## Abstract

There is a problem in the world of digital forensics. The demands on digital forensic investigators and resources will continue to increase as the use of computers and other electronic devices increases, and as the storage capacity of these devices increases. The digital forensic process requires that evidence be identified and examined, and resources to do this are constrained. This is creating a backlog of work as seized media and devices wait to be analysed, and some investigations or checks 'in the field' may be reduced or discarded as impractical. There is a technique which can be used to help quickly to collect and examine data to see if it is of interest. This technique combines statistical sampling and hashes as described by Garfinkel et al (2010).

This tool can use a Bloom filter to match the hashes from disk sectors against the stored hashes for a file which is being searched for. The tool was successfully implemented and the Bloom filter false positive rate was as predicted by theory (Roussev, Chen, Bourg, & Richard, 2006) which confirmed that the Bloom filter had been correctly implemented. This tool was written in Python which proved a simple to use programming language. This prototype tool can provide the basis for further work on a practical tool for use in real world digital forensics investigation.

## 1 Introduction

The objective was to develop and evaluate a proof of concept forensic triage tool which would sample sectors from a disk, generate hash values, then compare those hash values with a reference set of hash values for the sectors. This comparison would be done using both a database of values, and a Bloom filter representation of the values.

## 2 Material and methods

### 2.1 Small Block File Hash Analysis

It has been described how the sampling of small blocks off disk and hashing can be used in forensics (S. Garfinkel et al., 2010). This research has two main themes. One is identifying files types from small blocks, looking for discriminators such as are found in jpeg file types. The other theme is looking for known files by sampling small blocks. The concept of 'distinct blocks' is introduced, that is the idea that to compare the hashes of two small blocks to see if they come from the same file, then the small block must be distinct ,that is not found in other files. Sampling small blocks rather than reading a whole file speeds up the matching process.

### 2.2 Matching Hashes

The hashes of the sectors read off disk need to be checked for a match against the hashes from the sectors in the search file. The search file sector hashes can be stored in a database, and this is the most obvious approach. As the sectors are read off disk, they are hashed and the database is checked for a match on that hash value. As the size of the database increases, then it would be expected that this look up will take longer. In real world examples the database could conceivably contain the individual hashes for each sector for thousands of files. So, it is worthwhile to consider whether there are efficient alternatives to a database for doing this matching.

## 3 Theory

### 3.1 Bloom Filters

As speed of analysis is an issue, then Bloom filters are of interest as they offer a fast way to make a comparison, in this case between the hash of a sector from a disk and the reference set of sector hashes for the known bad file. Bloom filters provide a means to quickly test whether an object is a member of a set. They can return a true negative, but not a false negative. They can return a true or a false positive, The eponymous Bloom filter was developed by Burton H. Bloom (Bloom, 1970). The purpose of the Bloom filter is to represent members of a set in a bit vector, which can then be used to compare an object and see if it is equal to a member of the set. Bloom filters cannot give false negatives, that is an object identified as not being a member of the set definitely will not belong to it. The Bloom filter can give false positives, but at a predicted level of probability. The Bloom filter match does not absolutely confirm a match, but as effectively a pre-processor it reduces the amount of checking whether an object is the member of a set by eliminating many true negatives.

The Bloom filter is a single bit vector initialized to be 0 0 0 0 0 0 0 .. etc. It is a bit representation of the members of a set. There is a set of values, which could for example be data blocks, or dictionary words, or urls, against which objects are to be compared to see if they are in the set. Each of these values is hashed to produce a number. The hash value of this is used to set the corresponding bit in the Bloom filter to 1. There can be multiple hashes of the same value using different hash functions. This is repeated for the all the members of the set to result in a Bloom filter of the form 1 1 0 0 1 0 ..

etc. where each 1 means at least one member of the set has a hash value corresponding to that index in the Bloom filter. To query if another object is possibly a member of the set, the same algorithm is used to produce hashes for this object and a comparison made with the Bloom filter. If every bit set to 1 by the object is also set to 1 in the Bloom filter, then it may be a member of the set. However, if any bit set to 1 by the object is set to 0 in the Bloom filter then it cannot be a member of the set.

Let

m = the number of bits in the Bloom filter
k = the number of hashing function
H be the hashing function

an example where m = 16, k = 3, members = { X0, X1,X2)

Bloom filter initialised to 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

There are 16 bits in the Bloom filter, so the hashing function H has to return a value between 0 and 15 with equal probability of any particular value. With m the number of bits in the Bloom filter, then H has to generate an index value between 0 and 1-m. So if m is 16 then H has to generate values between 0 and 15 which would require H to be a 4 bit number as $2^4$ will give 16 values. Accordingly, sometimes Bloom filter lengths are expressed as an exponential of 2, and that gives the bit size needed from the hash, such that a Bloom filter of length $2^{16}$ bits needs a 16 bit hash which will give $2^{16}$ possible values. As in this case there are 3 hashing functions each to produce a value between 0 and 15 to be used to set that index in the Bloom filter to 1.

e.g. H(X0) = {2, 4, 6} , H(X1) = {0,3,6), H(X2) = {7,8,9}

adding the hash values as a bit vector to the initialised Bloom filter

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Initialised Bloom filter | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Hashed values of member X0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Resulting Bloom filter | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Then continue for the other members i.e.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Hashed values X1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Resulting Bloom filter | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Hashed values X2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Resulting Bloom filter | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

The final Bloom filter can then be used to see if an object may be a member of the set. If the object is a member of the set then it will have the hash values that correspond directly to set bits in the Bloom filter. If any of the hash values of the object correspond to a 0 in the Bloom filter then that proves that it cannot possibly be a member. A false negative is impossible. However, a false positive is possible. For instance in this example, and object could have hash values {2,3,6]. Those bits are set to 1 in the Bloom filter, however they have not all been set as a result from hashing a single member, but comprise results from hashing several members. This is a trivial example. With a small bit length a Bloom filter would soon be set to all ones, as Bloom filters are typically used to represent large sets the bit length in practice is much larger.

As well as being a compact representation, Bloom filters allow fast comparison, as a look up could be done of each of index of the bit set to 1 by the object. If any of those indices in the Bloom filter is a 0 then that is a negative, otherwise it is positive, but it may be a false positive or a true positive. As mentioned, Bloom filters can give false positives, but not false negatives. When looking if an object is a member of a set, a false negative would be of concern. However, a false positive should only mean that if the members of the set were then queried directly the false positive would become apparent. While this would have involved unnecessary processing, then how significant the false positive is would depend on the circumstances. For instance a false positive in a url cache simple requires reloading the url cache from the source.

Although the concept of the Bloom filter is clear and elegant, the mathematics involved and the calculation of false probability rates are not intuitive. From (Roussev, Chen, et al, 2006) who cite (Fan, 2000) and (Mitzenmacher, 2002).

Where, n = the number of elements in the set whose membership will be represented by the Bloom filter, m = the number of bits in the Bloom filter and k = the number of hash functions to be applied to each element and then update the Bloom filter

Then the probability of a false positive is given as;

$$P=(1-(1-\frac{1}{m})^{kn})^{k} \quad \text{which approximates to} \quad (1-e^{-(kn/m)})^{k}$$

The predicted probability for false positives can be compared with experiment results to check the validity both of the Bloom filter calculation, and its implementation in the experiment.

### 3.1.1 Programs to make and use the Bloom Filter

The program for producing the Bloom filter needs to first initialise all the indices to zero. The program then sequentially reads every sector of the file. Each sector is hashed and if the hash is distinct it is written to the database and also used to set indices in the Bloom filter bit array. This process is repeated until every sector in the file has been read. The program design can be illustrated as below.

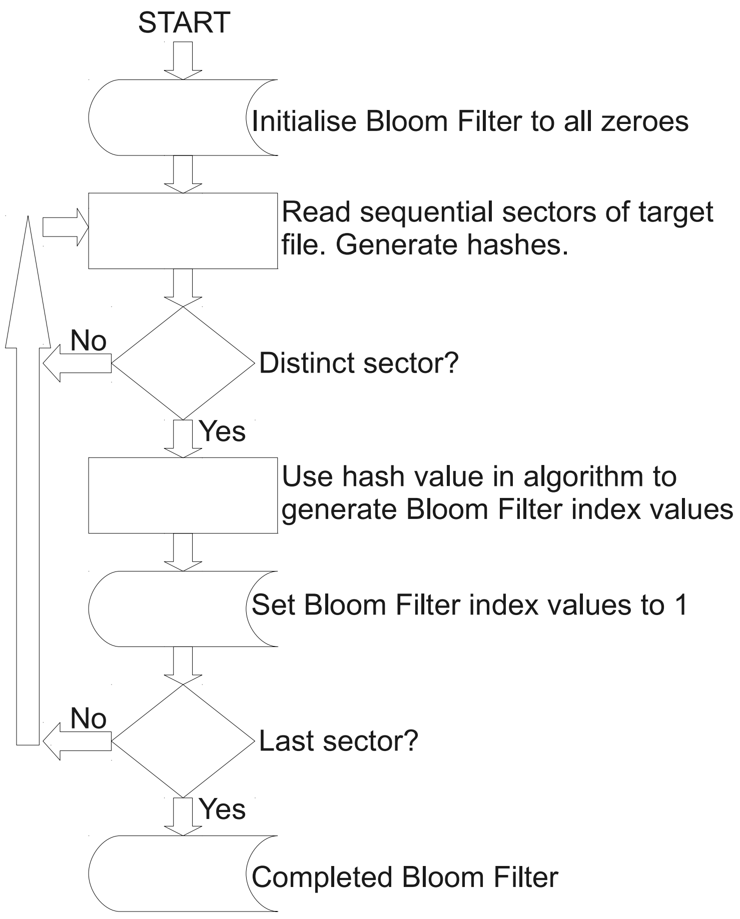
START
Initialise Bloom Filter to all zeroes
Read sequential sectors of target file. Generate hashes.
No
Distinct sector?
Yes
Use hash value in algorithm to generate Bloom Filter index values
Set Bloom Filter index values to 1
No
Last sector?
Yes
Completed Bloom Filter

*Figure 1: Schematic for constructing Bloom Filter*

The program to create the Bloom filter from the sector hashes from the search file was named makebf.py . Hashes for each sector in the source file (IMG_0036.JPG) are generated using the Unix dd command and md5sum operating system commands

```
 c = "dd if=" + ifile + " skip=%s count=1 | md5sum" % (s)
        os.system(c)
        result=os.popen(c).read()
```

The hash value is used to set indices in the Bloom filter bit array. The number of hash functions used to set the Bloom filter bit array is a variable k. The md5sum value is a 32 character hexadecimal. Where for example the Bloom filter bit length is $2^{16}$ bits, then a 4 character hexadecimal gives $(2^4)^4$ or $2^{16}$ possible values. So a sequence of 4 character hexadecimal values from the md5sum hash can be extracted, each one then converted to set to 1 an index in the Bloom filter. This is repeated for consecutive hexadecimal values k times. The result is that the Bloom filter bit array is set to 1 in k locations, each with an index value between 0 and $2^{16}-1$ .

```
for i in range(k):
                bf[int(result[i*b:(i*b)+b], 16)]=1
        with open(bloomfile, 'wb') as fh:
            bf.tofile(fh)
```

This is repeated for every sector in the jpeg file, to give a database of hashes for each sector, and a Bloom filter bit array with some indexes set to 1.

A full code listing is available at http://www.napier.ac.uk/...... As well as the Bloom filter, a database of the hashes was also created by the program.

From the literature review it was apparent that Bloom filters could offer speed benefits in matching (Farrell, Garfinkel, et al, 2008), It was also attractive to see if a Bloom filter could be implemented and how it would perform. The program to use the database is still useful because it can be used as a check on the Bloom filter results, and is also insurance against the possibility of the Bloom filter program not working as predicted. After creating a Bloom filter bit array which also 'stores' hashes for all the distinct sectors in the file, experiment can then be carried out against a disk, or disk image, to see if the file may be present. Again, this program samples sectors and does not read the entire disk image. Each sector is hashed and that hash is used to set indices using the same algorithm as used to generate the Bloom filter bit array. If these indices are all set in the Bloom filter then that is a match, although with the Bloom filter this could be a false positive. If there is a match this is displayed. Whether it matches or not the next sector is sampled and hashed. It would be possible to break the program on the first match, as at that stage the file has been located, however this may be a false positive match.

The program which used the Bloom filter to check for matches in the disk image was named useby.py . In this program sampled sector hashes are generated in the same way as in the usbdb.py i.e.

```
for s in range(sstart,sstop,sint):
       match = 0
       c = "dd if=" + ifile + " skip=%s count=1 | md5sum" % (s)
       os.system(c)
       result=os.popen(c).read()
```

The difference in the two programs is of course the comparison, which is using the Bloom filter bit array. For the md5sum hash value, the same algorithm is used as was used in the makebf.py program. If for the hash of a particular sector all k hash Bloom filter hash functions, there is a match in the Bloom filter bit array then that is a Bloom filter match and is displayed.

```
     for i in range(k):
           #print[int(result[i:i+4], 16) % 2**16] , bf[int(result[i:i+4], 16) % 2**16]
           if bf[int(result[i*b:(i*b)+b], 16)] == 0:
           #if bf[int(result[i:i+b], 16)] == 0:
               match = 0
               break
           else:
               match = 1
       if match == 1:
           print str(j) , 'bf Sector ' , s , ' hash ', result[:32] , ' is a match.'
```

A full code listing is available at http://www.napier.ac.uk/.....

## 4 Results

### 4.1 Triage

Combining the techniques of sampling, hash analysis and matching using Bloom filters which have been described above it is possible to conceive a tool which can look for a file quicker than would be the case with full disk collection and examination. The concept of rationally rationing limited resources by prioritizing needs is well known, and triage is best known for its medical application in life threatening situations. The parallel situation, of limited resources, can arise in computer forensics, where there is insufficient resource in the time available to comprehensively analyse seized or suspect data, and what is required is a technique to quickly identify suspicious data warranting further investigation, while eliminating data which is not suspicious. The role of triage in the digital forensics process has been clearly identified as contributing to closing the gap between growing demand for forensic examination and the limits on investigation resource (Cantrell, Dampier, Dandass, et al, 2012) (Overill & Silomon, n.d.). This triage approach has been modelled for computer forensics in the field, based on an informal but effective approach being used by investigators (Rogers, Mislan, Goldman, Wedge, & Debrota, 2006) and there are commercially available tools to support a triage approach such as STRIKE (System for TRIaging Key Evidence) (S. L. Garfinkel, 2010). It has also been applied to mobile devices (Walls, Learned-Miller, et al, n.d.) (Mislan, Casey, & Kessler, 2010). Another triage based approach to digital forensics categorises seized data according to a quick profile of the user (Grillo, Lentini, et al, 2009), this is an interesting approach, but in this study we are ignoring the user and only considering the data.

#### 4.1.1 HASTT

The tool is a proof of concept for a digital forensics investigation triage tool using disk sector hash analysis, sampling and Bloom filter or database for matching. For convenience this tool will be given the acronym HASTT (**H**ash **A**nalysis **S**ampling **T**riage **T**ool).

### 4.2 Bloom Filter Implementation

As has already been described, it is inherent with Bloom filters that they can give false positives as a result of collisions. This is where a value which is not in the reference set is hashed and matches with the Bloom filter because it overlaps with a composite of matches for several values which are in the reference set. These collisions are inevitable, although the rate depends on the Bloom filter settings.

Particular to sector hash analysis there is however another false positive which is avoidable. The concept of distinct sectors has already been described, that is that a sector is distinct if it is unique to a file and therefore its presence is a guarantee that the file it came from, and no other, is present. The possible combination of values at random for a sector is $512 x 2^8$ , an astronomical number. Of course sectors are not filled at random, and some sectors are 'indistinct' that is they appear in more than one file. It then follows that the detection of an indistinct sector is not a guarantee that the file it came from, and no other is present. The clearest, and commonest indistinct sector is 'all nulls'. This is common where nulls are used to stuff unused sectors, the 'slack space', in a cluster, and this slack space can be used covertly to conceal data (Berghel, Hoelzer, et al, 2006), but in normal use will be stuffed with nulls. For example, a file may be 15 sectors in size on a file system organised into clusters each of 8 sectors. The 15 sector file requires 2 such clusters, and the unused 16th sector may be stuffed with nulls.

The image file nps-2009-canon2 (30MB FAT16 SD card image) from the Digital Corpora website (Digital Corpora, 2012) was used to provide sample files. This image file contains 36 jpeg files. From this IMG_0036.JPG was chosen to be the 'bad' file searched for, and the other 35 jpeg files would be used for contrast. Before creating a Bloom filter of its sector hashes it was necessary to identify any indistinct sectors and then avoid using them when creating the Bloom filter. This was done by hashing all sectors on the 30MB corpora and then using the counter module within Python to identify any duplicate, and hence indistinct sectors. The duplicate sectors within the IMG_0036.JPG were thus identified, and as expected this included the sector of all nulls. There is of course the possibility that sectors would remain which might be found on other media in other files, but that is unavoidable.

Experimentally it would still be possible to detect any such indistinct sector matches later by noting when a positive match is found whether the match occurred on a sector within the known sector allocation for the bad file. False positives would be identified in a 2 step approach. When the Bloom filter matched a sector which was outwith the known sector allocation, that sector hash could then be looked up in the database of all the bad file sector hashes. If the hash value was not in there, then it was a false-positive caused by a Bloom filter collision. If the hash value was in the database then as in all cases the allocated sectors for the bad file would be known it was therefore a sector from another file but identical to a sector in the bad file, and was thus a false positive due to being an indistinct sector. Of the 1725 sectors in the bad file IMG_0036.JPG, 1718 were distinct, and 7 were indistinct.

A decision had to be made about the parameters to be used in the Bloom filter. It was convenient to generate hash functions for the Bloom filter indices by using bytes from the MD5 hash of the sector. The MD5 hash is 32 bytes (128 bits), which conveniently could generate 8 x 4 byte chunks (8 x 16 bits), each being 4 bytes was adequately distinct and each chunk mapping to one of $2^{16}$ possible values, which would map directly to a Bloom filter with a bit length of $2^{16}$ .

The predicted false positive rate for a Bloom filter has already been described and is

Pfp = $(1-e^{-(kn/m)})^k$

In this case the values will be;

| | |
|---|---|
| n = 1718 | This is the number of distinct sectors in the bad file IMG_0036.JPG, each of which will be hashed and used to set indices in the Bloom filter. |
| m = $2^{16}$ = 65536 | This is the bit length of the Bloom filter. |
| K = 8 | This is the number of hash functions applied to the MD5 sector hash in order to set an index in the Bloom filter bit array. |

This gives a value of Pfp = 0.00000164

For convenience Pfp was calculated for m = $2^{12}$ also, as that could be mapped to chunks of 3 bytes from the MD5 hash of the sector. For both m = $2^{16}$ and m = $2^{12}$ ,values of k =8 and k = 4 were used for comparison. A tabulation of the results for Pfp is;

| M (BF bit length) | K (# of hash functions) | Predicted FP rate |
|---|---|---|
| 65536 ( $2^{16}$ ) | 8 | 0.00000164 |
| | 4 | 0.00009820 |
| 4096 ( $2^{12}$ ) | 8 | 0.75266841 |
| | 4 | 0.43731715 |

Because of the advanced maths, and the complexity of understanding how the Bloom filter probabilities compute, at an early stage it was decided to run some experiments to assess whether the program written to compute matches using Bloom filters agrees with the predicted false positives. The experiment for false positive was run against the 30MB USB memory stick, containing the 'bad' file 'IMG_0036.JPG' and 35 other 'good' files. This was done sampling every 100th sector, and with 100% of all sectors.

| Initial evaluation of Bloom filter parameters, predicted false Positive vs actual (10 iterations) | | | | | | |
|---|---|---|---|---|---|---|
| M (BF bit length) | K | Sectors sampled | Predicted FP rate | Predicted FP count | Actual FP count | Actual/ Predicted |
| 65536 ( $2^{16}$ ) | 8 | 6080 | 0.00000164 | 0 | 0 | 0.00 |
| | 4 | 6080 | 0.00009820 | 1 | 0 | 0.00 |
| 4096 ( $2^{12}$ ) | 8 | 6080 | 0.75266841 | 4576 | 4468 | 0.98 |
| | 4 | 6080 | 0.43731715 | 2659 | 2481 | 0.93 |
| 65536 ( $2^{16}$ ) | 8 | 60799 | 0.00000164 | 0 | 0 | 0.00 |
| | 4 | 60799 | 0.00009820 | 6 | 10 | 1.67 |
| 4096 ( $2^{12}$ ) | 8 | 60799 | 0.75266841 | 45761 | 44445 | 0.97 |
| | 4 | 60799 | 0.43731715 | 26588 | 24833 | 0.93 |
| Constants: n = 1718; image file = 30Mnpsf1.raw; target file=img_0036.jpg; K= # of hash functions | | | | | | |

*Table 1: Bloom filter false positives - predicted vs actual (30MB)*

The results were encouraging, but not exact. The reasons for the discrepancy could be due to the particular distribution of sectors on this disk, but the high correlation gives confidence that the program is implementing the Bloom filter correctly, and is consistent with the false positive rate predicted by theory. It was decided to repeat the experiments on a larger scale, against a 256MB USB stick. First with a stick containing a single copy of the bad file, and with the 35 good files copied hundreds of times to fill the stick.

| M (BF bit length) | K | Sectors sampled | Predicted FP rate | Predicted FP count | Actual FP count | Actual/ Predicted |
|---|---|---|---|---|---|---|
| 65536 ( $2^{16}$ ) | 8 | 489471 | 0.00000164 | 1 | 0 | 0.00 |
| | 4 | 489471 | 0.00009820 | 48 | 85 | 1.77 |
| 4096 ( $2^{12}$ ) | 8 | 489471 | 0.75266841 | 368409 | 361720 | 0.98 |
| | 4 | 489471 | 0.43731715 | 214054 | 213984 | 1.00 |
| Image file = 256Mjcusbf1.dd | | | | | | |

*Table 2: Bloom filter false positives - predicted vs actual (256MB)*

For the $2^{16}$ bit length the actual and predicted totals are minuscule and consistent. Though the numbers are small the actual false positives for the $2^{12}$ bit length Bloom filter are very slightly less than predicted. Interestingly this is in contrast to a paper (Bose et al., 2008) which contends that the established formula for Bloom filter false positives will underestimate the actual false positive rate, though by a slight amount which can be negligible for a large value of m and a small value of K.

## 5 Discussion

Baseline experiments established the effect of what might be termed environment factors. These factors would include how the program is executed and the environment it is run in. The data source type would also be a factor to baseline. As well as establishing benchmark times, the base lining quantified the effect of these environmental factors to identify which might be significant so that later when the parameters for Bloom filter settings were varied then any change in output such as program times could more reasonably be attributed to the change in the parameters.

Once base lining had been established then the Bloom filter parameters were varied to see what effect that has on outputs. The two variable Bloom filter parameters are the bit length of the Bloom filter (**m**) and the number of hashing functions (**k**) used to set the Bloom filter. Other variables for experimentation are the source file size and the sampling intervals. Different permutations of these parameters were tested. These are the only parameters a Bloom filter can take, and it was important to assess if the false positive rates were as predicted, and also to see what the  effect on speed is of varying these.

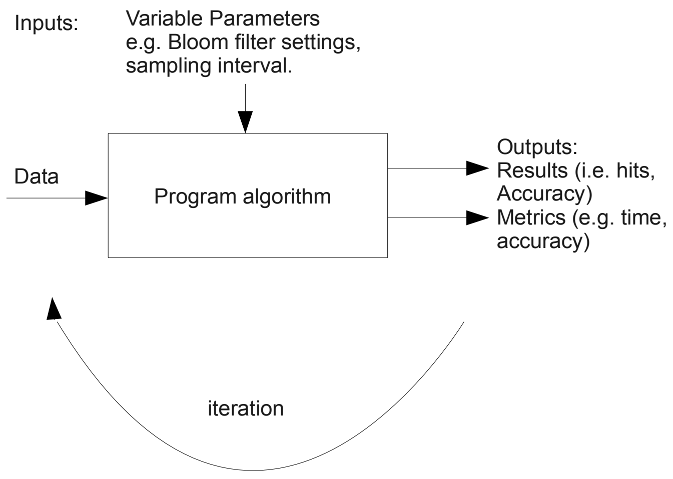

*Figure 2: General experiment program flow*

Initial experimentation was carried out on a local PC. This showed that the match time for comparing the hash value of the sector was much less than the time taken to read the sector and generate the hash value.

| **Command line 256MB** (all times in s) (20 iterations) | | | | | | |
|---|---|---|---|---|---|---|
| Sampling interval | Bloom filter match c/w checks & measures | | | | | |
| | Disk times | | | Match times | | |
| | Average | Min | Max | Average | Min | Max |
| 500 | **17.5792** | 17.2202 | 18.3854 | **0.0814** | 0.0792 | 0.0928 |
| 1000 | **8.7934** | 8.6020 | 9.0253 | **0.0407** | 0.0400 | 0.0421 |
| 2000 | **4.3903** | 4.2863 | 4.7863 | **0.0204** | 0.0198 | 0.0209 |
| Local PC. Bloom filter m/k settings $2^{16}$ /4. Image file = 256Mjcusbf1 | | | | | | |

Table 3: Baseline experiment (command line) 256MB - Disk times vs Match times

It is very apparent that while both times are inversely proportional to the sampling interval that the disk times are much more significant in overall time than the match time.

Experiment were repeated sampling and matching disk sector hashes and comparing using the database of hashes, and comparing using a Bloom filter. The times for the comparisons were as follows;

| **Command line 256MB** (all times in s) (20 iterations) | | | | | | |
|---|---|---|---|---|---|---|
| Sampling interval | Match times | | | | | |
| | Database Matching | | | Bloom Filter Matching | | |
| | Average | Min | Max | Average | Min | Max |
| 500 | **2.4542** | 2.4310 | 2.4763 | **0.0861** | 0.0824 | 0.0932 |
| 1000 | **1.2298** | 1.2169 | 1.2513 | **0.0417** | 0.0400 | 0.0456 |
| 2000 | **0.6491** | 0.6069 | 0.8940 | **0.0214** | 0.2030 | 0.0238 |
| Local PC. Bloom filter m/k settings $2^{16}$ /4. Image file = 256Mjcusbf1 | | | | | | |

The inferred huge speed advantage of the Bloom filter over the Database was confirmed. However, the database used, SQLite, might be low performance, and comparison with another database, such as MySQL could be useful.

All the baseline experiment so far had been done on the local PC. While this has a low specification it is a relatively controlled environment, and does not share resources with other users. It was expected that performance would be better on the remote VM. As was expected, because of the higher specification, the remote VM provided better performance than the local PC. Most testing was carried out using the remote VM. On the remote VM experiments were carried out to study the effect of varying m and k on Match Times with the following results.

| **256MB Match Times** (in s) for various m/k combinations | | | | | | |
|---|---|---|---|---|---|---|
| | m/k $2^{16}/8$ | | | m/k $2^{16}/4$ | | |
| Sampling interval | Average | Min | σ | Average | Min | σ |
| 500 | **0.02288** | 0.01738 | 0.00245 | **0.01825** | 0.01613 | 0.00488 |
| 1000 | **0.00970** | 0.00838 | 0.00145 | **0.00893** | 0.00815 | 0.00070 |
| 2000 | **0.00458** | 0.00413 | 0.00048 | **0.00445** | 0.00405 | 0.00045 |
| 4000 | **0.00228** | 0.00205 | 0.00028 | **0.00223** | 0.00200 | 0.00015 |
| | m/k $2^{12}/8$ | | | m/k $2^{12}/4$ | | |
| | Average | Min | σ | Average | Min | σ |
| 500 | **0.69795** | 0.66138 | 0.02615 | **0.41455** | 0.38968 | 0.01335 |
| 1000 | **0.35135** | 0.32975 | 0.01700 | **0.20900** | 0.19480 | 0.00950 |
| 2000 | **0.17523** | 0.16158 | 0.00928 | **0.10468** | 0.09075 | 0.00705 |
| 4000 | **0.08548** | 0.07623 | 0.00573 | **0.05258** | 0.04333 | 0.00505 |

Table 4: 256MB - Comparison of various Bloom filter m and k values – match times – Remote VM

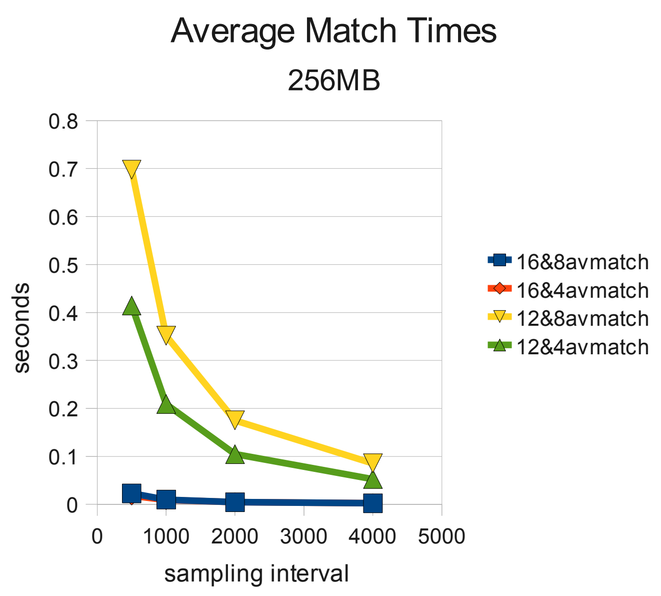

*Figure 3: 256MB various Bloom filter values – average match time vs sampling interval*

While match times remain the lesser component of overall program times, when studying the effect of m and k, the effect is most noticeable by looking at the match times. The match times for m and k values of $2^{12}$ and 8 are highest, followed by times for a $2^{12}$ nd 4, with times for $2^{16}$ and 8 slightly higher than times for a $2^{16}$ nd 4. The extra time expected to calculate 8 hashing functions compared to 4 was not noticeable. However, with m at $2^{12}$ the number of false-positives was so much greater than with m at 16 that program time was taken in recording and checking all these false-positives, each of which was checked in the database.

## 6 Conclusions

Implementing a Bloom filter for evaluation was not especially difficult. The predicted false positive rates were very close to those achieved in this case. While testing was limited to a single type of database, the Bloom filter gave an impressive speed gain in comparison times over the times achieved using the database. It is the opinion of the author that the principle of the Bloom filter should be widely understood, along with its' limitations of false positives. The Bloom filter can allow very rapid matching against a reference set of values with a predictable false positive rate.

When using a Bloom filter for comparisons the setting of m and k can have a significant, and predictable effect on accuracy. The choice of values for experiment was to some extent initially arbitrary but the comparison of values of m of $2^{16}$ and , $2^{12}$ and for k of 8 and 4 showed a very wide range of accuracy especially between the values for m of $2^{16}$ and . $2^{12}$ The results in this experiment environment showed that a Bloom filter gives a faster match than a database lookup. This can be achieved with a low, and predicted low, false positive rate by selecting appropriate parameters for the Bloom filter. In this case the key to this was in using the Bloom filter bit length of $2^{16}$ rather than. $2^{12}$ Whether the number of Bloom filter hashing functions was 4 or 8 made relatively little difference.

In this experiment environment when the total times for executing the disk sector hashing comparison were measured then the time for disk read and hashing was much more significant than the database or Bloom filter match time. However when looking at the comparison times in detail, then the Bloom filter was much quicker than the database.

## References

Berghel, H., Hoelzer, D., & Sthultz, M. (2006). Data hiding tactics for windows and unix file systems. *Berghel.net*. Retrieved from http://www.berghel.net/publications/data_hiding/data_hiding.php

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, *13*(7), 422–426. doi:10.1145/362686.362692

Bose, P., Guo, H., Kranakis, E., Maheshwari, A., Morin, P., Morrison, J., Smid, M., et al. (2008). On the false-positive rate of Bloom filters. *Information Processing Letters*, *108*(4), 210–213. doi:10.1016/j.ipl.2008.05.018

Cantrell, G., Dampier, D., Dandass, Y. S., Niu, N., & Bogen, C. (2012). Research toward a Partially-Automated, and Crime Specific Digital Triage Process Model. *Computer and Information Science*, *5*(2), 29–38. doi:10.5539/cis.v5n2p29

Digital Corpora. (2012). Digital Corpora - Disk Images. *Digital Corpora website*. Retrieved July 10, 2012, from http://digitalcorpora.org/corpora/disk-images

Fan, L. (2000). Summary Cache : A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM TRANSACTIONS ON NETWORKING*, *8*(3), 281–293.

Garfinkel, S. L. (2010). Digital forensics research: The next 10 years. *Digital Investigation*, *7*, S64–S73. doi:10.1016/j.diin.2010.05.009

Garfinkel, S., Nelson, A., White, D., & Roussev, V. (2010). Using purpose-built functions and block hashes to enable small block and sub-file forensics. *Digital Investigation*, *7*, S13–S23. doi:10.1016/j.diin.2010.05.003

Grillo, A., Lentini, A., Me, G., & Ottoni, M. (2009). Fast User Classifying to Establish Forensic Analysis Priorities. *2009 Fifth International Conference on IT Security Incident Management and IT Forensics*, 69–77. doi:10.1109/IMF.2009.16

Mislan, R. P., Casey, E., & Kessler, G. C. (2010). The growing need for on-scene triage of mobile devices. *Digital Investigation*, *6*(3-4), 112–124. doi:10.1016/j.diin.2010.03.001

Mitzenmacher, M. (2002). Compressed Bloom Filters. *IEEE/ACM TRANSACTIONS ON NETWORKING*, *10*(5), 604–612.

Overill, R. E., & Silomon, J. A. M. (n.d.). Digital Meta-Forenscs: Quantifying the Investigation. London. Retrieved from http://www.dcs.kcl.ac.uk/staff/richard/CFET_2010.pdf

Rogers, M. K., Mislan, R., Goldman, J., Wedge, T., & Debrota, S. (2006). Computer Forensics Field Triage Process Model, 27–40.

Roussev, V., Chen, Y., Bourg, T., & Richard, G. G. (2006). md5bloom: Forensic filesystem hashing revisited. *Digital Investigation*, *3*, 82–90. doi:10.1016/j.diin.2006.06.012

Walls, R. J., Learned-Miller, E., & Levine, B. N. (n.d.). Forensic Triage for Mobile Phones with DEC0DE.