Practical Applications of Bloom filters to the NIST RDS and hard drive triage.

Paul Farrell
Naval Postgraduate School
Monterey, CA

Simson L. Garfinkel
Naval Postgraduate School
Monterey, CA

Douglas White
NIST
Gaithersburg, MD

Abstract

Much effort has been expended in recent years to create large sets of hash codes from known files. Distributing these sets has become more difficult as these sets grow larger. Meanwhile the value of these sets for eliminating the need to analyze "known goods" has decreased as hard drives have dramatically increased in storage capacity.

This paper evaluates the use of Bloom filters (BFs) to distribute the National Software Reference Library's (NSRL) Reference Data Set (RDS) version 2.19, with 13 million SHA-1 hashes. We present an open source reference BF implementation and validate it against a large collection of disk images. We discuss the tuning of the filters, discuss how they can be used to enable new forensic functionality, and present a novel attack against bloom filters.

1. Introduction

Previous work has identified Bloom filters as attractive tools for representing sets of hash values with minimal error rates[17]. The National Institute of Standards and Technology (NIST) distributes a sample Bloom filter implementation in perl and two BFs containing a small subset of the NIST RDS 2.13[20]. Nevertheless, to date there has been no published research on large-scale BF implementations optimized for speed; on-disk representations for BFs have not been standardized; and BFs have not been publicly incorporated into open source forensic tools.

Meanwhile the NSRL's RDS [19] continues to grow, with the July 2008 RDS 2.21 release mapping 47,553,722 known files to 14,563,184 unique hashes. NIST has also announced its intentions to distribute dramatically expanded hash sets "of each 512-byte block of every file we process"[20].

While storing these 14 million SHA1 hashes requires 291 megabytes—a small amount of storage in today's world—the ancillary material that accompanies these hash codes expands the size of the RDS to almost 6GB, making the files somewhat difficult to distribute and work with. The time to search this database is also increasing, since most tools use either a binary search or some kind of index tree, producing access speeds that scale with the log of the dataset size. In our testing with very capable reference hardware, we could only perform between 17 thousand and 85 thousand RDS hash lookups per second using SleuthKit's `hfind` command[1], and only 4 thousand lookups per second when the hashes were stored in a MySQL InnoDB database.

Bloom filters are an attractive way for handling very large hash sets. In this paper we present a new BF implementation that can perform between 98 thousand and 2.2 million lookups per second on that same hardware.[2]

The RDS includes a significant amount of metadata for each file, including the file's name, size, the software package and operating systems for which it was distributed, and the publisher. This information is not used by SleuthKit's `hfind` command, Guidance Software's EnCase[11], or other tools we have evaluated. This metadata is largely unexploited which, when accessed efficiently, can assist in the rapid analysis and triage of newly acquired hard drives.

1.1. This paper's contribution

This paper applies and extends previous work on BFs to the National Software Reference Library's (NSRL) Reference Data Set (RDS), specifically:

- We present nsrl_bloom, a new, efficient, highly configurable, open source Bloom filter implementation in C.
- We modified `fiwalk`, an automated forensics tool based on SleuthKit, to use on our BF implementation to automatically exclude "known goods."
- We evaluate the performance of BFs created with different parameters and compare actual results with the NSRL RDS to the results predicted by theory.
- We evaluate performance of BFs compared to lookups in sorted flat-files (what SleuthKit uses) and lookups of hashes stored in a large MySQL database.

[1] The range results from the fact that `hfind`'s binary search algorithm terminates early when it finds a hash that is in the database. As a result, looking up a hash that is in the data set is roughly 5 times faster than looking up a hash that is not. Surprisingly, MySQL exhibits similar performance for both kinds of hashes.

[2] Once again, the range is the result of the difference in time between looking up a hash that is not in the database and one that is. Unlike binary searches on sorted data, BFs terminate faster when searching for data that is not present.

1063-9527 2008

DOI 10.1109/ACSAC.2008.12

- We evaluate the coverage of RDS over fresh installs of Windows 2000, XP and Vista.
- We present a novel attack against the use of BFs to eliminating "known goods."
- We evaluate the use of BFs for cross-drive analysis and the distribution of extracted features.

1.2. Related Work

Bloom introduced "hash coding with allowable errors" in 1970[2]. Dillinger and Manolios showed how BFs could be produced that are fast accurate, memory-efficient, scalable and flexible[8]. Fan *et al.* introduced *Counting Bloom Filters*, in which small integers are stored in the vector instead of individual bits[9]. *Bloomier Filters*[5] use layered BFs for mapping entries to one of multiple sets. Broder presents equations for computing the optimal number of hashing functions can be calculated to provide a minimal false positive rate[3]. Manolios provides a simple online calculator for computing these values[14].

BFs have since been applied to various forensic applications[3]. Roussev *et al.* proposed using BFs for storing file hashes[17]. The researchers also stated object versioning could be detected by piecewise hashing file parts, noting "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." Unfortunately the source code for their program, `md5bloom`, was never released. In 2007, Roussev *et al.* expanded this research with Multi-Resolution Similarity Hashing to detect files that were similar but different[18].

White created a sample BF implementation in Perl and distributed it from the NIST website[19]. This code was flawed, in that each bit of the MD5 or SHA1 code was used multiple times to compute multiple Bloom hash functions. Furthermore, because it was written in perl, excessive memory consumption prevented this program from being able to digest the entire RDS.

1.3. Outline of paper

Section 2 discusses BFs and our BF implementation. Section 3 discusses our experience at applying our implementation to the NSRL RDS. Section 5 discusses the implications of this work to mainstream forensics research and practice. Section 6 concludes.

2. High-performance BFs

2.1. Introduction to Bloom filters

Fundamentally the Bloom filter is a data structure that allows multiple values $V_0 \ldots V_n$ to be stored in a single finite bit vector F. As such, BFs support two primitive operations: storing a new value V into a filter F, and querying as to whether or not a value V' is present in F.

BFs work by computing a hash of V and scaling this hash to an ordinal between 0 and m, producing a bit i. This bit is then set in the bit vector F which is also of size m. To query the filter to see if V' is present, the value V' is hashed and scaled to produce bit i'. If bit i' in the filter is not set, then value V' could not have been stored in the filter.

If bit i' is set, the bit may be set because V' was previously stored in the filter. Alternatively, another value V'' may have been stored that has a scaled hash i'' that is equal to i'. The values V' and V'' are *aliases*: users of the filter cannot determine which of these two values was previously stored. Because of this property, BFs are said to offer *probabilistic* data retrieval: if a BF says that a value was not stored in the filter, then it was definitely not stored. But if the BF says that a value was stored, the value might have been stored; alternatively, an alias may have been stored. As more information is stored in a BF, the probability of aliases and false positives increase.

In practice, multiple hash functions $f_1 \ldots f_k$ are used to store a single value V into filter F. This constellation of bits is referred to as an *element*. Storing data thus requires setting k different bits ($i_1 \ldots i_k$) in filter vector F while querying requires checking to see if those bits are set.

BFs can thus described by four parameters:

- m: the number of bits in the filter. (In this paper we additionally use M to denote $log_2(m)$.)
- k: the number of hash functions applied to produce each element
- b: the number of bits set per element in the filter
- f: the hash function

Once data is stored in the filter, additional parameters can be used to describe the filter's state:

- n the number of elements stored in the filter
- p the probability that a value V, reported to be in the filter, was actually stored in the filter.

As discussed elsewhere[16, 15, 17], the probability that an element in a filter will *not* be set is:

$$P_0 = (1 - 1/m)^{kn} \tag{1}$$

This can be approximated as:

$$P_0 = e^{-kn/m} \tag{2}$$

The theoretical false positive rate is approximated as:

$$P_{fp} = (1 - e^{-kn/m})^k \tag{3}$$

2.2. Performance Characteristics

Bloom filters are similar to hash tables (HTs), in that a large number of sparse values can be stored compactly in a single data structure. Once stored, the structure allows a values' presence or absence to be queried in constant time irrespective of the amount of data that has been stored. BFs have an advantage over HTs in that data can be stored in significantly less space; they have the disadvantage that the retrieval is probabilistic—a given BF might say that the data is present, when in fact it is not.

Another commonality between BFs and HTs is that neither enjoys *locality of reference*[7] because data is hashed throughout the structure. Because of real performance characteristics of memory caches and memory hierarchies, considerable performance advantages can be achieved on modern computers by minimizing the memory footprint of a data structure. This is especially true for structures like BFs and HTs that do not access memory in any predictable order: there is no way to achieve locality of reference other than shrinking the data structure to fit within a cache.

Consider a modern Macintosh iMac desktop based on an Intel Core 2 Duo microprocessor. This processor runs with an internal clock speed of 2.4Ghz. But because there is no locality of reference, the speed of accessing each BF bit will be roughly equal to the speed of the memory system in which that element is stored (Table 1), ignoring the overhead associated with the BFs hashing and bookkeeping. A 16K BF that fits entirely into the computer's 32K L1 data cache can be accessed at the rate of roughly 40 million 20-function queries per second (the remainder of the cache is required for the state associated with the aforementioned hashing and bookkeeping). On the other hand, a filter that is 500MB in size and fits entirely in main memory can only support 710,000 hash lookups per second, because the computer's high-performance memory subsystem nevertheless requires 70 nanoseconds to fetch each hashed bit.

Most memory systems are pipelined, allowing multiple reads to be outstanding at any given time. Furthermore, the Core 2 Duo can execute 4 instructions per cycle. This capability will be used to perform bookkeeping activities such as incrementing loop counters and shifting bits; eventually the thread will stall until the requested bit is fetched from the memory subsystem in which it resides. If we can reduce the delay in fetching the data from memory and the number of times we must fetch per lookup, we can significantly speed the hash lookup process. By varying the size and number of hash functions for a BF, we can optimize our data set representation for performance.

2.3. nsrl_bloom

White's original code[19] had a flaw that caused each bits of the MD5 or SHA1 hash in multiple Bloom hash functions. As a result, the bits were correlated and BFs created with the perl code showed a factor of 10 more false positives than predicted by theory. For BF's to be effective, the hashing functions must be truly random and independent. To avoid this correlation, our implementation simply divides the hash into pieces based on the size of the BF: a Bloom filter with $k = 4$ and $M = 28$ uses the first 28 bits of the SHA1 for the Bloom hash function f_1, the second 28 bits for f_2, and so on. Because these hash functions are strong, bits are not correlated with one another. But as a result, $k \times M$ must be less than 128 a Bloom filter built from MD5 hashes and 160 for SHA1 hashes.

Starting with this code base, we implemented a fast and configurable BF implementation in C with both C and C++ bindings. Filter vectors are stored in binary files, with the first 4096-bytes of each file containing the filter's parameters and a comment. The implementation has a simple but usable API consisting of six C functions:

- `next_bloom_create()` — Creates a Bloom filter with a specified hash size, M, k and an optional comment. The Bloom filter can reside in memory or be backed to a file.
- `nsrl_bloom_open()` — Opens a previously created Bloom filter, reading the parameters from the file.
- `nsrl_bloom_add()` — Adds a hash value to the Bloom filter.
- `nsrl_bloom_query()` — Queries the Bloom filter for the membership of the hash.
- `nsrl_bloom_set_encryption_passphrase` — Adds *string* as a cryptographic passphrase for the Bloom filter. (The hash of the string is calculated; this is used as a key of a HMAC for future adds and queries.)
- `nsrl_bloom_free()` — Frees the memory associated with the Bloom filter.

Our C implementation uses `memmap` to map the BF vector into memory for speedy lookup: in practice, this means that the computer's virtual memory subsystem pages the bit vector into memory as needed and performs no unneeded copies. If the entire BF is likely to be needed, the filter can be paged into RAM all at once in order to minimize hard drive latency due to random seeks.

Besides having a correct hash function, this new implementation dramatically faster and more memory efficient than the original `perl` version, allowing us to test it on the complete RDS.

There is also a C++ class that allows zero-overhead access to the C API.

3. Bloom filters for the RDS

Having completed our BF implementation, we proceeded to create a number of BFs that contained the SHA1 hashes from the NSRL RDS.

3.1. Building the filters

The RDS is distributed as four ISO images in ISO9660 format. Each image contains several text files. Details of RDS are available online.[19]

We downloaded the ISO images for RDS 2.19 from the NIST website, RDS 2.20 being published too late for inclusion in this paper. Each ISO consists of several text files and a ZIP file containing more text files. We processed these images with two programs: `nsrlutil.py`, a Python program which mounted the disk images as files on a Linux server, opened the compressed ZIP file, and sent the hashes to standard output; and `bloom`, a program which created a new BF using parameters provided on the command line

Memory System	Size	Cycle time	Latency	Time to access 10,000 random bits	# hash lookups per second
L1 Data Cache	32K	3 cycles	1.25 nsec	12.5 μsec	40,000,000
L2 Cache[1]	4MB	14 cycles	5.83 nsec	58.3 μsec	8,500,000
667 Mhz DDR2 SDRAM[2]	4GB	5-5-5-15	70 nsec	700 μsec	710,000
Disk	1000GB	n/a	8.5 msec	85 sec.	6

Table 1. Relative speeds to access a bit of a Bloom table or hash table stored in different memory subsystems on a modern iMac computer (2.4Ghz Intel Core 2 Duo Processor E6600). Memory latency information from [6, 12]. Disk access times are approximate, based on 8.5ms average seek time. A "hash lookup" requires accessing 20 random bits.

and then loaded the filter with hash codes read from standard input. By piping these two programs together we were able to rapidly create a large number of BF files, each with a specific set of parameters.

3.2. Accuracy and Validation

Our goal was to create RDS BFs that would be small enough to distribute on CD if necessary and to fit into the main memory of older machines or smaller PDA style devices. We arbitrarily decided to evaluate BF of sizes 32MB, 64MB, 128MB, 256MB and 512MB. Such BFs can be created with 2^{28} through 2^{32} one-bit elements ($M = 28 \ldots 32$). But how many hash functions should be applied to each element? That is, what is the optimal value of k, and is it necessary to choose the optimal value?

Given that we knew the desired sizes of our filters and RDS 2.19 has 13,147,812 unique hashes, we plugged these numbers into the optimal filter equations and discovered that a 512MB filter would require 226 hash functions for a false positive rate of 6.89×10^{-69}. Clearly, this false positive rate is far lower than needed—it is, for example, considerably smaller than the failure rate of the hard drive or electronic media that would be used to store the filter. Furthermore, there is not sufficient entropy in a 160-bit hash value to provide data for 226 hash functions, each providing 32-bits of uncorrelated output.

Limiting ourselves to the real-world requirements of the RDS requires choosing the correct parameters for k and m giving the fact that there are only 160 bits of data to divide up for the hash. Of course, the values of $k = 1$ and $m = 2^{160}$ would produce no false positives at all, since each bit in the BF would correspond to a unique hash value, but such a BF would be impossibly large. The good news is that with values of $k = 5$ and $m = 2^{32}$ we see no false positives in our sample set; these settings allow the full 160 bits of the SHA-1 value to be used ($5 \times 32 = 160$) in the BF calculation, with a theoretical false positive rate of 6.2×10^{-16}). Bloom Filters of this size can be comfortably downloaded, stored on USB memory sticks, and stored in memory of 32-bit workstations. Although we would see a similar false positive rate with $k = 4$ (and have an implementation that is mildly faster), using $k = 5$ gives our BFs the ability to accommodate additional information without a degradation in accuracy.

To validate our code, we wrote a regression program which tested each BF with a million hash values that we knew were in the RDS and a million hash values that we knew were not in the RDS. In keeping with the theory of BFs, in no case was a value that was known to be in the RDS reported to be absent from any of our filters. But also in keeping with theory, we did observe occasional false positives for lower values of k and m (Table 2) than those we recommend.

3.3. Performance

In this section we compare the performance of looking up hash values in a BF, in a sorted text file, and in the MySQL database—two systems that are commonly used by today's forensic tools for storing RDS hash codes. We also explore the impact on BF performance of adjusting the k and m parameters.

3.3.1 BF vs. hfind and MySQL

With the BF, each lookup for a match takes f operations, while a lookup for a non-match takes $1 \ldots f$ operations. Storing that same data sorted in a text file and performing a binary search consumes roughly $\log_2(n)$ operations. MySQL was configured to use InnoDB tables which is designed to deliver high performance transaction processing, with row level locking and multi-version concurrency control. Data is stored in B-trees.[13]

Tests were performed on a Red Hat FC6 server with two quad-core Xeon processors with 2MB L2 caches running at 3.2Ghz and 8GB RAM. code was compiled with `gcc` 4.1.2 and run on a 2.6.22.9-61 kernel.

RDS is distributed as four ISO images. We combined the hashes from all of these images into a single file. This RDS 2.19 file was imported into clean BF with $m = 2^{32}, k = 5$; imported to SleuthKit[4] with hfind using the command `hfind -i nsrl-sha1 flatfile.txt`; and imported into a single MySQL database located on the test machine to reduce network latency overhead affecting test results.

	m (BF size, in bits (MB))										
	2^{22} (512KB)	2^{23} (1MB)	2^{24} (2MB)	2^{25} (4MB)	2^{26} (8MB)	2^{27} (16MB)	2^{28} (32MB)	2^{29} (64MB)	2^{30} (128MB)	2^{31} (256MB)	2^{32} (512MB)
k	**Predicted number of false positives for 1 million random values:**										
1	956,486	791,401	543,274	324,184	177,920	93,314	47,799	24,192	12,170	6,081	3,045
2	996,217	914,866	626,315	295,146	105,096	31,656	8,707	2,285	585	148	37
3	999,753	973,016	740,548	330,423	87,780	16,510	2,552	355	47	6	1
4	999,986	992,448	836,980	392,271	87,111	11,045	1,002	76	5	0	0
5	999,999	998,027	904,503	467,768	95,013	8,708	484	20	1	0	0
6	1,000,000	999,506	946,760	548,411	109,179	n/a	n/a	n/a	n/a	n/a	n/a
7	1,000,000	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	
k	**Actual rate of false positives for 1 million random values:**										
1	956,483	791,511	543,165	323,971	178,416	93,558	48,042	24,448	12,373	6,213	3,156
2	996,126	914,938	625,765	295,401	105,465	31,921	8,735	2,282	582	144	48
3	999,758	972,802	740,271	330,808	88,079	16,518	2,556	378	46	3	0
4	999,989	992,175	836,379	392,427	87,606	10,956	1,049	60	4	0	0
5	999,999	997,916	904,260	467,288	95,662	8,755	479	15	0	0	0
6	1,000,000	999,463	946,587	548,083	109,982	n/a	n/a	n/a	n/a	n/a	n/a
7	1,000,000	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	

Table 2. Predicted and actual number of false positives for 1 million pseudo-random hashes queried against a BFs loaded with the 13.1 million hashes of RDS 2.19. Entries with 1,000,000 at the left of the table indicates that every hash value was a false positive; entries marked with 0 at the right of the table indicate that there were no false positives. Entries marked n/a cannot be computing because there are not sufficient number of bits in the 160-bit SHA-1. This analysis indicates that $m = 2^{32}, k = 5$ appears to be the optimal value for a Bloom filter designed to hold SHA-1 values.

We measured the time that it took to perform two sets of queries against each object. The first set was 1 million SHA1 hashes taken from RDS 2.19—hashes that were guaranteed to be in the data set. Next we measured the time that it took to perform 1 million pseudo-random queries of hashes that were known not to be in the data set. From this number we could determine the number of queries per second that each configuration delivered (Figure 1).

As expected the BFs were faster than both the binary searches through the sorted text file used by SleuthKit and MySQL's InnoDB tables. Also as expected, it is dramatically faster to lookup a hash that is not in the BF than one that is in the BF—this is because our search routine stops searching the moment it retrieves the first unset bit i' from vector F.

3.3.2 Effects of m on speed

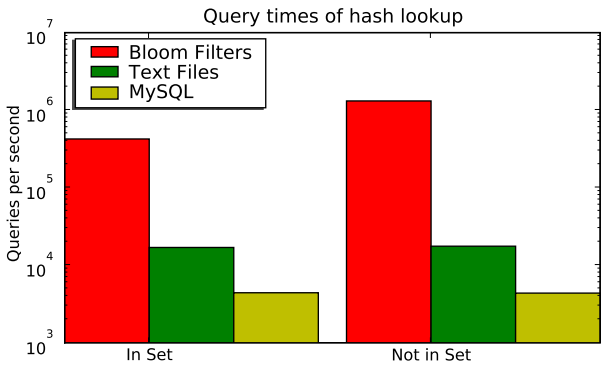
In Section 2.2 we asserted that smaller BFs would have higher performance on modern hardware due to L1 and L2 cache performance. To test this assertion we constructed multiple filters with $k = 5$ and m stepping from 2^8 to 2^{32}. We then inserted 1 million pseudorandom values into the hash and searched for each of these values.[3]

This attempting to measure the impact of the L1 and L2 cache was frustrated because these tests were done on an Internet-accessible multi-user machine running Linux: a lot of other processes were competing for the cache. On the other hand, this configuration is similar to what most practitioners will be using—a complex operating system that is running multiple tasks at once. Nevertheless, we did observe a significant decrease in the BF's lookup performance as the filter increased in size (Figure 2). The graph also shows an inflection point as the size of the graph reaches the size of the benchmark system's L2 cache, as indicated by the dashed line.

3.3.3 Effects of k on speed

In Section 3.1 we asserted that BFs with smaller numbers of hash functions (i.e. BFs with smaller k) would have higher performance due to the computational overhead of each hash function and retrieving the corresponding bit from memory. To look for this effect, we constructed multiple filters with $m = 2^{20}$ and k ranging from 1 to 5. When then followed the same procedure of inserting 1 million pseudorandom values and looking up those values. As expected, performance decreased as k increased. Indeed, the number of lookups per second is roughly proportional

[3] Searching for known values is the slowest operation for our BF implementation; searching values not to be present with a BF of $k = 5$ has the same performance as searching values not to be present in a filter of $k = 1$ but with the same constant overhead. Since we were interested in measuring the performance of the BF and not the overhead of our implementation, we used $k = 5$.



Method	In Set	Not in Set
Bloom Filters	424,808	1,314,060
SleuthKit	16,942	17,595
MySQL InnoDB	4,415	4,369

Figure 1. Queries per second for 160-bit SHA1 hashes against $M = 32, k = 5$ Bloom Filters, SleuthKit's `hfind`, and MySQL's `SELECT` statement (with InnoDB tables). "In Set" refers to hashes that are in RDS 2.19, while "Not in Set" refers to hash values that are not in RDS 2.19.

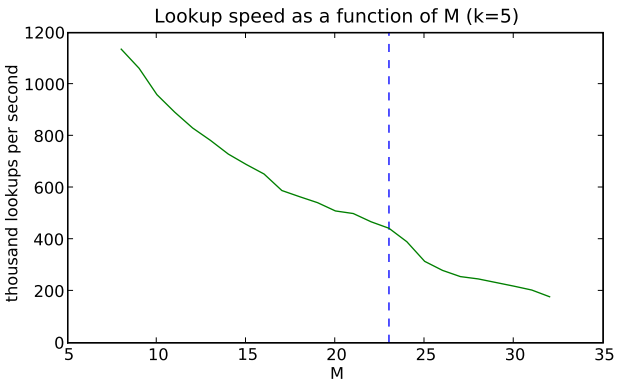


Figure 2. Increasing the size of the Bloom filter decreases its speed due to caching issues. The dashed line indicates the 2 MB (8 Mbit) size of the benchmark system's L2 cache. (Note: these speeds are for successful lookups; the speed of looking up hash values not in the filter are roughly $12\times$ faster.)

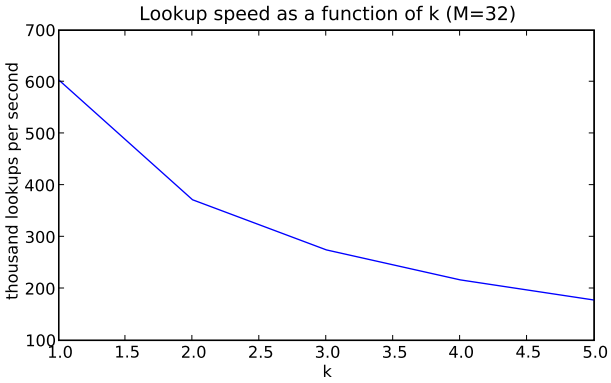


Figure 3. Increasing the number of bits per element in the Bloom filter decreases its speed, since more work needs to be done to look up each element.

to $\frac{1}{k}$ (Figure 3).

3.4. Batch forensics with `fiwalk`

We have incorporated our BF implementation into `fiwalk`, an open source batch disk forensic analysis program. `fiwalk` uses Carrier's SleuthKit to perform a batch analysis and extraction of allocated and deleted files from all of the partitions resident in a disk image that is to be analyzed. `fiwalk`'s output is a *walk file* that includes a list of every file, the file's metadata, and optionally the file's MD5 or SHA1 cryptographic hash.

The current version of `fiwalk` allows files to be specified by their name or extension. We modified `fiwalk` to allow the use of BF for inclusion or exclusion as well. We further modified `fiwalk` so that it can generate a BF based on the files that it finds in a disk image.

3.5. RDS Coverage of Windows

We created VMWare machines Windows 2000 Service Pack 4, Windows XP Service Pack 2 and Windows Vista Business. The `.vmdk` files were converted to raw files with `qemu-img`[1] and processed with `fiwalk` to create "walk" files containing all files on the image and the SHA1 hashes of the files. These walk files were then compared against our baseline RDS v2.19 bloom filter. Next we patched all of the virtual machines with the latest hot fixes from Microsoft Update and reprocessed the VMs. The percentage of files in the virtual machines that appeared in RDS 2.19 are presented in Table 3.

These results yield several interesting data points. First, even on base operating system installs with hot fixes installed, only 60–70% of files are covered by the RDS. Why not more? Some files are unique per system, a result of hardware signatures, chargeable registration keys, and usernames. Swap files will invariably differ between machines. It also shows the amount of updates that Microsoft has issued since the RDS 2.19 was released in December, 2007.

Table 3 shows an analysis of the files that were present

PNF files in the WINDOWS/inf directory	707
Windows PC Health Offline Cache files	321
VMWare Tools installation files[a]	130
Start Menu links	95
Other Windows System files	77
Miscellaneous system log files	69
Windows wbem autorecover files	53
Other PC Health files	40
Windows System Restore Files	38
Other Documents and Settings files	41
Windows Prefetch files	31
Miscellaneous system text files	15
File system metadata files	8
Other system shortcuts	7

[a] Artifact of VMWare; not part of the Windows XP base release.

Table 3. Breakdown of the 1635 files in the Windows XP base installation which were not present in NSRL RDS.

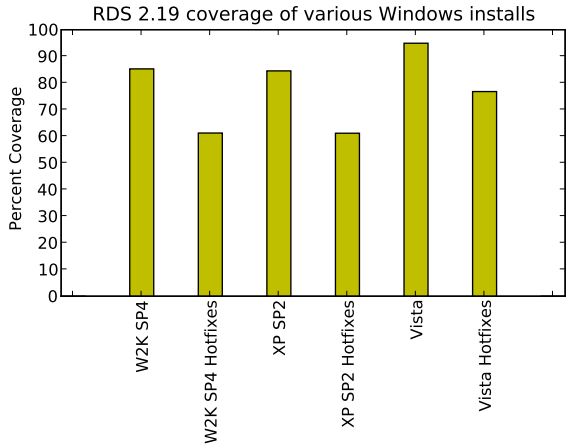


Figure 4. RDS 2.19 coverage of Windows installations

in our base installation of Microsoft Windows XP but which were not in the RDS. The majority of them are files resulting from the installation of software during installation or logfiles resulting from actually running the system.

Overall, the number of files *not* covered by RDS in these virtual machines is a cause for concern: although many files are removed for the potential examiner, a significant number remain. If the primary purpose for RDS is to eliminate known good files so that they do not need to be examined, then it is not delivering on this promise.

3.6. Coverage of Real Data

To evaluate the RDS against real-world data, we performed file system walks of 891 hard drives purchased on the secondary market between 1998 and 2006. Overall, 45 drives had greater than 80% coverage by RDS, 33 which contained a significant number of files. For the 280 drives with > 100 files, RDS covered on average 36.64% of files. For the 186 drives with > 5000 files, RDS coverage averaged 36.62%. Once again, RDS helps reduce what must be consulted, but not by much.

3.7. Profiling Hard Drives

Although the main use of RDS today is to eliminate "known goods" from analysis, we believe that the coverage of RDS that we have seen limits this use. An alternative use of this resource is to use the RDS metadata to profile uses that a hard drive may have had.

We have created an application which attempts to profile a hard drive by looking up each file's SHA1 in the RDS database and retrieving a list of all the RDS objects with a matching hash value, and then retrieving a list of the product names as identified by NIST. Each hash code may appear multiple times in the RDS, each appearance corresponding to a different product. In the cases where only a single product name is matched, that product name is added to a list.

Using RDS in this manner allows us to get a rapid handle on the kind of software that is present on the hard drive even in cases where the programs themselves cannot be recovered because of file deletion.

4. Attacks on Bloom Filter Distributions

There is on significant problem with distributing a hash set as a Bloom Filter: it is dramatically easier for an attacker to construct a hash collision within the BF than it is to find a hash collision with a collision-resistant function such as SHA-1. With collisions easier to find, an attacker could use this approach as a way of hiding contraband data from forensic tools that use BFs to eliminate "known goods."

Assuming that SHA-1 is a strong hash function, the only way to find a hash collision is by a brute force attack—with odds of 14 million out of 2^{160} (assuming 14 million unique hashes in the RDS). Using a hash collision to hide contraband data, then, would require appending a block of data to the contraband and then making small, incremental changes to that block of data until the hash collision is found. In practice, a collision will never be found with a brute force approach such as this.

However, if those 160 bits are divided into 5 groups of 32 and stored in a BF with $m = 2^{32}$ and $k = 5$, finding a collision becomes much easier. The 14 million hash codes represent a maximum of $14 \times 5 = 70$ million distinct 32 bit codes, all stored in the same filter. Finding a false positive for $k = 5$ requires finding a single hash for which each of its 5 groups of 32 bits are set in that filter. The probability for each group of 32 is 70 million : 2^{32} or

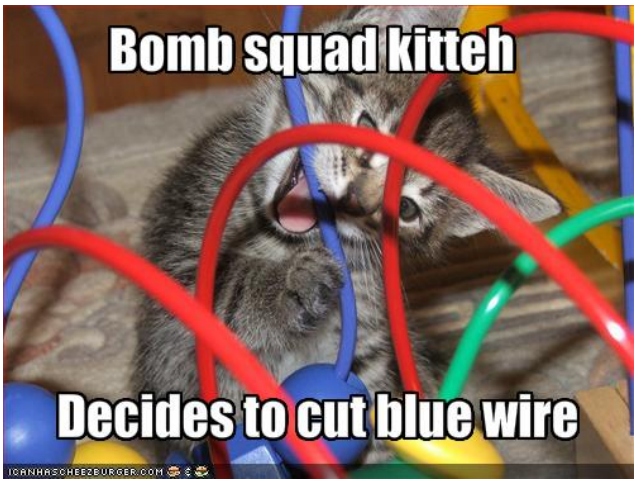


Figure 5. A test image for creating Bloom filter false positives.

$p = 0.016$. Finding all five together is $p = (0.016)^5$, or roughly 2^{-30}, which makes the difficulty of finding such a collision roughly equal to the task of cracking a 30-bit encryption key.

We tested this hypothesis by taking a JPEG of a kitten (Figure 5), appending a binary counter, computing the SHA-1 and checking for a false positive. If no false positive was found we incremented the counter and repeated. We found a collision with the $M = 32$ $k = 4$ BF after 110,223,107 iterations by appending the hex bytes `03 df 91 06`. Total computation time to find the alias was roughly 5.5 CPU hours. Details are in Table 4.

The only defense against this attack is to use an encrypted bloom filter—for example, by hashing each 160-bit SHA-1 with a 160-bit random key that is kept secret. Since the adversary does not know the key, she cannot construct an alias. Unfortunately, the key must be kept secret, and the BF cannot be used without it.

In practice, this means that while BFs are a useful tool for distributing hash sets within an organization, publishing the BFs in a public forum makes those BFs unsuitable for use if there is an adversary who might wish to hide data by creating false positives.

The permutation attack is also a useful defense against the use of BFs for finding *known bads* or for performing cross-drive analysis, but that attack is also useful against traditional hash analysis as well. That is, making minor changes to hacker tools or contraband content necessarily changes the hash value of these files. Therefore, using BFs for finding *known bads* does not introduce any more vulnerabilities than using traditional hash tools, but does make the searches dramatically faster.

5. Implications

This section explores a variety of forensic applications that we are developing based on the BF implementation presented in the previous section.

5.1. Watch Lists

BFs can be used to store hashes of any kind. In particular, they can store hashes of features[10] extracted from files or bulk data.

We are developing a bulk extraction application which extracts features, computes the HMAC using a selectable secret key, and stores the results in a BF. This program can be applied to list of email addresses, credit card numbers, or other kinds of pseudounique information to create watch list filters. These filters can be taken into the field and used to triage suspect hard drives while minimizing the risk that the features in the filter will be compromised.

5.2. Cross-Drive Analysis

Boolean operations can be applied directly to BFs. This allows BFs to be used for cross drive subsection. The procedure is straightforward. First, a BF containing the hashes of the extracted features is created for each drive in the corpus. A threshhold is set for the maximum number of drives for which a filter will be considered (a threshhold of $\frac{n}{3}$ where n is the number of hard drives in the corpus is a good starting point). Next, a threshold vector $\bar{F}$ of integers is created where the size of the vector is equal to the number of bits in the BFs. Each BF is scanned; for every bit i that is encountered, the corresponding integer in $\bar{F}$ is incremented. At the conclusion of the first pass, all integers in $\bar{F}$ that are larger than the threshold are zeroed. Those that remain are used as a filter of relevant pseudounique features. A correlation vector $\bar{F}_{i,j}$ can now be constructed for each (i, j) pair of drives by computing $\bar{F} + F_i + F_j$.

5.3. Segmenting RDS

File hashes present a useful way of eliminating known files that are unlikely to be modified from the set of files of interest to a forensic investigator. However, a set of known files of interest can provide much more useful information, though it can be harder to find.

Instead of having a single BF for all of RDS, an alternative is to divide the data set into smaller sets: one containing Windows installation files, one containing files from common desktop application, one for video production applications, and so on.

Dividing RDS in half and storing each in its own BF that is half the size significantly decreases the false positive rate, since there are fewer opportunities for aliasing. Although the time to search the BFs increases because each BF must be searched sequentially, the advantage is that the BFs can be used to characterize the files beyond simply known/unknown. This is equivialent to using a Bloomier filter[5].

Original SHA1:	df7ce34d	f723ae1a	675cd06f	e202d060	bd82bd9b
SHA1 of modified file:	cb6b989b	97ad04fb	eaa0ef99	4b8a4059	f2d51dc6
SHA1 of "Index.htm" from "WIN"	C076275E	694CC871	C9624246	CB6B989B	BFFEA55F
SHA1 of "MEMLABEL.PCT" from "Mac"	B961495D	3CDCC1C6	97AD04FB	4CF2CFFE	1BDA3025
SHA1 of "1TXT047.gif" from "WIN2000"	EAA0EF99	B62CD185	3A9DD81A	1FF458C3	734767DF
SHA1 of "H8499.GIF" from "Gen"	4B8A4059	19B1E394	85B7B439	E3A0B940	AD65865F

Table 4. Results of a brute force attack against Bloom filter with M=32 and k=4. The first line shows the SHA1 of the original JPEG (Figure 5); The second line shows the SHA1 of the file modified by appending the hex bytes 03 DF 91 06. The remaining lines show the SHA1 of the specific files names and application distributions within RDS 2.19 that contributed to the false-positive, with the specific aliases boxed.

5.4. Prefiltering with Bloom Filters

An alternative to segmenting the RDS into multiple BFs is to use the BFs in conjunction with a slower lookup service that returns additional information. That is, instead of viewing the RDS BF as an alternative to storing the RDS in a MySQL database, the BF can be used as an *accelerator* for database: hashes that are to be looked up can first be checked against the BF and, if the hash is in the RDS, then the MySQL database can be consulted to obtain the additional metadata.

To this end, our MySQL schema stores significantly more information for each hash: we store *all* of the information distributed with the RDS, including file name, size, operating system ID, application ID, language, and RDS release. This information is stored in structured many-to-one and many-to-many SQL tables. The information can be accessed directly using a MySQL connector or through a web-based XMLRPC server.

6. Conclusions

Validating previous work, we find that BFs are good tools for performing high-speed matches against hash sets. However, we learned that BFs are not a good tool for distributing hash sets of "known goods" if the adversary can be reasonably expected to get access to the filter or the filter's parameters, because it is relatively easy for an adversary to modify hostile content to create a false positive. The way to defend against this attack is to create bloom filters in the field with a randomly chosen cryptographic key.

By testing RDS with a variety of different BF parameters, we found that a filter with 2^32 one-bit elements (512MB in size) with 5 hash functions produced excellent performance. We have made our BF implementation freely available for download; the implementation is Public Domain and can be freely used or modified by anyone for any purpose. Finally, we have shown how BFs can be used to build secure watch lists and for cross drive analysis.

6.1. Availability

All of the programs discussed in this paper are distributed in source code form and build with GNU build tools. We have tested them on MacOS 10.5, Linux, and FreeBSD. The code may be downloaded from our web server at `http://www.afflib.org/`.

6.2. Future Work

We are in the process of evaluating the use of BFs for hash sets of individual file blocks or disk sectors.

We are modifying our web-based hash lookup service so that new hashes can be automatically submitted by members of the community; we hope to implement a reputation system and voting algorithm to prevent database poisoning. We may further modify the system so that users will be able to download BFs constructed "on the fly" to match specific SQL queries based on the RDS (e.g., a BF that matches the files that were part of a specific application program).

We are working on a new release of our BF code that will be able to directly open BFs that have been compressed with the ZIP or GZIP algorithms. Given that Java has memory-mapped files and there are persistent reports that well-written Java code can out preform the equivalent C, we are also writing a compatible Java implementation of our BF code.

6.3. Acknowledgments

The authors wish to express their thanks to Brian Carrier, Jesse D. Kornblum, Beth Rosenberg and Vassil Roussev, all of whom have provided useful feedback on this research and this paper. Thanks also to the anonymous reviewers who saw and corrected significant errors in a previous version of this paper and recommended exploring the possibility of false-positive attacks.

This research was supported in part by the Naval Postgraduate School's Research Initiation Program. The views expressed in this report are those of the author and do not nnecessarily reflect the official policy or position of the Department of Defense, the National Institute of Standards and Technology, or the U.S. Government.

References

[1] Fabrice Bellard. Qemu: Open source processor emulator, 2008. `http://bellard.org/qemu`.

[2] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. ISSN 0001-0782.

[3] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1 (4):485–509, May 2004.

[4] Brian Carrier. The Sleuth Kit & Autopsy: Forensics tools for Linux and other Unixes, 2005. `http://www.sleuthkit.org/`.

[5] Bernard Chazelle, Joe Kilian, and Ronitt Rubinfeld. The blomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 30–39, 2004.

[6] Franck Delattre and Marc Prieur. Intel core 2 duo – test. July 4 2006. `http://www.behardware.com/articles/623-6/intel-core-2-duo-test.html`.

[7] Peter J. Denning and Stuart C. Schwartz. Properties of the working-set model. *Commun. ACM*, 15(3):191–198, 1972. ISSN 0001-0782.

[8] Peter C. Dillinger and Panagiotis Manolios. Bloom filters in probabilistic verification. In *Formal Methods in Computer-Aided Design*. Springer-Verlag, 2004. `http://www.cc.gatech.edu/fac/Pete.Manolios/research/bloom-filters-verification.html`.

[9] Li Fan, Pei Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8:281–293, June 2000.

[10] Simson Garfinkel. Forensic feature extraction and cross-drive analysis. In *Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS)*. Lafayette, Indiana, August 2006. `http://www.dfrws.org/2006/proceedings/10-Garfinkel.pdf`.

[11] Guidance Software, Inc. EnCase Forensic, 2007. `http://www.guidancesoftware.com/products/ef_index.asp`.

[12] William Henning. Intel core 2 duo e6600 review. *Neoseeker*, September 19 2006. `http://www.neoseeker.com/Articles/Hardware/Reviews/core2duo_e6600/6.html`.

[13] Ken Jacobs and Keikki Tuuri. Innodb: Architecture, features, and latest enhancements. In *MySQL Users Conference 2006*, 2006. `http://www.innodb.com/wp/wp-content/uploads/2007/04/innodb-overview-mysql-uc-2006-pdf.pdf`.

[14] Panagiotis Manolios. Bloom filter calculator, 2004. `http://www.cc.gatech.edu/~manolios/bloom-filters/calculator.html`.

[15] Michael Mitzenmacher. Compressed bloom filters. pages 144–150, 2001.

[16] James K. Mullin. A second look at bloom filters. *Commun. ACM*, 26(8):570–571, 1983. ISSN 0001-0782.

[17] Vassil Roussev, Yixin Chen, Timothy Bourg, and Golden G. Richard III. *md5bloom*: Forensic filesystem hashing revisited. *Digital Investigation*, 3(Supplement-1):82–90, 2006.

[18] Vassil Roussev, Golden G. Richard III, and Lodovico Marziale. Multi-resolution similarity hashing. *Digital Investigation*, 4(Supplement-1):105–113, 2007.

[19] Douglas White. NIST national software reference library (NSRL), September 2005. `http://www.nsrl.nist.gov/documents/htcia050928.pdf`.

[20] Douglas White, August 17 2006. `http://www.nsrl.nist.gov/RDS/rds_2.13/bloom/`.