdb*), (2) hierarchical Bloom filter trees (*hbft*) and (3) flat hash maps (*fhmap*).

LMDB/hashdb. To store the considered blocks Garfinkel and McCarrin (2015) make use of *hashdb*, a database which provides fast hash value lookups. The idea of *hashdb* is based on Foster (2012) and Young et al. (2012). In 2018, the current version (3.1[3]) introduces significant changes compared to the original version mentioned by Garfinkel and McCarrin (2015).

The former implementation of *hashdb* originally supported B-Trees. Those have been replaced by the *Lightning Memory Mapped Database* (LMDB) which is a high-performance and fully transactional database (Chu, 2011). It is a key-value store based on B + Trees with shared-memory features and copy-on-write semantics. The database is read-optimised and can handle large data sets. The technique originally focused on the reduction of cache layers by mapping the whole database entirely into memory. Direct access to the mapped memory is established by a single address space and by features of the operating system itself. Storages are considered as primary (RAM) or secondary (disk) storages. Data which is already loaded can be accessed without a delay as the data is already referenced by a memory page. Accessing not-referenced data triggers a page-fault. This in turns leads the operating system to load the data without the need of any explicit I/O calls. Summarised, the fundamental concept behind LMDB is a single-level store, the mapping is read-only and write operations are performed regulary. The read-only memory and the filesystem are kept coherent through a Unified Buffer Cache. The size is restricted by the virtual address space limits of an underlying architecture. As mentioned by Chu (2011), on a 64 bit architecture which supports 48 addressable bits, this leads to an upper bound of 128 TiB of the database (i.e., 47 bits out of 64 bits).

hbft. The concept of hierarchical Bloom filter trees (*hbft*) is fairly new. This theoretical concept was introduced by Breitinger et al. (2014b) and later implemented by Lillis et al. (Lillis et al., Scanlon). The lookup differs from the approximate matching algorithm *mrsh-v2*, as *hbft* only focuses on fragments to identify potential buckets of files. A parameter named *min_run* describes how many consecutive chunk hashes need to be found to emit a match. A good recall rate was accomplished for $min\_run = 4$. The tree structure is then traversed further if a queried file is considered a match in the root node. Each of the nodes is represented by a single Bloom filter which empowers to traverse the tree. A traditional pairwise comparison can be done at possible matching leaf nodes. For details of the actual traversing concept we refer to the original paper (Lillis et al., Scanlon).

Just like previous *mrsh* implementations the lookup structure can be precomputed in advance. First, the tree is constructed with its necessary nodes. Then the database files are inserted. Thus, the time for construction can be neglected in the actual comparison phase. More precisely the tree structure is represented as a space efficient array where each position in the array points to a Bloom filter. The implementation uses a bottom up construction which fills trees from the leaf nodes to the root. The array representation does not store references to nodes, its children, or leaves explicitely. Every reference needs to be calculated depending on the index in the array. Efficient index calculations are only applicable for binary trees. The lookup complexity within the tree structure is $O(\log_x(n))$, where x describes the degree of the tree and n is the file set size.

fhmap. Flat hash maps have been introduced as fast and easy to realize lookup strategies. Up to now, they have been mainly discussed in different fields of application. Similar to *hbft*, the actual implementation of *fhmap* represents a proof of concept implementation with good capabilities but limited features.

The concept of flat hash maps is an array of buckets which contain multiple entries. Each entry consists of a *key-value* pair. The *key* part represents the identifier for the *value* and is usually unique in the table. An index of the bucket is determined by a hash function and a modulo operation. The position i equals to: $hash(key) \bmod size(table)$. A large amount of inserts into a small table causes collisions, where multiple items are inserted in the same bucket. A proper hash function needs to be chosen in order to maintain a lookup complexity of $O(1)$. The function needs to spread the entries without clustering. The amount of inserted items is denoted by the *load factor*, i.e., the ratio of entries per buckets. A high load factor obviously causes more collisions. The table gets slower since the buckets have to be traversed to find the correct entry. The lower the load factor the faster the table. However, more memory is required since buckets will be left empty on purpose. If a slot is full the entries are re-arranged.

The whole table is implemented as a contiguous (*flat*) array without buckets which allows fast lookups in memory. With linear probing the next entry in the table is checked if its free. If not the next one is checked until either a free slot is found or the upper probing limit is reached. The table is re-sized as soon as a defined limit is reached. The default *load factor* of this table is 0.5. Specific features should speed up the lookup phase: open addressing, linear probing, Robin Hood hashing, prime number of slots and an upper limit probe count. Robin Hood hashing introduced by (Celis et al., 1985) ensures that most of the elements are close to their ideal entry in the table. The algorithm rearranges entries: elements

[3] http://downloads.digitalcorpora.org/downloads/hashdb/hashdb_um.pdf (last accessed 2018-10-23).

Table 1
Features of hashdb, hbft and fhmap. New implementations are marked with asterisks (symbol *, table cell is coloured green) and potential techniques are marked with a caret (symbol ˆ, table cell is coloured red).

	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	✓	✓	*

which are very far will be positioned closer to their original slot, even if it is occupied by another element. The element which occupies this specific slot will also be rearranged from its possibly ideal slot to enable approximately equal distances for each element to its ideal position in the table. The algorithm takes slots from rich elements, which are close to their ideal slot, and gives those slots to the poor, elements which are very far away, hence the name.

Feature analysis

In what follows we shortly inspect the capabilities and properties of all considered techniques. We discuss the current state of each approach in the case of existing features and properties. Table 1 provides a summary of the discussed capabilities. Note, the marks (*) and (ˆ) mean that these attributes are introduced or discussed, respectively, in the course of this work. For instance, an important extension in the case of *fhmap* is the integration of an appropriate chunk extraction and insertion technique.

Block Building. In case of *hashdb* the database building and scanning of images is now possible without the use of *bulk extractor* which was originally proposed to extract chunks. It builds and hashes the blocks with a fixed sliding window which shifts along a fixed step size *s*. Obviously this produces quite a lot of block hashes to be stored in the database. Similar to the original *mrsh-v2* algorithm, the current *hbft* implementation identifies chunks by the usage of a Pseudo-Random-Function (PRF). As soon as the current byte input triggers a previously defined modulus a new chunk boundary is defined. The current implementation sticks to the originally proposed rolling_hash (Breitinger and Baier, 2012). Thus, the extraction of chunks relies on the current context of an input sequence and not on a previously defined block size. Those Context-Triggered Piecewise-Hashing (CTPH) algorithms prevent issues by changing starting offsets of an input sequence. The fixed defined modulus b_m approximates the extracted block size in average. Unlike *hashdb* or *hbft*, the *fhmap* implementation itself does obviously not feature any block building or hashing. As it will just serve as a container, we have to extend the capabilities to extract, hash and store fragments during evaluation.

Block-Hashing Algorithm. In case of *hashdb* the authors propose the cryptographic hash function *MD5* and an initial blocksize *b* of 4 KiB. A value which is obviously inspired by the common cluster size of todays' file systems. *hbft* makes use of the FNV hash function to hash its chunks and sets 5 bits in its leaf nodes and the corresponding ancestor nodes. Since the root Bloom filter is considerably large, the adapted version of FNV which outputs 256 bits is required. Finally, *fhmap* makes use of FNV-1 for hashing an input.

Multithreading support. With respect to multithreading support *hashdb* (or *Lightning Memory Mapped Database*) allows multithreaded reading operations to improve the runtime performance. However, up to now no theoretical or practical concepts are available to integrate mulithreading support into *hbft* and *fhmap*, respectively. Nevertheless the block building phase may use multithreading for both *hbft* and *fhmap*.

Multihit handling. *hashdb* associates hashed blocks with meta data. A meta data describes the count of matching files for a specific block. The saved counts are used to remove duplicates from a database and to rule out multi matches in advance. In detail, all hashed blocks are removed which have an associated counter value higher than one (Garfinkel and McCarrin, 2015). The current prototypes of both *hbft* and *fhmap* do not provide any functionality of deduplication. Thus, a feature for filtering common blocks and common shared chunks is missing. We address this problem in Section 3 and extend both prototypes.

Add and remove hashes. *hashdb* supports adding new hashes into a database (with integrated deduplication). First the new files are split into blocks and hashed. Afterwards, hashes are inserted into the database. The implementation also supports deletion of hashes from a given database by subtracting a database from the original. In order to insert new files into an existing *hbft* database, the tree needs to be rebuilt with the new file hashes if the tree was limited for the original file set. One can save the original block hashes in order to avoid rehashing. The current concept of the structure does not provide any functionality for deleting a given chunk hash. This is obviously primarily caused by the nature of Bloom filters. The original implementation of *fhmap* supports adding and deleting entries by default. After determining the index in the table, the entries are re-arranged as soon as a slot is full. After adding or deleting, the table is optionally re-sized to a final *load factor* of 0.5.

Prefiltering of non-matches. *hashdb* provides prechecking by a *Hash Store*. A *Hash Store* is described as a highly compressed optimized store of all block hashes in the database.[4] In case of *hbft* the root Bloom filter provides an easy discrimination between a match and a non-match and thus, yielding prefiltering of non-matches. The current version of *fhmap* does not provide any prechecking or prefiltering mechanisms. The additional implementation of a Bloom filter may solve this problem and is object of future research.

False Positives. Similar to a probabilistic lookup strategy like

[4] http://downloads.digitalcorpora.org/downloads/hashdb/hashdb_um.pdf (last accessed 2018-10-23).

Bloom filters, the currently proposed *Hash Store* of *hashdb* causes false positives. Even if the concept of prechecking produces false positives, *hashdb* still performs a complete lookup of the queried hash value. Thus, the overall lookup does not suffer from any false positives. A major disadvantage of utilizing a probabilistic lookup strategy like in case of *hbft* is the possible collision of lookups. Thus, the lookup strategy suffers from false positives. The expected value of false positives is controlled by the size of the root Bloom filter and the handled amount of inserts. As *fhmap* performs a full lookup on the stored hash values, the approach does not suffer from any possible false positives.

Limited to RAM. The current implementation of *hashdb* integrates capabilities of loading and storing entries from and to disk. Thus, the approach is not limited to any memory boundaries. The overall design and construction of the *hbft* tree heavily relies on the memory constraints. The original implementation was created as a RAM-resident solution only. The proposed parametrization and initialization mainly focuses on memory boundaries of a target system. In case of *fhmap*, the structure of the contiguous array is directly created in memory and thus, the current approach is limited to the given memory boundaries of system.

Persistent Database The database of *hashdb* is persistent. The current implementation of *hbft* offers the possibility to save and load a database to and from disk. The recent prototype of *fhmap* was only proposed as a simple proof of concept. No features for saving or restoring a disk-based database have been considered so far.

Extensions to *hbft* and *fhmap*

In this section we discuss extensions of our candidates *hbft* and *fhmap* with respect to both already implemented and potential future ones. The extended code is available for both *hbft*[5] and *fhmap*.[6] In contrary, *hashdb* already fulfills most of the required capabilities and is the only ready-to-use approach for our considered context. In Section 'Multihit Prevention fhmap' we first discuss strategies for the handling of multihits in the case of *hbft*, benchmark them, and depict a proper candidate. In Section 'Chunk Extraction hbft and fhmp' we discuss the possible multihit prevention via deduplication for *fhmap*. The integration of persistence for *fhmap* will not be outlined in detail in the course of this work. In Section 3.3 we discuss the chunk extraction process in the case of hbft and fhmap. We will additionally introduce a concept for the parallelization of the chunk extraction via a rolling hash function. Finally, we will outline in Section 'Theoretical Extensions' some theoretically extensions and thoughts.

Multihit prevention hbft

The prevention of multihits (i.e., the filtration of common blocks) could be differentiated at the construction or at the lookup phase. In the following we discuss two approaches of multihit prevention, one realized during the construction phase and one during the lookup phase.

Tree-filter based. By the utilization of a temporary *hbft* for each file, multihit chunks could be marked during the construction phase. Each file is processed sequentially one after another. As can be seen in Fig. 1, a temporary *hbft* stores the chunks of a currently selected file. The following files are compared against the temporary *hbft*. A multihit is highlighted within the tree by a counter. The currently compared chunk of a processed file is also labeled with a

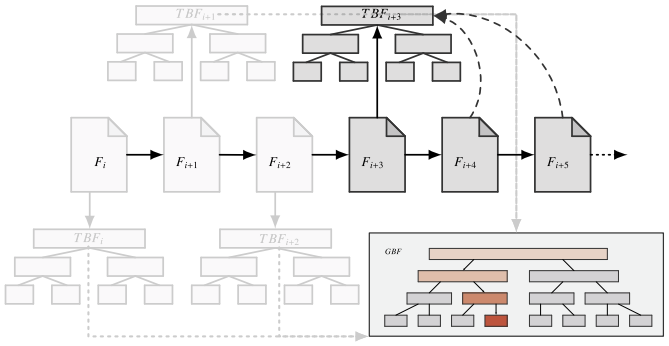

Fig. 1. Tree-filter based multihit prevention with temporary hbfts per file. Temporary Bloom filter trees (TBF_i) are used to filter multihits for a current file F_i and all its subsequent files. A global hbft (*GBF*) represents all unique elements in the processed set.

counter. After processing all of the subsequent files, unique chunks of the current tree are saved into a global hbft. The next file generates a temporary *hbft* again. However, only chunks with a zero counter are considered during the pass. The final tree stores unique chunks in different leaf nodes. Thus, it could be guaranteed that the tree does not feature the same chunk in a different leaf node.

Obviously, this approach has a higher building time but offers additional features. A possible advantage of utilizing counters for each chunk could be the definition of a threshold for accepted multihits. Thus, by the definition of a threshold a tree is generated which stores the maximum amount of elements in each leaf node. This empowers to identify chunks related to multiple files. Considering an interleaved multihit between two unique hits, an investigator could infer the gap between both.

Global-filter based. A straightforward approach could be the utilization of a separate Bloom filter which represents all multihits for a target file set. Therefor, two Bloom filters are generated with an adequate size (i.e., a size which respects an upper bound of false positives).

As shown in Fig. 2, a single *global* Bloom filter stores all chunks of a file set. A second *global multihit* Bloom filter will store all multihits for a corresponding set. A temporary *local* Bloom filter is generated for a specific file and gets zeroed out before another file will be processed. The local filter empowers to distinguish multihits within a file itself. Recalling the informal definition of a multihit, a multihit within a file itself but with no matches in other files could still be used for unique identification and is desired to keep. The local filter emits possible multihits on a file base. Those are ignored and not further processed in the global filter. If a chunk is neither in the local filter, nor in the global filter, it will be inserted in the global filter. We consider such a chunk as a unique chunk until proven otherwise. If a chunk is already in the global filter, an identical chunk has been seen before. Such a chunk will be further considered as multihit and gets inserted into the multihit filter. This process gets repeated for each file. The result is a global filter which stores all occurred multihits. A set (including multihits) could be

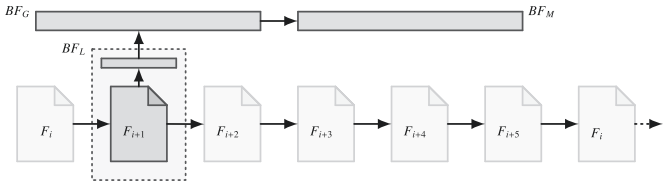

Fig. 2. Global-filter based prevention of multihits. Three different Bloom filter are used to filter multihits: A local Bloom filter (BF_L), a global multihit Bloom filter (BF_M), and a global Bloom filter (BF_G).

[5] https://github.com/ishnid/mrsh-hbft.

[6] https://github.com/skarupke/flat_hash_map, further extensions will be available via https://dasec.h-da.de/staff/lorenz-liebler/in April 2019.

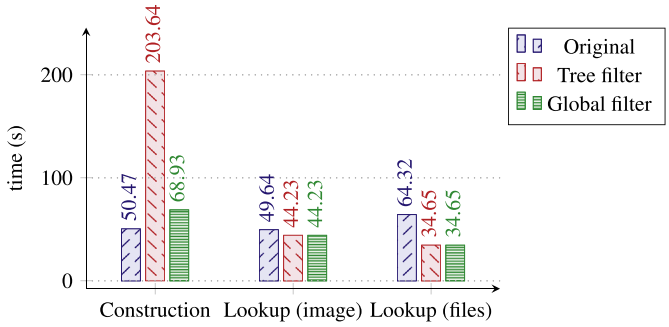

Fig. 3. Benchmark of multihit prevention approaches for *hbft*.

stored in a global *hbft* with an additional check against the multihit filter. An additional step of deduplication could shift the prevention of multihits to the construction phase.

Evaluation and selection. We benchmarked[7] the two introduced approaches in the case of construction and lookup runtime. For our benchmark we make use of the t5-corpus[8] and each node in the tree will represent one file of the corpus. The corpus consists of 4457 files with a total size of approximately 1.9 GiB. The interface was extended to additionally handle a single large image as input. We constructed an image by concatenating all 4457 files of the set. The amount of extracted chunks by the utilization of the *rolling_hash* with a block size $b = 160$ produces 8,311,785 chunks. In total 457,793 of the chunks are multihits (i.e., shared in more than one file).

Fig. 3 outlines the results of the benchmark. The construction phase does not differ for the *file* or *image* lookup. The prevention is handled during construction phase, as we focus on an improved lookup performance. The deduplication handling causes longer construction times in both cases. The tree filter approach clearly outreaches the global filter approach in terms of construction time. Depicting an appropriate candidate (i.e., Tree-filter or Global-filter) is a trade off between performance and matching capabilities. By the utilization of a global filter, we loose the ability to match multihits to their root files.

Lookup timings are identical for both techniques as both approaches filter multihits before the construction of the tree. Once the search for a file hits a leaf node it can be assumed that the query will not match any other leaf nodes. This holds true considering the false positives caused by the probabilistic Bloom filters. Even if it depends to the overall use case how often a filter has be rebuilt from scratch, the time needed to construct a Tree-filter becomes unbearable for larger data sets. Therefore, this enhancement will not be pursued any further and we propose the usage of a Global-filter.

Multihit prevention fhmap

Originally and naturally hash tables do not feature our considered handling of multihits by design. Each of the inserted keys should be unique. Implementations behave differently upon multihit insertions. Some databases will simply overwrite existing values, while others won't insert the value at all as soon as a key is already occupied. However, there are hash tables which support duplicated keys. This section presents an algorithm which prevents multihit insertions in any hash table.

Each inserted chunk is represented by a hash value, i.e., the actual *key*. The value of the corresponding key is a reference to the filename the chunk originates from. We assume that the corresponding chunks are multihits if two hash values are identical. In order to rule them out in the final database, each chunk will be looked up first. If a key is not in the table it can be safely inserted. If a key is already present in the table it is very likely a multihit. As already explained, multihits which occur in a file itself, but not in any other, should be kept. If the found value of the key (i.e., the filename) is equal to the value of the key which needs to be inserted, it is a multihit in a file itself. The insertion is ignored since the chunk is already represented in the table. If the found value is not equal to the query value, the chunk is a multihit within the file set. The entry will be marked as a duplicate in the table. This procedure is done for all chunks in the file set and comparable.

In the second processing step each entry is checked for the duplicate mark. If a mark is found the entry will be deleted from the hash table. This algorithm also reduces the amount of inserted chunks while keeping unique insertions only. Fewer insertions also mean less collisions and the increase of lookup speed. We will inspect the runtime, also in terms of deduplication, in the following section. Keeping the multihits is possible and would be comparable to the concept of hashdb which keeps internal counters to inserted chunks. However, this would force the database to handle multihits during the lookup phase.

Chunk extraction hbft and fhmp

Chunk extraction means the following. A given input sequence of bytes is divided into chunks by the definition of a fixed modulus b (common values are $64 \leq b \leq 320$ bytes). The extraction algorithm iterates over the input stream in a sliding window fashion, rolls through the sequence byte-by-byte, and processes 7 consecutive bytes at a time. A current window is hashed with a *rolling_hash function*, which returns a value between 0 and b. If this value hits the value $b - 1$, a trigger point is found and thus, defines the boundary of a current chunk.

We consider an example of block building and querying an image of 2 GiB. Reading in the image, constructing the block hashes via *rolling_hash*, and hashing the blocks via FNV-256 takes approximately 43 seconds. Querying each chunk against the *hbft* takes approximately 8.5 seconds (with each chunk present in the *hbft*). Thus, the extraction without lookup takes about 83.4% of the overall query time for a single process.

Beside the evaluation of lookup strategies, we also discuss the possible parallelization of the extraction process itself. A possible parallelized version of the *rolling_hash* is depicted in Fig. 4. We evenly split the input S into parts P_i which start at byte s_i (offset) and end at e_i (offset). The splitted blocks are further processed by a

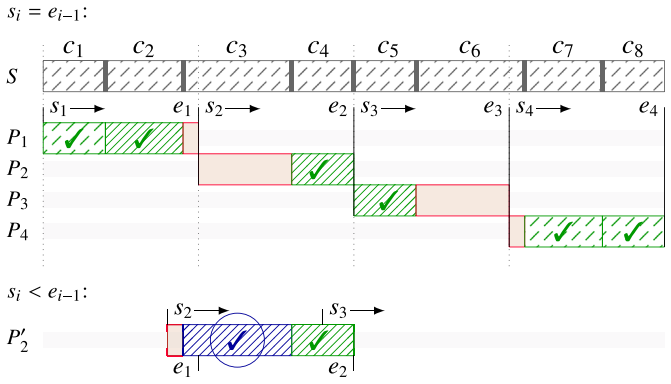

Fig. 4. Introducing wrong blocks at the borders of image blocks.

[7] performed on a laptop with Intel I7 2.2 GHz, 8 GiB of RAM and a SSD.
[8] http://roussev.net/t5/t5.html (last accessed 2018-10-23).

rolling_hash function which subsequently defines the chunk boundaries within each block. Thus, each P_i contains 0 or more chunks C_i. The challenge arises at each chunk boundary, i.e., the last chunk in P_i and the first in P_{i+1} (marked red). The chunk boundaries are implicitly defined by the block boundaries and do not match with the original chunk boundaries of S. The naive alignment of blocks, which would set the start address of a block to the end address of a subsequent block (i.e., $s_{i+1} = e_i$), would process chunks at the borders incorrectly (e.g., compared to non-parellized processing of S). To produce consistent chunk boundaries in the parallelized and non-parallized version, we could move the starting point into the range of a preceding block. This would create an overlap which gives the rolling_hash a chance to re-synchronize. We could also run over the end e_i until we identify a match with the leading chunks of its successor (i.e., block processed by P_{i+1}).

Evaluation rolling_hash. Our prototype implementation uses the producer consumer paradigm. The image is read as a stream of bytes by the main thread. Depending on the amount of cores the first image_size/cpu_num bytes are read. Those bytes are passed to a thread which performs the rolling hash algorithm and hashes identified blocks. In parallel, the next (image_size/cpu_num)+ overlap bytes are read and processed by another thread. In Table 2 times are shown needed to process a 2 GiB image into block hashes with and without threading. In the case of multithreading, the time needed for resynchronization is already included. Obviously the needed CPU time will increase since it is spread over multiple threads. The actual elapsed time is remarkably lower. A speedup of approximately 30 seconds can be achieved. This increases the processing speed by a factor of 3.2 by keeping consistent block boundaries. The query could be parallelized as well, but was not implemented in the course of this work.

Theoretical extensions

This subsection will continue with an analysis of possible extensions which have not been integrated.

New file insertion in *hbft*. Adding new elements to a hbft is considerable easy, as long as the tree does not reach a critical load factor. A naive approach is the initial design of an oversized tree structure to create additional empty leaf nodes. A parameterization always has to consider the impact on the overall false positive rate. The new files can then be splitted into blocks and hashed into the tree. The already introduced *Global-filter* can be used to filter multihits with low dependencies. Therefore, only the original *global* and *multi* Bloom filters have to be updated and saved after a finished session. With a further growing amount of additional files being inserted, the empty leaf node pool will starve and the false positive rate for the structure will become unacceptable. At this point the database needs to be resized and rebuilt. Original block hashes can be saved to disk to shorten the build time. However, in many cases, this suggestion is infeasible. Adding additional files to the tree requires careful and storage-intensive pre-planning. Most of the times the database is optimized for a given file set. Introducing empty leaf nodes possibly adds additional levels to the tree. This pessimistic growth would slow down the lookup phase and adds memory overhead which indeed never could be required.

File deletion in hbft. Hashes cannot be deleted from Bloom filters (except Counting Bloom filters). This in turn leads to the major drawback that hbft structures need to be partially rebuilt without the deleted file. In order to delete a specific file from the data structure, all related nodes from a leaf node up to its final root node are affected. Such affected filters need to be deleted and re-populated again. Also file chunks which are represented by the affected nodes need to be re-inserted again. Concerning the depicted tree structure in Fig. 5, lastly, every file hash is needed since the root filter holds the block hashes for every file in the set. Splitting the root node into several filters would reduce the amount of recreated hashes from scratch. During a lookup, this would require to horizontally process a sequence of root-filters first. Fig. 5 describes the problem of deleting files in a hbft structure. Deleting a file from the tree comes with considerable effort and computational overhead.

Table 2
Times of building/hashing blocks of an 2 GiB image in seconds (s).

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

Evaluation

The following performance tests focus on a runtime comparison between *hashdb*, *hbft* and *fhmap*. Each phase ranging from creating a database to the actual lookup is measured individually. Beside the overall *Memory Consumption* (4.2), we consider three major phases: *Build Phase* (4.3), *Deduplication Phase* (4.4) and *Lookup Phase* (4.5).

The assessment of required resources and performance limitations of the candidates should respect the proposed environmental conditions. In particular, the considered techniques are scaled for specific environments where *hashdb* explicitly targets large scale systems with multiprocessing capabilities. Even if the presence of an adequate infrastructure is a considerable assumption, we aim for similar evaluation conditions and therefore will limit resources (e.g. the number of processing cores). Again, it should be clear that *hashdb* as a single-level store clearly stands out compared to our memory-only candidates *hbft* and *fhmap*. However, we strive for a comprehensive comparison in our introduced use case by including a fully equipped database with desirable features.

Testsystem and testdata

Testsystem. All of the tests were performed on a laptop with Ubuntu 16.4 LTS using an underlying ext4 filesystem. The machine features an Intel I7 Processor with 2.2 GHz, 8 GB RAM, a built-in HDD, and a built-in SSD. After each evaluation run the memory and caches have been cleared in order to avoid run time or storage benefits in a subsequent evaluation pass (i.e., we make use of a 'cold' machine). Each test was repeated three times and the results have been averaged. Building and lookup phase are influenced by the underlying storage drive. A benchmark of both drives reported a read data rate of 128 MiB per second for the HDD and 266 MiB per second for the SSD. Thus, reading a 2 GiB file into memory takes approximately 16 seconds from the HDD and 8 seconds from the SSD. We further used the SSD throughout the following tests. Both drives have been benchmarked using the linux tool *hdparm*.

Testdata. Tests are performed on random data and synthetic images. The considered file set consists of 4096 files. Each file has a size of 524,288 bytes totaling in an image of 2 GiB. The image was created by concatenating all files together. We depict a global blocksize b of 512 bytes for all candidates and all tests. This should lead to a comparable compression rate and equal treatment in

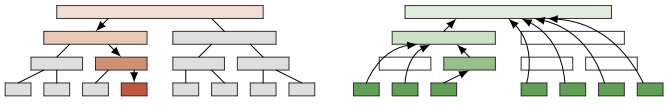

Fig. 5. Deletion of elements in a *hbft* could be performed in two steps. First: delete all Bloom filters which were influenced by the deleted file in a top-down or bottom-up approach. Second: Reinsert block hashes affected by the deletion in a bottom-up approach.

Table 3
Harddisk and Memory consumption of hashdb, fhmap and hbft for processing 2 GiB of input data.

Technique	DISK	RAM
hashdb	405.9 MiB	900.0 MiB
fhmap	100.0 MiB	200.0 MiB
hbft	168.0 MiB	168.0 MiB

different phases of processing. Since each file is a multiple of b in length, a fixed size extraction of blocks will not have to cope with any alignment issues.

Further extensions. Fixed block size hashing was additionally implemented for *hbft* and *fhmap* to allow an uniform comparison. However, the originally proposed chunk hashing function FNV-256 remained unchanged. FNV-1 is used for *fhmap* since the length of FNV-256 is not necessary. In case of the parallelized rolling hash all cores can be kept busy as long as the process of reading is faster than the actual processing of extracted chunks. If the wait time for I/O is slower than the processing time, only one thread will process the image blocks. Remaining cores will not be utilized and the block building is bottlenecked by a slow I/O bus. *hashdb* can operate multi-threaded in all reading steps. As introduced in the course of this work, *hfbt* and *fhmap* feature multi-threading in its block building phase only. To allow a comparison, timings for single threaded usage of *hashdb* is given as well, where *hashdb* implements a check of available system cores. This function was temporarily altered to always return a value of one. However, the implementation uses a producer-consumer approach and will spawn an additional thread anyways. By the usage of *taskset* we finally forced the execution on a single core.

Memory consumption

The three approaches feature different memory requirements. After ingesting the 2 GiB test set hashdb produces files totaling in 405.9 MB on disk. Since there is no theoretical background to calculate the data structures size in main memory, it is approximated using the top command. The in memory size of the structure is about 900 MB.

In the case of fhmaps, the author[9] mentions some storage overhead for the handling of key-value-pairs. The overhead will be at a minimum of 8 bits per entry and will be padded to match with the actual key length. Assuming a key length of 64 bit, then the overhead for fhmap would be 64 bit as well. Assuming the test set of 2 GiB with a block size $b = 512$, then the total amount of blocks n will be $4,194,304$. The total size s in main memory with a *load factor* of 0.5 would then result in $s = \frac{n \cdot 64\ bits \cdot 3}{0.5} \approx 200\ MiB$. The allocation size on disk would be halved to approximately 100 MiB caused by the *load factor*.

The size s of a *hbft* depends on various parameters and is mainly influenced by the data set size, the block size b, and a desired false positive rate *fpr*. In our scenario we approximate the root filter size for the parameters = 2 GiB, $b = 512$ and $fpr = 10^{-6}$. This would lead to an approximated root filter size of $m_1 = \mu * 2^{15.84} \approx 14$MiB. The tree consists of $\log_2(4096) = 12$ levels. Thus, the total amount of needed memory is approximately $12 \cdot 14$MiB $= 168$MiB. The size on disk will be approximately the same since the array and corresponding Bloom filters need to be saved. Fur further details of the hbft parametrization and calculation, we refer to Lillis et al. (Lillis et al., Scanlon).

An overview of the required storage for each technique can be seen in Table 3. In conclusion, *hbft* is the most memory efficient approach, followed by *fhmap* and *hashdb*. Nevertheless, it should be noted that only hashdb is able to work with databases which do not completely fit into RAM. In contrary *hbft* and *fhmap* will only work if the databases fit into RAM.

Build phase

The creation of a database consists of several steps including the initialization of the different structures for data storage and handling. The extraction of blocks by splitting an input stream is considered for fixed blocks (similar to hash-based carving) or varying blocks (similar to approximate matching). In detail, we integrated a fixed block extraction (*fi*) and the extraction per *rolling_hash* (*ro*) for hbft and fhmap. Afterwards, in all cases the blocks are hashed. In case of *hashdb* MD5 is used while *hbft* and *fhmap* use FNV-2561 and FNV-1 respectively.

The overall runtime is presented in Fig. 6a for a single-threaded execution. Results for both block building approaches are displayed and evaluated in the case of *hbft* and *fhmap*. The high discrepancy of runtime in case of *hashdb* is also caused by the setup of its metadata and relational features. The high difference from CPU to real time is related to read and write operations. Fig. 6b shows the results for a multi-threaded execution (8 threads). The timings for the *hbft* and *fhmap* block building algorithms keeps constant since multi-threading is not implemented in the building phase yet. With the utilization of eight threads, *hashdb* cuts down its processing time tremendously. However, the approach is still slower than both block building algorithms of *hbft* and *fhmap*.

Deduplication phase

Recalling the utilized set of random data, randomly generated data does not feature any multi hits and thus, nearly all of the extracted chunks result in unique hits. Our considered version of *hashdb* allows deduplication of an existing database per configuration. The associated counter value for all hashed blocks are checked and all blocks with a value higher than one are deleted. The remaining chunks are written to a new database which finally ensures unique matches during lookup. The size of the newly established database stays the same. The in Section 'Extensions to hbft and fhmap' introduced and implemented multihit mechanisms of *hbft* and *fhmap* are executed in memory only. The runtime results of the deduplication phase are displayed in Fig. 7a and b.

Timings do not differ remarkably for single- or multi-threaded scenarios. The deduplication procedure for hashdb is slightly slower since it needs to read the database from disk first. As the *rolling hash* produces less blocks than a fixed-size extraction, the

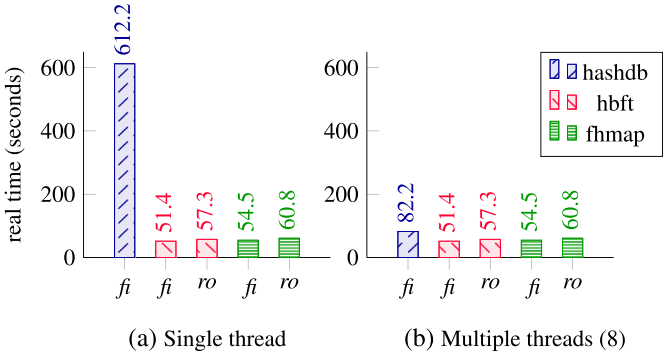

Fig. 6. Build time performance of *hashdb*/*hbft*/*fhmap*. In case of *hbft* and *fhmap* we consider fixed blocks (*fi*) and the extraction per rolling_hash (*ro*).

[9] https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/.

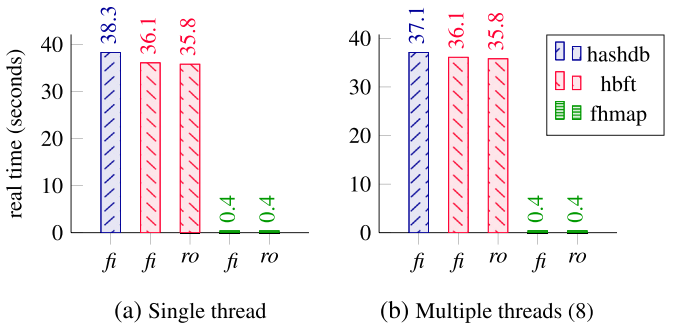

Fig. 7. Deduplication performance of *hashdb*|*hbft*|*fhmap* with zero occurring multihits in a set.

overall amount of inserted chunks decreases and thus, the runtime of deduplication improves. The fast deduplication time of *fhmap* is caused by three facts: First, there are no special structures which have to be additionally set up or evaluated. Second, the deduplication happens in the building phase as well, so there is no clear separation in building and deduplication for hash tables. Last, the random set does not feature any multi hits. *hashdb* and *hbft* need to process their temporary databases and filter out multi hits before inserting unique chunks in the actual database. In the case of *fhmap*, previously marked multihit-entries are simply deleted from the database.

Lookup phase

The lookup consists of splitting an image in blocks, hashing those blocks, and query them against the database. As we are only interested in efficiency (but not in detection performance), we make use of a simple approach to simulate full and partial detection scenarios. We create four different images which are queried against the databases. All images have a fixed size of 2 GiB. Each of the four images is constructed to match either 100%, 75%, 50%, or 25% of the database. Again we point out that the different file matching sizes are only used to investigate the efficiency behaviour dependent on different matching rates, i.e., the matching rate is the input parameter. Images with matching rates below 100% are partially filled with random bytes to reach the desired size of 2 GiB. The size of every inserted file is a multiple of $b = 512$ bytes. Thus, images are crafted which do not cause alignment issues for a fixed block extraction. It has to be considered that the rolling hash produces less blocks which additionally vary in size.

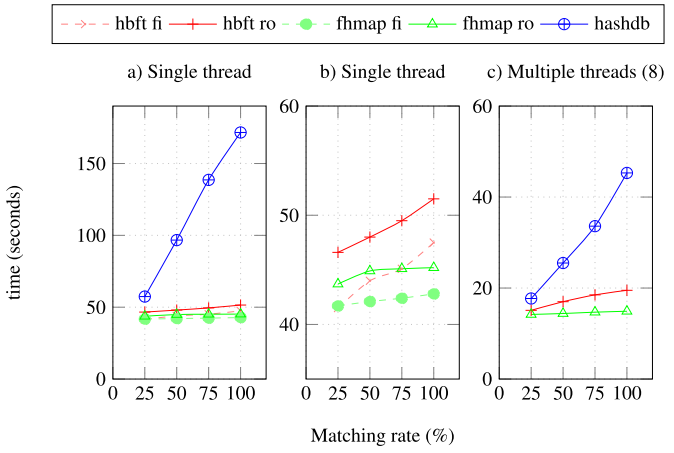

Fig. 8. Lookup performance evaluation (real time).

Results of the benchmark are shown in Fig. 8. In Fig. 8a and b the lookup performance in the case of single-threaded evaluations are shown. Fig. 8c displays the parallelized version with a total amount of eight running threads.

As shown fhmap features the fastest lookup followed by hbft and lastly hashdb. The results underline the performance of fhmap in all cases and the impact of fixed blocks, in contrary to the overhead caused by computing a rolling hash. A significant speeding up is gained by our proposed parallelization of the rolling hash. The plot shows stable lookup results for all matching rates with fhmap outperforming its competitors. Lookup times for hbft and fhmap increase slightly by a rising matching rate. The lookups of hashdb are higher due to its complex internal structure.

Pre-filters for hbft and hashdb speed up the lookup time for non-matches notably. In case of hbft the root Bloom filter will rule out non matches instantly. Otherwise, a query needs to inspect subsequent nodes, shown by the slightly increased lookup times for higher matching rates. If a chunk does not match *hashdb's* compressed *Hash Store* the actual database is not queried either. A *Hash Store* claims to have a false positive rate of 1 in 72 million with a database containing 1 billion hashes. However, every hash will be queried against this store first before searching the actual database. The presented flat hash map does not feature any pre-filtering so far. Each key will be queried against the database. Performed tests with different sized databases did not differ remarkably.

Conclusion

In this work we discussed and evaluated three different implementations of artifact lookup strategies in the course of digital forensics. Several extensions have been proposed to finally perform a comprehensive performance evaluation of hashdb, hbft, and fhmap. We introduced concepts to handle multihits for hbft and fhmap by the implementation of deduplication and filtration features. Moreover, we interfaced fhmap with a rolling hash based extraction of chunks. For a better comparison to hashdb, we additionally parallelized the extraction of chunks.

Results show that fhmap outperforms hbft in most of the considered performance evaluations. While *hbfts* are faster than *hashdb* in nearly all evaluations, the concept introduces false positives by the utilized Bloom filters. Even if hbfts have small advantages in case of memory and storage efficiency, their complexity, fixed parametrization, and limited scope of features make such an advantage negligible. However, specific use cases with tight memory constraints could make hbfts still valuable.

Discussions of hashdb in terms of performance should consider the underlying concept of single-level stores. Shifting the discussion to offered features and a long term usage with an ongoing maintenance, *hashdb* and *fhmap* are more suitable. One thing to note is that *hashdb* is the only implementation that is able to deal with databases which do not fit into main memory. In addition it supports transactional features.

In Table 4 a final comparison of all three candidates in terms of performance and supported features is given. The final overview underlines the trade-offs between the concepts, where fhmap shows a constant performance in most of the mentioned categories.

Future work

A concept similar to single-level stores for digital artifacts with stable results in all of the mentioned categories is desirable. Where most of the considered challenges rely on an high amount of engineering effort first, the direct integration of a multihit prevention into a single-level store could be an interesting field of research.

Table 4
Final comparison of hashdb/hbft/fhmap in case of performance and offered features.

	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	++	++

Concepts to close the gap between performance-oriented memory-resistant lookup strategies and transactional database are needed.

Acknowledgment

This work was supported by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry of Higher Education, Research and the Arts within CRISP (www.crisp-da.de).

References

Breitinger, F., Baier, H., 2012. Similarity preserving hashing: eligible properties and a new algorithm mrsh-v2. In: International Conference on Digital Forensics and Cyber Crime. Springer, pp. 167–182.

Breitinger, F., Baier, H., White, D., 2014. On the database lookup problem of approximate matching. Digit. Invest. 11, S1–S9. Supplement 1.0 (2014). Proceedings of the First Annual DFRWS Europe, ISSN: 1742-2876.

Breitinger, F., Rathgeb, C., Baier, H., 2014. An efficient similarity digests database lookup-a logarithmic divide & conquer approach. The Journal of Digital Forensics, Security and Law: JDFSL 9, 155.

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

Chu, H., 2011. Mdb: a memory-mapped database and backend for openldap. In: Proceedings of the 3rd International Conference on LDAP, Heidelberg, Germany, p. 35.

Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.D., 2014. Cuckoo filter: practically better than bloom. In: Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies. ACM, pp. 75–88.

Foster, K., 2012. Using Distinct Sectors in Media Sampling and Full Media Analysis to Detect Presence of Documents from a Corpus, Technical Report. NAVAL POSTGRADUATE SCHOOL MONTEREY CA.

Garfinkel, S.L., McCarrin, M., 2015. Hash-based carving: searching media for complete files and file fragments with sector hashing and hashdb. Digit. Invest. 14, S95–S105.

Gupta, V., Breitinger, F., 2015. How cuckoo filter can improve existing approximate matching techniques. In: International Conference on Digital Forensics and Cyber Crime. Springer, pp. 39–52.

Harichandran, V.S., Breitinger, F., Baggili, I., 2016. Bytewise approximate matching: the good, the bad, and the unknown. Journal of Digital Forensics, Security and Law 11, 4.

Liebler, L., Breitinger, F., 2018. mrsh-mem: approximate matching on raw memory dumps. In: International Conference on IT Security Incident Management and IT Forensics. IEEE, pp. 47–64.

Lillis, D., Breitinger, F., Scanlon, M., 2017. Expediting mrsh-v2 approximate matching with hierarchical bloom filter trees. In: International Conference on Digital Forensics and Cyber Crime. Springer, pp. 144–157.

Pagani, F., Dell'Amico, M., Balzarotti, D., 2018. Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis. In: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy. ACM, pp. 354–365.

Young, J., Foster, K., Garfinkel, S., Fairbanks, K., 2012. Distinct sector hashes for target file detection. Computer 45, 28–35.