Hashing and Data Fingerprinting in Digital Forensics

Hashing is a primary, yet underappreciated, tool in digital forensic investigations. Recent R&D has demonstrated that, with clever design, we can construct robust fingerprinting and similarity hashes that can significantly speed up an investigation.

Vassil Roussev
University of New Orleans

Digital forensic analysis aims to reconstruct a chain of events that have resulted in the current observable state of a computer system or digital artifact. Generally, an investigation involves answering four questions:

- What happened?
- When did it happen?
- How did it happen?
- Who did it?

With the persistent societal trend of digitizing all information, such analysis is becoming increasingly critical in investigating the entire range of illegal activities, from minor infractions to capital cases.

In computer security, forensic analysis—also called *incident response* in this context—is the first step in identifying, understanding, and mitigating security breaches. In the corporate environment, most aspects of the business already depend heavily on massive computer systems, and the capability to examine them forensically in a timely fashion has become essential. According to the Computer Security Institute, insider abuse of Internet access has eclipsed virus attacks as the number one computer security concern, with 59 percent of companies reporting such incidents.[1] Insider threats pose significant privacy concerns, such as leaks of sensitive information (both accidental and malicious), and expose companies to liabilities from employee misuse of the IT infrastructure. In all cases, response time is critical.

To quickly and efficiently screen data, forensic examiners rely heavily on hash-based techniques. Recent research has considerably expanded the range of such techniques to include adaptations of data-fingerprinting methods from other domains. Here, we describe the driving problems that motivate R&D in this area, and survey both established practices and recent research advances.

The Problem of Scale

One of the biggest obstacles to rapid response in digital forensics is scale. As the generation of digital content continues to increase, so does the amount of data that ends up in the forensic lab. According to US Federal Bureau of Investigation (FBI) statistics, the average amount of data examined per criminal case has been increasing by 35 percent annually—from 83 Gbytes in 2003 to 277 Gbytes in 2007.[2] However, this is just the tip of the iceberg—the vast majority of forensic analyses support either civil cases or internal investigations and can easily involve terabyte-scale data sets.

Ultimately, a tiny fraction of that information ends up being relevant—the proverbial needle in a haystack. So, there's a pressing need for fast, efficient methods that can focus an inquiry by eliminating known content that's irrelevant and by pinpointing content of interest on the basis of prior knowledge. As an illustration of the problem's difficulty, consider the 2002 US Department of Defense investigation into a leaked memo detailing Iraq war plans. According to *Computerworld* Australia, authorities seized 60 Tbytes of data in an attempt to identify the source.[3] Several months later, the investigation closed with no results. Another widely publicized example, the Enron case, involved more than 30 Tbytes of raw data and took

many months to complete.[4] Although these examples might seem exceptional, it isn't difficult to envision similar, plausible scenarios in a corporate environment involving large amounts of data. As media capacity continues to double every two years, huge data sets will increasingly be the norm.

Finding Known Objects: Basic Hashing

The first tool of choice in investigating large volumes of data is hashing—it's routinely used to validate data integrity and identify known content. At a basic level, hash-based methods are attractive because of their high throughput and memory efficiency. A hash function takes an arbitrary string of binary data and produces a number, often called a *digest*, in a predefined range. Ideally, given a set of different inputs, the hash function will map them to different outputs.

Intuitively, a hash function is *collision resistant* if finding two different inputs with the same output is computationally infeasible. Cryptographic hash functions, such as MD5, RIPEMD-160, SHA-1, SHA-256, and SHA-512, are explicitly designed to be collision resistant and to produce large, 128- to 512-bit results. Because the probability that two different data objects will produce the same digest by chance is astronomically small, we can assume that two objects having the same digest are identical. Another way to look at this property is that we have a compression mechanism by which we can generate a unique, fixed-size representation for data objects of any size. Clearly, this is an irreversible computation because we can't recover the original object from the digest.

Researchers have developed many other hashing algorithms, such as checksums, polynomial hashes, and universal hashes, but these have mostly found limited use in digital forensics. The main reason is that cryptographic hash functions are quite affordable on modern hardware—a good workstation can easily sustain bulk MD5 hashing (the most popular choice) at 400 Mbytes per second on a single core, whereas large commodity hard drives—the source of all data—are limited to approximately 100 Mbytes/s. Other classes of hash functions are either slower to compute or provide less collision resistance, so there's little incentive to use them.

The state of the practice is to apply a cryptographic hash function, typically MD5 or SHA-1, to either the entire target (drive, partition, and so on) or individual files. The former approach validates the forensic target's integrity by comparing before-and-after results at important points in the investigation. The latter method eliminates known files, such as OS and application installations, or identifies known files of interest, such as illegal ones. The US National Institute of Standards and Technology (NIST) maintains the National Software Reference Library (NSRL; www.nsrl.nist.gov), which covers most common OS installation and application packages. Similarly, commercial vendors of digital forensics tools provide additional hash sets of other known data.

From a performance perspective, hash-based file filtering is attractive—using a 20-byte SHA-1 hash, we could represent 50 million files in 1 Gbyte. So, we could easily load a reference set of that size in main memory and filter out, on the fly, any known files in the set as we read the data from a forensic target.

Besides whole files, we're often interested in discovering file remnants, such as the ones produced when a file is marked as deleted and subsequently partially overwritten. A common method to address this problem is to increase the hashes' granularity—we can split the files into fixed-size blocks and remember each block's hashes. Once we have a block-based reference set, we can view a forensic target as merely a sequence of blocks that can be read sequentially, hashed, and compared to the reference set. Typically, the block size is 4 Kbytes to match the minimum allocation unit used by most OS installations. This scheme has two main advantages: we can easily identify pieces of known files and avoid reading the target hard drive on a file-by-file basis. (File-based data access tends to generate a nonsequential disk access pattern, which can seriously degrade throughput.)

Efficient Hash Set Representation: Bloom Filters

Faced with a large reference hash set, a forensic tool needs an efficient mechanism to store and query it. Most tools sort the hashes, lay them out sequentially in memory, and query them using binary search. To facilitate this process, NSRL and other reference sets are already sorted. Although this organization isn't unreasonable, as the set's size grows, the query mechanism's performance degrades substantially, regardless of the host machine's computational capabilities. For example, for every query in a set of 50 million reference hashes, we would expect approximately 26 main memory accesses, each of which will cause a delay of tens of CPU cycles. (Owing to the randomized pattern in which the referenced set is accessed, cache benefits would be marginal.) Obviously, such a memory-constrained workload severely underutilizes the CPU.

One promising approach to speed up lookup operations and to reduce space requirements is Bloom filters. First introduced by Burton Bloom,[5] they're widely used in areas such as network routing and traffic filtering. A Bloom filter is simply a bit vector of size m, with all bits initially set to zero. The basic idea is to represent each element of the set as a (hopefully) unique combination of k bit locations. For that purpose, we need a set of k independent hash functions, $h_1, \ldots, h_k$, that produce

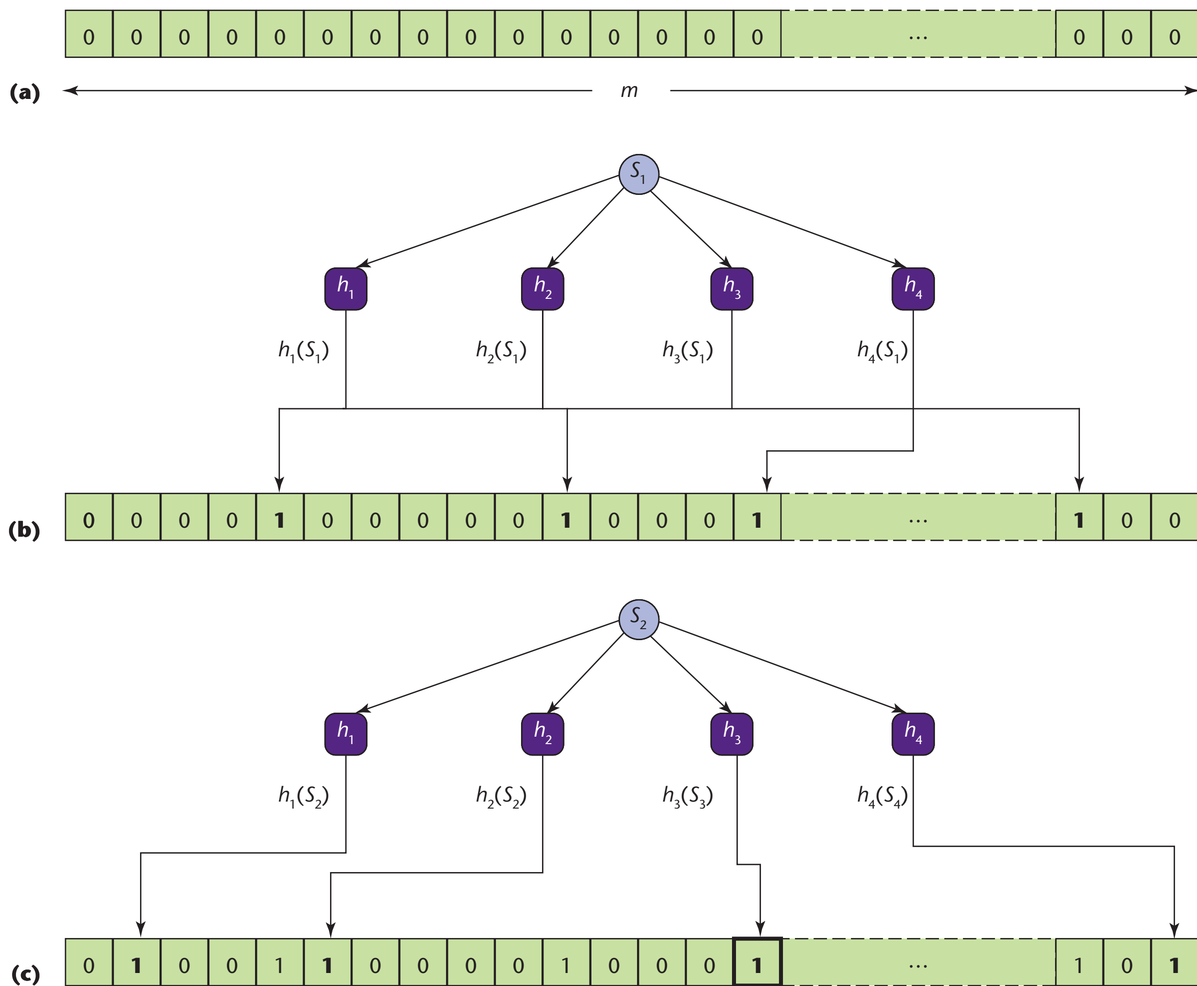

Figure 1. The insertion of two elements into a Bloom filter using four hash functions: (a) an empty Bloom filter; (b) a Bloom filter after the insertion of one element, S_1; and (c) a Bloom filter after the insertion of a second element, S_2. Each insertion sets four bits in the filter; some bits might be selected by different elements—$h_4(S_1) = h_3(S_3)$—which can lead to false positives.

values in the range of 0 to $m - 1$. To insert an element (a binary string) S_1, we apply each hash function to it, which gives us k values. For each value—$h_1(S_1)$, …, $h_k(S_1)$—we set the bit with the corresponding number to one (setting a bit twice has the same effect as setting it once). Figure 1 shows an example insertion of two consecutive elements—$S_1$and S_2—using four hash functions: h_1, h_2, h_3, and h_4.

To look up an element, we hash it with all the hash functions and check the corresponding bits—if all of them are set to one, we return "yes"; otherwise, "no." The filter will never return a false negative; that is, if the element was inserted, the answer will always be "yes." However, we could have a false positive—a "yes" answer for an element that has never been inserted but whose bits have been set by chance by other element insertions.

False positives are the price we pay for the compression gains. The silver lining is that because we can quantify false-positive rates analytically, we can control them.[6] Generally, after the insertion of n elements, the probability that the filter will return a false positive is a nonlinear function of the bits-per-element ratio m/n and the number of hash functions k. Table 1 lists different parameter combinations and their corresponding false-positive rates.

As it turns out, the routine use of cryptographic hashes in digital forensics makes it easy to introduce Bloom filters into the process. Instead of computing k separate hashes, we can take an object's cryptographic hash, split it into several nonoverlapping subhashes, and use them as if different hash functions had produced them. For example, we could split a 128-bit MD5 hash into four 32-bit hashes, which would let us work with a 1-Gbyte filter and four hash functions. If we insert 50 million hashes, the expected false-

Table 1. Example Bloom filter parameters and predicted false positives.

NO. OF HASHES	FALSE-POSITIVE RATE (ACCORDING TO BITS PER ELEMENT)			
	8	10	12	16
4	0.0240	0.0117	0.0064	0.0024
6	0.0216	0.0083	0.0036	0.0009
4	0.0255	0.0081	0.0031	0.0006

positive rate would be less than 0.3 per million, which in almost all cases would be quite acceptable. What we gain in return would be four memory accesses instead of 26. In many situations, such as the initial screening of evidence, much higher false-positive rates are acceptable to reduce the volume of data under consideration. For example, we could increase the number of hashes from 50 to 500 million and expect a false-positive rate of 0.2 percent.

Finding Similar Objects: Data Fingerprints

So far, we've considered searches for objects that are an exact copy of a reference object; a much more challenging problem is to find similar objects. For example, modern software has a propensity for frequent online updates, which tends to age static hash sets rather quickly. We want to be able to identify executables and libraries that are likely to be newer versions of known application installations. Similarly, given a text file, we want to be able to automatically find different versions of it, perhaps as an HTML file or as part of another document.

By design, hashes are fragile—even if a single bit in a file changes, the hash will be completely different. If we insert a single character into a file, all the block hashes following the change will also change. So, block-based hashing will do little to fundamentally address the fragility problem. Instead, we need a smarter mechanism, called *data fingerprinting*, that can generate a signature of the data that's more resilient to modifications. The term "digital fingerprint" is heavily overloaded and signifies different things to different people. In particular, in the security and authentication domain, it usually refers to the message digest produced by a cryptographic hash function. Here, we consider a more relaxed form of fingerprinting that doesn't aim to be unforgeable.

The essential idea has been around for decades and is fairly generic and simple. For every object, we select characteristic features and compare them to features selected from other objects, using some measure of correlation to draw conclusions. We could apply this approach at different abstraction levels, from comparing raw binary data all the way up to natural language processing, where we could extract the semantics of text, for example, to make more abstract connections. In forensics, analysis at multiple levels can be relevant to an investigation; however, the computational cost tends to grow rapidly with the abstraction level. Here, we focus entirely on raw data analysis—we will consider data objects mere strings of bytes; our goal is to find their similarities.

The seminal research on data fingerprinting, by Michael Rabin, dates back to 1981.[7] It's based on random polynomials, and its original purpose was "to produce a very simple real-time string-matching algorithm and a procedure for securing files against unauthorized changes."[7] In essence, we can view a Rabin fingerprint as a checksum with low, quantifiable collision probabilities that can be used to efficiently detect identical objects. Rabin and Richard Karp soon extended this research to improve pattern-matching algorithms.[8] The 1990s saw renewed interest in Rabin's work in the context of finding all things similar, with an emphasis on text. For example, Udi Manber created the *sif* tool for Unix to quantify similarities among text files.[9] Sergey Brin and his colleagues, in Brin's pre-Google years, used Rabin fingerprinting in a copy-detection scheme,[10] and Andrei Broder and his colleagues applied it to find syntactic similarities among Web pages.[11]

The basic idea, called *anchoring*, *chunking*, or *shingling*, uses a sliding Rabin fingerprint over a fixed-size window to split the data into pieces. For every window of size w, we compute the hash h, divide it by a chosen constant c, and compare the remainder to another constant m. If the two are equal ($h \bmod c \equiv m$), we declare the beginning of a chunk (an anchor), slide the window by one position, and continue the process until we reach the data's end. For convenience, the value of c is typically a power of two ($c = 2^k$), and m can be any fixed number between zero and $c - 1$.

Once we've determined our baseline anchoring, we can use it in numerous ways to select characteristic features. Figure 2 illustrates three examples:

- Choose the chunks (or shingles) between anchors as our features (Figure 2a).
- Start at the anchor position, and pick the following x number of bytes (Figure 2b).
- Use multiple, nested features (Figure 2c).

Although shingling schemes pick a randomized sample of features, they're deterministic and, given the same input, produce the exact same features. Furthermore, they're locally sensitive in that an anchor point's determination depends only on the previous *w* bytes of input, where *w* could be as small as a few bytes. We can use this property to solve our fragility problem in traditional file and block-based hashing. Consider two versions of the same document—we can view one of them as derived from the other by means of inserting and deleting characters. For example, converting an HTML page to plain text will remove all the HTML tags. Clearly, this would modify several features. However, we would expect chunks of unformatted text to remain intact and to produce some of the original features, letting us automatically correlate the versions. For the actual feature comparison, we store the selected features' hashes and use them as a space-efficient representation of the object's fingerprint.

We have some control over our feature's average size through the selection of *c*: on random input, we expect features to be *c* bytes long on average. We face an important design choice. Smaller features provide better coverage and higher sensitivity at the expense of more storage and higher comparison costs. Larger features are easier on storage and the CPU but provide fewer details.

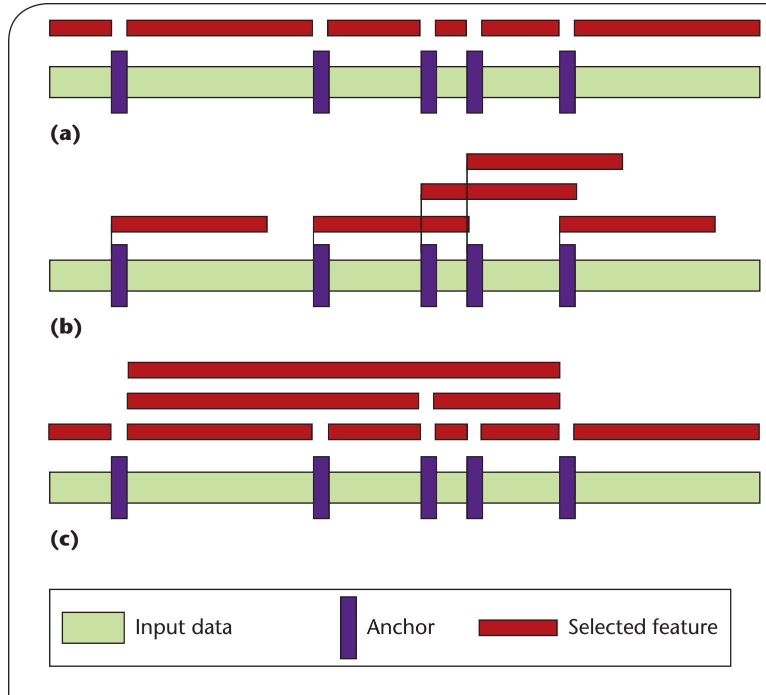

Figure 2 Rabin-style feature selection: (a) nonoverlapping, (b) fixed size, and (c) nested multilayer. Different feature selection techniques allow the baseline fingerprinting scheme to be customized for the intended applications.

Optimizing Fingerprint Coverage

Many techniques strive to maximize coverage and minimize the possibility that a common feature as small as a few hundred bytes will be missed. One important area that requires this is malware detection. Autograph is an automated tool for generating signatures for traditional (static) malware.[12] It uses a Rabin scheme to break up the payloads of collected suspicious traffic into pieces of average size of 256 bytes ($c = 256$). It then compares them and extracts the statistically most likely signature for the specific piece of malware, in effect performing real-time, automated network forensics and defense. Because of the feature selection scheme's resilience to various payload alignments, we can scale up the system to a distributed environment where multiple instances collaborate to generate the signatures more quickly.

Payload attribution systems (PASs) are a major area of network forensics research that continues to advance fingerprints. Basically, a PAS collects a network trace in a digested, compressed form. Afterward, it provides a query interface that lets us ask whether the trace contains a specific byte sequence. The actual trace isn't stored, so we can't go back and review the traffic. This is the price we pay for a typical compression ratio of 50:1 and up, but the trace won't be an added security risk. An essential building block in modern PASs is a Bloom filter, which provides most of the compression.

One specific implementation is the use of *hierarchical Bloom filters* (HBFs). One common way to utilize Bloom filters is to slice a payload into a sequence of fixed-size blocks, $B_1B_2 \ldots B_n$, and insert each of them into the filter. We can split the query excerpt into same-size blocks and ask the filter whether it has seen them. The problem is that the filter maintains no ordering information, so asking for B_1B_2 and B_2B_1 would both yield positives. An HBF alleviates this problem by inserting additional levels of superblocks produced by the successive concatenation of 2, 4, 8, … blocks.[13] For example, if the original payload consists of seven blocks—$B_1B_2B_3B_4B_5B_6B_7$—the HBF will insert into the filter these elements: B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_1B_2, B_3B_4, B_5B_6, and $B_1B_2B_3B_4$. Although this doesn't eliminate the problem, it dramatically improves confidence in the results as the query excerpt's size grows.

A recently proposed alternative to the hierarchical approach is a *rolling Bloom filter* (RBF),[14] which also stores aggregated results but does so linearly. In the previous example, we could use an aggregation factor of 3 and a step of 2. This would result in the RBF inserting into the filter the superblock excerpts $B_1B_2B_3$, $B_3B_4B_5$, and $B_5B_6B_7$. This method performs better in that it consistently achieves the best-case performance of an HBF.

Winnowing aims to improve the original Rabin

scheme's coverage by ensuring that exactly one hash is picked from every data window of size *w*.[15] It achieves this by first computing the hashes of all sequences of size *k* and then selecting the hash with the highest value as the fingerprinting feature. This is guaranteed to achieve more even distribution of fingerprinting features than the original technique, which tends to have nontrivial variation in feature size and distribution, especially for low-entropy data.

Winnowing multihashing (WMH) is among the most recent advances in PASs; it combines multiple Rabin polynomials, shingling, and winnowing.[16] The result is a system that, for HTML traffic, produces a false-positive rate of less than 1 in 10,000 for excerpts of at least 200 bytes and a compression ratio of 50:1. For 130:1, WMH achieves the same false-positive rate for 500-byte excerpts.

Similarity Hashing

So far, we've looked at systems that basically solve a particular type of needle-in-a-haystack problem—given a small query string (the needle), does the data haystack contain it? A related problem is to establish "haystack similarity"—given two (500-Gbyte) haystacks, how similar are they on the basis of their needles? Similarity hashes aim to efficiently make such a comparison by selecting and compactly storing a set of characteristic features for each object of interest and by providing a mechanism to compare the hashes directly.

Inspired by earlier spam-filtering research, Jesse Kornblum proposed *fuzzy hashing*.[17] This approach uses shingling to split the file into chunks, generates small, 6-bit hashes for every chunk, and concatenates them to produce the final hash, which is base64 encoded. To determine similarity, it treats the two hashes as text strings and compares them using an edit distance measure, which produces a number between 0 and 100. An interesting design choice is to limit the hash's size to 80 symbols—this is mainly to give investigators a fixed hash value per file similar to the ones produced by cryptographic hashes (MD5, SHA-1, and so on). To achieve this, the algorithm requires the overall length of the data. On the basis of this length, it estimates a value for *c*.

Furthermore, after the algorithm calculates the hash, if the result is longer than the target 80 symbols, it doubles the *c* parameter and recalculates the hash from scratch. For example, for a set of MS Word documents, the algorithm performs the calculation twice on average. Because the actual file signature is sensitive to object size, it produces two hashes for two different resolutions—*c* and 2*c*. This takes the edge off the problem, but if the difference in size between the data objects exceeds a factor of four, the two hashes aren't comparable. In practice, the hash does seem to effectively identify objects that are versions of each other and aren't too big or dissimilar. However, the hash quickly loses resolution for larger files and can't be applied to stream data where the size is unknown.

Around the same time as Kornblum, I proposed a similarity hash measure, primarily for detecting versions of executable files and libraries.[18] Essentially, this method breaks the object into known components (coded functions and resources), hashes each component, and combines them into a Bloom filter to produce the similarity hash. It then compares hashes by counting the number of corresponding bits in common between the two filters and comparing them with the theoretical expectations—any statistically significant deviation indicates similarity. Performance results for system libraries demonstrate that this method can readily detect versions, even for filters with very high compression rates.

My follow-up *multiresolution similarity* (MRS) hashing work addressed two main problems: accommodating arbitrarily large objects and comparing objects of widely disparate sizes (for example, a file and a drive image).[19] The key idea is to simultaneously select features at multiple resolutions (see Figure 1c) using a Rabin shingling scheme, by using a set of multiplicative parameters for $c = 256$ for level 0, 4,096 for level 1, 65,536 for level 2, and so on. Small objects are naturally compared at the finest resolution (level 0); for large ones, we can start at the lowest resolution to find out quickly whether any large-scale similarities exist. Depending on the target application and time constraints, we can choose to move progressively to the appropriate resolution level. For objects of different magnitudes, we can pick any level that's common to both of them, typically level 0. In terms of storage requirements, the MRS hash is approximately 0.5 percent of the original object size, which lets us, for example, load in main memory the hashes of two 500-Gbyte targets and compare them. The storage footprint is dominated by level-0 features that are, on average, approximately 256 bytes long. So, we can drop the finest resolution from consideration, reduce storage requirements by another factor of 16, and still achieve disk-block resolution with level-1 features.

Implementations

You can find the baseline hashing capabilities—hashing of entire targets, file-based hashing, or block-based hashing—in virtually every commercial tool, such as AccessData's FTK and Guidance Software's EnCase, as well as in open source tools such as SleuthKit (sleuthkit.org), maintained by Brian Carrier. The data-fingerprinting techniques are only now being adapted to the forensics domain and, at this stage, implementa-

tions are primarily research prototypes. The HBF and WMH methodologies are relatively mature and have been implemented at the Polytechnic Institute of New York University as part of the ForNet research system for network forensics (http://isis.poly.edu/projects/fornet). Kornblum maintains an open source version of his fuzzy-hashing scheme called *ssdeep* (http://ssdeep.sf.net). I maintain a research prototype of the MRS and am happy to provide the source on request.

Hashing is a primary, but underappreciated, tool in digital forensic investigations. Recent R&D has demonstrated that, with clever design, we can construct robust fingerprinting and similarity hashes that can significantly speed up an investigation in several ways. We can also quickly and efficiently find versions of known objects, with which we can effectively narrow an inquiry's focus by filtering data in or out. We can use the same techniques to screen targets for trace evidence (for example, remnants of a JPEG file) without relying on file system metadata, which allows the processing of corrupted or formatted targets. We can quickly trace pieces of evidence (for example, a file) across multiple data sources—a file system image, a memory dump, network capture, and so on. Achieving this with traditional "exact" techniques is daunting because it requires parsing numerous formats and reconstructing the objects. Payload attribution schemes allow efficient, reliable searches of network traces for small byte sequences (for example, a piece of malware), thereby supporting effective incidence response and network forensics. Overall, hash-based techniques are currently the only ones promising quick, high-level estimates of the content of large targets and fast multi-drive correlation on a terabyte scale. □

References

1. R. Richardson, "2007 CSI Computer Crime and Security Survey," Computer Security Inst., 2007.
2. *Regional Computer Forensics Laboratory Program Annual Report FY2007*, US Federal Bureau of Investigation, 2007; www.rcfl.gov/downloads/documents/RCFL_Nat_Annual07.pdf.
3. P. Roberts, "DOD Seized 60TB in Search for Iraq Battle Plan Leak," *Computerworld* (Australia), 31 Jan. 2005; www.computerworld.com.au/index.php/id;266473746.
4. *RCFL Program Annual Report for Fiscal Year 2006*, US Federal Bureau of Investigation, 2006; www.rcfl.gov/downloads/documents/RCFL_Nat_Annual06.pdf.
5. B. Bloom, "Space/Time Tradeoffs in Hash Coding with Allowable Errors," *Comm. ACM*, vol. 13, no. 7, 1970, pp. 422–426.
6. A. Broder and M. Mitzenmatcher, "Network Applications of Bloom Filters: A Survey," *Proc. Ann. Allerton Conf. Communication, Control, and Computing*, 2002; www.eecs.harvard.edu/~michaelm/NEWWORK/postscripts/BloomFilterSurvey.pdf.
7. M.O. Rabin, *Fingerprinting by Random Polynomials*, tech. report 15-81, Center for Research in Computing Technology, Harvard Univ., 1981.
8. R. Karp and M. Rabin, "Efficient Randomized Pattern-Matching Algorithms," *IBM J. Research and Development*, vol. 31, no. 2, 1987, pp. 249–260.
9. U. Manber, "Finding Similar Files in a Large File System," *Proc. Usenix Winter 1994 Technical Conf.*, Usenix Assoc., 1994, pp. 1–10.
10. S. Brin, J. Davis, and H. Garcia-Molina, "Copy Detection Mechanisms for Digital Documents," *Proc. 1995 ACM SIGMOD Int'l Conf. Management of Data*, ACM Press, 1995, pp. 398–409.
11. A. Broder, S. Glassman, and M. Manasse, "Syntactic Clustering of the Web," SRC Technical Note 1997-015, Digital Equipment Corp., 25 July 1997.
12. H. Kim and B. Karp, "Autograph: Toward Automated, Distributed Worm Signature Detection," *Proc. 13th Usenix Security Symp.*, Usenix Assoc., 2004, pp. 271–286.
13. K. Shanmugasundaram, H. Brönnimann, and N. Memon, "Payload Attribution via Hierarchical Bloom Filters," *Proc. 11th ACM Conf. Computer and Communications Security*, ACM Press, 2004, pp. 31–41.
14. C.Y. Cho et al., "Network Forensics on Packet Fingerprints," *Security and Privacy in Dynamic Environments*, Springer, 2006, pp. 401–412.
15. S. Schleimer, D. Wilkerson, and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," *Proc. 2003 ACM SIGMOD Int'l Conf. Management of Data*, ACM Press, 2003, pp. 76–85.
16. M. Ponec et al., "Highly Efficient Techniques for Network Forensics," *Proc. 14th ACM Conf. Computer and Communications Security*, ACM Press, 2007, pp. 150–160.
17. J. Kornblum, "Identifying Almost Identical Files Using Context Triggered Piecewise Hashing," *Proc. 6th Ann. Digital Forensics Research Workshop Conf.* (DFRWS 06), Elsevier, 2006, pp. S91–S97; www.dfrws.org/2006/proceedings/12-Kornblum.pdf.
18. V. Roussev et al., "md5bloom: Forensic Filesystem Hashing Revisited," *Proc. 6th Ann. Digital Forensics Research Workshop Conf.* (DFRWS 06), Elsevier, 2006, pp. S82–S90; www.dfrws.org/2006/proceedings/11-Roussev.pdf.
19. V. Roussev, G.G. Richard III, and L. Marziale, "Multi-resolution Similarity Hashing," *Proc. 7th Ann. Digital Forensics Research Workshop Conf.* (DFRWS 07), Elsevier, 2007, pp. S105–S113; www.dfrws.org/2007/proceedings/p105-roussev.pdf.

__Vassil Roussev__ is an assistant professor in the University of New Orleans Department of Computer Science. His research interests are digital forensics, computer security, and security issues related to distributed and collaborative computing. Roussev has a PhD in computer science from the University of North Carolina, Chapel Hill. Contact him at vassil@cs.uno.edu.