2012-12

Distinct Sector Hashes for Target File Detection

Young, Joel; Foster, Kristina; Garfinkel, Simson; Fairbanks, Kevin

IEEE Computer, December 2012
https://hdl.handle.net/10945/44287

Distinct Sector Hashes for Target File Detection

Joel Young, Kristina Foster, and Simson Garfinkel, *Naval Postgraduate School*
Kevin Fairbanks, *Johns Hopkins University*

Using an alternative approach to traditional file hashing, digital forensic investigators can hash individually sampled subject drives on sector boundaries and then check these hashes against a prebuilt database, making it possible to process raw media without reference to the underlying file system.

Forensic examiners frequently search disk drives, cell phones, and even network flows to determine if specific known content is present. For example, a corporate security officer might examine a suspicious employee's laptop for unauthorized documents; law enforcement officers might search a suspect's home computer for illegal pornography; and network analysts might reconstruct Transmission Control Protocol streams to determine if malware was downloaded. In these and many other cases, examiners typically identify files by computing their cryptographic hash—often with MD5 or SHA1 hash algorithms—and then searching a database for the resulting hash value.

Use of hash values for file identification is pervasive in digital forensics—every popular forensics package has built-in support. One of the most widely used databases is the National Software Reference Library (NSRL) Reference Data Set (RDS). Version 2.36, released in March 2012, contains 25,892,924 distinct file hashes (www.nsrl.nist.gov). Other databases are available to customers of specific companies and to law enforcement organizations.

There are many limitations when using file hashes to identify known content. Because changing just a single bit of a file changes its hash, pornographers, malware authors, and other miscreants can evade detection simply by changing a comma to a period or appending a few random bytes to a file. Likewise, hash-based identification will not work if sections of the file are damaged or otherwise unrecoverable. This is especially a problem when large video files are deleted and the operating system reuses a few sectors for other purposes: most of the video is still present on the drive, but recovered video segments will not appear in a database of file hashes.

SECTOR HASHING

We are developing alternative systems for detecting target files in large disk images using cryptographic hashes on sectors of data rather than entire files. Modern file systems align the start of most files with the beginning of a disk sector. Thus, when a megabyte-sized video is stored on a modern hard drive, the first 4 kibibytes are stored in one disk sector, the second 4 KiBytes are stored in another disk sector, typically the adjacent one, and so on. (In our work, we distinguish between power-of-two-based sizes of digital artifacts, such as kibibytes, and power-of-ten-based sizes, such as kilobytes. See the "Decimal versus Binary Prefixes" sidebar for more details.) Furthermore, by sampling randomly chosen sectors from the drive, it is only necessary to read a tiny fraction of the drive to determine with high probability if a target file is present. This enables rapid triage of drive images.

We compare drive sector hashes to a hash database of fixed-sized file fragments, which we call *blocks*. The terms "sector" and "block" are often used incorrectly as syn-

 Published by the IEEE Computer Society

DECIMAL VERSUS BINARY PREFIXES

Today there are two standards for representing sizes of files, storage systems, and memory banks: SI (International System of Units) decimal prefixes and IEC (International Electro-technical Commission) binary prefixes. SI decimal prefixes are commonly used to represent metric quantities. For example, the SI prefix giga- multiplies the value that follows by 10^9; thus, a gigabyte (Gbyte) is 10^9 = 1,000,000,000 bytes. In contrast, the IEC prefix gibi- multiplies the value that follows by 2^{30}; a gibibyte (GiByte) is thus 2^{30} = 1,073,741,824 bytes.

The confusion over prefixes dates back to the early days of computing, when K and M meant 1,024 and 1,048,576 when describing memory systems but 1,000 and 1,000,000 when describing storage systems. The difference in terminology resulted from the way that these systems were addressed. Memory was addressed by a series of binary lines, while electromechanical drums and disks were addressed by specifying a head, a track, and sector numbers: such numbers only map to even powers of two when the number of heads, tracks, and sectors are also even powers of two, and this is rarely the case due to manufacturing concerns.

For much of computing history, the fact that 1K sometimes meant 1,000 and sometimes 1,024 was not a major problem, as the correct size could be inferred from context and, in any event, the difference between 1,000 and 1,024 is not that great. However, the distinction became an issue in the 1990s as memory capacity mushroomed and commonly used prefixes went from Ks to Ms and then Gs, resulting in a larger divergence between the power-of-two measurement and the corresponding power-of-ten measurement. The IEC accordingly proposed binary prefixes in 1996 and standardized their use in 1999. In 2008, the International Organization for Standardization adopted the IEC standard with the addition of prefixes for describing exbi- (2^{60}), zebi- (2^{70}), and yobi- (2^{80}) byte quantities.

Despite this standardization effort, we live in a world in which 4-Gbyte memory sticks sold as system RAM can store 4,294,967,296 bytes of data but 4-Gbyte microSD (Secure Digital) cards for cell phones are only warranted to store 4,000,000,000 bytes of data. However, since those 4 billion bytes are organized in 512-byte logical sectors, the microSD card typically stores 7,812,500 (or more) sectors, a number that does not make much sense technically but makes a great deal of sense when the design of flash-based storage systems is considered. That is, flash systems contain more physical memory than they advertise, with the system removing bad blocks from service as the device ages. Thus, a "4-Gbyte" microSD card might actually have 8 million or even 9 million physical sectors, but those extra physical sectors are invisible to the operating system.

We expect use of IEC binary prefixes to increase with time. We use them here to describe block size and sector size, as they are typically multiples of 512 (2^9). We use SI decimal prefixes to describe disk sizes, since that is the way they are sold by manufacturers.

onyms. For clarity, we use "sector" and "block" to refer to chunks of data extracted from drive images and files or file systems, respectively. Our approach depends on the existence of file blocks that only occur in a single distinct file. Experiments show that such *distinct blocks* comprise the vast majority of both executable files and user-generated content. Matches against block hashes shown not to occur elsewhere are strong evidence that a corresponding target file is or was present.

As the "Previous Work" sidebar describes, little work has been done on the use of sector hashes for file identification. However, sector hashing has numerous advantages over file hashing in forensic analysis. In many cases, using sector hashing with full media analysis—comparing every sector of the drive to an appropriate database—can detect a single block from a file that was once present. Alternatively, sector hashing can be combined with random sampling, making it possible to scan a terabyte-sized drive for the presence of select data in just a few minutes.

While sector hashing offers advantages when used for file detection in a forensic context, it also presents technical difficulties.

BLOCK SIZE AND HASH ALGORITHM

Two important design choices for using sector hashes are the block size and the hash algorithm.

Clearly, the block size must be small enough so that file blocks will align with drive sectors. The easiest way to assure this is to use a block size of 512 bytes, the sector size of most mass storage systems from the 1970s until quite recently. When presented with a device that has a larger sector size—for example, 2 KiBytes in CD-ROMs or 4 KiBytes in modern drives—the sectors could be divided into 512-byte blocks and hashed accordingly.

However, 512 bytes might be smaller than necessary. Many file systems use a 4-KiByte allocation size (NTFS has a default cluster size of 4 KiBytes for drives smaller than 16 Tbytes). In addition, using a 4-KiByte block size would reduce the hash value database's size by a factor of eight. The danger with a 4-KiByte block size is that a file system with a 4-KiByte allocation size might be used to write to a device with 512-byte sectors. If the blocks are not aligned on an eight-sector boundary, there is a risk that each set of eight sectors hashed would contain part of one block and part of another. The result is that no distinct blocks would be found.

This problem can be avoided in devices with 512-byte sectors by reading 15 sectors at a time, producing eight hashes: the first from sectors 0-7, the second from sectors 1-8, and so on. While multiple hashing does increase the computational costs of both hashing and database operations, the need for such hashing will decrease over time as 512-byte-sector devices are phased out of use.

We chose the MD5 hash algorithm, which is widely used within the forensic community and computationally fast. Although MD5 is no longer collision resistant, our technique relies on using hashes to match adversary data to target content—in fact, collisions actually facilitate the process.

PREVIOUS WORK

While at the US Department of Defense Cyber Crime Center, Nick Harbour developed the dcfldd disk imaging tool (http://dcfldd.sourceforge.net), based on GNU dd, that would compute a hash on a disk image as it was created. Harbour subsequently modified dcfldd to compute hashes over segments of the disk image so that if it was inadvertently modified, a chain of custody could be maintained for at least part of the image. He called this *piecewise hashing*. Jesse Kornblum's md5deep (http://md5deep.sourceforge.net) extended piecewise hashing to multiple files.

As part of his solution to the 2006 Digital Forensics Research Workshop (DFRWS) Data Carving Challenge, Simson Garfinkel introduced a new technique dubbed "the MD5 trick."[1] After finding the original challenge documents based on text fragments from the challenge description, Garfinkel computed the MD5 hash of 512-byte file blocks and searched the challenge drive for matching 512-byte sectors. Using this technique, he identified all of the challenge files including a fragmented Microsoft Word file.

Three years later, Naval Postgraduate School researchers released frag_find, a tool that automates this process.[2] Sylvain Collange and colleagues called this approach *hash-based data carving*[3] and explored the use of GPUs to speed the hashing load. They found that, with a powerful enough GPU, it is possible to simultaneously hash a block of data on subsector boundaries—for example, 1,024 bytes of data can be hashed in 512-byte chunks on 4-byte boundaries, creating 128 distinct hash values—although doing so dramatically increases pressure on the database.

In 2009, Simon Key developed the File Block Hash Map Analysis (FBHMA) EnScript, a dual-purpose tool that creates a hash map of file blocks from a master file list and searches selected areas of a target drive for the blocks.[4] Like frag_find, however, FBHMA EnScript does not support billion-block hash databases or sufficiently fast lookup speeds to use sector hashing in full media analysis or random sampling.

References

1. S.L. Garfinkel, "DFRWS 2006 Challenge Report," 2006; http://sandbox.dfrws.org/2006/garfinkel/part1.txt.
2. S. Garfinkel, "Announcing frag_find: Finding File Fragments in Disk Images Using Sector Hashing," 1 Mar. 2009; http://tech.groups.yahoo.com/group/linux_forensics/message/3063.
3. S. Collange et al., "Using Graphics Processors for Parallelizing Hash-Based Data Carving," *Proc. 42nd Hawaii Int'l Conf. System Sciences* (HICSS 09), IEEE CS; http://hal.archives-ouvertes.fr/docs/00/35/09/62/PDF/ColDanDauDef09.pdf.
4. S. Key, "File Identification and Recovery Using Block Based Hash Analysis," lab presented at the annual Computer Enterprise and Investigations Conf. (CEIC), 2012; www.ceicconference.com/AJAX/courseScheduleLightbox.aspx?id=1000018721.

UNDERSTANDING DISTINCT BLOCKS

Identifying files with sector hashes relies on the presence of distinct file blocks. A distinct block is one that does not occur anywhere more than once except as a block in a copy of the original file. Using distinct blocks as a forensic tool leverages two hypotheses:

- if a block of data from a file is distinct, then a copy of that block found on a data storage device is evidence that the file is or was once present; and
- if the blocks of that file are shown to be distinct with respect to a large and representative corpus, then those blocks can be treated as if they are universally distinct.[1]

The first hypothesis is trivially true if we could know that a particular block is indeed distinct. Unfortunately, it is impossible to know this. On the other hand, we can determine the frequency of blocks in large collections of real files.

The frequency of distinct blocks

We counted the number of blocks in several million-file corpora that occurred once, twice, or more frequently. We call these *singleton*, *paired*, and *common* blocks, respectively. If paired and common blocks are extremely unusual, then it is reasonable to believe that singleton blocks are indeed universally distinct. Also, by examining the context of paired and common blocks, we might understand the root causes of their nondistinctness: a common method used to generate the data, an extrinsic process that created similar files, or some other kind of data-sharing mechanism.

For these experiments, we used three corpora modified to remove all duplicate files:

- Govdocs, a collection of 974,741 freely redistributable files downloaded from US government webservers (average file size: 493 KiBytes);[2]
- OpenMalware 2012, a collection of 2,998,898 malware samples (average file size: 417 KiBytes);[3] and
- the 2009 NSRL RDS, a set of 12,236,979 block hashes for a collection of known, traceable software applications (average file size: 235 KiBytes).

To our knowledge, no previous studies have analyzed the co-occurrence of blocks across such a large number of files and file types. Using these corpora let us make some general conclusions about the frequency of distinct blocks.

We analyzed each corpus using both 512-byte and 4-KiByte blocks—the sector size of older and modern hard drives, respectively—except in the case of the 2009 NSRL RDS, for which 512-byte block hashes were not yet available. We also compared OpenMalware 2012 to the 2009 NSRL RDS to find the most common blocks across legitimate and malicious executables. Table 1 lists the incidence of singletons, pairs, and common sectors in the three corpora.

The vast majority of blocks in the corpora correspond to single, specific files. This is not surprising given that high entropy data approximates a random function. A truly random 512-byte block contains 4,096 bits of entropy. There are thus $2^{4,096} \approx 10^{1,200}$ possible different blocks, and all are equally probable. It is inconceivable that two

randomly generated blocks would have the same content. The randomness of user-generated content is less than 8 bits per byte, of course, but even for content that has entropy of 2 bits per byte, a 512-byte block still contains 1,024 bits of entropy, again making it very unlikely that two blocks will be the same.

As Table 1 shows, all kinds of user-generated content, including word processing files, photos, and video, contain sectors that are not seen elsewhere—that is, distinct blocks according to our definition. The frequency of distinct blocks in the OpenMalware 2012 and 2009 NSRL RDS datasets is significantly lower but still quite high. However, our experiments make it clear that it is impossible to assume a priori that a given singleton block is distinct.

Table 1. Incidence of singleton, paired, and common sectors in three file corpora.

No. of blocks	Govdocs	OpenMalware 2012	2009 NSRL RDS
Block size: 512 bytes			
Singleton	911.4 M (98.93%)	1,063.1 M (88.69%)	N/A
Pair	7.1 M (.77%)	75.5 M (6.30%)	N/A
Common	2.7 M (.29%)	60.0 M (5.01%)	N/A
Block size: 4 kibibytes			
Singleton	117.2 M (99.46%)	143.8 M (89.51%)	567.0 M (96.00%)
Pair	0.5 M (.44%)	9.3 M (5.79%)	16.4 M (2.79%)
Common	0.1 M (.11%)	7.6 M (4.71%)	7.1 M (1.21%)

Origin of nondistinct blocks

To better understand the root causes of nondistinct blocks, we analyzed the most common blocks from each corpus. Our original intuition was that blocks that had low entropy or that contained repeating byte patterns would occur frequently. We found that many of the common blocks indeed had these characteristics.

As expected, the block of all NUL (0×00) bytes was the most common block across all corpora. But we found other examples as well. For instance, there were more than 200,000 occurrences of an Adobe PDF internal data structure in the Govdocs corpus. Likewise, we found several common blocks that contained Microsoft Office internal structures.

Several high-entropy blocks were common in the OpenMalware 2012 dataset. We found that these blocks occurred in different files but always at the same byte offset. Further analysis revealed that the containing files were actually different variants of the same malware, as reported by several antivirus tools on VirusTotal.com. The repeated blocks did not appear in any legitimate files listed in the 2009 NSRL RDS corpus. Clearly, these blocks are unique to a specific malware family and not general executables or other system files.

Although traditional file identification techniques require each variant's hash, our findings show that shared blocks can identify some malware variants. We suspect that these common malware blocks are the result of hand-patching existing malware and code reuse, or elementary attempts to change a file hash by adding bytes to the end of the file.

BLOCK HASH DATABASE

To develop a useful system for performing sector analysis, it is not enough to choose which or what size blocks should be used to capture a target dataset. It is necessary to, first, efficiently store the hashes for the target blocks and, second, check quickly enough to determine whether disk sectors are present in the dataset.

Performance requirements

Our goal is to create a database of one billion file block hashes that can be field deployed on a laptop. The database should be fast enough to support searches of hashes that are created by reading a consumer hard drive at the maximum I/O transfer rate (assuming that hashing is free). Given that it takes approximately 200 minutes to read the contents of a Tbyte-size hard drive, this translates to a database that can perform roughly 150,000 hash lookups per second. With a billion 512-byte block hashes, the database would allow identification of 512 gigabytes of known content, a number that is sufficient for many applications. Because hash values are evenly distributed, the database can be trivially parallelized using prefix routing.[4] A cluster with 1,000 such databases could thus support 10^{12} block hashes and address half a petabyte of known content.

Instead of hashing every sector of the drive, it is possible to conduct an exhaustive investigation sampling only one million randomly chosen sectors. Although the sample contains only 0.05 percent of the drive, there is a 98.17 percent chance of detecting 4 Mbytes of known content, provided that each of those 8,000 blocks is in the database.

This is an instance of the well-known "urn problem" in statistics, which describes the probability of pulling some number of red beans out of an urn that contains a mix of randomly distributed red and black beans. In this case, the red beans are distinct sectors, there are 8,000 (C) of them distributed randomly, there are two billion beans in total (N), and one million (n) are selected randomly. The probability p of not finding even a single red bean in n draws is

$$p = 1 - \prod_{i-1}^{n} \frac{((N-(i-1))-C)}{(N-(i-1))}.$$

Applying this equation to 500,000 and 250,000 randomly selected sectors, we find that the chance of detecting

4 Mbytes of known content, provided each of the 8,000 blocks is in the database, is 86.47 percent and 63.21 percent, respectively.

Note that the 4 Mbytes might be a single high-resolution JPEG or 40 medium-resolution JPEGs—the key issue is that there are 8,000 distinct blocks stored on sectors of the drive, and each random choice represents another chance to find one of them. Furthermore, because each sample is random, the distribution of the sectors on the drive is irrelevant—the chance of finding them with a random search is the same whether they are randomly distributed or clustered in a single location.

A 7,200-rpm hard drive can perform approximately 300 seeks per second. If the million randomly chosen sectors are sorted in advance, most systems could read all of them in 30 minutes; it is possible to read more data in the same time by increasing the read size to 8, 64, or even 128 sectors, although the statistical calculation becomes more complicated because many of the samples are now correlated, not strictly random. Thus, for the random sampling application, a database lookup of a few thousand transactions per second might be sufficient.

Our goal is to create a database of one billion file block hashes that can be field deployed on a laptop.

Designing the database

Neither conventional SQL databases such as MySQL, PostgreSQL, and SQLite nor NoSQL databases such as MongoDB have sufficient performance to support even high-speed random sampling. Using recent versions of each database on a Dell R510 server equipped with Dual Xeon E5620 2.4-GHz processors (each with 16 cores, a 12-mibibite cache, and 128-gibibite main memory), we got less than 1,000 lookups per second for databases containing one billion hashes.

To achieve better performance, we created our own purpose-built key-value pair store, where the key is a cryptographic block hash and the value identifies the source file and offset. We tested various custom-built solutions using hash maps, B-trees, red/black trees, and sorted vectors. In keeping with our goal for field deployment, the database is precomputed, finalized, and distributed to the client as a single file.

When looking for known content, we expect few of the sector hashes from a subject drive to actually be present in the database. We leverage this by checking a Bloom filter[5] before checking the database. Bloom filters facilitate efficient probabilistic set-membership checking with a zero false-negative rate and a false-positive rate dependent on the filter's parameters—the number of bits used in each hash (M) and the number of hash functions used (k).

When storing an item in the Bloom filter, we first hash the item k times, yielding k M-bit integers. We then set the corresponding bits in the filter. To test membership, we repeat the process, but instead of setting the bits, we check them, and if one or more bits are not set, the item cannot be in the filter. Note, however, that if all k bits are set, the item might or might not be present, as the bits might be aliases set for other items.

As we are storing the 128-bit MD5 hash values for the block, we do not need to compute k new hashes, but instead can partition the MD5 hash into M bit chunks. The resulting Bloom filter consumes 2^M bits or $2^M/8 = 2^{M-3}$ bytes. When $M = 32$, for example, the result is only $k = 128/32 = 4$ hashes, and the Bloom filter occupies 512 MiBytes of disk space. The theoretical false-positive rate of such a filter with a billion items is 13.48 percent, approximated by $P_{fp} = (1 - e^{-kn/m})^k$. Doubling the size of the filter lowers the false-positive rate to 1.92 percent.[6]

One typically implements red/black trees, flat maps (essentially sorted vectors), and hash maps as in-memory data structures. To achieve persistence, we developed a data structure based on the `boost::interprocess` library, which allows transparently placing Boost C++ container implementations into memory-mapped files. For the B-tree back end, we selected Beman Dawes' proposed `boost::btree` library (https://github.com/Beman/Boost-Btree) and adapted our framework to support this back end. We used the Naval Postgraduate School's Bloom filter implementation.

After the user finalizes the database, the framework packs the data structures and releases extra space. In addition, it rewrites the B-tree with fully packed nodes at maximum density, enabling it to preload part of the tree into RAM.

Finally, the framework supports sharding the database into multiple chunks by the high-order bits in the key type.

Our key type is the 128-bit MD5 hash, and the record type is 64 bits partitioned to represent a file identifier and an offset yielding a 24-byte-per-record minimum cost. The flat map and B-tree back ends are most efficient, using less than 25 bytes per element, while the red/black tree and hash maps are less efficient, using 64 and 61 bytes per element, respectively. The red/black tree overhead comes from the tree nodes, while the hash map overhead results from unused buckets.

As cryptographic hashes are designed to be unpredictable, there is no similarity from one hash to the next, thus there is little locality of reference that the operating system can exploit when building the database. System RAM thus becomes the dominant factor in determining the time required. On one Intel Xeon E5620-based server (2.4 GHz, 12-MiByte L2 cache) with 32 GiBytes of RAM, it took 29

Table 2. Total transactions per second (TPS) for best execution.

Bloom filter			Database		TPS at 1 M lookups		TPS at 1,200 seconds	
k	*M*	Size	Strategy	Size	Present	Absent	Present	Absent
100 million records								
3	31	257 MiBytes	B-tree (preload)	2.3 GiBytes	35.3 K	49.5 K	161.3 K	1.8 M
3	31	257 MiBytes	B-tree	2.3 GiBytes	11.6 K	565.8 K	156.8 K	2.3 M
3	31	257 MiBytes	Hash map	5.3 GiBytes	13.9 K	656.9 K	641.9 K	3.0 M
3	31	257 MiBytes	Flat map	2.2 GiBytes	28.2 K	746.9 K	356.4 K	2.6 M
3	31	257 MiBytes	Red/black tree	6.0 GiBytes	12.9 K	694.5 K	187.0 K	2.7 M
1 billion records								
3	34	2.1 GiBytes	B-tree (preload)	23 GiBytes	2.2 K	6.1 K	3.6 K	23.1 K
3	33	1.1 GiBytes	B-tree	23 GiBytes	2.6 K	85.8 K	3.7 K	114.9 K
3	33	1.1 GiBytes	Hash map	57 GiBytes	–	–	0.3 K	3.1 K
3	34	2.1 GiBytes	Flat map	22 GiBytes	–	–	0.4 K	4.0 K
3	33	1.1 GiBytes	Red/black tree	60 GiBytes	–	–	0.1 K	1.4 K

Dashes indicate that 1 million queries were not completed in the 1,200 seconds allowed.

days to create a billion-record hash map, while it took less than four hours on a slower, AMD Opteron 6174-based system (2.2 GHz, 512-KiByte L2 cache) with 256 GiBytes of RAM.

We found that creating some locality by first building the database as a flat map and then converting to either a B-tree or hash map was faster than generating the B-tree or hash map directly. Likewise, we found that tuning the Linux operating system parameters `dirty_ratio`, `dirty_background_ratio`, and `dirty_expire_centisecs` to allow dirty pages to stay in memory longer improved performance by helping the OS use the disk cache more efficiently.

When fielding systems using the block hash database, system memory and I/O speed are the prime drivers. A drive triage system must be able to read disk sectors as fast as possible from a subject drive and test hashes of those sectors against the database. Large RAM allows caching more of the database, reducing I/O pressure. The database should be stored on a solid state drive (SSD) to further speed I/O, since every lookup will require one or more random seeks within the database file.

For systems supporting fixed sites, such as a customs and immigration checkpoint, a large memory server or cluster can maintain the entire database in RAM and support several triage stations over a gigabit network.

Back-end testing

We performed back-end testing with databases containing 100 million and 1 billion records. The tests were done on a laptop with 8 GiBytes of RAM, a 2.67-GHz processor, and a 250-Gbyte SSD attached via eSATA and USB2 drives. We performed additional testing on a desktop system with 24 GiBytes of RAM and spinning media. All runs were performed with 50/50 random blends of database hits and misses, which might be unrealistically pessimistic. To guarantee that no part of the database was already loaded in memory, we directed the OS to stop caching all disk files by syncing the disks and then writing a "3" into `/proc/sys/vm/drop_caches` between each run.

Table 2 shows the read transactions per second against the 100 million and one billion record databases after one million lookups (2-384 seconds, depending on the row) and at 1,200 seconds, obtained with the four back-end strategies and B-tree with and without preload. Performance graphs for all of the runs are available at http://domex.nps.edu/deep.

The hash map offered the best performance at 100 million records, followed in order by the red/black tree, the flat map, and the B-trees. There was a factor-of-eight difference for queries that were present, but only a 40 percent spread for queries that were not present. In all cases, we observed that database misses were dramatically faster than hits, a result of prefiltering with the Bloom filter. The back-end performance is still relevant for misses, however, due to the false positives. We also observed that very large Bloom filters negatively impacted speed because of increased memory pressure. At one billion records, we obtained the best performance with $M = 33$ for the no-preload B-tree. Note that while the hash map outperformed the other strategies at 100 million records, B-trees overall dominated all other strategies by a factor of almost 30 (300 times better than the classic databases). The USB2 drive was roughly half the speed of the eSATA drive.

In sum, for billion-record hash databases, the B-tree is the best choice. For smaller datasets, the hash map

is the fastest, but the flat map offers the best compromise between space and time (being among the smallest and tied for second place in speed), while the B-tree offers the poorest (requiring the most space and being the slowest). In either case, database lookups can be performed faster than sectors can be read from a drive being triaged.

SECTOR HASHING IN DIFFERENT FILE SYSTEMS

An advantage of sector hashing is the ability to process raw media without reference to the underlying file system. Doing so requires aligning the file data on sector boundaries. Fortunately, most file systems in use today align files in data units that consist of multiple disk sectors. These allocation units are variously called clusters, blocks, or sectors.

Sector hashing can aid in revealing the presence of encrypted files, provided they have not been re-encrypted.

Current file systems

The FAT (File Allocation Table) file system, introduced by Microsoft with DOS, has become the de facto format for storage devices such as thumb drives, external hard disks, and Secure Digital (SD) cards. All three FAT variants (12, 16, and 32) block-align data.

Microsoft developed exFAT (extended FAT) to address FAT's file size and performance limitations. It lacks the NTFS security features, but it can support file sizes greater than 4 GiBytes. Like its predecessors, exFat has a block-aligned data region.

NTFS (New Technology File System) is the default file system for the current generation of Windows. It uses a master file table (MFT) that has an entry for every file and directory. NTFS block-aligns large files but not files smaller than 1,024 bytes, which can be contained entirely in the MFT. The advent of 4-KiByte physical sector drives raises an issue, as they are not supported by products prior to Windows 8 and Server 2012. Instead, NTFS uses an "emulation mode" 512e to return a logical sector. While emulation can occur transparently, it also can induce file system clusters to cross physical sector boundaries, causing every physical sector to contain parts of different contiguous clusters. The techniques for working with 512-byte physical sector sizes also address alignment problems.

Ext4 is the default file system for most Linux distributions, including newer versions of Android. A major difference between Ext4 and Ext3 is that the former uses extents, while the latter employs a block pointer system. Despite this, Ext3 and Ext4 base their allocation on blocks, so file data is invariably aligned with the underlying storage media.

Next-generation file systems

Newer file systems handle data storage quite differently than their predecessors. Differences include data and metadata integrity mechanisms, copy-on-write transactions instead of journals, and built-in support for snapshots. However, sector hashing should work on these systems.

ZFS is the most mature next-generation file system, and the only one used in production environments. ZFS "blocks" are dynamically sized extents consisting of multiple sectors. If a file requires more space than the maximum block size, the system allocates multiple blocks. Since blocks are always aligned with the underlying storage media, there is no impact on sector hashing.

The B-tree file system (Btrfs) is poised to become the file system of choice for Linux. Although Btrfs uses extents of blocks to store large files, it can pack small files into the leaf block of the B-tree used to store file attributes. Thus, the system might not sector-align files smaller than 4 KiBytes.

The Resilient File System (ReFS) is Microsoft's upcoming innovation. Like Btrfs, ReFS makes extensive use of B-trees and extents. ReFS will block-align data, but whether it will pack small files into the B-trees is currently unknown.

Encrypted file systems

If an application such as Adobe Acrobat encrypts a file and transfers it to a different system, the encrypted data blocks will remain the same on the target media. Thus, sector hashing can aid in revealing the presence of encrypted files, provided they have not been re-encrypted.

Encrypted file systems, in contrast, present a significant problem. BitLocker for NTFS and ReFS and FileVault 2 for Apple's HFS+ encrypt data blocks as they are written to the storage medium and decrypt them when they are read back. Because each drive is encrypted with a different key, the same data will be encrypted differently on different drives. Thus, sector hashing will not work with these drives unless the block device is read through the file system after the decryption key has been loaded or the drive is otherwise decrypted.

Quickly detecting documents or images of interest in digital media is critical to the forensic investigation process. Given a large disk or set of disks, an investigator requires an efficient triage process to determine if known bad or illegal files or previously unseen files that require additional analysis are present. Traditionally, forensic investigators use file-hashing tools to analyze the file system. However, file hashing has several shortcom-

ings: it does not work with files that have been modified in any way, it requires files to be intact, and it requires the ability to extract both allocated and deleted files from the subject media in a forensically sound manner.

Our approach for forensic identification of data searches disk sectors for distinct file blocks, rather than searching the file system for distinct files. Our method is agnostic to the file system and file type, and can analyze all portions of the media including unallocated space, metadata, and encrypted content. Sector hashing can also be parallelized since each sector is processed independently of all others.

Using sector hashes presents several challenges. The first is choosing an appropriate file block size that balances the ability to identify distinct chunks of files with the amount of data that needs to be stored and analyzed. A block hash database's large size makes it necessary to design a custom data store for this application. It is also useful to identify disk sectors that are likely to be nondistinct to minimize the number of queries made to the database without missing critical distinct file blocks.

A potential critique of our approach is that an attacker could defeat it by adding or removing semantically empty data to files, thereby changing the sector alignment. In response, we note that it would be easier to encrypt the entire drive and that many people are still not doing this.

Although sector hashing will be a powerful tool for media forensics, the size of the block hash database will surely hamper widespread adoption. Consequently, sector hashing is more likely to appeal to large organizations searching for stray copies of their own files and new variants of malware that they have already encountered, rather than by small organizations seeking to match their media against a database distributed by a vendor or the US government.

Acknowledgments

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the US government. The US government is authorized to reproduce, distribute, or authorize reprints for any reason notwithstanding any copyright annotations thereon.

References

1. S. Garfinkel et al., "Using Purpose-Built Functions and Block Hashes to Enable Small Block and Sub-File Forensics," *Digital Investigation*, Aug. 2010, pp. S13-S23; www.dfrws.org/2010/proceedings/2010-302.pdf.
2. S. Garfinkel et al., "Bringing Science to Digital Forensics with Standardized Forensic Corpora," *Digital Investigation*, Sept. 2009, pp. S2-S11; www.dfrws.org/2009/proceedings/p2-garfinkel.pdf.
3. D. Quist, "State of Offensive Computing," blog, 7 July 2012; www.offensivecomputing.net/?q=node/1868.
4. E.M. Bakker, J. van Leeuwen, and R.B. Tan, "Prefix Routing Schemes in Dynamic Networks," *Computer Networks and ISDN Systems*, Dec. 1993, pp. 403-421.
5. B.H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Comm. ACM*, July 1970, pp. 422-426.
6. P. Farrell, S.L. Garfinkel, and D. White, "Practical Applications of Bloom Filters to the NIST RDS and Hard Drive Triage," *Proc. Ann. Computer Security Applications Conf.* (ACSAC 08), IEEE CS, 2008, pp. 13-22.

Joel Young *is an assistant professor in the Department of Computer Science at the Naval Postgraduate School. His research interests include computer forensics, algorithm design, and machine learning. Young received a PhD in computer science from Brown University. He is a member of the Association for the Advancement of Artificial Intelligence. Contact him at jdyoung@nps.edu.*

Kristina Foster *is a student at the Naval Postgraduate School. Her research interests include computer forensics and computer security. Foster received an MS in engineering, electrical engineering, and computer science from the Massachusetts Institute of Technology. Contact her at kmfoster@nps.edu.*

Simson Garfinkel *is an associate professor in the Department of Computer Science at the Naval Postgraduate School. His research interests include computer forensics, security visualization, and information policy. Garfinkel received a PhD in computer science from the Massachusetts Institute of Technology. He is a member of IEEE and ACM. Contact him at slgarfin@nps.edu.*

Kevin Fairbanks *is a cybersecurity research engineer in the Applied Physics Laboratory at Johns Hopkins University. His research interests include digital forensics and computer security. Fairbanks received a PhD in electrical and computer engineering from the Georgia Institute of Technology. He is a member of IEEE. Contact him at kevin.fairbanks@jhuapl.edu.*