MODIFIED BLOOM FILTER CASE STUDIES

Real-World examples for Digital Forensics.

/ INTERMEDIATE

DRDC | RDDC

Unfortunately, due to a small formatting problem, there was an error in Part 1 of this article in Issue 24 – found at the bottom of the 3rd column on Page 14.

To calculate the ideal number of elements (m) in the bit array:
- **Where p is the probability of a false positive**
- **To determine the rate of false positives**
- **To calculate the ideal number of hashes**

CORRECTION:
The following equations may help in working out the various tweaks:
1. **To calculate the ideal number of elements (m) in the bit array:**
   m = -n ln p / (ln 2)^2
   Where p is the probability of a false positive
2. **To determine the rate of false positives:**
   ln p = -m (ln 2)^2 / n
3. **To calculate the ideal number of hashes:**
   K = m (ln 2) / n

In the previous article, we looked at modified Bloom filters and piecewise hashing (PWH) in terms of what they are and what they can do for digital forensics. Now it is time for two real-world examples. Both examples will examine the search for malware as this is becoming even more common. It is a never-ending threat to governments, corporations and individuals alike. At the same time, we will also look at the relevance of multithreading and parallelization as it applies to PWH.

Both case studies involved deleted and partially damaged data for recovery. The first uses direct sector-by-sector analysis using the md5deep/hashdeep toolkit while the second uses fuzzy hashing (FH) via ssdeep along with much larger block sizes.

Of course, the tools and techniques we discuss in this article are equally applicable to allocated files. The only reason for actually using them against files would be if one or more files were suspected of being corrupted or damaged.

Many time-consuming analyses were redone until we could find no faster method or tool. We chose to work with Linux because the tools we needed were readily available. SHA1 was used to lower the risk of hash collision, although more complex hashes could be used. We had hoped to examine PWH and parallelization metrics but it was not possible given the complexity of thoroughly explaining and documenting these two case studies. We will instead explore these metrics in our next and final follow-up article concerning this subject.

/ TEST COMPUTER SYSTEM AND INSTALLED SOFTWARE, ABOUT PERMISSIONS AND LIMITATIONS

Computer benchmarking tests were done on an Intel i7 980X 6-core processor (12 with Hyper-threading) equipped with 24 GiB RAM. The system is running a customized installation of Fedora 21 64-bit, outfitted with 5TB of disk space disks. At the time of this writing, the computer is running Linux kernel 4.0.6-200.fc21.x86_64 #1 SMP with GCC version 4.9.2 20150212. No swap, RAID, LVM or SELinux were in use and all real-time anti-virus protection was disabled.

All commands we ran over the course of our two case studies were run as root. This might not apply, given your particular context and level of access to a given system. Certain commands (chattr, mount, etc.) require root-level permission. Where this is not possible, the sudo command can be used in lieu.

We used the latest version of ssdeep (written by Kornblum), 2.13, a precompiled 64-bit Linux executable obtained from the Fedora/Forensics repository. The version of md5deep/hashdeep (the toolkit by Kornblum/Garfinkel which includes sha1deep) used in this article is also the latest version, 4.4, which was installed from the Fedora application repository. Finally, the disktype tool and The Sleuth Kit were at versions 9-15 and 4.1.3, respectively. The repositories used for the installation of the former and latter were from Fedora applications and from Fedora/Forensics, respectively.

In this article, we use the words "chunk size" and "block size" interchangeably. We also use GiB (2^30) in lieu of GB (10^9).

/ CASE STUDY 1 – PIECEWISE HASHING USING SHA1

A while back, an acquaintance brought in a disk needing our attention. He said it was a case involving a drive-by download. He wanted us to validate whether his network protection solution was working correctly. Having imaged the disk we then started working with a copy of the image. Nowhere within it could any evidence of malware be found. He was adamant that something had happened but his contracted-out technical support pulled the plug on the machine before a memory capture could be done. So we were asked to determine if this disk image had in fact been touched by the malware. He told us that he had all the Pcap (raw network) logs and that contract support succeeded in recovering the malware from the logs. Finally, according to the information he had, this malware was self-deleting.

While working on the image we performed file system timelining [1]; after completing this consuming analysis, we discovered that the malware had in fact touched the disk but that it then deleted itself shortly thereafter. Data recovery [2] was attempted as was advanced data carving [2] using the most recent versions of the tools outlined in [2] but all to no avail. Trying to find the deleted malware on disk proved more difficult than we originally anticipated – it made for quite the exercise in frustration.

TOP FACT: BEST HIGH PERFORMANCE LINUX FILE SYSTEM

A subject of Linux flame wars on the web, many benchmarks have been carried out over the years and to date, there is no clear winner with respect to all possible needs. However, for high performance RAIDs under Linux, some are better than others, depending on requirements. We do not recommend software RAID, ever, even though this statement too might dissatisfy some. Where many large data files and a minimum of small ones will be used, XFS is very likely the best choice, and it is a very mature choice at that. JFS, on the other hand, while just as mature and scalable as XFS, does not provide the same high levels of I/O for large files as XFS. Instead, it works well with large and small files alike. Ext4, unlike JFS and XFS, is a highly tweakable file system and works well as it is supported by almost all Linux kernels running out there, but it has certain limitations that the others do not have. Why not Btrfs or ZFS? Btrfs is still not considered "production ready" and ZFS is very robust, but for sheer performance, this is not the right file system; for data integrity, it absolutely is. We recommend reading references 3-7 to learn more about these various file systems.

"IN MANY CASES, TIME CAN BE SAVED BY EXTRACTING UNALLOCATED DISK SPACE, SINCE THIS IS MOST LIKELY WHERE WE WANT TO WORK FROM WHEN RECOVERING DATA."

None of the tools we used were able to identify anything remotely similar to the Pcap-recovered malware. We tried over 10 anti-virus scanners, ran both against the disk image itself and everything recovered from it – nothing was found. We even tried various trialware data recovery tools; they too found nothing. None of the recovered data could be even partially matched to the recovered malware. This exercise had already consumed more time than it should have so we wanted to try one last thing – Bloom filter-based matching.

The disk image (suspect.dd) was 500 GB (500,107,862,016 bytes) in size and the Pcap-recovered malware (malware.bin) was about 120KB (119,560 bytes) in size. The disk image yields 976,773,168 512-byte disk sectors to analyse. However, it would take more than a single hash match to confirm that the malware was still on disk. Thus, the way to confirm that the malware was still on disk, if only in a deleted state, was to hash every disk sector and to attempt to identify hash matches established against the actual malware itself, it too being hashed into smaller chunks.

What we describe below are the various commands used along with a short summary of what was produced by them. Please note that locations /mnt/evidence, /mnt/analysis and /mnt/scratch are each independent Ext4-formatted disks. The first location was used to store the evidence files, the second for the processed data files and the third for temporary storage. We used 512-byte PWH but this number can be replaced by whatever chunk size is most suitable, be it for 1, 2 or 4 KiB (or more) according to the underlying disk sectors. Note, where possible, limit the chunk size to equal or less than the underlying cluster size.

The first thing we did was set the evidence files (malware sample malware.bin and disk image suspect.dd) to immutable to preserve their integrity:

```
Command 1: $ sudo chattr +i /mnt/
evidence/malware.bin
$ sudo chattr +i /mnt/evidence/
suspect.dd
```

With a copy of the malware in hand, we conducted PWH using sha1deep, which completed almost instantaneously generating an output file 26,709 bytes in size with 234 hashes:

```
Command 2: $ sha1deep – p 512 /
mnt/evidence/malware.bin >
/mnt/analysis/malware.bin.
pw.sha1.txt
```

In order to perform PWH of the disk image, after having tried GNU Parallel and other methods of parallelization, we found that using the straightforward approach was best. This generated an output file 126,546,484,060 bytes (117.86 GiB) in size containing 976,773,168 hashes:

```
Command 3: $ sha1deep – p 512 /
mnt/evidence/suspect.dd >
/mnt/analysis/suspect.dd.pw.sha1.txt
```

On our test system, Command #3 took 149 min. 13 sec. to complete.

With the PWH SHA1 text files generated (suspect.dd.pw.sha1.txt and malware.bin.pw.sha1.txt) we extracted the hashes therein, sorted and identified the number of repeated hashes using these commands:

```
Command 4: $ cat /mnt/analysis/
suspect.dd.pw.sha1.txt | awk
'{print $1}' |
sort | uniq – c | sort – r – n –
T /media/scratch >
/mnt/scratch/suspect.dd.pw.sha1.
sorted.uniq.txt
$ mv /mnt/scratch/suspect.dd.pw.
sha1.sorted.uniq.txt
/mnt/analysis

Command 5: $ cat /mnt/analysis/
malware.bin.pw.sha1.txt | awk
'{print $1}' |
sort | uniq – c | sort – r – n | >
/mnt/analysis/malware.bin.
pw.sha1.txt.sorted.uniq.txt
```

Command #4 took 135 min 39 sec. to complete producing a file 5,852,170,305 bytes (5.45GiB) in size containing 119,432,047 hashes while Command #5 was 11,270 bytes in size with 230 hashes (4 less than malware.bin which indicated that there were pattern repetitions therein). The sort parameter – T instructed the tool to use only user-specified temporary space because Command #4 required a huge amount of temporary space.

/ PIECEWISE HASHING

Bloom filters are mathematical constructs that reduce arbitrarily long and complex datasets into various unique encodings. Commonly used to determine whether a given element or set of elements belong to a given set, Bloom filters have become a fast data structure. They are used to represent arbitrary data of varying size and are highly efficient for use in computer memory. Their application is useful for testing disk-based storage to determine if it contains a partial piece or subset of some data set. They are also used to identify the similarity between different but identically sized data sets. Probabilistic in nature, they are susceptible to false positives, but not false negatives. On the other hand, modified Bloom filters, implemented as piecewise hashing, is a technique for breaking up data sets in other manageable-sized chunks and then hashing the chunks using some arbitrary hashing function. In so doing, it may be possible to identify lost, damaged or deleted fragments or pieces of data.

```
Regular file, size 465.8 GiB (500107862016 bytes)
DOS/MBR partition map
Partition 1: 465.8 GiB (500105740288 bytes, 976769024 sectors from 2048, bootable)
  Type 0x07 (HPFS/NTFS)
  NTFS file system
    Volume size 465.8 GiB (500105739776 bytes, 976769023 sectors)
```

Figure 1. Output of command disktype.

Now that we have reduced data files malware.bin.pw.sha1.txt.sorted.uniq.txt and suspect.dd.pw.sha1.sorted.uniq.txt, we must look at them to identify if there are excessively repeated hashes. Typically, these repetitions are repeating disk blocks, usually pattern-based disk sectors including those that are zero-filled. We then inspected these two newly generated files using standard console pages (e.g. more, less, etc.). Then we deleted the excess repetitions (only 1 line being deleted for the former file and 10,000 for the latter – your mileage will undoubtedly vary), using the following commands:

```
Command 6: $ cd /mnt/analysis
$ sed – e '1,1d' < malware.bin.
pw.sha1.sorted.uniq.txt >
malware.bin.pw.sha1.sorted.uniq.
reduced.txt
$ sed – e '1,10000d' < suspect.
dd.pw.sha1.sorted.uniq.txt >
suspect.dd.pw.sha1.sorted.uniq.
reduced.txt
```

The third command took several minutes to complete whereas the first two were instantaneous. Now, in order to match the hashes from file malware.bin against disk image suspect.dd we needed to prepare for using the grep command by removing the first column of data from these two files:

```
Command 7: $ cat suspect.dd.pw.
sha1.sorted.uniq.reduced.txt |
awk '{print $2}' >
suspect.dd.pw.sha1.sorted.uniq.
reduced.txt2
$ cat malware.bin.pw.sha1.sorted.
uniq.reduced.txt |
awk '{print $2}' >
malware.bin.pw.sha1.sorted.uniq.
reduced.txt2
```

The first command took several minutes to complete whereas the second was instantaneous. Now we could actually match one file against the other to find out which hashes match:

```
Command 8: $ grep – f malware.
bin.pw.sha1.sorted.uniq.reduced.
txt2
suspect.dd.pw.sha1.sorted.uniq.
reduced.txt2 > matched.txt
```

This command took 38 sec. to complete and the resulting file was 9,348 bytes in size with 228 matched hashes. Now the final command that was needed to determine where in the disk image these matches occurred was:

```
Command 9: $ grep – f matched.txt
suspect.dd.pw.sha1.txt > offsets.txt
```

Taking 20 min. 3 sec. to complete, the resulting file was 7,068 bytes in size with 228 entries, which upon inspection, were all one next to the other. Remember, the offsets need to be multiplied by 512.

Using our favourite hex editor, we set it to the first offset (after having been corrected for by multiplying by 512), then copied the data until the last offset was reached and then saved it as /mnt/analysis/recovered.bin. We passed our scanners on it only to discover that it was identified as infected by the same malware found for the Pcap-recovered sample. An in-depth analysis revealed that recovered.bin was in fact a Windows PE file but was missing its first few sectors. This follows as several zero-filled disk sectors were encountered while copying out the data.

This study case was quite the adventure and took many hours to complete. Had we conducted piecewise hashing against a much larger disk image it would likely have taken days, not hours to complete. In all, we spent a couple of weeks putting it all together, documenting our actions, one step at a time all the while trying to determine if our commands could be optimized or even parallelized (which they could not be, unfortunately).

Having finished this particular exercise, we remain convinced that using conventional hashing would not have yielded tangible results.

/ SPEEDING THINGS UP A BIT

In many cases, time can be saved by extracting unallocated disk space from the disk image under investigation, since this is most likely where we want to work from when recovering data. Once extracted, we can then perform the aforementioned commands against the new data file. To extract unallocated space we use both

DATA RECOVERY & CARVING

There is a multitude of data recovery solutions available Some really stand out but most are unexceptional. Those we used in [2] in our case studies are among the most popular . However, when performing data recovery and carving they fare poorly when dealing with fragmented and damaged data, particularly when identifying signatures are also damaged or deleted. This makes recovering them a near impossible feat although some tools claim to be able to do just this, but only for a few select formats. Sometimes there is confusion between data recovery and carving. Data recovery requires that the file system metadata remains more or less intact to identify and recover still in place data; however, this does not preclude the use of identifying signatures. In contrast, data carving uses known signatures to identify and recover potential data. Obviously, the latter is less reliable than the former, but is often the only recourse when working with unallocated space. Then, when there are no known signatures and recovery is not possible using standard methods, tools and techniques, what we have shown in this article can be of immense help.

Q&A

Should I use PWH, or FH or data recovery/carving?

If there is no evidence against which to draw comparisons then data recovery/carving is likely the best. On the other hand, if what you are looking for is exactly or almost alike (a few bytes difference at the most) then PWH may yield tangible results. However, if what is expected will differ by more than a few bytes then FH is the way to go. We often do not know what is in unallocated space until we perform data recovery/carving, a time consuming task. What it really comes down to is expectations. If there is an expectation for exactness, then perform only PWH. In contrast, if there is a prospect for variance albeit with high levels of similarity and size, then FH should be conducted.

"IF THE DISK IMAGE'S FILE SYSTEM CONTAINS MOSTLY UNALLOCATED SPACE, AS OURS DID, THEN THE TIME SAVED MAY BE MINIMAL"

the disktype tool (output in Figure 1 above) and The Sleuth Kit (TSK), both of which support the most commonly encountered file systems, as shown below:

```
Command 10: $ disktype /mnt/
evidence/suspect.dd
```

```
Command 11: $ blkls – A – b 512 –
o 2048 /mnt/evidence/suspect.dd >
/mnt/analysis/unallocated.dd
```

Of course, if the disk image's file system contains mostly unallocated space, as ours did, then the time saved may be minimal. However, extracting unallocated space does not extract the space between partitions and file systems – that has to be done manually.

Obviously, neither disktype nor TSK is needed when piecewise hashing file system allocated files.

CASE STUDY 2 – FUZZY HASHING (SSDEEP)

A couple of years ago a different acquaintance, who also owned his own business complete with in-house technical support, discovered that an Advanced Persistent Threat (APT) had targeted it. His technical support succeeded in identifying two different versions of the same malware (malware1.bin and malware2.bin, respectively); the second one the attacker actually forgot to delete. His tech support suspected that other versions might be lurking somewhere, but they could not find any additional evidence to support this train of thought. They tried running ssdeep against all pertinent machines on the network including all disk-based file system objects, but found nothing new. My acquaintance asked us if anything could be done to find evidence of additional APT malware as he was adamant that one small server disk, which had been particularly active according to his network logs, might contain some yet to be identified malware.

We told him we would consider the matter, until it hit us that we could combine FH and modified bloom filters. The malware samples, along with a copy of the disk image (suspect.dd), were stored on the forensic workstation's /mnt/evidence disk. Our first order of business was to perform data recovery and carving, as done in Case Study 1, which gave us no useful results even after having ssdeep'ed all recovered files against the two malware samples that had been given to us.

And so, off we went and tried to ssdeep every single disk sector from our first suspect disk using a script. The disk image, only 500 GB disk (500,107,862,016 bytes) in size, did not succeed in hashing more than about 7% of the disk after a bit more than a day of processing. The reason it was so slow is that ssdeep does not provide a means for specifying an input chunk size for hashing. In order to make up for ssdeep's inability to process user-specified chunk sizes, the script used dd to make up for this lacking feature. Unfortunately, when doing small read/writes the tool becomes entirely dependent on the rotational and access/seek speeds of the underlying disk (this includes dd and most other I/O based tools). Thus, the bottleneck was entirely I/O related. To compensate for this the read/write chunk size of dd was increased from one disk sector (512 bytes) to a much larger size. But because our malware samples were each 212,807 bytes in size and their ssdeep comparisons had a 99% similarity, we decided to use a chunk size of 212,992 bytes (or 416 512-byte disk sectors). Your mileage and performance will vary so some pre-testing may be required.

Before rerunning ssdeep, we extracted all unallocated disk space from the file using previously mentioned commands #10 and #11. Command #11 took about 85 min to complete using two different disks (one for the source and target images). The newly generated unallocated image file, /mnt/evidence/unalloc.dd, was approximately 252 GiB in size (270,324,813,824 bytes) and its integrity was ensured using the chattr command (see Command #1). After modifying the script to the new block size, we created a small 128 MiB RAM disk to reduce any additional I/O slowdowns in our processing chain:

```
Command 11: $ sudo mkdir /mnt/
ramdisk
$ sudo mount – t tmpfs – o
size=128M tmpfs /mnt/ramdisk
```

/ DISK SECTOR SIZE

Sometimes it is not obvious to determine the actual disk sector size for a given disk. For optical discs this is almost always 2048, but again there are exceptions to every rule. As for hard disk drives, sometimes we cannot be certain if it is 512 or 4KiB in size. And don't forget there do exist non-standard disk drives out there. So long as the Linux kernel supports a given block device (HDDs, opticals, software RAIDs, etc. (but not tapes or other character devices)) then it is easy to determine the actual physical block size for a device. Simply use the cat command to dump the contents of pseudo file /sys/block/sdX/queue/physical_block_size, where sdX represents the SATA disk (hdX for IDE and sgX for SCSI).

/ COMPILING SOFTWARE VS. REPOSITORY PRECOMPILED SOFTWARE

Although it is not always the case, often it is not worth trying to compile repository based software oneself. But to be thorough and to minimize the time we spent on our data processing, we did compile the software packages for ssdeep, md5deep/hashdeep and using more aggressive compilation options. After analysing the difference in processing time between the compiled and precompiled forms of these two packages, the difference between them was on the scale of seconds or several minutes for the longest processing. These differences are statistically negligible and can easily be attributed to various the internal mechanisms inherit of a running operating system.

"TO CONFIRM THE MALWARE WAS STILL ON DISK, IF ONLY IN A DELETED STATE, WAS TO HASH EVERY DISK SECTOR AND ATTEMPT TO IDENTIFY HASH MATCHES ESTABLISHED AGAINST THE ACTUAL MALWARE ITSELF."

The new script, saved as /mnt/analysis/sector-ssdeep.sh, was set to use a block size of 212,992 bytes. Running the script with this block size required only 1,269,178 loop iterations while using a 512-byte dd read requires 527,978,152 iterations. Thus, we could readily see why this operation is so dependent on the disk's underlying capabilities (i.e. rotational and disk access/seek speed).

To make the script runnable, its permissions were set to 755 (although other permissions sets are entirely useable too) using the chmod command. Its contents were as follows:

```
#!/bin/sh
size=`ls – al $1 | awk '{print
$5}'`
numblocks=$(($size/212992))
echo "File $1 is $size bytes"
echo "It will require $numblocks
iterations to process"
echo ""
sleep 10
blockskip=0
for ((blockskip=0;
blockskip<=numblocks;
blockskip++))
do
`dd if=$1 bs=212992 count=1
skip=$blockskip of=/mnt/
ramdisk/$blockskip ; ssdeep – b /
mnt/ramdisk/$blockskip >> /mnt/
analysis/unalloc.ssdeep.txt ; rm
– rf /mnt/ramdisk/$blockskip `
done
```

The script extracts the specified chunk-sized blocks to the RAM disk where it is then fuzzy hashed and whose output is in turn stored into hash file /mnt/analysis/unalloc.ssdeep.txt. The script was run as follows:

```
Command 12: $ cd /mnt/evidence
$ ./sector-ssdeep.sh unalloc.dd
>log&>log
```

The script took 309 min 37 sec (about 5h 10 min) to complete and generate an ssdeep output file with 2,579,920 hash entries and was 157,967,610 bytes in size. One thing noticed using this script was that there were duplicate entries for certain disk blocks within the output hash file. Fortunately, removing these duplicates was a straightforward task. At the same time, each time ssdeep was called by the script it added an ssdeep header to the hash file that had to be removed. Thus, to remove all these duplicate entries and headers we used the following commands:

```
Command 13: $ cd /mnt/analysis
$ cat unalloc.ssdeep.txt | grep –
v ssdeep > temp.txt
$ cat temp.txt | sort | uniq >
temp2.txt
$ mv temp2.txt unalloc.ssdeep.
nodup.txt
$ rm –rf temp.txt temp2.txt
```

Having removed all duplicates, we still found that the reprocessed hash file unalloc.ssdeep.nodup.txt still contains excess data. We expected a hash file containing exactly 1,269,178 entries but the new file contained 1,289,194 entries. This gave us 20,016 excess entries that we surmised were spurious in nature and were added by ssdeep. Since no additional data processing can be readily preformed on data file unalloc.ssdeep.nodup.txt, we had to add just one ssdeep header to the very beginning of the hash file so that it was recognized as a hash file. This was done by prepending the header to the hash file as follows:

```
Command 14: $ sed
– i '1i ssdeep,1.1--
blocksize:hash:hash,filename'
unalloc.ssdeep.nodup.txt
```

Once the file was prepended, it was time to perform the actual hash comparisons. This was done using the following command:

```
Command 15: $ ssdeep – a – d
– m unalloc.ssdeep.nodup.txt
malware*.bin >
compare.txt
$ cat compare.txt | grep – v
"(0)" | sort
```

These commands resulted in the following important output:

```
/mnt/analysis/malware1.bin matches
unalloc.ssdeep.txt:303647 (31)
/mnt/analysis/malware1.bin matches
unalloc.ssdeep.txt:303648 (76)
```

BIOGRAPHIES

R. Carbone has been working for Defence R&D Canada since 2001. He has been working as a digital forensics investigator for the last six years. He is also an open source software expert and was the co-author of a Government of Canada study that influenced federal government policy on the adoption of open source. He is a certified digital forensic investigator, incident handler and malware reverse engineer. val-forensics@drdc-rddc.gc.ca

C. Bean also works for the Canadian federal government. She is a certified digital forensic investigator and has been interested in the application of forensic techniques to disk-based forensic problems for some time.

```
/mnt/analysis/malware2.bin matches
unalloc.ssdeep.txt:303647 (32)
/mnt/analysis/malware2.bin matches
unalloc.ssdeep.txt:303648 (77)
```

There it is. We see that the two samples we already had in hand clearly matched something found between blocks 303,647 and 303,648. Remember that the block size we were working with was 212,992 bytes so these blocks had to be multiplied by this factor and that these block numbers were actually dd-based offsets, therefore had to have a "1" added to them as well. Thus, what is of interest for us to look at in a disk editor was found between offsets 64,674,594,816 and 64,675,020,799 within file unalloc.dd (remember that we were working from this disk file and not the actual disk image suspect.dd). This left us some 425,983 bytes to inspect manually.

Closer manual inspection revealed a damaged Windows PE file missing its first two sectors (when compared to malware1.bin and malware2.bin) and missing one more disk sector near the middle. The recovered file size was sector-size aligned (512-byte sectors) and when saved to disk was 212,992 bytes in size. Ssdeep comparison to the original malware samples indicated a 98% and 99% similarity, respectively

Thus, we have presented a technique for manually recovering deleted/damaged data from a disk image using only fuzzy hashes, and as far as we know based on an extensive search of the public literature, is the first time it has been shown to work.

REFERENCES
1. Carbone, R. and Bean, C. Generating computer forensic super-timelines under Linux: A comprehensive guide for Windows-based disk images. Technical Memorandum. TM 2011-216. Defence R&D Canada. October 2011. http://cradpdf.drdc-rddc.gc.ca/PDFS/unc113/p535374_A1b.pdf.
2. Carbone, R. File recovery and data extraction using automated data recovery tools: A balanced approach using Windows and Linux when working with an unknown disk image and filesystem. Technical Memorandum. TM 2009-161. Defence R&D Canada. January 2013. http://cradpdf.drdc-rddc.gc.ca/PDFS/unc150/p531895_A1b.pdf.
3. Kernel.org. Ext4 Filesystem. Online technical documentation. Unknown date. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt. Accessed: August 3, 2015.
4. Arch Linux. JFS Filesystem. Online technical documentation. Arch Linux. September 2015. https://wiki.archlinux.org/index.php/JFS_Filesystem. Accessed: October 14, 2015.
5. Sweeney, Adam et al. Scalability in the XFS File System. Technical paper. Silicon Graphics Inc. Presented at 1996 USENIX San Diego. 1996. http://oss.sgi.com/projects/xfs/papers/xfs_usenix/index.html. Accessed: October 14, 2015.
6. Unknown author. StackExchange Question and Answer. Online Q&A. July 2014. http://unix.stackexchange.com/questions/146620/difference-between-sync-and-async-mount-options. Accessed: August 3, 2015.
7. Kobayashi, Yoshitake. Evaluation of Data Reliability on Linux File Systems. Presentation. Toshiba Corporation. December 2012. http://elinux.org/images/e/e2/EvaluationOfDataReliabilityOnLinuxFileSystems-Kobayashi.pdf. Accessed: August 13, 2015.
8. Charikar, Moses S. Similarity Estimation Techniques from Rounding Algorithms. Technical paper. Princeton University. 2002. http://www.cs.princeton.edu/courses/archive/spring04/cos598B/bib/CharikarEstim.pdf. Accessed: August 11, 2015.
9. Sadowski, Caitlin and Levin, Greg. SimHash: Hash-based Similarity Detection. Technical paper. University of California, Santa Cruz. December 2007. http:// https://simhash.googlecode.com/svn/trunk/paper/SimHashWithBib.pdf. Accessed: August 11, 2015.
[10] Roussev, Vasil et al. Multi-resolution similarity hashing. Technical paper. Presented to DFRWS 2007. 2007. http://www.dfrws.org/2007/proceedings/p105-roussev.pdf. Accessed: August 11, 2015.

PREVIOUS SSDEEP BLOCK HASHING WORK

While we believe we are the first to propose using fuzzy hashes for data recovery for deleted/damaged files, others have proposed analogous work over the years. What we have done is not entirely novel yet with respect to the use of PWH and FH (using ssdeep) our technique or similar to it has not yet been found in the literature. However, others have proposed their own set of tools and techniques and we will briefly explore them here.

The first of note was Charikar [8] who proposed using rounding hashing algorithms to help determine the similarity or closeness between strings of information. His work appears to be the first in the literature to propose such a technique for evaluating the similarity between strings. While this is not the first work to do this it is considered the seminal work.

Sadowski and Levin [9] proposed a tool called SimHash and accompanying technique to identify similarities between files using a novel binary string-based approach and their tool is freely available from code.google.com.

Roussev et al. [10] proposed a method for breaking larger data files into smaller chunks and then combine this with block-based hashing, PWH and Bloom filtering into a new tool called MRS hash. This work is similar to what we have done and we are inspired by it, but ours is readily straightforward, automatable and in our opinion it is simpler to understand what it does.

CONCLUSION

We have shown how damaged or deleted data can be successfully recovered even if data recovery and data carving tools fail to identify and recover it. Depending on the context, either PWH or FH can be used to determine where in a disk image (or subset thereof) a given piece of data which must be either the same or very similar to some evidence already in hand, can be recovered. While the steps shown are manual and technical in nature, they can be largely automated. The authors have not attempted to reconstruct this work under Windows; others may do so.

Our case studies are based on real world situations but have been slightly amended to protect the identities of the parties involved.

While we had hoped to examine various performance metrics as it pertains to PWH and FH based parallelization, it was not possible given the current time and space constraints. We anticipate exploring this in our next and final follow-up article concerning this subject.