md5bloom: Forensic Filesystem Hashing Revisited

By

Vassil Roussev, Timothy Bourg, Yixin Chen, Golden Richard

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2006 USA

Lafayette, IN (Aug 14th - 16th)

available at www.sciencedirect.com

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

Digital Investigation

md5bloom: Forensic filesystem hashing revisited

Vassil Roussev*, Yixin Chen, Timothy Bourg, Golden G. Richard III

Department of Computer Science, University of New Orleans, New Orleans, LA 70148, USA

Keywords:
Digital forensics
Bloom filters
Similarity hashing
Filesystem investigation
md5bloom

ABSTRACT

Hashing is a fundamental tool in digital forensic analysis used both to ensure data integrity and to efficiently identify known data objects. However, despite many years of practice, its basic use has advanced little. Our objective is to leverage advanced hashing techniques in order to improve the efficiency and scalability of digital forensic analysis.

Specifically, we explore the use of Bloom filters as a means to efficiently aggregate and search hashing information. In this paper, we present *md5bloom*—an actual Bloom filter manipulation tool that can be incorporated into forensic practice, along with example uses and experimental results. We also provide a basic theoretical foundation, which quantifies the error rates associated with the various Bloom filter uses along with a simulation-based verification. We provide a probabilistic framework that allows the interpretation of direct, bitwise comparison of Bloom filters to infer similarity and abnormality. Using the similarity interpretation, it is possible to efficiently identify versions of a known object, whereas the notion of abnormality could aid in identifying tampered hash sets.

1. Introduction

The main attraction of hash functions is that they can map an arbitrarily large sequence of bytes to a random number within a fixed range. Furthermore, a collision-resistant hash function will map sequences that differ even by one bit to completely different hash values. Thus, given enough bits, we can get a compact object representation that is unique for all practical purposes. Another interesting aspect of the hashes is that, although they uniquely represent the entire contents of a data object, it is not possible to recover the original data by knowing the hash. The downside is that it is not possible to examine the original object; however, the upside is that it is possible to collect some information about the object without incurring privacy concerns.

Traditionally, cryptographic hashes, such as MD5 and SHA-1, have been used in filesystem (forensic) analysis to efficiently answer yes–no questions, such as object equality. The most important use is to compute and record the hash of a forensic target during imaging in order to demonstrate the integrity of the working copy. This process may also include the storing of finer-grain hashes, typically at the block level. Another common use is to collect hashes of known files in order to filter them out during the examination process. NIST, for example, maintains the National Software Reference Library—an extensive list of approximately 50 million known file (MD5, SHA-1) hashes from common operating systems and application packages (http://www.nsrl.nist.gov). A similar approach is used in host-based intrusion-detection systems (HIDS) where a clean copy of the installation is hashed and later verified to flag any suspicious modifications. There is no reason to believe that the traditional uses will change any time soon with the exception of the planned transition to stronger hash functions. In that sense, the proposed uses of hashes should be viewed as *additional* tools facilitating the forensic inquiry and are not intended as replacements.

* *Corresponding author.*
E-mail addresses: vassil@cs.uno.edu (V. Roussev), yixin@cs.uno.edu (Y. Chen), tbourg@cs.uno.edu (T. Bourg), golden@cs.uno.edu (G.G. Richard).

1742-2876/$ – see front matter © 2006 DFRWS. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2006.06.012

1.1. New scenarios

While the developed tools could be used in a variety of case-specific scenarios, we use two generic scenarios—multiple target correlation and fine-grained change detection—to motivate the rest of our discussion.

1.1.1. Multiple target correlation

The goal is to pick from a set of forensic images the one(s) that are most like (or perhaps most *un*like) a particular target. This problem comes up in a number of different variations, such as comparing the target with previous/related cases, or determining the relationships among targets in a larger investigation. The goal is to get a high-level picture that will guide the following in-depth inquiry. It is fairly obvious that the already existing problems of scale in digital forensic tools are further multiplied by the number of targets, which explains the fact that in other forensic areas comparison with other cases is routine and massive, whereas in digital forensics it is the exception.

As a simple illustration, consider the following approach that can be implemented with existing tools—compute the number/fraction of common block-level MD5 hashes for every pair of targets and use that as a measure of similarity. While this scheme is relatively simple to implement, its resource requirements exceed the capacity of the typical high-end workstation. For example, the raw storage requirement for the block-level MD5 hashes of a 512 GB hard drive is 16 GB. For efficient access, the hashes themselves must be stored in RAM in a hash table which is about 50% full. Thus, the total amount of main memory needed is approximately 32 GB, whereas a workstation has only 2–4 GB. Relying on virtual memory is not an option due to the random access pattern of hash tables, which rules out efficient pre-fetching from secondary storage. Therefore, current hardware can reasonably handle the hashes of 32–64 GB HDD in the above scenario. In other words, we need to compress our representation at least eight times.

Another important question is the interpretation of the results—how significant it is if 50, 60, 70% of the content of two targets is the same?

1.1.2. Object versioning detection

Often, the problem is not to discover that a target object is different from a reference object but to identify it as a likely version of a known object. This would be particularly useful in dealing with composite file objects, such as executables and office documents.

The problem with executables is that installed software is no longer a static entity as operating systems and applications (of various sizes and quality) update themselves frequently over the network. Hence, a file-granularity reference hash list will age very quickly and will have a declining recognition rate. Therefore, it would be useful to have a tool that recognizes the file as variation of a known file. Depending on the case, this would either serve to focus attention (e.g. intrusion investigation), or ignore it (e.g. data recovery).

Regardless of the usage scenario, the results from the proposed tools are, essentially, *hints* to the investigator. Therefore, they must come with statistical measures of confidence to allow the quick identification of the most promising leads and the filtering out noise.

1.2. Bloom filters as hash bags

What we need to address in the presented problems is a way to store a *set* of hashes representing the different components of a composite object as opposed to a single hash. For example, hashing the individual routines of libraries or executables would enable fine-grained detection of changes (e.g. only a fraction of the code changes from version to version).

The problem is that storing more hashes (perhaps many more) presents a scalability problem even for targets of modest sizes. Therefore, we propose the use of Bloom filters as an efficient way to store and query large sets of hashes. As our following discussion will show, in addition to being able to reduce storage requirements by an order of magnitude, a Bloom filter can be used as a first-class object and be directly compared with other filters in a statistically meaningful way. The upshot is that we can build 'big-picture' tools that can capture relationships among entire hash sets without per-member queries.

2. Related work

This section briefly presents work in three related areas: file similarity tools, general results on Bloom filters, and applications of Bloom filters to digital forensics and security.

2.1. File similarity

Discovering similarity among files has been a topic of research for decades. For example, Manber (1994) developed the *sif* tool, which seeks to identify file similarity based on approximate fingerprinting (essentially, selective hashing). It is primarily suited for text files, however, and does not provide any statistical interpretation of the results. A more recent wave of research has focused on web objects to filter out similar search results and to use delta-encoding for object versions.

2.2. Bloom filters

Since their introduction in 1970 by Bloom, Bloom filters have enjoyed a lot of attention both from theoreticians and practitioners. Here, we briefly introduce the basic mathematical framework we need for our own analysis and do not attempt to exhaustively survey the literature. Our discussion and notation follow the framework presented in Fan et al. (2000) and Mitzenmacher (2002) and the reader is referred to these publications for a more comprehensive explanation.

A Bloom filter B is a representation of a set $S = \{s_1, \ldots, s_n\}$ of n elements from a universe (of possible values) U. The filter consists of an array of m bits, initially all set to 0. As our following discussion will show, the ratio $r = m/n$ is a key design element and is usually fixed for a particular application. To represent the set elements, the filter uses k independent hash functions $h_1, \ldots, h_k$, with a range $\{0, \ldots, m-1\}$. All hash functions are assumed to be independent and to map elements from U uniformly over the range of the function.

To insert an element s from S, the hash values $h_1(s), \ldots, h_k(s)$ are computed and the corresponding k bit locations are set to 1. The same process is repeated for every element in S. Note that setting a bit to 1 multiple times has the same effect as doing it once. To verify if an element x is in S, we compute $h_1(x)$, …, $h_k(x)$ and check whether *all* of those bits are set to 1. If the answer is no, then we *know* that x *is not* an element of S, i.e., there are no false negatives. Otherwise we *assume* that x is a member, although there is a distinct possibility that we are wrong and the bits we checked were set by chance. In other words, Bloom filters are subject to false positives; however, the false positive rate can be quantified and controlled, as shown below.

First, we calculate the probability that a specific bit is still 0, after all the elements of S are hashed. Since our functions are assumed perfectly and uniformly random, probability is given by:

$$p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

For a false positive to occur, each of the k locations verified must not contain 0. For all practical purposes, it is safe to assume that entries in B are independently set 0 with probability p and to 1 with probability $1-p$ (a more strict mathematical justification can be found in Mitzenmacher, 2002). Hence, the probability of a false positive P_{FP} is given by:

$$P_{FP} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k = (1-p)^k$$

It is clear that the false positive rate depends on three factors: the size of the filter m, the number of elements n, and the number of hash functions k, and that those can be traded off. Thus, if the goal is to minimize the false positive rate (the usual objective), then for fixed m and n, we can vary the number of hash functions. Note that this optimization is not straightforward: increasing the number of hash functions increases the chance of finding a 0 bit for a non-element; however, reducing the number of hash functions increases the fraction of 0 bits in the filter.

Optimizing the expression for k yields a false positive rate of $(0.6185)^{m/n}$ for $k = \ln 2(m/n)$. Obviously, for practical purposes, k must be an integer and must be relatively small. Returning to our motivating example from the introduction, to achieve the minimum compression ratio of eight times we need a ratio of $r = m/n$ of 16. Table 1 gives the false positive rates for $m = 1024$ and various combination of k and r.

2.3. *Applications of Bloom filters*

Bloom filters have been used in a number of applications, including hyphenation (Bloom's original application, described in Bloom, 1970), spell checking (Manber, 1994), database optimization, e.g. in speeding up semi-join applications (Valdurez and Gardarin, 1984), efficient web cache sharing (Fan et al., 2000), peer-to-peer networks (Ledlie et al., 2002), routing (Feng et al., 2001; Whitaker and Wetherall, 2002) and network traffic measurement (Estan and Varghese, 2001). The interested reader is referred to Broder and Mitzenmacher's (2005) survey paper on applications of Bloom filters for a good introduction. In the remainder of this section we concentrate on applications of Bloom filters in computer security.

Spafford (1992) was perhaps the first person to use Bloom filters to support computer security. The OPUS system (Spafford, 1992) uses a Bloom filter which efficiently encodes a wordlist containing poor password choices to help users choose strong passwords. Bellovin and Cheswick (in preparation) present a scheme for selectively sharing data while maintaining privacy. Through the use of encrypted Bloom filters, they allow parties to perform searches against each other's document sets without revealing the specific details of the queries. The system supports query restrictions to limit the set of allowed queries.

Aguilera et al. (2003) discuss the use of Bloom filters to enhance security in a network-attached disks (NADs) infrastructure. A NAD accepts block level, reads and writes operations over a network, eliminating the need for a file server to transfer disk blocks. The file server is therefore freed to process only filesystem metadata, to support, for example, file lookup and deletion operations. This greatly improves disk bandwidth, since the file server is no longer a bottleneck. To improve NAD security, Aguilera encrypts network traffic between the NADs and clients and uses capabilities to describe which blocks a client may access. To prevent replay attacks, every request is verified against a Bloom filter of recently performed disk accesses.

Dharmapurikar et al. (2004) use Bloom filters to match known signatures for network intrusion detection. Dharmapurikar et al., in press describe a scheme for longest prefix matching using Bloom filters. This work is to support content filtering. Shanmugasundaram et al. (2004) introduce hierarchical Bloom filters to perform payload attribution in network forensics. The goal is to be able to derive the set of hosts involved in transmission of a payload fragment, in order to track the propagation of malware such as worms or viruses. A hierarchical Bloom filter (HBF) is a collection of block-based Bloom filters (where the elements hashed for the Bloom filter contain not only data, but a concatenated offset indicating where the data fits in a larger data stream) with geometrically increasing block sizes. Hierarchical Bloom filters allow determination of

Table 1 – Example of false positive 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

whether two payload strings which "match" the Bloom filter occur in adjacent locations in a data stream (e.g. they appeared within the same packet) or whether they appeared in different packets, while traditional Bloom filters would only be able to determine where the payloads contained the strings. In addition to the storage advantages of using Bloom filters rather than storing raw network data, their system enhanced privacy, since the original network data cannot be recovered from the Bloom filters. Experimental results show that their system is adept at tracking the propagation of some real examples of malware.

3. Bloom filter comparison

So far we have only discussed the possibility of issuing membership queries against a Bloom filter. Relative to the customary use of hashing, the only advantage is that we can realize an order of magnitude space savings at price of a false positive rate of well under 1%. In this section, we present a probabilistic argument for interpreting the direct bitwise comparison of two Bloom filters and provide illustrative examples.

3.1. *Probabilistic justification*

Specifically, let n_1 and n_2 be the size of sets S_1 and S_2, respectively. S_1 is represented by a Bloom filter B_1 with m bits using hashing functions $h_1, \ldots, h_k$. Similarly, S_2 is represented by Bloom filter B_2 with m bits using the same set of hashing functions. Note that for the comparison of two filters to be meaningful, the size of both filters and the set of hash functions used must be the same; however, the number of elements in each filter could be different.

Under the above assumptions, the only direct observation we can perform is to count the number of positions at which the corresponding bit is set to 1 in both filters. We refer to such bits as *matching* bits. Below, we derive the probability that the two filters, B_1 and B_2, have exactly C matching bits.

Let p_1 (p_2) denote the probability that a specific bit is 1 in B_1 (B_2) after all the elements of S_1 (S_2) are hashed into the Bloom filter. Using the reasoning from the previous section, we get:

$$p_1 = 1 - \left(1 - \frac{1}{m}\right)^{kn_1} \approx 1 - e^{-kn_1/m}$$

$$p_2 = 1 - \left(1 - \frac{1}{m}\right)^{kn_2} \approx 1 - e^{-kn_2/m}$$

Thus, the probability that two filters have *specific* C bits in common is $p_1^C p_2^C \hat{p}(C)$, where $\hat{p}(C)$ denotes the probability that there are no matching bits among the rest of the $m - C$ bits. After accounting for all the possible ways in which the bits can be chosen, we get:

$$\hat{p}(C) = \sum_{i=0}^{m-C} \left\{ \binom{i}{m-C} (1-p_1)^i (1-p_2)^i \times \left[p_1(1-p_2) + p_2(1-p_1)\right]^{m-C-i} \right\}$$

Therefore, the probability that two filters have *any* C matching bits is:

$$p(C) = \binom{C}{m} (p_1 p_2)^C \hat{p}(C)$$

In other words, given the number of matching bits in two filters we can calculate the probability that this happened by chance. The threshold value *below* which the result is considered statistically significant is determined by the user, with typical values (from hypothesis testing) of 0.01 or 0.05.

On the practical side, the expression for $\hat{p}(C)$ is obviously unwieldy and computationally unattractive; however, it is fairly straightforward to obtain an approximation based on the normal distribution.

3.2. *Interpretation*

Figs. 1 and 2 show families of probability density functions for a set of realistic parameters for the m/n ratio (16, 8) and the number of hash functions (2–6) (only probabilities greater than 1×10^{-4} are shown).

Evidently, the bell-shaped curves are indicative of a normal distribution, which should not come as a surprise given the basic terms for p_1 and p_2. More importantly, this offers the opportunity to use available methods for efficiently computing $p(C)$, in the future.

The presented data provide an interesting insight into choosing the appropriate number of hash functions. The rationale in previous work has been that fewer functions mean less computation, which makes perfect sense, in theory. In practice, the different hash values are often obtained as appropriately-sized bit subsets of the values produced by a cryptographic function. For example, if we need 16-bit values for the hashes, we can obtain up to eight of these by breaking up a 128-bit MD5, which means that for up to eight hashes the computational cost is constant. The graphs seem to suggest that using a higher number of hashes will make it somewhat more difficult to make confident conclusions about direct comparisons of Bloom filters.

Using the presented examples as a starting point, let us briefly demonstrate how the result could be used in data forensics, in particular, in determining target similarity. For that purpose, it is useful to have some of the actual data behind the graphs to widen/narrow the scope of an investigation. Assuming that goal is to determine the images closest

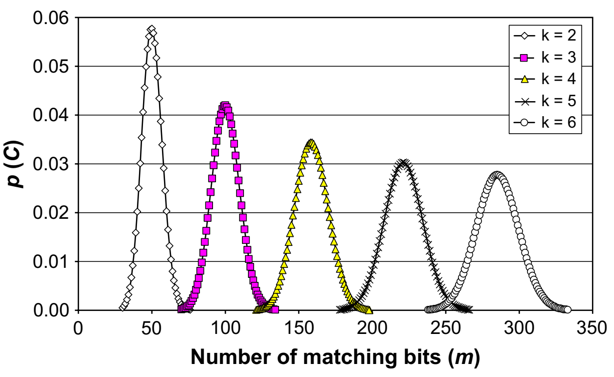

Fig. 1 – PDF for $n_1 = n_2 = 128$, $m = 1024$.

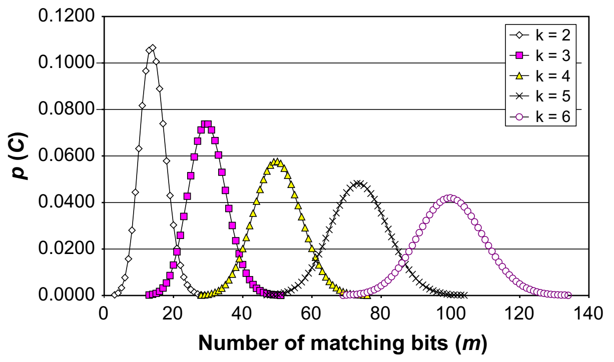

Fig. 2 – PDF for $n_1 = n_2 = 64$, $m = 1024$.

in binary content to a target, we could prioritize them by starting with the one most likely to be similar and gradually move away, i.e., we start with $p < 0.01$, then $p < 0.05$, 0.1, etc.

From Tables 2 and 3 we can conclude that a bit match that has fewer than the $C_{\min}$ or more than the $C_{\max}$ has the corresponding probability p of being random. As the [$C_{\min}$, $C_{\max}$] range expands away from the mean, the statistical significance of the result increases. Note that this ordering could still be useful even if it does not carry statistical significance for all values of p.

Another interesting observation is that it is possible for two filters to have *too few* matching bits. In other words, if the actual number of matching bits is to the left of the bell in the distribution, it could be interpreted as an indication that there is a *deliberate* effort to reduce the number of matching bits. For cryptographic hash functions, it means that it is likely that somebody has modified a hash set *after* it has been generated, since any change to the source data would lead to uniform random perturbations in the hash results that would be undetectable. This result is significant because, for the first time, it provides a statistical means to detect tampering of hash sets. Section 5 presents some simulation results that lend support to this conjecture.

4. Implementation: *md5bloom*

In this section we present a prototype stream-oriented Bloom filter implementation called *md5bloom*. The ultimate intent is to produce a well-tested open-source tool that can be incorporated into the arsenal of the digital forensic investigator either directly, or as part of an integrated environment.

Table 2 – Intervals for Fig. 2 ($n_1 = n_2 = 128$, $m = 1024$)

p	$k = 2$		$k = 4$		$k = 6$	
	$C_{\min}$	$C_{\max}$	$C_{\min}$	$C_{\max}$	$C_{\min}$	$C_{\max}$
0.25	10	18	43	58	89	110
0.10	8	20	39	61	85	116
0.05	7	21	37	63	82	118
0.01	6	25	32	67	76	125

Table 3 – Intervals for Fig. 2 ($n_1 = n_2 = 64$, $m = 1024$)

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

4.1. *Overview*

There are three different functions associated with a Bloom filter: creation, query evaluation, and filter comparison. In addition, those could be used on static data sets, dynamic sets, or some combination of these. To increase compatibility with existing hashing tools, we provide basic conversion utilities that allow direct reuse of existing hash sets. To avoid creating a multitude of command-line utilities, we have wrapped all the functions into a single utility (with a multitude of options).

Conceptually, the filter management is split into front- and back-end components. The back-end module deals with the basic filter operations—creation, loading, storing, inserts, and queries—and is independent of the actual hash functions used. Due to the fact that it handles membership queries we refer to it as a *bloom server* (or daemon). The front-end (the *bloom client*) is the one responsible for feeding the server with the desired hash sets and for performing the queries. Depending on the usage scenario, the client and the server could be running in the same process, or in separate processes communicating over the network. In a large-scale scenario, a single client may query multiple servers and a single server can process simultaneous requests from multiple clients.

Central to the operation of the filter is the generation of hash function values. Our scheme piggybacks on the ubiquitous MD5 function to produce the necessary hashes, although it would be straightforward to use any cryptographic function, such as the ones in the SHA family. Despite the recent upheaval about the possibility of MD5 being vulnerable to attack, MD5 is very suitable for our purposes for a number of reasons: we do not use it to validate data integrity of individual objects but to compare groups of objects; the results are always treated as a hint; MD5 has much lower computational costs than stronger hash functions.

Recall that, given an arbitrary sequence of bytes, the MD5 function returns 128 bits: $h^{\mathrm{MD5}} = \mathrm{MD5(object)}$. Since MD5 is assumed to be a collision-resistant function, any individual bit of the hash value can be viewed as an independent random variable and, by extension, any subset of the 128 hash bits can be selected to produce a value within a desired range. Let the bits in h^{MD5} be numbered 0.127. We use the notation $h_{d_1:d_2}$ and the term *subhash* to denote the selection of a continuous of bits numbered d_1 through d_2, inclusively. Thus, $h^{\mathrm{MD5}} = h_{0:127}$ and can also be expressed as the concatenation of subhashes, e.g.:

$$h^{\mathrm{MD5}} = h_{0:15}h_{16:31}h_{32:47}h_{48:63}h_{64:79}h_{80:95}h_{96:111}h_{112:127}$$

For convenience, we assume that the size of the filter m is a power of two: $m = 2^l$. We need a selection of $k \times l$ bits from the original hash and one straightforward way to achieve is as by follows:

$\{h_{0:l-1}, h_{l:2l-1}, \ldots, h_{(k-1)l:kl-1}\}$

For practical purposes, however, it is more convenient to first breakup the hash into byte-aligned pieces and then mask out the unneeded bits. We use 1-byte alignment if $l \leq 8$, 2-byte alignment if $8 < l \leq 16$, and 4-byte alignments if $17 < l \leq 32$ (2^{32} is currently the maximum supported filter size). After the breakup, the extra subhash bits are masked out, starting with the most significant one first.

4.2. *Filter creation*

As already mentioned, the creation of the filter is handled by the server module. There are three alternative options for creating the filter—from a byte stream, from existing MD5 hashes, and from client hashes. Below, we provide a brief description of the options of our tool. For brevity, we omit the name of our tool—md5bloom—from the command lines and present only the options:

-genstream ⟨l⟩ ⟨k⟩ ⟨r⟩ ⟨block_size⟩ [-daemon port]

The *–genstream* version reads from the standard input blocks of the given size, hashes them, and inserts them into a filter with the given parameters. When the filter reaches its capacity (as determined by the *m/n* ratio) a new one is started. The default output behavior is to send the binary content of the full filter to the standard output, preceded by a small header describing its basic parameters. If the *–daemon* option is specified, the filter is kept in memory and the module acts as a server daemon accepting request on the given port. Example:

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

The process will create the filter $m = 2^{10}$, $k = 4$, $m/n \leq 8$, and will hash *stdin* every 512 bytes. Resulting filters will stay in memory and will be available for queries on port 2024.

-genmd5 ⟨l⟩ ⟨k⟩ ⟨r⟩ [-daemon port]

The *–genmd5* version reads from the standard input a stream of hexadecimal MD5 text strings (one per line) and inserts them into a filter with the specified parameters. As before, the output/server behavior is controlled by the *–daemon* option.

-genclient ⟨l⟩ ⟨k⟩ ⟨r⟩ [-daemon] ⟨port⟩

The *–genclient* version creates an empty filter with the given parameters and awaits a connection from a client on the given port. Since the module must act as a server, the *–daemon* option only affects the output.

-load [-daemon] ⟨port⟩

The *–load* option instructs the module to load a binary stored version of the filter from standard input. The server option is implied so *–daemon* controls the output only. Note that outputting to a stream allows the addition of new elements to an existing filter set.

4.3. *Filter query/comparison*

-clientstream ⟨host⟩ ⟨port⟩ ⟨block_size⟩

The client reads in a stream of data from standard input, generates MD5 hashes for every block of the given size, and passes on the result to the server on the specified host and port to be placed in a filter.

-clientmd5 ⟨host⟩ ⟨port⟩

The same as the previous option except for that the hashes are not generated from source data but are read from standard input (hexadecimal, one per line).

-querystream ⟨host⟩ ⟨port⟩ ⟨block_size⟩

The client reads in a stream of data from standard input, generates MD5 hashes for every block of the given size, and sends them as queries to the specified server host and port. Hashes that are found in the filter are printed to standard output.

-querymd5 ⟨host⟩ ⟨port⟩

The same as the previous option except for that the hashes are not generated from source data but are read from standard input (hexadecimal, one per line).

-diff ⟨file_1⟩ ⟨file_2⟩

Performs a bitwise comparison of two filters and prints to standard output the number of common bits, the number of set (to 1) bits in the first filter, and the number of set bits in the second one.

-print ⟨file⟩

Pretty prints the header information with all the parameters of the filter stored in the file.

5. Experiments

In this section we describe some initial experiments we performed to validate our work and to ensure that the probabilistic framework presented earlier does indeed perform as expected. We are still at an early stage of the development cycle (alpha) and are not ready at the time of this writing to present performance measurement, which are obviously very important to establish the viability of the proposed tool.

5.1. *MD5 false positive rate validation*

Since our entire construct hinges on the behavior of the MD5 function, our first order of business is to perform a few sanity checks and to observe the actual false positive rate. To observe the latter, we used independently generated streams of data from the *Linux* `/dev/random` device. The essential idea is that, if the filter is filled with hashes from random blocks and those are later compared with hashes of other random blocks, then any positives are the result of random collisions. The difference between the predicted rate and the observed is essential to understand the actual behavior of our tools.

The following command creates a filter using blocks from the random device:

```
dd -if=/dev/random -count=⟨n⟩ -cbs 512 | \
md5bloom -genstream ⟨l⟩ ⟨k⟩ ⟨r⟩ 512 -daemon \
1234
```

where n, l, k, and r vary for the different filter configurations tested. The corresponding client looks as follows:

```
dd -if=/dev/random -count=⟨N⟩ -cbs 512 | \
md5bloom -querystream ⟨host⟩ 1234 512
```

By default, the client outputs the hashes that have been recognized by the server so the false positive rate is given by the ratio of the number of recognized elements and the overall number of elements N (note that $N \gg n$ to produce a good estimate). The long-term averages over multiple runs produced results that closely match the predicted ones from Table 1. For the sake of brevity and to avoid repetition, we omit the actual numbers.

Another sanity check is to verify that, given the same input, the recognition rate will be 100%. That was trivially accomplished by creating a random file and feeding it both to the client and the server. (The randomness here is optional and it is only to ensure that we are not testing against files with large blocks of zeros.)

5.2. *Matching bit ratio versus PDF*

Let us call the ratio of matching bits to the average number of set (to 1) bits in the filters a *matching bit ratio* (*MBR*). We can now answer the question from the introduction of the paper—if the MBR is 50, 60, 70, …, %, is that a statistically significant result? In other words, at what point the chance of this happening becomes very small.

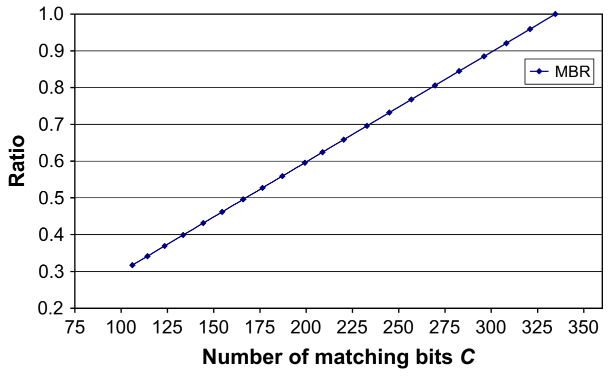

Fig. 3 – MBR for 0–100% overlapping files.

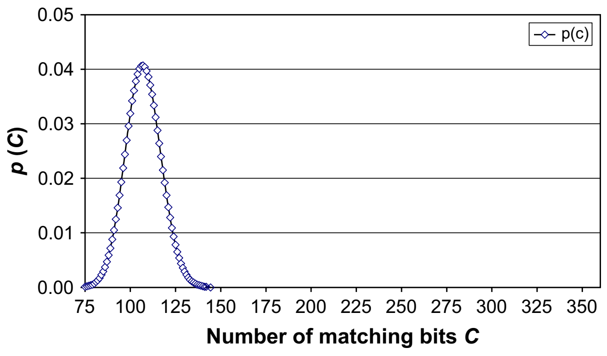

Fig. 4 – PDF corresponding to Fig. 3.

Since it is difficult to find 'in the wild' files with such specific MBRs, we performed a controlled experiment, in which file overlap was known ahead of time. Using the *Linux* random device, we generated two files of 100 MD5 hashes with no overlap and compared them. Then, we mixed 95% of the first file with 5% of the second (95/5 mix) and compared the resulting mix to the second file. Then, we compared a 90/10 mix to the second file and so on until we reached 0/100 mix (second file compared to itself). Note that since we directly use the MD5 hash records, there is a linear relationship between the number of matching bits and the number of overlapping records. The results are presented in Fig. 3 with the corresponding probability density function shown in Fig. 4.

Taken together, the two graphs tell the following story—the Bloom filters (with the chosen parameters) of two completely random files are expected to have about 33% matching bits. As the matching bits increase beyond 150, which correspond to 45% bits in common the *p*-value drops below 0.0001. Note that the 45% matching bits in the filter is achieved with only 20% overlap in the original hash files.

5.3. *Detecting hash tampering*

It is also possible to derive the MBR from the previous section analytically using the formula for the expected magnitude of the matching bits (Broder and Mitzenmacher, 2005). Yet, we decided to obtain simulation results to better understand the behavior of the system and, in particular, to verify our conjecture that the only way to observe a matching bits number to the left of the bell curve is to deliberately introduce non-randomness. Indeed, over the course of a 1000 runs of the experiment, the minimum number of matching bits C we recorded was 89—which fits comfortably under the bell of the curve (Fig. 4).

The other part of the experiment was to experimentally demonstrate a lower C. Clearly, achieving this by tampering the object data is tantamount to breaking MD5, which is still an open problem. Instead, we used simple techniques to gradually eliminate randomness from the MD hashes.

We tried two approaches—increasing the number of bits set to 1 and increasing the number of bits set to 0. One simple way to achieve these goals is to (globally) replace digits from the hexadecimal MD5 representation with 'F' and '0', respectively. This has the effect of introducing more repeat bit patterns, which results in fewer bits set and less randomness in the filter. Fig. 5 summarizes the results.

Correlating Figs. 4 and 5 shows that globally replacing about three to four digits in either case drives the expected overlap to the left of the curve: for $C < 85$ there is >99% chance that this is not random. There is a mild suggestion that introducing more of 1 bits (replace w/'F') is more easily detectable, which could be explained by the fact that 0 bits outnumber 1 bits 2-to-1 in these specific filters. More generally, the number of ones should not exceed the number of zeros in a properly constructed filter.

5.4. *Detecting object versioning*

In the introductory discussion, we pointed out that fine-grained hashing based on the logical components of an object can be a generic way to recognize unknown versions of known objects. In our proposed scheme, that would be accomplished by decomposing the object into logical modules, hashing every module, and inserting the resulting hashes into a Bloom filter, which becomes the hash representation of the object. At verification time, if the old and the new versions are identical, so will be their filter representation. Otherwise, a comparison of the number of matching bits versus the probability density function as discussed in Section 3.2 can provide a (potentially strong) hint that the new version is, indeed, a close relative of the old one.

To perform an initial test of the feasibility of this approach, we picked at random three pairs of *Linux* libraries with names that suggested version relationships. For the test, we used `objdump` to extract the content of the .text sections with all the library functions, extracted the machine code from the dump, generated the MD5 hashes for each individual function, and filtered out all duplicates (Table 4).

For each of the pairs, we created one or two versions of the Bloom filter representation by varying m. Based on our goals for a realistic m/n ratio between 8 and 16, we fixed the number of hash function $k = 4$, which yields false positive rates between 0.24 and 2.4% (Table 1). Based on initial observations, we also experimented with m/n ratio values of 4 or less. Normally, such Bloom filters would give positive rates that are too large to be used in a traditional way (>16% for $k = 4$); however, for hash set comparisons, they appear to be a reasonable choice.

The results are summarized in Table 5, where the notation is as follows: the column headers identify which libraries were

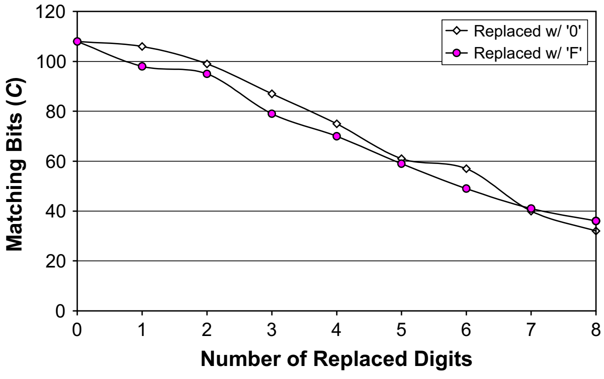

Fig. 5 – Effects of replacing MD5 digits.

Table 4 – List of tested libraries

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

compared in the experiment, e.g. 'fs' means that `fs1` and `fs2` were compared. The rows for each experiment contain: the size of the filters used (m); the number of elements in the first library (n_1); the number of elements in the second library (n_2), m/n ratio, where $n = (n_1 + n_2)/2$; the number of common elements ($S_1 \cap S_2$); the number of matching bits in the filter comparison (C); and the p-value with which we identify the libraries as versions of each other. Recall that p gives the probability that the matching bits happened randomly so a value <0.01 means that there is less than 1% chance of this happening by chance.

We believe that the results strongly support our hypothesis that versions of known objects can, indeed, be identified by comparing their Bloom filter representations. Furthermore, the solution is very efficient both in terms of space and time requirements. For example, the first column shows that we can represent about 50 elements with 256 bits (~5 bits per element) and expect to reliably a version which modifies approximately 2/3 of them. However, a version which contains about 1/4 modified elements can be discovered using only three bits per element (last column). The counting of the matching bits is straightforward and is naturally parallelizable, which is increasingly important with emerging multi-core processors and hardware multi-threading.

6. Conclusions

In this paper, we motivated and justified the use of Bloom filters as natural extension of existing filesystem hashing techniques with application to correlation of multiple targets and identification of object versioning. Specifically, our work makes the following contributions.

We built a general purpose stream-oriented tool, called *md5bloom* that supports the management of Bloom filters. In

Table 5 – Library version identification

	`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

addition to creating Bloom filters from source data, it allows for the direct use of existing hash sets to create Bloom filters.

We present an initial probabilistic framework that justifies the direct, bitwise comparison of Bloom filters and provides a means to interpret the results.

We provide simulation results that both validate our framework and demonstrate that it is possible to identify tampering with a hash set by comparing it to a random one.

We present initial tests on system libraries that demonstrate the practicality of identifying object versioning through Bloom filters. In particular, the result suggests that it is possible to use filters with relatively high false positive rates (>15%) and low number of bits per element (3–5) and still identify object versioning with high degree of confidence.

7. Future work

We expect to quickly beta-test *md5bloom* and release it as an open-source tool to be used by practitioners. We also plan to add support for other types of Bloom filters, such as counting and hierarchical (Shanmugasundaram et al., 2004).

We plan to perform large-scale testing with complete installations (e.g. compare different versions of *Linux*) to gain more practical insight.

On the theoretical side, we would like to obtain a more computationally attractive expression for the PDF of the matching bits. We expect that a normal distribution approximation will be a reasonable approach.

We plan to investigate the utility of the presented approach to security problems, such as intrusion prevention, detection, and mitigation.

Acknowledgements

We would like to thank the anonymous reviews for their contribution in improving the final presentation of this work.

References

Aguilera M, Ji M, Lillibridge M, MacCormick J, Oertli E, Anderson D, et al. Block-level security for network attached disks. In: Proceedings of the second Usenix conference on file and storage technologies; 2003.

Bellovin S, Cheswick W. Privacy-enhanced searches using encrypted bloom filters, draft paper, in preparation.

Bloom B. Space/time tradeoffs in hash coding with allowable errors. Communications of the ACM 1970;13(7):422–6.

Broder A, Mitzenmacher M. Network applications of bloom filters: a survey. Internet Mathematics 2005;1(4):485–509.

Dharmapurikar S, Attig M, Lockwood J. Design and implementation of a string matching system for network intrusion detection using FPGA-based bloom filters. Washington University in St. Louis, Computer Science Technical Report WUCSE-2004-12; 2004.

Dharmapurikar S, Krishnamurthy P, Taylor D. Longest prefix matching using bloom filters. IEEE Transactions on Networking, in press.

Estan C, Varghese G. New directions in traffic measurement and accounting. In: Proceedings of the 2001 ACM SIGCOMM internet measurement workshop; 2001.

Fan L, Cao P, Almeida J, Broder A. Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Transactions on Networking 2000;8(3):281–93.

Feng W-C, Shin K, Kandlur D, Saha D. Stochastic fair blue: a queue management algorithm for enforcing fairness. In: Proceedings of the twentieth annual joint conference of the IEEE computer and communications societies (INFOCOM 2001); 2001.

Ledlie J, Taylor J, Serban L, Seltzer M. Self-organization in peer-to-peer systems. In: Proceedings of the in tenth ACM SIGOPS European workshop; 2002.

U. Manber. Finding similar files in a large file system. In: Proceedings of the USENIX winter 1994 technical conference; Jan 1994. p. 1–10.

Mitzenmacher M. Compressed bloom filters. IEEE/ACM Transactions on Networks October 2002;10(5):613–20.

Shanmugasundaram K, Bronnimann H, Memon N. Payload attribution via hierarchical bloom filters. In: Proceedings of the ACM symposium on communication and computer security (CCS'04); 2004.

Spafford E. Opus: preventing weak password choices. Computer and Security 1992;11:273–8.

Valdurez P, Gardarin G. Join and semijoin algorithms for a multiprocessor database machine. ACM Transactions on Database Systems 1984;9(1):133–61.

Whitaker A, Wetherall D. Forwarding without loops in icarus. In: Proceedings of the fifth IEEE conference on open architectures and network programming; 2002.

Vassil Roussev is currently an Assistant Professor of computer science at University of New Orleans and also leads the Distributed Computing and Software Integration Lab. He holds a Ph.D. in Computer Science from the University of North Carolina—Chapel Hill and a B.S. and M.S. Computer Science degrees from Sofia University, Bulgaria. His main research interests include digital forensics, computer-supported cooperative work (CSCW), distributed computing, and software engineering.

Yixin Chen received a Ph.D. degree in computer science from The Pennsylvania State University in August 2003. Since then, Yixin has been an Assistant Professor of computer science at the University of New Orleans. Yixin's research interests include machine learning, data mining, biomedical informatics, artificial intelligence, computer vision, and robotics and control.

Timothy Bourg holds a B.S. in computer science from the University of New Orleans and is a graduate student member of the research group in digital forensics at the Department of Computer Science at the University of New Orleans. His thesis work is focused on building efficient digital forensic tools with quantifiable error rates.

Golden G. Richard III holds a B.S. in computer science from the University of New Orleans and M.S. and Ph.D. degrees in computer science from The Ohio State University. His primary research interests are in digital forensics, mobile computing, and operating systems. He is an Associate Professor of Computer Science at the University of New Orleans, a GIAC-certified Digital Forensics Investigator, and co-founder of Digital Forensics Solutions, LLC.