available at www.sciencedirect.com

SCIENCE DIRECT®

journal homepage: www.elsevier.com/locate/diin

Digital Investigation

Identifying almost identical files using context triggered piecewise hashing

Jesse Kornblum

ManTech SMA

Keywords:
Memory analysis
Forensics
Windows
Reverse engineering
Microsoft

ABSTRACT

Homologous files share identical sets of bits in the same order. Because such files are not completely identical, traditional techniques such as cryptographic hashing cannot be used to identify them. This paper introduces a new technique for constructing hash signatures by combining a number of traditional hashes whose boundaries are determined by the context of the input. These signatures can be used to identify modified versions of known files even if data has been inserted, modified, or deleted in the new files. The description of this method is followed by a brief analysis of its performance and some sample applications to computer forensics.

© 2006 DFRWS. Published by Elsevier Ltd. All rights reserved.

1. Introduction

This paper describes a method for using a context triggered rolling hash in combination with a traditional hashing algorithm to identify known files that have had data inserted, modified, or deleted. First, we examine how cryptographic hashes are currently used by forensic examiners to identify known files and what weaknesses exist with such hashes. Next, the concept of piecewise hashing is introduced. Finally a rolling hash algorithm that produces a pseudo-random output based only on the current context of an input is described. By using the rolling hash to set the boundaries for the traditional piecewise hashes, we create a Context Triggered Piecewise Hash (CTPH). Such hashes can be used to identify ordered homologous sequences between unknown inputs and known files even if the unknown file is a modified version of the known file. We demonstrate the spamsum algorithm, a CTPH implementation, and briefly analyze its performance using a proof of concept program called ssdeep.

The algorithm explained in the remainder of this paper was adapted from a spam email detector called spamsum (Andrew, 2002) written by Dr. Andrew Tridgell. Spamsum can identify emails that are similar but not identical to samples of known spam. The spamsum algorithm was in turn based upon the rsync checksum (Tridgell, 1999) also by Dr. Tridgell. Although the application of this algorithm to computer forensics is new, the author did not develop the spamsum algorithm. This paper explains the new application and analyzes its effectiveness.

2. Background

Computer forensic examiners are often overwhelmed with data. Modern hard drives contain more information that cannot be manually examined in a reasonable time period creating a need for data reduction techniques. Data reduction techniques aim to draw the examiner's attention to relevant data and minimize extraneous data. For example, a common word processing application is not worth examining, but a known malicious program should be highlighted.

E-mail address: jesse.kornblum@mantech.com
1742-2876/$ – see front matter © 2006 DFRWS. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2006.06.015

To date, forensic examiners have used cryptographic hashing algorithms such as MD5 and SHA-1 for data reduction (White, 2005). These algorithms take an input of arbitrary size and produce a fixed-length value corresponding to that input. Cryptographic hashes have many properties, but forensic examiners take advantage of two of them in particular. First, if even a single bit of the input is changed, the output will be radically different. Second, given an input and its hash, it is computationally infeasible to find another input that produces the same hash.

These two properties can be used to identify known files in sets of unknown files. An examiner gathers a set of known files, computes their cryptographic hash values, and stores those values. During future investigations, the examiner can compute the hash values for every file in the investigation and compare those hash values to the known values computed previously. If any of the new hash values match the known values, the investigator has almost certainly found the known files (White, 2005). If the file in question is a known good file, the file can be eliminated from consideration. If it is a known bad file, the examiner has a new investigative lead to follow.

Once sets of hashes have been created they can be passed from one investigator to another. The National Institute of Standards and Technology, the U.S. Department of Justice (White, 2005), and a number of forensic software vendors have all established repositories of cryptographic hashes for this purpose.

Malicious users can frustrate this technique, however, by making even a one-bit change to known files (Foster and Liu, 2005). As noted above, changing even a single bit of the input changes the cryptographic hash of the file. For example, altering a string found in most Microsoft Windows programs, from "This program cannot be run in DOS mode" to "This program cannot be run *on* DOS mode" (emphasis added), will radically alter the hash for that file. As a result, systems that use sets of known cryptographic hashes cannot match this file to the original.

Files with one-bit changes are almost entirely identical and share a large ordered homology. Borrowing from genetics, two chromosomes are homologous if they have identical sequences of genes in the same order. Similarly, two computer files can have ordered homologous sequences if they have large sequences of identical bits in the same order. The two files are identical except for a set of insertions, modifications, and deletions of data.

In practice, these almost identical files could be a Microsoft Word document and an edited version of that document or a JPEG and a truncated version of that JPEG. Such files would have different cryptographic hashes and could not be identified as homologous using algorithms such as MD5. Although the human eye can detect the similarity between the two, there is currently no automated method to do so.

3. Context triggered piecewise hashes

3.1. Piecewise hashing

Originally developed by Nicholas Harbour for dcfldd (Harbour, 2002), piecewise hashing uses an arbitrary hashing algorithm to create many checksums for a file instead of just one. Rather than to generate a single hash for the entire file, a hash is generated for many discrete fixed-size segments of the file. For example, one hash is generated for the first 512 bytes of input, another hash for the next 512 bytes, and so on. See Fig. 1 for a set of sample piecewise hashes. The technique was originally developed to mitigate errors during forensic imaging. If an error occurred, only one of the piecewise hashes would be invalidated. The remainder of the piecewise hashes, and thus the integrity of the remainder of the data, was still assured.

Piecewise hashing can use either cryptographic hashing algorithms, such as MD5 in dcfldd or more traditional hashing algorithms such as a Fowler/Noll/Vo (FNV) hash. Regardless of the algorithm, for the purposes of this paper the algorithm used to compute the piecewise hashes is called the *traditional hash* to distinguish it from the *rolling hash* described below.

3.2. The rolling hash

A rolling hash algorithm produces a pseudo-random value based only on the current context of the input. The rolling hash works by maintaining a state based solely on the last few bytes from the input. Each byte is added to the state as it is processed and removed from the state after a set number of other bytes have been processed.

Assuming we have an input of n characters, we say the ith byte of the input is represented by b_i. Thus, the input as a whole consists of bytes b_1, b_2,..., b_n. At any position p in the input, the state of the rolling hash will depend only on the last s bytes of the file. Thus, the value of the rolling hash, r, can be expressed as a function of the last few bytes as shown in Eq. (1).

$$r_p = F(b_p, b_{p-1}, b_{p-2}, \ldots, b_{p-s}) \tag{1}$$

The rolling hash function F is constructed so that it is possible to remove the influence of one of the terms. Thus, given r_p, it is possible to compute r_{p+1} by removing the influence of b_{p-s}, represented as the function $X(b_{p-s})$, and adding the influence of b_{p+1}, represented as the function $Y(b_{p+1})$, as seen in Eqs. (2) and (3).

$$r_{p+1} = r_p - X(b_{p-s}) + Y(b_{p+1}) \tag{2}$$

$$r_{p+1} = F(b_{p+1}, b_p, b_{p-1}, \ldots, b_{(p-s)+1}) \tag{3}$$

3.3. Combining the hash algorithms

Whereas current piecewise hashing programs such as dcfldd used fixed offsets to determine when to start and stop the traditional hash algorithm, a CTPH algorithm uses the rolling hash. When the output of the rolling hash produces a specific

```
0 - 512: 24a56dad0a536ed2efa6ac39b3d30a0f
512 - 1024: 0bf33972c4ea2ffd92fd38be14743b85
1024 - 1536: dbbf2ac2760af62bd3df3384e04e8e07
```

Fig. 1 – Sample piecewise MD5 hashes.

output, or trigger value, the traditional hash is triggered. That is, while processing the input file, one begins to compute the traditional hash for the file. Simultaneously, one must also compute the rolling hash for the file. When the rolling hash produces a trigger value, the value of the traditional hash is recorded in the CTPH signature and the traditional hash is reset.

Consequently, each recorded value in the CTPH signature depends only on part of the input, and changes to the input will result in only localized changes in the CTPH signature. For instance, if a byte of the input is changed, at most two, and in many cases, only one of the traditional hash values will be changed; the majority of the CTPH signature will remain the same. Because the majority of the signature remains the same, files with modifications can still be associated with the CTPH signatures of known files.

The remainder of this paper demonstrates one implementation of a CTPH called the spamsum algorithm in honor of its origin. This algorithm is implemented in the program ssdeep to be published concurrently with this paper at `http://ssdeep.sourceforge.net/`.

4. The spamsum algorithm

The spamsum algorithm uses FNV hashes for the traditional hashes (Andrew, 2002) which produce a 32-bit output for any input (Noll, 2001). In spamsum, Dr. Tridgell further reduced the FNV hash by recording only a base64 encoding of the six least significant bits (LS6B) of each hash value (Andrew, 2002).

The algorithm for the rolling hash was inspired by the Alder32 checksum (Andrew, 2002) and pseudocode for it is in Fig. 2. The original spamsum algorithm had an additional logical AND with `0xffffffff` following the shift left which has been deleted. This AND statement originally limited the values in the algorithm to 32-bit values. Because the author has explicitly stated that all values in the algorithm are unsigned 32-bit values, the AND statement has no effect and has been omitted.

Before processing the input file, we must choose a trigger value for the rolling hash. In the spamsum algorithm, and thus for the remainder of this paper, the trigger value will be referred to as the block size. Two constants, a minimum

x, y, z, and c are unsigned 32-bit values initialized to zero. *window* is an array of *size* unsigned 32-bit values all of which are initialized to zero.

To update the hash for a byte d:

$y = y - x$
$y = y + size * d$
$x = x + d$
$x = x - window\ [c \bmod size]$
$window\ [c \bmod size] = d$
$c = c + 1$
$z = z \ll 5$
$z = z \oplus d$
return $(x + y + z)$

Fig. 2 – Pseudocode for rolling hash.

block size, b_{min}, and a spamsum length, S, are used to set the initial block size for an input of n bytes using Eq. (4). The block size may be adjusted after the input is processed under certain conditions defined below. Thus, the block size computed before the input is read is called the initial blocksize, b_{init}.

$$b_{init} = b_{min} 2^{\lfloor \log_2 (\frac{n}{Sb_{min}}) \rfloor} \tag{4}$$

After each byte of the input is processed, the rolling hash is updated and the result is compared against the block size. If the rolling hash produces a value that is equal, modulo the block size, to the block size minus one, the rolling checksum has hit a trigger value. At such time a base64 encoded value of the LS6B of the traditional hash is appended to the first part of the final signature. Similarly, when the rolling checksum produces a value that is equal, modulo twice the block size, to twice the block size minus one, a base64 encoded value of the LS6B of the traditional hash is appended to the second part of the spamsum hash. Note that two separate states of the traditional hash are maintained, one for each block size calculation.

After every byte of the input has been processed, the final signature is examined. If the first part of the signature is not long enough after all of the input is processed, the block size is halved and the input is processed again.

The final spamsum signature consists of the block size, the two sets of LS6Bs, and the input's filename in quotes. The first set of LS6Bs is computed with block size b and the other $2b$. Psuedocode for the spamsum algorithm is shown in Fig. 3 and a sample spamsum signature is shown in Fig. 4.

5. Comparing spamsum signatures

Two spamsum signatures can be compared to determine if files from which they were derived are homologous. The examination looks at the block size, eliminates any sequences, and then computes a weighted edit distance between them as defined below. The edit distance is scaled to produce a match score, or a conservative weighted measure of the ordered homologous sequences found in both files.

Because the triggers for the traditional hash are based upon the input file and the block size, only signatures with an identical block size can be compared. The spamsum algorithm generates signatures for each input based on block sizes b and $2b$, so it is possible to compare two signatures if the block sizes given in the signatures are within a power of a two. For example, with two signatures, the first with a block size of b_x and the second with b_y, the first signature has CTPH values for block sizes b_x and $2b_x$, the second for y and $2b_y$. We can compare these two signatures if $b_x = b_y$, $2b_x = b_y$, or $b_x = 2b_y$.

After the block sizes have been resolved, any recurring sequences are removed. These sequences indicate patterns in the input file and usually do not convey much information about the content of the file (Andrew, 2002). Finally, the

```
b = compute_initial_block_size(input)

done = FALSE
while (done = FALSE) {
    initialize_rolling_hash(r)
    initialize_traditional_hash(h1)
    initialize_traditional_hash(h2)
    signature1 = ""
    signature2 = ""

    foreach byte d in input {
        update_rolling_hash(r, d)
        update_traditional_hash(h1, d)
        update_traditional_hash(h2, d)
        if (get_rolling_hash(r) mod b = b − 1) then {
            signature1 += get_traditional_hash(h1) mod 64
            initialize_traditional_hash(h1)
        }
        if (get_rolling_hash(r) mod (b * 2) = b * 2 − 1) then {
            signature2 += get_traditional_hash(h2) mod 64
            initialize_traditional_hash(h2)
        }
    }

    if length (signature1) < S/2 then
        b = b/2
    else
        done = TRUE
}

signature = b + ":" + signature1 + ":" + signature2
```

Fig. 3 – Pseudocode for the spamsum algorithm.

weighted edit distance between the two hashes is computed using dynamic programming. Given two strings s1 and s2, the edit distance between them is defined as "the minimum number of point mutations required to change s1 into s2", where a point mutation means either changing, inserting, or deleting a letter (Allison, 1999). The spamsum algorithm uses a weighted version of the edit distance formula originally developed for the USENET newsreader trn (Andrew, 2002). In this version, each insertion or deletion is weighted as a difference of one, but each change is weighted at three and each swap (i.e. the right characters but in reverse order) is weighted at five. For clarity, this weighted edit distance is referenced as $e(s_1,s_2)$ for signatures s_1 and s_2 of length l_1 and l_2, respectively, and is shown in Eqs. (5)–(7). In these equations, i is the number of insertions, d is the number of deletions, c is the number of changes, and w is the number of swaps.

$$e = i + d + 3c + 5w \tag{5}$$

$$c + w \leq \min(l_1, l_2) \tag{6}$$

$$i + d = |l_1 - l_2| \tag{7}$$

The edit distance is then rescaled from 0–64 to 0–100 and inverted so that zero represents no homology and 100 indicates almost identical files. The final match score, M, for strings of length l_1 and l_2 can be computed using Eq. (8). Note that when $S = 64$, which is the default, that the S and 64 terms cancel.

The match score represents a conservative weighted percentage of how much of s_1 and s_2 are ordered homologous sequences. That is, a measure of how many of the bits of these two signatures are identical and in the same order. The higher the match score, the more likely the signatures came from a common ancestor and the more likely the source files for those signatures came from a common ancestor. A higher match score indicates a greater probability that the source files have blocks of values in common and in the same order.

$$M = 100 - \left(\frac{100Se(s_1, s_2)}{64(l_1 + l_2)}\right) \tag{8}$$

The CTPH proof of concept program, ssdeep (Kornblum, 2006b), indicates any two files with a match score greater than zero as matching. During anecdotal testing the author did not find any dissimilar files that had a match score greater than zero, but further research should be conducted in this area.

It is possible that two files can have identical CTPH signatures but still be different files. If a modification is made to a file that does not affect when the rolling hash triggers the recording of the traditional hash, the value of the traditional hash for that piecewise segment may be different. But because the value of the traditional hash is reduced from a 32-bit value to a 6-bit value, by the pigeonhole principle, many values from the traditional hash will map to the same value recorded in the signature. Specifically, there is a 2^{-6} probability that this new block will have the same CTPH signature as the original file. Thus, even if two files have the same spamsum signature, there is no proof that they are byte for byte identical and the examiner must further examine the files in question. Granted, such a discrepancy can easily be resolved using cryptographic hashing algorithms such as MD5. Given that such algorithms can indicate for certain if the files are identical and run much faster than CTPH, they should probably be computed first regardless.

5.1. *Comparing identical files*

Because the spamsum algorithm is completely deterministic, identical files will produce the same spamsum hash. Since the hashes match exactly they will have an edit distance of zero. When scaled they have a match score of 100 – a perfect match.

```
1536:T0tUHZbAzIaFG91Y6pYaK3YKqbaCo/6Pqy45kwUnmJrrevqw+oWluBY5b32TpC0:
T0tU5s7ai6ptg7ZNcqMwUArKvqfZlMC0,"/music/Dragostea Din Tei.mp4"
```

Fig. 4 – Sample spamsum signature, line breaks added.

5.2. Without changing triggers

If a change is made in the input file such that none of the contexts that trigger the rolling hash are altered, one of the traditional hash values will still be changed. As there are only 64 possible values of the traditional hash, there is still a 2^{-6} possibility that this new file will have the same signature as the original file. Even if the signatures are different, they will only differ by one character and therefore have an edit distance of one. Applying the formula from Eq. (8), we can compute the final match score for these two files as $(100-(100/(l_1+l_2)))$.

5.3. Changing triggers

If a modification is made in the input file such that one of the context triggers is changed, at most two of the traditional hash values will be changed. First, the rolling hash will trigger earlier than in the original file, changing the current value of the traditional hash. Then, because the computation of the next value of the traditional hash started at a different point in the input file, the next value of the traditional hash will most likely be changed too. Finally, for each of those values of the traditional hash there is a 2^{-6} probability that the value of the traditional hash will be the same. Thus, the odds of a file having a modification that affects a trigger point producing the same spamsum signature as the original file are 2^{-12}.

Even if two piecewise hash values are different in the final CTPH signature, this new signature will have an edit distance of two from the original signature and a match score of $(100-(98/(l_1+l_2)))$.

5.4. Random files

Two completely random files will match if the edit distance between them is small enough. The author demonstrated earlier that the odds of any two characters in the signature being the same is 2^{-6}. For two signatures of length l_1 and l_2 where $l=\min(l_1,l_2)$, the odds of a match score of 100 are therefore $(2^{-6})^l$. In the case where $l=32$, for example, the odds of an exact match are 2^{-192} and is thus highly unlikely.

6. Performance issues

Because the input may need to be processed more than once to compute a CTPH, the running time for CTPH programs may be considerably longer than comparable cryptographic hashing programs. As an example, the author generated a few files of pseudo-random data using `dd` and `/dev/urandom` on a computer running Fedora Linux. These files were then hashed using both cryptographic hashing programs (Kornblum, 2006a) and ssdeep. The running times are recorded in Table 1. Note that ssdeep was significantly slower than the cryptographic hashing algorithms with the only exception of Whirlpool. It should also be noted that two files of equal length may require different running times in a CTPH program because of the number of times the input must be processed. Assuming that a single CTPH processing of the input takes $O(n)$ time, the block size may need to be adjusted $O(\log n)$ times, making the total running time $O(n \log n)$.

Table 1 – Time to hash pseudo-random data

Algorithm	1 MB	10 MB	50 MB
MD5	9	49	223
SHA256	24	184	897
Ssdeep	71	669	6621
Whirlpool	156	1505	7518

All times are measured in milliseconds.

When matching sets of CTPH, the program must compare the signature for the unknown file to each known signature. Cryptographic hashes can be sorted, or ironically enough, put into a hash table, to reduce the time for matching a given hash to an unknown hash to $O(\log n)$ where n is the length of the hash, usually a constant, so $O(1)$. Because every known CTPH must be compared with each unknown hash, the edit distance must be computed for each pair. Computing the edit distance takes $O(l^2)$ time, where l is variable but limited to a constant. The time required to compare a single CTPH signature against a set of n known signatures is thus $O(n)$.

7. Applications

The CTPH technique described in this paper is quite applicable to computer forensics. This section gives some examples of these applications and a brief analysis of the results. The proof of concept program developed for this paper, ssdeep (Kornblum, 2006b), was used to generate all of the results below.

7.1. Altered document matching

CTPH can be used to identify documents that are highly similar but not identical. For example, the author constructed a document containing the 272 words of the "Bliss" version of Abraham Lincoln's Gettysburg Address (Lincoln, 1953) saved in Microsoft Word format. The author recorded a CTPH signature for this document. The author then changed all of the text in the document to another size and font. Two paragraphs of new text were inserted at the beginning of the file and one of them was put into a different color. The author made numerous insertion and deletions, for example, (changes emphasized) "Four score and *like* seven years ago our fathers brought forth on this continent, a new *like* nation, conceived in Liberty *and stuff*." The author reversed every clause regarding consecration ("But, in a larger sense, we must dedicate – we must consecrate – we must hallow – this ground.") and the final clause about how a government of the people shall not perish from the Earth did, in fact, perish. Finally, the author appended the phrase "I AM THE LIZARD KING!" a few dozen times to the end of the document. The ssdeep program was able

```
$ ssdeep -b gettysburg-address.doc > sigs.txt

$ ssdeep -bm sigs.txt gettysburg-modified.doc
gettysburg-modified.doc matches gettysburg-address.doc (57)
```

Fig. 5 – Matching the Gettysburg Address.

to generate a match between the modified document and the original as seen in Fig. 5.

7.2. *Partial file matching*

Another application of CTPH technology is partial file matching. That is, for a file of *n* bits, a second file is created containing only the first *n*/3 bits of the original file. A CTPH signature of the original file can be used to match the second file back to the first. This simulates matching the partial files found during file carving to known files. Additionally, CTPH can be used to match documents when only footers are available. That is, if a new file is created with only the last *n*/3 bits of the original, this file can also be matched back to a known file using CTPH. The reader will see in Fig. 6 how a sample file of 313,478 bytes was used to generate two partial files. The first partial file contained the first 120,000 bytes of the original file and second partial file contained the last 113,478 bytes of the original file. Both of these new files can be matched back to the original using ssdeep, as seen in Fig. 6.

The cutoff of *n*/3 bytes comes from the block size comparison. Below this level there are still matches, but the block sizes are too different to make a meaningful comparison. That is, a smaller block size is needed to generate more bits for the signature in the smaller file. Given that there are no common block sizes in the signatures, they cannot be compared.

The comparison of partial files, especially footers, is significant as it represents a new capability for forensic examiners. A JPEG file, for example, missing its header cannot be displayed under any circumstances. Even if it contained a picture of interest, such as child pornography, the examiner would not be able to determine its content. By using CTPH to detect the homology between unviewable partial files and known file, the examiner can develop new leads even from files that cannot be examined conventionally.

8. Conclusion

Context triggered piecewise hashing is a powerful new method for computer forensics. It will enable examiners to associate files that previously would have been lost in vast quantities of data that now make up an investigation. By creating associations between files that are homologous but not identical, investigators will be able to quickly find relevant pieces of material in new investigations. Although CTPH is not a new technology, its application to computer forensics represents a step beyond the traditional use of cryptographic hashing.

Acknowledgments

This research was inspired by Greg Fricke. The author is grateful to ManTech's Computer Forensics and Intrusion Analysis Group for providing the resources and time necessary to complete this project. Special thanks to Dr. Andrew Tridgell for developing and freely releasing the spamsum algorithm, without which this paper would not exist. Invaluable assistance was provided by Devin Mahoney, Greg Hall, Larissa O'Brien, Reid Leatzow and Ben McNichols. Extra special thanks to S —.

```
$ ls -l sample.jpg
-rw-rw---- 1 usernm usernm 313478 Apr 11 11:05 sample.jpg

$ dd if=sample.jpg of=first bs=1 count=120000
100000+0 records in
100000+0 records out

$ dd if=sample.jpg of=last bs=1 skip=200000
113478+0 records in
113478+0 records out

$ file first last
first: JPEG image data, JFIF standard 1.01
last: data

$ ssdeep -b sample.jpg > sigs

$ ssdeep -bm sigs first last
first matches sample.jpg (54)
last matches sample.jpg (65)
```

Fig. 6 – Matching partial files.

REFERENCES

Allison Llyod. Dynamic programming algorithm (DPA) for edit-distance. Monash University. Available from: http://www.csse.monash.edu.au/~simlloyd/tildeAlgDS/Dynamic/Edit/; 1999.

Foster James C, Liu Vincent T. Catch me, if you can. Blackhat Briefings. Available from: http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-foster-liu-update.pdf; 2005.

Harbour Nicholas. Dcfldd. Defense Computer Forensics Lab. Available from: http://dcfldd.sourceforge.net/; 2002.

Kornblum Jesse. md5deep. Available from: http://md5deep.sourceforge.net/; 2006a.

Kornblum Jesse. Ssdeep. Available from: http://ssdeep.sourceforge.net/; 2006b.

Lincoln Abraham. Collected works of Abraham Lincoln. In: Basler Roy P, editor. The Abraham Lincoln Association; 1953.

Noll Landon C. Fowler/Noll/Vo Hash. Available from: http://www.isthe.com/chongo/tech/comp/fnv/; 2001.

Tridgell Andrew. Efficient algorithms for sorting and synchronization. PhD thesis. Canberra, Australia: Department of Computer Science, The Australian National University; 1999.

Tridgell Andrew. Spamsum README. Available from: http://samba.org/ftp/unpacked/junkcode/spamsum/README; 2002.

White Douglas. NIST National Software Reference Library. National Institute of Standards and Technology. Available from: http://www.nsrl.nist.gov/; 2005.

Jesse Kornblum is a Principal Computer Forensics Engineer for ManTech SMA's Computer Forensics and Intrusion Analysis Group. Based in the Washington DC area, his research focuses on computer forensics and computer security. He has authored a number of computer forensics tools including the widely used md5deep suite of cryptographic hashing programs and the First Responder's Evidence Disk. A graduate of the Massachusetts Institute of Technology, Mr. Kornblum has also served as a Computer Crime Investigator for the Air Force Office of Special Investigations, an instructor in the U.S. Naval Academy's Computer Science Department, and as the Lead Information Technology Specialist for the Department of Justice Computer Crime and Intellectual Property Section. According to Mr. Kornblum, the most useful animals are members of the *Capra* family.