Effective Whitelisting for Filesystem Forensics

Sudarshan S. Chawathe

Abstract— **Forensic analysis of the large filesystems commonly found on current computers requires an effective method for categorizing and prioritizing files in order to avoid overwhelming the investigator. A key technique for this purpose is *whitelisting* files, i.e., skipping the detailed analysis of files that match files in a well known reference collection of files. Effective use of this technique requires an efficient method to match files, detecting not only exact matches, but also near matches or approximate matches. This paper outlines the requirements for such matching, formalizes them as the bounded best match and approximate bounded near-match problems, and describes methods to solve these problems. In particular, the approximate bounded near-match problem is mapped to the problem of finding near neighbors in a high-dimensional metric space and solved using locality-sensitive hashing.**

I. Introduction

It is increasingly common for criminal investigations to include forensic analysis of computer equipment. Given the widespread and growing use of computers in many aspects of everyday life, information valuable to a criminal case may be recovered by examining email messages, digital photographs, software logs, and other artifacts of a computer's operation. An important source of such information is data found in various files on a typically large hard disk or similar nonvolatile storage medium, such as flash memory or solid-state disks.

The large, and growing, capacities of current hard disks (several terabytes) are both a boon and a curse for forensic analysis. On the one hand, the large capacities have led to the growth of applications (e.g., email, instant messaging, chat rooms, digital photography, games, and video) and usage patterns (large logs, almost perpetual retention of messages and downloaded content) that are likely to assist investigators. On the other hand, analyzing such large amounts of data effectively within the time constraints of an investigation poses serious challenges. While human inspection of a few megabytes of data may be feasible, such inspection of terabytes of data is impractical. There is a critical need for semi-automatic tools and techniques that can assist an investigator in quickly locating those files that are worth further analysis.

Whitelisting is an important technique that has emerged to partly address this problem. This technique is based on the observation that a significant number of files in a computer system are used by the operating system software (e.g, GNU/Linux, Windows XP), various software components (e.g., Java, .NET), server software (e.g., Web and database servers), end-user applications (e.g., Office, Firefox), and so on. The investigator's workload can be substantially reduced if these files can be automatically or semi-automatically detected and flagged. Such detection is, at first glance, simple because most of these files change very rarely, if at all. For example, the executable files that are used to run Adobe Acrobat and similar applications are likely to change only when the software is upgraded, and not on a daily basis. This observation has led to the creation of several *whitelists*, i.e., lists of files belonging to known applications, operating systems, and other code-bases. These whitelists, created by one or more trusted entities, list the name, creator, and other relevant attributes of each known file along with its *signature*. A file's signature is essentially a small amount of data ($\approx$ 100 bytes) that is easily computed from the file's contents. A notable feature of such signatures is that even a small change in a file results in a significant change in the signature and, further, it is practically infeasible to create a file with a given signature, or to modify a file in a way that retains its signature.

In what follows, we use the phrase *reference collection* to refer to the files, signatures, and auxiliary information associated with a whitelist of files. We use the phrase *target filesystem* to refer to the filesystem that is being analyzed for evidence. The tasks outlined in this paper occur during two very different kinds of time periods. The first, which we call *preparation time* or simply *advance* refers to the time during which an investigator and colleagues prepare the tools for later use. The second time period, which we call *query time* or *analysis time*, refers to the time during which an investigator is actively studying evidence related to one or more investigations.

Section II describes how simple signatures may be used to detect and filter out files in the target filesystem that match files in the reference collection. Unfortunately, this technique is easily defeated by an adversary who may make small, inconsequential changes in the standard files, resulting in no matches with the whitelist signatures. Addressing this issue requires methods for matching files that are not overly sensitive to small changes in files. We outline two simple methods for this purpose: The first is based on the idea of computing a signature for each suitably-sized *block* of a file and forming a composite file signature by concatenating the block signatures (Section III-A). The second method

Author's address: Department of Computer Science; University of Maine; Orono, ME 04469, USA. chaw@cs.umaine.edu. This work was supported in part by the U.S. National Science Foundation with grant CNS-0426683.

improves on the first by replacing fixed-sized blocks with block boundaries that are computed dynamically based on file contents (Section III-B).

These enhancements to the basic technique make it harder for an adversary to defeat the whitelisting of files by making trivial changes. However, we also need methods for efficiently matching signatures of files from the target filesystem to signatures of "approximately equal" files in the reference collection. Section III-C outlines a simple method based on measuring the dissimilarity of files using the string edit distance between their signatures. Section III-D presents a faster method based on hashing composite file signatures using a locality-sensitive hashing scheme. Section IV describes related work and the conclusion appears as Section V.

It is worth emphasizing that the methods of this paper address only a very small, albeit important, part of the task of forensic analysis of a computer system. Not only do the methods not address important aspects such as live memory analysis and network forensics, they are also not designed to be used in exclusion. Rather, tools based on these methods are applicable in the early stages of forensic investigation of a computer, specifically the storage subsystem, in order to determine which files are the best candidates for a more thorough investigation using more sophisticated, but slower and typically more labor intensive tools. Additional remarks on the context of this problem appear in Section V.

II. Whitelist Matching

Given the expected large sizes of the target filesystems (several hundred gigabytes to terabytes) as well as the reference collections (tens to hundreds of gigabytes), matching files by comparing their contents is a task that would require hours to days and is therefore not practical in the early stages of an investigation. It is not surprising, therefore, that signature-based methods are popular in this context. These methods use one of several popular signature functions, such as MD5 and SHA-1 [1], [2]. (These signatures are also known as digests or hashes.) Recall from Section I the notion of preparation time, in advance of any investigation. At this time, each file in the reference collection is mapped to its signature using one or more such functions, yielding a reference collection of signatures. At query time, each file in the target filesystem is also mapped to its signature using the same functions. If the resulting signature is found in the reference collection, a match is reported. Since signatures are very small in comparison to the original files, searching for matching signatures is computationally much less demanding than a search based on the original files, although the search is still not trivial. For example, the uncompressed size of the reference collection from the NIST National Software Reference Library is 5.8 GiB [3].

Such direct signature-based methods have two key benefits when applied to the whitelist matching problem, both derived from the properties of the signature algorithms. First, since signatures are small objects (64 bytes for MD5, 128 for SHA-1), they incur low storage overheads and are efficiently matched. Second, it is computationally impractical to modify a file without modifying its signature, or to edit a file that matches a desired signature. As a result, when an unknown file's signature matches a signature in the reference collection, it is extremely unlikely that the contents of the two files differ.

Unfortunately, it is very easy for a knowledgeable adversary to practically defeat signature-based whitelisting schemes by making small, inconsequential changes in all, or many, files in a target filesystem. It is usually easy to make a small change to a text string encoding an error message in a program's executable file without otherwise affecting the operation of the program. For example changing a text string "invalid input" to "Invalid data." in an executable file will result in a completely different signature. When a large number of files in a target system have been modified in this manner, direct signature-based whitelisting is rendered useless because the result is a large collection of un-whitelisted files that an investigator must examine.

In order to overcome the above difficulty, it is useful to devise a system that finds not only exact matches in the reference collection, but also approximate, or close, matches. For instance, if a file in the reference collection matches an unknown file in all but a few bytes, it should be reported as a potential match. In deployment, such potential matches should be associated with a measure of confidence and distinguished clearly from the exact matches. If we quantify the dissimilarity of files f_1 and f_2 using a function $D(f_1, f_2)$ then we may pose the problem as one of determining a file f_m in the reference collection that is most similar to an unknown file f_u, provided the dissimilarity is no greater than some limit L.

> *Bounded Best Match:* Given (1) in advance, a domain S of elements, a finite set $R \subset S$ (reference "whitelist" elements), a distance metric $D : S \times S \to \mathbb{Z}^+$, and a positive integer L, and (2) at query time, an element $v \in S$, find an element $r^* \in R$ such that $D(r^*, v) \leq D(r', v) \leq L$ for all $r' \in R$, reporting failure if no such r^* exists.

While this modification to the exact matching task is conceptually simple, it poses two related difficulties: First, it is clear that standard signature schemes such as MD5 cannot be used directly because, by design, a single-byte change to a file results in a markedly different signature. Second, without some sort of signature scheme, determining matching files is likely to be computationally impracticable.

III. Detecting Near Matches by Hashing

A. Fixed Blockwise Hashing

For efficient matching, it is useful to be able to use some kind of signature or hashing scheme to match unknown files to files in the reference collection. In order to find near

matches in addition to exact matches, it is desirable that small changes to a file map to only small changes to the signature or that there be a significant correlation between the magnitudes of the difference in the file contents and the difference in signatures.

Perhaps the simplest method to address this requirement is to replace a file's single signature with a collection of signatures, one for each suitably sized portion of the file. For example, we may generate an MD5 signature for each consecutive 512-byte block of a file, and use the concatenation of all the block signatures as the file's composite signature. Concatenating entire MD5 signatures (or other similarly large signatures) yields a composite signature that is too large for our purposes, and we may include only a small portion of the signature of each block.

As in simple hash-based matching, identical file contents are mapped to identical composite signatures, and these matches are easily detected. In addition, the composite signatures of files that differ only in a few blocks will differ only in the components corresponding to those blocks. Consider the example, from Section II, of an executable file that has been modified by changing a few bytes. If the file size is 100 KiB (a fairly typical size for executable files), then its composite signature has 200 components, of which only one or two will change in response to the text modification of that example. (A change of two components will result only if the edited text spans a block boundary.) With a suitable threshold for matching signatures (such as 95% or 99% match) the modified file will be matched to the appropriate unmodified file from the reference collection.

The trouble with this method is that it too, like the scheme of Section II, may be defeated by a knowledgeable adversary. It is true that a single change, or a few changes, no longer defeat the scheme. Nevertheless, it is not very difficult to make small changes in a large number of a file's blocks. It is prudent to assume that the details of the whitelisting scheme, such as the block size and the thresholds for matching signatures, are known to the adversary. The adversary then only needs to change enough blocks to cross the threshold. Further, for many file types, it is not necessary to preserve the offsets of critical portions of the file (such as function-call addresses in an executable file). In such cases, the insertion of a single byte will result in a change in all subsequent block boundaries and a resulting change in the corresponding signature components. Even when some offsets must be preserved, an automated script can easily rewrite the necessary addresses to patch the file after an insertion.

B. Dynamic Blockwise Hashing

In order to make the above composite-signature scheme more robust to inconsequential changes made by a knowledgeable adversary, it is possible to remove its dependence on predetermined block boundaries that are easily manipulated by an adversary. The key idea is to dynamically determine whether a particular byte in the file marks a block boundary for the purpose of signature-generation. This determination is made by examining a few bytes in the vicinity of the byte, and scrambling those bytes using a suitable hash function, separate from the one used for the component signatures. We may use a *rolling hash* scheme similar to that used by the popular *rsync* and *spamsum* programs [4], [5]. The rolling hash is a simple hash function that is evaluated over a sliding window of a few bytes of the file as the window moves from the beginning to the end of the file. In order to enable efficient evaluation of this function at each byte, a scheme similar to the Adler-32 hash function is used, so that the function's working output value is easily modified by adding the effect of a new byte and removing the effect of the old byte as the window slides down the file.

If some byte position in the file produces a rolling hash function value that equals a predetermined constant (usually a pattern of all 1s), then that byte position is deemed a block boundary for signature-generation purposes. That is, a standard signature function, such as MD5, is computed over the portion of the file between the previous block boundary (or the file's beginning) and the current position, yielding a component of the composite signature. As in the fixed blockwise hashing scheme, in order to limit the composite signature to a manageable size, we use only a small portion of the standard signature of each dynamically determined block. Following rsync and spamsum, we use the six least-significant bits of the MD5 signature of each block.

C. Near Matches

The above scheme addresses the need for mapping files that differ only slightly to signatures that also differ only slightly, thus allowing for the detection of near matches for whitelisted files. However, it does not, on its own, provide a method for efficiently determining a suitable near match, or approximate match, for a given file. Following *spamsum*, we may measure the dissimilarity of signatures by their *string edit distance* [6]. Briefly, the edit distance between two strings is the weighted sum of the number of edit operations necessary to transform one to the other. The edit operations typically used include byte insertions, deletions, updates (in-place replacements), transpositions, etc., each with an associated weight. When the edit operations are not too complex, as above, the edit distance may be determined using a dynamic programming algorithm in time proportional to the square of the lengths of the input strings (i.e., the composite signatures) [6].

We note that, unlike typical applications of string-to-string editing (spelling correction, file comparison), the edit model used to determine the dissimilarity of composite signatures is not strongly motivated by the underlying processes. It is used here mainly because it is relatively easily computed and is roughly correlated with the needs of our application: that files differing in many blocks be farther apart than files differing in fewer blocks.

D. Approximate Near-Matches by Hashing

In order to determine files in the reference collection that potentially match an unknown file, the scheme of Section III-C requires that we compute, at query time, the edit distance between the signature of the unknown file and the signatures of all the files in the reference collection. We now describe a method that avoids this exhaustive search over the reference collection for each unknown file at the cost of providing only an approximate answer, in a sense made precise below.

In what follows, it is beneficial to view composite signatures as points, or vectors, in a high-dimensional space, and to model their dissimilarity using a suitable distance metric. Since we are working with signatures only, we may dispense with the files themselves and model the reference collection as a set of d-dimensional vectors with each dimension representing a component (block) signature of a composite (file) signature. The whitelisting problem may then be phrased as that of determining a reference vector that most closely matches a given vector (that of the unknown file), provided the dissimilarity is at most L. Further, we will relax our problem definition to permit an *approximate* solution in the following sense:

> *Approximate Bounded Near-Match:* Given (1) in advance, a finite set $R \subset \mathbb{Z}^d$ of d-dimensional reference vectors, a distance metric $D : R \times R \to \mathbb{Z}^+$, a positive integer L, and a positive real number ϵ, and (2) at query time, another d-dimensional vector $v \in \mathbb{Z}^d$, find a vector $r^* \in R$ such that $D(r^*, v) \leq (1 + \epsilon)L$, reporting failure if $r \geq (1 + \epsilon)L$ for all $r \in R$.

In order to avoid time-consuming query-time computation of distances between signatures of unknown files and all signatures in the reference collection, we appeal once again to hashing: We use a scheme that hashes the composite signatures in a manner that allows us to determine approximate near matches efficiently. In order to distinguish it from the earlier hashing steps (such as the rolling hash and the standard hash) we refer to this step as the *embedding hash*. The key idea is to use *locality sensitive hashing* for the embedding hash [7]. Intuitively, a hashing scheme is locality sensitive if items that are close to each other are very likely to be mapped to the same hash bucket while those that are distant are very likely to be mapped to different buckets.

The hash function used for this embedding hash step may be described as the composition of three simpler functions. The first function, h_1, maps a vector v (composite signature) to the concatenation of the unary representations of its components, with each component 0-padded on the right to a fixed length. Let z denote the largest value of a vector dimension (over all vectors). Then a component with value x is mapped to the binary sequence composed of x 1s followed by $z - x$ 0s. Thus each d-dimensional vector is mapped to a binary string of length zd. For example, if the largest value of a dimension is 5, then the vector $(3, 1, 4)$ is mapped to 11100 10000 11110, where the spaces are for legibility only. The second function, h_2, maps a binary string $b_1, b_2, \ldots, b_{zd}$, such as that produced by h_1, to a smaller binary string by selecting the k bits $b_{i_1}, b_{i_2}, \ldots, b_{i_k}$, where the indices i_j are selected uniformly randomly, and with replacement, from $\{1, 2, \ldots, zd\}$, and where k is a parameter. Both k and the random choices are determined in advance (during reference collection preparation), and remain fixed thereafter. Finally, the third function, h_3 maps a large number, such as the binary interpretation of the result of h_2, to the identifier of a hash bucket in the range $0, 1, \ldots, M$ using conventional methods (such as a simple additive function, modulo M). This entire process is conceptually repeated l times (where, like k, l is a parameter determined in advance), so that each signature v in the reference collection is inserted into the appropriate bucket $h_3(h_2(h_1(v)))$ in each of l hash tables. Note that, each hash table uses a different h_2 due to the separate random selections.

When the whitelisted files in a reference collection are processed in preparation for later work, each file is first mapped to a dynamic blockwise composite signature using the method of Section III-B. The composite signature is in turn hashed using the above method and inserted into l hash tables that are stored as indexes over the collection. When processing a target filesystem, each unknown file is processed in a similar manner, except that instead of inserting the file's signature into the hash tables, the reference signatures in the matching hash buckets are collected as potential near-matches. The best match among the signatures collected in this manner is determined by computing their distances from the signature of the unknown file. Thus the expensive distance-computation operations are performed for only the signatures collected from the hash buckets, which are very few in number. Using standard techniques, each hash-table lookup incurs exactly one bucket access with high probability.

The above method for finding approximate near-matches using locality-sensitive hashing depends on three key parameters: the threshold L that defines near matches (part of the problem definition); the length k of the bit-string produced by the second stage function h_2; and the number l of hash tables used for indexing the reference collection. At first glance, it would appear that the value for L must be chosen dynamically, depending on the characteristics of the unknown files in the target filesystem. However, both experimental and theoretical observations indicate that the distribution of the distances of query vectors and reference vectors are determined mainly by the characteristics of the reference collection [8]. Thus, once a reference collection has been built, the hash indexes need only be built for one statically selected value of L. A suitable choice of parameters k and l is guided by a result [7, Theorem 1], suggesting $k = \log_{10}(n/B)$ and $l = (n/B)^{0.77}$, where n is the size of the reference collection (number of composite signatures) and B is the number of signatures in one hash bucket.

IV. Related Work

Carrier and Spafford present a process model for the digital investigation process. [9]. They introduce the model of a digital crime scene and outline the relationships between the digital and physical crime-scene investigation processes. The challenges posed by file system forensics have been widely discussed, and the presentation by Shewmaker outlines some key issues, focusing on the *ext* and *ufs* file systems [10]. The guide by Grundy [11] provides a good overview of digital forensic tools and techniques, with an emphasis of Linux and the Sleuth Kit [12]. Kiley, Shinbara, and Rogers present methods for obtaining digital evidence from Apple's *iPod* devices [13]. Gupta, Hoeschele, and Rogers describe the forensic challenges posed by hidden areas on disk, such as the Host Protected Areas (HPA) and Device Configuration Overlays (DCO) [14]. Purcell and Lang describe some filesystem artifacts specific to Microsoft Windows Vista, such as folders named using an account's security identifier (SID) in the *recycle bin*, and the thumbnail cache [15]. The book by Carrier provides a comprehensive overview of forensic analysis of filesystems [16].

Mead presents an overview [3] of the NIST National Software Reference Library's Reference Data Set (NSRL RDS) and its use in helping investigators separate known files from unknown files requiring further analysis. The need for approximate matches in this context is highlighted in the presentation by White [17], which describes the use of hash values computed for 4096-byte blocks of files. The value 4096 was determined to be a good choice for the dataset, tools, and statistical properties in this environment. White reports that when using whole-file hashes, approximately 30% of the disk requires human examination. The use of block hashes reduces this number to approximately 15%.

Locality-sensitive hashing, as used in Section III-D was developed by Gionis, Indyk, and Motwani [7]. Tridgell's thesis describes the design of the *rsync* algorithm that introduced the methods for blockwise hashing used in Sections III-A and III-B [18]. Further refinements are found in the implementation of the *spamsum* program [5]. Although the description of blockwise hashing in this paper used MD5 signatures for block hashes, simpler and faster hashes such as the popular Fowler-Noll-Vo hash [19] may be effective replacements. Kornblum [20] describes the use of context-triggered piecewise hashes (essentially, the method of Section III-B) and its implementation in the *ssdeep* program [21]. His work shares many of the motivations and key ideas with the work in this paper. However, one important difference is that the method of Section III-D allows us to avoid the exhaustive search over the reference collection of signatures and efficiently locate the likely matches. Roussev et al. [22] describe an interesting application of Bloom filters to storing a compressed representation of the reference collection of signatures. By allowing the possibility of a few false positive matches, the reference collection is substantially compressed using a standard Bloom-filter encoding.

There is a large body of work on computing the edit distance between strings, as used in Section III-C. In particular, the widely available GNU *diff* program [23] is based on an algorithm by Myers [24] that runs in time $O(nd)$ where n is the size of the input and d is the length of the edit script (output). By using this, or a simlar, output sensitive algorithm and limiting it to small values of d, it should be possible to substantially increase the speed of signature matching during exhaustive search (either in the method of Section III-C or in the very last stage of the method of Section III-D).

Liu, Zhang, and Zeng present a method to detect money-laundering activity by matching sequences of transactions to a reference collection of sequences [25]. While the application domain is quite different from the one in this paper, it should be interesting to study the adaptation of some of their methods for our purposes. Zeng and Li have found tags to be effective in improving similarity calculations in a user-based Web page recommendation framework [26]. Their work suggests that it may be fruitful to investigate whether a similar distributed, user-based scheme may be applicable to whitelisting files that are newly created or that change frequently and therefore cannot be addressed by infrequently updated lists [3]. In general, there are interesting connections to be made between cybercrime and social computing [27]: While social computing gives rise to new kinds of cybercrime, it also provides new opportunities, such as distributed detection, processing, and tagging, to address cybercrime.

V. Conclusion

We motivated the need for methods that assist an investigator in quickly locating the files in a target system that are most likely to yield useful information. The use of whitelists, lists of attributes of well known files published by a trusted authority, is valuable in this regard, but its effectiveness is often hampered by the lack of suitable matching techniques. In particular, there is a need for tools that are robust to small changes in files and that efficiently determine not only the exact matches between target systems and whitelists but also approximate matches; otherwise, an adversary may easily overwhelm the system. We highlighted and formalized the problem requirements and parameters as two key problems: bounded best-match and approximate bounded near-match. We described methods for solving these problems and discussed several variations. A key contribution is the method for efficiently locating approximate matches to a target file using locality-sensitive hashing.

The method of Section III-D maps composite file signatures to a high-dimensional metric space. This mapping affords some flexibility in how bits of the signature are mapped to dimensions. Instead of mapping the component signature of each block (with block boundaries dynamically determined as in Section III-B), we may map the concatenated component signatures of several adjacent blocks to a single dimension. The reduction in the dimensionality of the

resulting nearest-neighbor problem is likely to improve efficiency. However, that improvement must be weighed against the potential loss of efficiency due to an increase in the length of the unary representation conceptually produced by h_1. Although this unary representation need not (and should not) be explicit in an implementation, a larger representation results in slower hashing in general. These and other details related to the mapping are part of our continuing work on this problem.

For simplicity, this paper has described the analysis of the storage system (typically hard disks) at the filesystem level. However, much valuable information may be gleaned by examining the storage system at lower levels of abstraction, such as logical and physical volumes, raw disk devices accessed through the controller, and physical media (platters) analyzed using specialized hardware. For example, by accessing the raw disk device (through the controller) we may uncover data that has been erased at the filesystem level. Similarly, by reading physical media on special hardware we may uncover data that has been erased at the controller level. Although we do not address these complex tasks directly, some aspects of our methods may be applicable in these situations. For example, raw disk sectors recovered from an analysis of physical media may be matched to a whitelist of sector-sized portions of well-known files.

It is natural to consider the dual problem of blacklisting: matching files in the target system to a list of known malevolent, or otherwise notable, files. Certainly this problem has received much attention in the context of anti-virus software and similar tools. However, there are some important differences in a forensic environment. Some, such as the ability to study a frozen system, provide new opportunities, while others, such as the need to preserve the original state, make the problem harder. Also, while it is reasonable to assume a whitelist that changes infrequently and in a controlled manner, with changes initiated and monitored by trusted entities, blacklists must change frequently and unpredictably, in response to newly discovered malware.

In continuing work, we are developing a software toolkit based on these methods. In addition to potentially assisting other researchers and practitioners, an important goal of this toolkit is to enable large-scale studies of diverse whitelists and target systems. In particular, it should be interesting to conduct a long-term study of how whitelists, and target systems, both benign and adversarial, co-evolve. Although not a primary objective, this toolkit is also likely to provide some interesting datasets for the study of the approximate nearest-neighbor problem in high-dimensional spaces.

REFERENCES

[1] Ronald L. Rivest, "The MD5 message-digest algorithm," IETF Network Working Group. Request for Comments 1321. http://www.ietf.org/, Apr. 1992.

[2] "Secure hash standard," FIPS PUB 180-1. National Institute of Standards and Technology. U.S. Department of Commerce, Apr. 1995. [Online]. Available: http://csrc.nist.gov/publications/

[3] Steve Mead, "Unique file identification in the national software reference library," http://www.nsrl.nist.gov/Documents/analysis/, May 2005.

[4] Andrew Tridgell and Paul Mackerras, "The rsync algorithm," Department of Computer Science, Australian National University, Canberra, ACT 0200, Australia, Tech. Rep., 1998.

[5] Andrew Tridgell, "`spamsum` source code and documentation," http://www.samba.org/ftp/unpacked/junkcode/spamsum/, 2002.

[6] David Sankoff and Joseph B. Kruskal, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.

[7] Aristides Gionis, Piotr Indyk, and Rajeev Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, Edinburgh, Scotland, Sept. 1999.

[8] Paolo Ciaccia, Marco Patella, and Pavel Zezula, "A cost model for similarity queries in metric spaces," in *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS)*, Seattle, Washington, June 1998, pp. 59–68.

[9] Brian Carrier and Eugene H. Spafford, "Getting physical with the digital investigation process," *International Journal of Digital Evidence*, vol. 2, no. 2, pp. 1–20, Fall 2003, 2003.

[10] James Shewmaker, "File system forensics," Presentation notes. http://www.bluenotch.com/resources/, May 2008.

[11] Barry J. Grundy, "The law enforcement and forensic examiner's introduction to Linux," http://www.linuxleo.com/, Dec. 2008, version 3.78.

[12] Brian Carrier *et al.*, "The Sleuth Kit tool overview," http://wiki.sleuthkit.org/, Dec. 2008.

[13] Matthew Kiley, Tim Shinbara, and Marcus Rogers, "iPod forensics update," *International Journal of Digital Evidence*, vol. 6, no. 1, pp. 1–9, Spring 2007.

[14] Mayank R. Gupta, Michael D. Hoeschele, and Marcus K. Rogers, "Hidden disk areas: HPA and DCO," *International Journal of Digital Evidence*, vol. 5, no. 1, pp. 1–8, Fall 2006.

[15] Daniel M. Purcell and Sheau-Dong Lang, "Forensic artifacts of Microsoft Windows Vista system," in *Pacific Asia Workshop on Cybercrime and Computer Forensics (PACCF)*, Taipei, Taiwan, June 2008, pp. 304–319.

[16] Brian Carrier, *File System Forensic Analysis*. Addison-Wesley, 2005.

[17] Douglas White, "Hashing of file blocks: When exact matches are not useful," Presentation notes, American Academy of Forensic Sciences 60th Anniversary Meeting. http://www.nsrl.nist.gov/Presentations.htm, Feb. 2008.

[18] Andrew Tridgell, "Efficient algorithms for sorting and synchonization," Ph.D. dissertation, The Australian National University, Feb. 1999.

[19] Landon Curt Noll, "Fowler/Noll/Vo hash," http://isthe.com/chongo/, 2003.

[20] Jesse Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3S, pp. S91–S97, 2006.

[21] ——, "`ssdeep` source code and documentation," http://ssdeep.sourceforge.net/, Jan. 2009, version 2.1.

[22] Vassil Roussev, Yixin Chen, Timothy Bourg, and Golden G. Richard III, "`md5bloom`: Forensic filesystem hashing revisited," *Digital Investigation*, vol. 3S, pp. S82–S90, 2006.

[23] Mike Haertel, David Hayes, Richard Stallman, Len Tower, Paul Eggert., and Wayne Davison, "The GNU diff program," Texinfo system documentation, 1998, available through anonymous FTP at `prep.ai.mit.edu`.

[24] E. Myers, "An O(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 2, pp. 251–266, 1986.

[25] Xuan Liu, Pengzhu Zhang, and Dajun Zeng, "Sequence matching for suspicious activity detection in anti-money laundering," in *Proceedings of the Pacific Asia Workshop on Intelligence and Security Informatics (PAISI)*, Taipei, Taiwan, June 2008, pp. 50–61.

[26] Daniel Zeng and Huiqian Li, "How useful are tags?—an empirical analysis of collaborative tagging for Web page recommendation," in *Proceedings of the Workshop on Social Computing (SOCO)*, June 2008, pp. 320–330.

[27] Danel Zeng, Sheau-Dong Lang, Raymond Hsieh, Michael Chau, and Christopher C. Yang, "Cybercrime and social computing," in *Panel at the IEEE International Conference on Intelligence and Security Informatics (ISI)*, Taipei, Taiwan, June 2008.