available at www.sciencedirect.com

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

Visualization in testing a volatile memory forensic tool

*Hajime Inoue**, *Frank Adelstein*, *Robert A. Joyce*

ATC-NY, Ithaca, NY, United States

Keywords:
Volatile memory
Memory forensics
OS X
Memory dump
Visualization
Dotplot
Density plot

ABSTRACT

We have developed a tool to extract the contents of volatile memory of Apple Macs running recent versions of OS X, which has not been possible since OS X 10.4. This paper recounts our efforts to test the tool and introduces two visualization techniques for that purpose. We also introduce four metrics for evaluating physical memory imagers: correctness, completeness, speed, and the amount of "interference" an imager makes to the state of the machine. We evaluate our tool by these metrics and then show visualization using dotplots, a technique borrowed from bioinformatics, can be used to reveal bugs in the implementation and to evaluate correctness, completeness, and the amount of interference an imager has. We also introduce a visualization we call the density plot which shows the density of repeated pages at various addresses within an image. We use these techniques to evaluate our own tool, Apple's earlier tools, and compare physical memory images to the hibernation file.

1. Introduction

Most research on volatile memory forensics is on analysis of the physical memory images, not on the process of recording them. Researchers assume that operating systems provide tools which reliably provide accurate images. This is, however, a bad assumption.

The area of volatile memory forensics has been rapidly growing since the 2005 DFRWS memory challenge (DFRWS, 2005). Previously, memory analysis consisted mostly of string extraction. In recent years, tools like PTFinder (Schuster, 2007) and Volatility (Waters, 2007) allow investigators to reconstruct what was in the contents of memory. Halderman et al. were able to reconstruct encryption keys from key-schedule components retrieved from volatile memory (Halderman et al., 2009). With disk sizes continuing to increase, as well as the increasing use of full-disk encryption, volatile memory forensics will play an increasingly important role in digital investigations.

Volatile memory forensics has not been possible on recent Apple Macs using only software.[1] After Mac OS X 10.4, Apple removed the `/dev/mem` file, which is the device used in all Unix-like systems to access physical memory. Matthew Suiche suggested reimplementing the `/dev/mem` device as a kernel extension at BlackHat 2010 (Suiche, 2010a). Suiche never released his tools, so we wrote our own kernel extension which replaces this functionality.[2]

The Mac Memory Reader uses our new `/dev/mem` and a new device we created called `/dev/pmap` which outputs the physical memory map in the same format as the `showbootermemorymap` macro in the Apple Kernel Debug Kit (Apple Corporation). This map is used by the reader to exclude

* *Corresponding author.*
E-mail addresses: hinoue@atc-nycorp.com (H. Inoue), fadelstein@atc-nycorp.com (F. Adelstein), rob@atc-nycorp.com (R.A. Joyce).

[1] Physical memory can be acquired from Macs that support Firewire using the approach first demonstrated by Maximillian Dornseif (Dornseif, 2004).

[2] The Mac Memory Reader is available at http://cybermarshal.com/index.php/cyber-marshal-utilities or as part of the Mac Marshal product.

1742-2876/$ – see front matter © 2011 Inoue, Adelstein & Joyce. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2011.05.006

addresses mapped to devices. The reader outputs a mach-o core dump file which can be read by `gdb`. Once we implemented the reader, however, we faced a problem – How do we test it?

Recording volatile memory is often an awkward operation. Valid memory address ranges are interleaved with device memory which must not be read—reading device memory often causes an immediate system crash. Operating systems themselves often get it wrong, or have limitations that are not obvious to users.

We are aware of three programs to record physical memory on Windows XP: `mdd` (ManTech International Corporation, 2009), `win32dd` (Suiche, 2010b), and the OnLine Digital Forensic Suite memory dump tool (`dumpmmf`) (Cyber Security Technologies, 2003). All three record physical memory to a raw image file by accessing `\Device \PhysicalMemory`, the Windows equivalent of `/dev/mem`. We found that `mdd` produces an image that is 118 pages smaller than the image produced by `win32dd` or `dumpmmf` on a 4 GB XP machine, and that image is only 3.3 GB. Non-server versions of Windows XP do not support more than 4 GB of physical address space, including devices, so there is only 3.3 GB of addressable RAM (Russinovich, 2008). It is unclear why `mdd` gives a different number of pages than `win32dd` or `dumpmmmf`.

Linux has a long history of problems with memory-backed devices. In 2005, Steven Rostedt discovered that `/dev/kmem`, the device used to access raw kernel memory, had been broken "for some time" on Linux without anyone noticing (Corbet, 2005).

Linux's `/dev/mem`, while technically not broken, was purposely crippled so that it does not behave like a regular file. Several distributions use a configuration called STRICT_DEVMEM (previously NONPROMISC_DEVMEM). This prevents `/dev/mem` from returning any results from locations that are not from device or BIOS locations. Memory mapping is not allowed at all with `/dev/mem`, except for device memory. Only file system functions are allowed. Mapping a page of non-device memory for some versions returns a page of all zeros (Corbet et al., 2005), allowing programs which require mapping `/dev/mem` to run, but not to run correctly. Similar to our approach, Kollar and Anderson have independently implemented kernel modules that restore `/dev/mem` functionality (Kollar, 2010; Anderson, 2008).

On modern versions of OS X, no `/dev/mem` equivalent exists. Apple removed the capability with the Intel version of OS X 10.4. On 10.4 PowerPC `/dev/mem` only works correctly if the computer is started using a `kmem=1 diag=16` boot time argument. Otherwise, reading `/dev/mem` will return pages of content, but they will in no way be related to the address asked for by the read operation.

It is clear that recording physical memory is difficult. An operating system may not provide access to access physical memory, there may be bugs in its implementation if it does, and it may have restrictions that are difficult to discover or understand.

Because no `/dev/mem` exists on recent versions of OS X, we decided to implement one that could be used by forensics investigators to record physical images before the machines are powered down. Once we implemented it, we needed to test it.

2. Evaluating physical memory imagers

We want our `/dev/mem` to record physical memory completely, correctly, quickly, and with a minimum of interference with the system it is recording. What that actually means has been interpreted differently by various tools' authors. Below we describe our design decisions for our tool.

We define a physical memory image to be **complete** if all of the physical address space which is not allocated to devices or the BIOS is recorded.[3] This does not account for all of the physical memory in the machine. We also find that a small amount of memory is not allocated in the physical memory map. On a 8 GB MacBook Pro, for example, we find that 7703 physical pages (about 31 MB) are not addressable. This same decision was made by the authors of Windows tools to recover physical memory also, and is forced by the fact that memory must be addressable to be accessible.

By **correct**, we mean that the physical address of a page in the image was the actual physical address of that page in memory. Because physical memory is not contiguous, we chose not to write to raw files. Instead our tool writes to a Mach object core dump file. Other tools, such as `dumpmmmf`, record memory as raw binary files, inserting pages of zeros when pages are not accessible.[4] This creates differences between the two types of files. First, there is an extra page of header information in the object file. Second, in ranges that are not in the memory map, raw files contain pages of zeros where the object file omits them entirely. Finally, because our tool records in the order of the physical memory map, not by address, the pages are not always recorded in order of ascending address.

Many physical memory analysis tools (such as Volatility), depend on the physical memory image being self-consistent. The image should appear as a snapshot of memory, and therefore the tool should record the image as **quickly** as possible. Our tool is designed to save the image to a removable disk because it is fast. We try to minimize disruption in the kernel file cache by turning off caching.[5]

Finally, the tool itself should not **interfere** with the computer it is recording. It should minimize the amount of memory it alters on the machine. Some tools are designed to record over the network to minimize memory disruptions. However, this takes significantly longer than recording to disk.

2.1. *Testing our* `/dev/mem`

We wanted to demonstrate that our physical memory imager completely, correctly, and quickly images OS X systems while minimally interfering with the system. This is a difficult problem. Ideally, we would like to compare the output of our

[3] Specifically, we record physical memory ranges tagged as type 1 (LoaderCode) through type 7 (available), as described by the OS X PE_state.bootArgs->MemoryMap kernel variable (Singh, 2007). It is possible to record BIOS code as well, and we are currently evaluating whether to include it in a default image.

[4] Such an approach is not feasible on 64-bit hardware; gaps between physical memory ranges can be substantial.

[5] Section 2.1 describes the side effects of the OS X file cache incurred when using `dd`.

tool with one which is known to work. Unfortunately, there are no existing ways to extract physical memory without disturbing the system. The closest to “ground truth” is Apple’s own `/dev/mem`, which works on OS X 10.4 PowerPC when certain boot flags are used.

Therefore the best, most straightforward test is to generate an image using Apple’s `/dev/mem` and then compare it with one generated immediately after by our own tool. We performed this comparison using an PowerPC G4 Power Mac running OS X 10.4.11. We used `dd` and `netcat` to avoid file caching problems to copy the physical memory image from `/dev/mem` to another machine. Copying over the network took several minutes. We generated an image using our own tool as soon as the network copy finished.

We then compared them. The `dd`-generated image was 131,072 pages; our tool’s image was 131,073 pages, which is consistent, since ours has an extra page for the Mach object header. We can conclude that the tool for 10.4 PPC is complete. We found that 93% (121,916 of 131,072) of pages were identical between the two images. Because copying perturbs the system and copying the image over the network is quite slow, allowing other processes to perturb the system during the copy, this is not an unreasonable result. It gives us confidence that, at least on OS X 10.4, our `/dev/mem` implementation is correct.

We measured speed as well. Our code is faster than the Apple implementation. We find that our implementation runs on an eMac PowerPC G4 with 512 MB of memory in an average of 14.2 ($\pm$0.2) seconds. The Apple implementation with `dd` runs with an average of 16.4 ($\pm$0.2) seconds with 99% confidence for a sample size of 5 runs each. In this experiment, `dd` copied the image to the local disk, unlike in our earlier testing, because a network copy takes several minutes.

Finally, we cannot know if our implementation minimally perturbs with the system. Again, we can compare our implementation with Apple’s. We ran each implementation 5 times in succession. We found that an average of 89,747 pages had changed after each `dd` run, with a standard deviation of 302 pages, giving an average similarity of 32% between runs. Our own tool modified an average of 4406 pages between runs with a standard deviation of 200 pages, giving an average similarity of 97%. Clearly, `dd` should not be used to record physical memory on OS X.[6]

This highlights an aspect of physical memory imaging that has not been well noted before. Apple’s implementation of `/dev/mem`, paired with `dd`, is complete, correct, and is within 15% of the speed of our imager implementation. However, it causes the kernel’s file cache to overwrite most inactive memory when it runs; ours causes *much* less interference with physical memory.

2.2. *Testing on recent versions of Mac OS X*

Testing our physical memory imager on Intel Macs and more recent versions of OS X is more difficult than on OS X 10.4 because there is no implementation we can compare with our tool. We must rely on tests with OS X 10.4 and examining the output on newer versions to see if it meets our expectations. This is difficult because volatile memories can be very large. For example, an 8 GB machine has 2,097,152 pages—far too large to examine by hand.

We performed several tests on newer versions of OS X. We knew where some content exists in physical memory because the kernel symbol table gives addresses and the kernel’s virtual to physical mapping is predictable. These gave us some evidence that our tool is correct.

We also performed what we call a string injection test. We generated a kernel extension that contained a known string that was unlikely to exist in memory from other sources. After we loaded the kext, we generated an image and searched for the string. We were able to find the string. Unfortunately, we found the string several times. On analysis, we found that this is the correct behavior. The string can be copied into memory by the file cache multiple times, since both the binary and source contain the string. Viewing the kernel extension in an editor can also create copies. As long as the pages the string is embedded in are not identical, then the copies are legitimate.

We did this test on OS X 10.6 with both the 64-bit and 32-bit kernels. We did not find true repeats (identical pages) with the string on the 64-bit kernel. However, we did find that the injected string did appear on two identical pages in the 32-bit kernel. We could not conclude from this that there must be a bug, because it is not uncommon to find identical pages. In fact, we found that some pages are repeated hundreds or thousands of times.

Suspecting a bug, we then extracted all the strings from the 32-bit image, which reduced its length dramatically, and made it easier to inspect. We observed that thousands of strings were repeated in the same order and concluded that our 32-bit `/dev/mem` was buggy.

On examination of the kernel source code, we found that one of the structures we were using to hold the physical

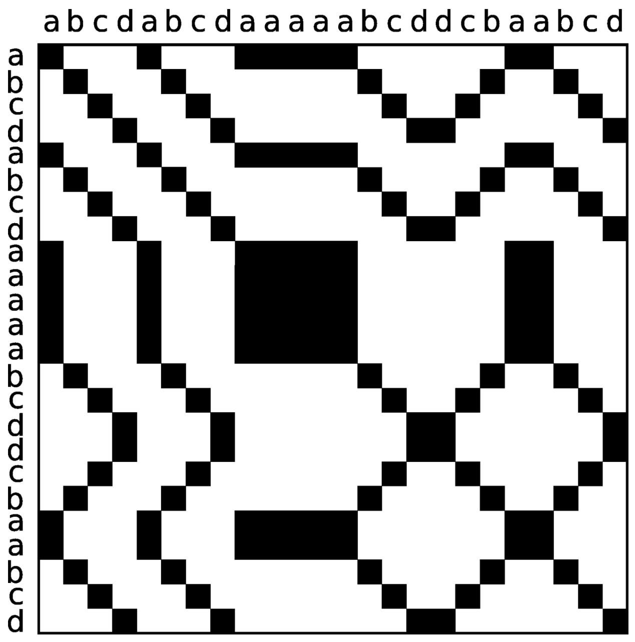
a b c d a b c d a a a a a b c d d c b a a b c d
a b c d a b c d a a a a b c d d c b a a b c d

Fig. 1 – Example dotplot.

[6] While we provide our own imager, OS X’s `dd` utility can still be used with our own `/dev/mem` implementation. We recommend against this.

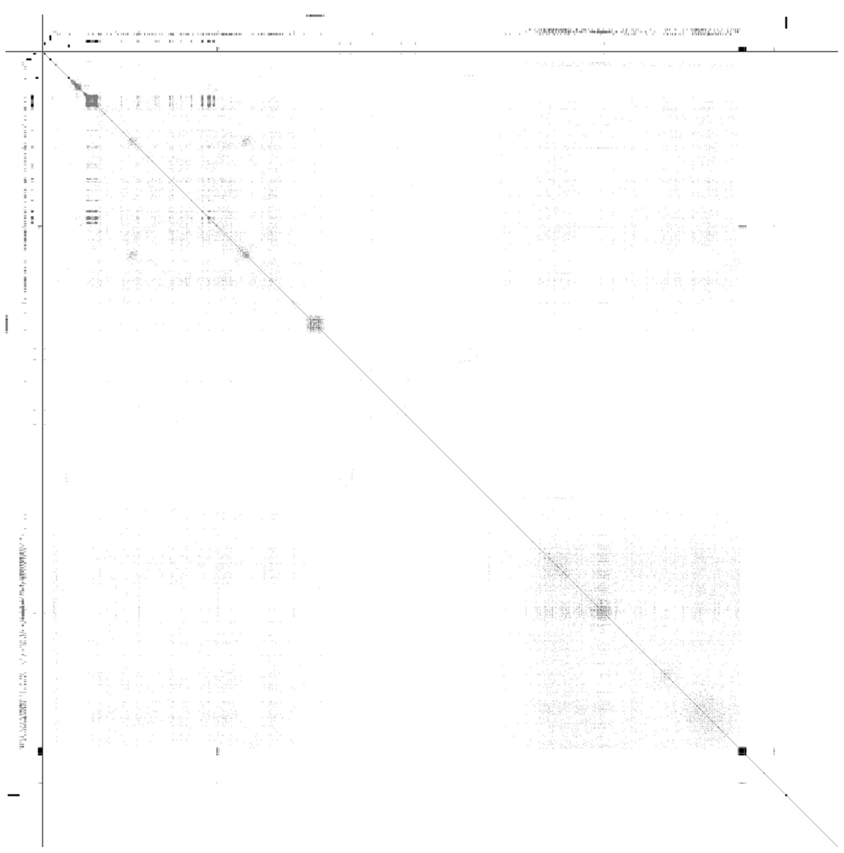

Fig. 2 – Dotplot of a correct image on OS X 10.6 with a 32-bit kernel with zero and one-pages removed (palette inverted).

address was a 32-bit value in the 32-bit kernel and was overflowing, which caused problem on Macs with physical memory addressed over the 4 GB boundary. This is common, even on machines with less than 4 GB of RAM.

After fixing that bug, we wanted to be confident that there were no others. With no ground truth for more recent, relevant OS X versions, and the huge size of current memories making manual inspection impossible, we turned to visualization.

3. Visualizing physical memory

We wanted a way to visualize the contents of hundreds or thousands of megabytes of memory in a way that would allow us to see unknown, but systematic errors in our implementation. As Lyle notes, most errors in forensic tools are systematic, and should lead to patterns (Lyle, 2010). We therefore turned to a visualization method used in bioinformatics called a dotplot.

Dotplots are used to visualize the similarity matrix of two sequences. The similarity matrix is the matrix of scores which express the similarity between the two sequences at each point in the sequence. Dotplots are primarily used in bioinformatics to show similarities between the genetic sequences of proteins.

For our purposes, we use the same sequence to examine self-similarity. Biologists do this to find similarities between regions of a single protein. Fig. 1 shows a small dotplot using an alphabet of four symbols. The region signifying the similarity between two symbols is black if the symbols are identical and white if they are not. Because we are calculating self-similarity, there will always be a diagonal black line from the upper left to lower right. Other lines denote copied regions, such as the repeated "abcd" sequence. Blocks show regions where the same symbol is repeated ("aaaa"). Inversions show where symbols are copied in reverse order ("abcd" vs "dcba").

In our dotplots, the quantum of similarity is the page. We define similarity by comparing the SHA-1 hashes of pages. Thus, two pages are similar only if they are identical. Because there are far more pages in our memory images than pixels in the dotplot, we use a black-body radiation palette to show the relative similarity of a region. If a region is entirely filled with 0 bytes, it is blue.[7] The value of the pixel is calculated by dividing the sum of all similar pages assigned to the pixel by the total number of pages assigned to the pixel. This is then normalized by dividing each pixel's value with the maximum similarity value found in the entire matrix. To summarize, the more similar pages in a region assigned to a pixel, the brighter the its color. For clarity, in the gray-scale version of this paper the palette is inverted in some figures.

For efficiency, we sample regions when generating dotplots of large images (>1 GB). If an image contains 2,097,152 pages (8 GB), a non-sampled similarity matrix requires 4,398,046,511,104 (2^{42})

[7] A full-color version of this paper is available at http://cybermarshal.com/index.php/cyber-marshal-utilities.

comparisons, or 4,194,304 (2048 × 2048) comparisons per pixel for a dotplot 1024 pixels wide. Because small differences in values cannot be distinguished in a single pixel, sampling reduces execution time dramatically without visibly changing the output. We usually sample about 10% of the similarities before assigning a value to a pixel.

For dotplots of smaller images there is no need to sample. A dotplot of a 512 MB image takes several minutes on our Intel Core 2 2.66 GHz machine without sampling. The dotplot of an 8 GB image takes many hours without sampling.

3.1. *The 32-bit kernel bug*

Self-similarity reveals systematic errors in our physical memory images. For example, our 32-bit overflow bug is immediately revealed by a dotplot visualization. The four repeated boxes in the upper left of Fig. 5(a) indicate that a large region of memory has been repeated (the first box starts at the origin and uses about a third of the address space). In Fig. 5(b) we have removed zero-pages (pages that contain all zeros) and one-pages (pages where each byte is `0xff`), which are repeated throughout memory. In this image, many of these pages were found high in memory, and therefore little of the region resulting in the black box in the lower right of (a) is shown. With the pages removed, the diagonal lines make it easy to see that a region has been copied.

The density plot, Fig. 5(c) is also revealing. This is a visualization method that has not been described before to our knowledge. We use a black-body palette to indicate the quantity of pages with a particular hash value in a region of memory. In our dotplots, we calculate the histogram of page hashes and then plot the density plots of the 16 most common above and to the left of the axes. Each of the 16 rows is a density plot, in order of decreasing frequency, starting from the bottom (the bottom row represents the zero-page). Fig. 5(c) is an enlarged version of the histogram density plot shown in Fig. 5(a) that omits the higher addresses which are mostly zero-pages. The two arrows above the plot indicate the copied region.

Fig. 2 shows a correct dotplot. It was generated after filtering out zero-pages and one-pages to make it clearer. This diagram is almost entirely without similarity, with the exception of the diagonal from top left to bottom right. This is what we expect from an OS that is using memory efficiently. Copied pages are wasted pages, since the same physical page can be mapped to many different processes.

3.2. *Visualization comparing our* `/dev/mem` *to Apple's* `/dev/mem`

We then applied this visualization technique to our earlier tests on OS X 10.4. The dotplot will visually display the differences between Apple's `/dev/mem` implementation and our own. Fig. 6(a) and (b) shows the dotplot of the image generated with Apple's implementation and one generated by our own tool, respectively. Except for section near the center,

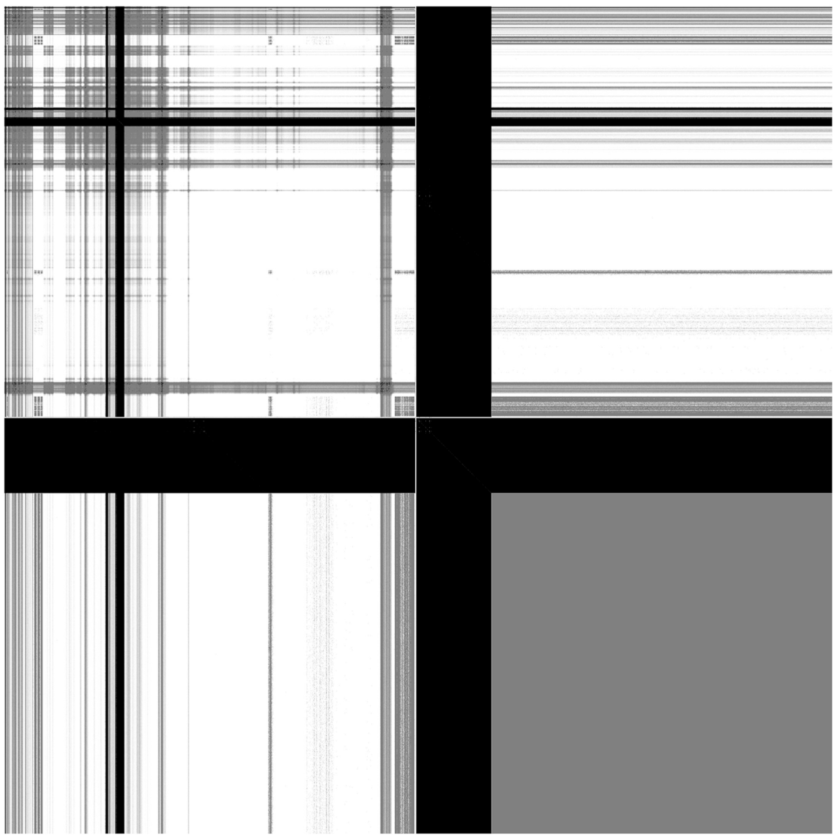

Fig. 3 – A dotplot comparing the physical memory image of a OS X 10.6 MacBook Pro running the 64-bit kernel and a hibernation file recorded immediately before the image. The memory image is first. The hibernation file has been compacted and inactive memory replaced by the zero-page.

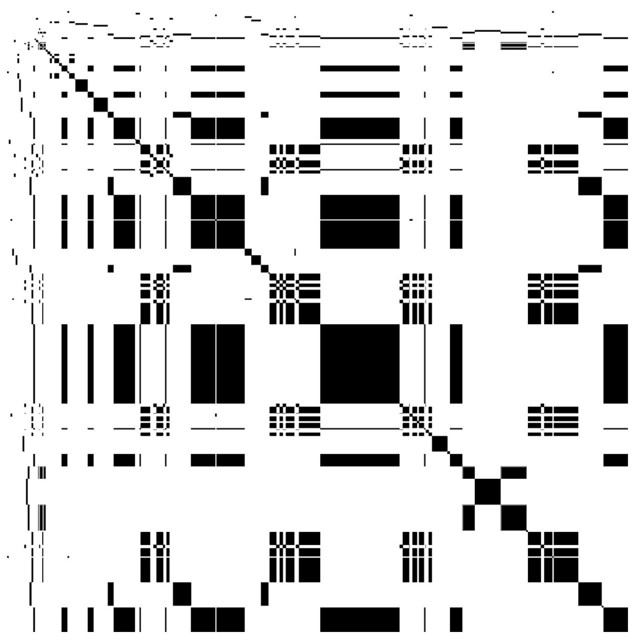

Fig. 4 – Dotplot of `/dev/mem` **image from Intel OS X 10.4. The large self-similar blocks clearly show it is broken (palette inverted).**

the two appear similar. This is confirmed by Fig. 6(c), which is a plot of the difference of the corresponding pixels in Fig. 6(a) and (b). The “cross” indicates that most perturbation was localized to a particular physical region.

In Fig. 6(d) and (e), we show a gray-scale density plot of the most frequent pages from the images used to produce Fig. 6(a) and (b). This also clearly shows the similarity. Finally, Fig. 6(f) shows a dotplot of the two images concatenated together. A line denotes the individual images. The top right and bottom left boxes show the similarity matrix of one image compared with the other. The diagonal line demonstrates the substantial similarity of the two images.

3.3. *Comparisons to the hibernation file*

Mac OS X stores the contents of RAM in a hibernation file, `/var/vm/sleepimage`, when it enters the sleep state. The hibernation file is often seen as an alternative to a physical memory image. The hibernation file is by definition complete, correct, and consistent. If the hibernation file was created by making a copy of physical memory that was then copied back when the machine resumed from sleep this would be an excellent way to obtain physical memory images. Unfortunately, this is not the case, as can be seen in Fig. 3. The bottom right quadrant is blue, indicating all zero-pages. The hibernation file compacts used memory in lower address spaces. It also has more than 34% more zero-pages than the image, indicating some or all inactive memory is not recorded. This has tremendous forensic implications, since it is memory from terminated processes or file caches that is no longer needed but is useful to investigators. Generating a hibernation file also fails the “minimize interference” test in that it requires putting the machine to sleep.

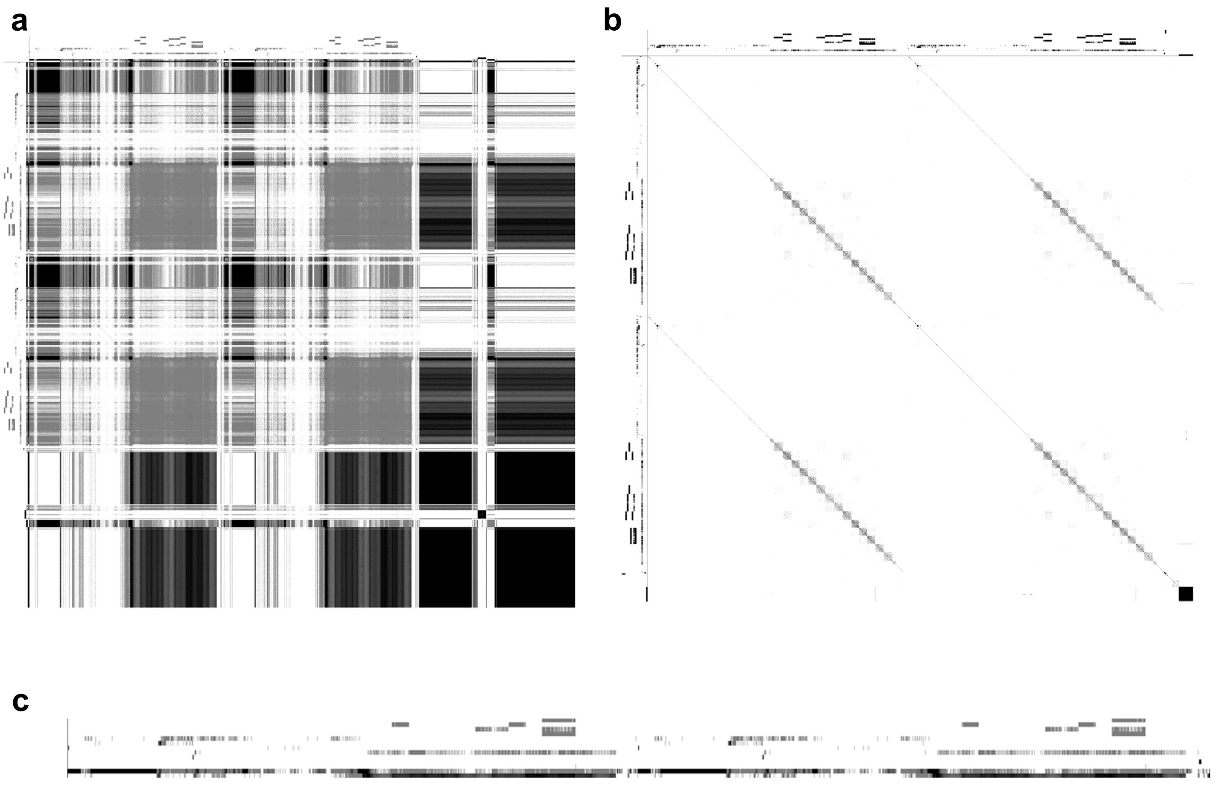

Fig. 5 – Depiction of our initial 32-bit overflow bug. The left dotplot (a) includes all pages. The right dotplot (b) has zero-pages (all zeros) and one-pages (each byte is `0xff`**) removed before plotting. The bottom figure (c) shows a blowup of the density plots of the most frequent pages. Arrows above the plot indicate the copied region (palette of all images is inverted).**

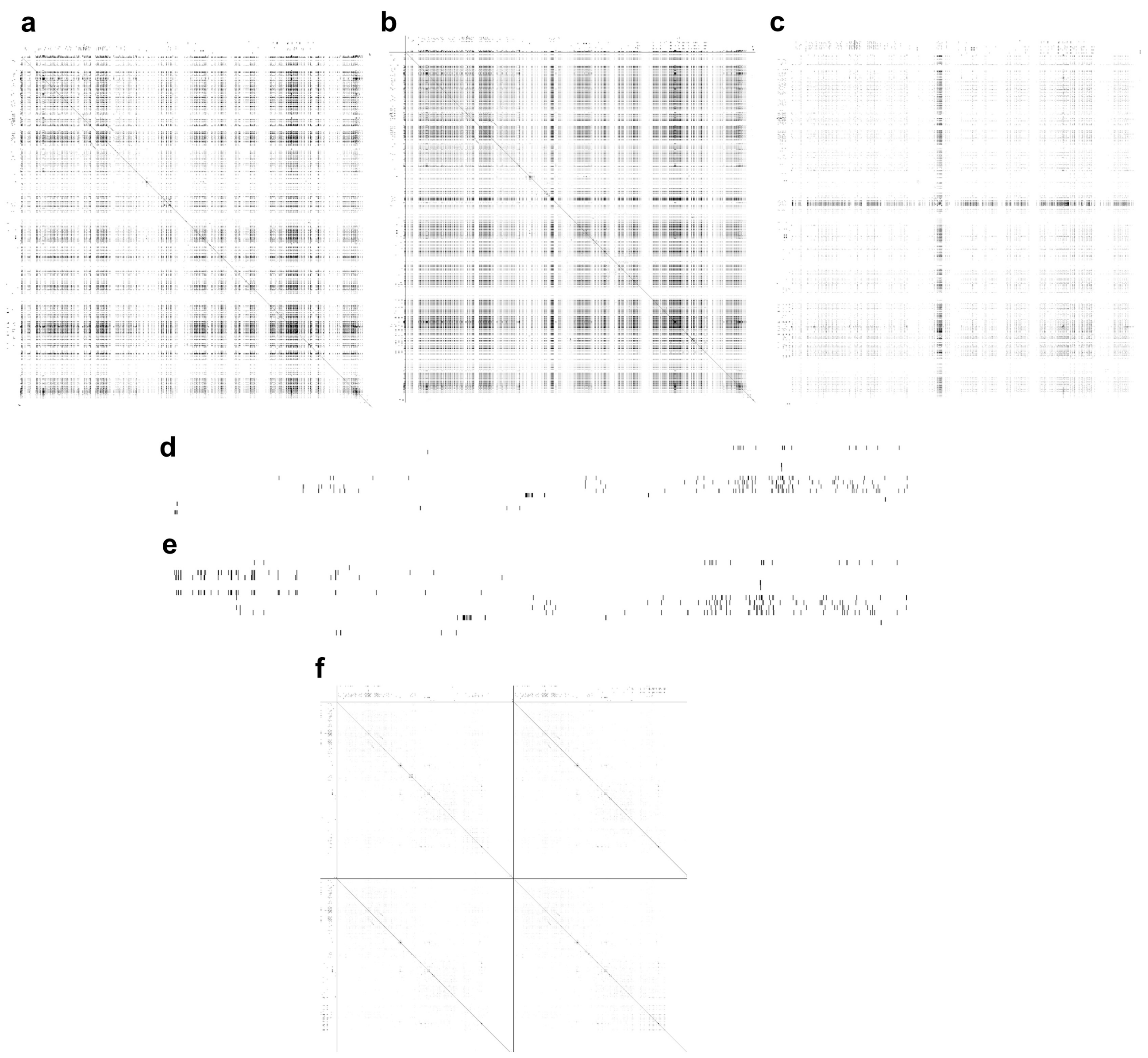

Fig. 6 – (a) Dotplot of Apple's `/dev/mem` collected using `dd` and copied over the network. (b) Dotplot of an image generated using our tool. (c) Plot showing the difference in values of each pixel between (a) and (b). (d) and (e) are density plots of the physical memory images used in (a) and (b), respectively, after zero and one-pages have been removed. (f) is a dotplot made from the file generated after concatenating the images from (a) and (b). The diagonal lines starting from the center of each axis in (f) indicate that the two images are substantially similar. The cross in (c) indicates that most differences are in a small, clustered region of pages (palette is inverted).

3.4. *Visualizing* `dd` *images*

The `dd` (data description) utility is the standard Unix utility for copying from devices. This and similar utilities, such as the aforementioned `mdd` and `win32dd` are used to make physical memory images by copying from `/dev/mem` or its equivalent.

In Section 2.1 we compared our own implementation of `/dev/mem` with Apple's, using the native `dd` application as the imager. We demonstrated that `dd` should not be used to when recording the image to the local disk. The dotplot in Fig. 7(a) reveals why. It is clear that `dd` creates four separate copies of memory as images. This is not its behavior when recording over the network, by piping the output of `dd` through a utility such as netcat. We believe that `dd` on OS X does not turn off file caching. OS X uses all available memory for its cache. The utility reads a portion of memory and then writes it to a file. The OS then caches these blocks in the file system cache, which are apparently allocated linearly in physical memory. When the copy reaches the file system cache, the copy is then copied again. This process continues until the copying completes.

These copies are not in-order or complete, however. Fig. 7(b), an enlargement of the upper right copy, shows that some copies end up like palindromes, mirrored about a center.

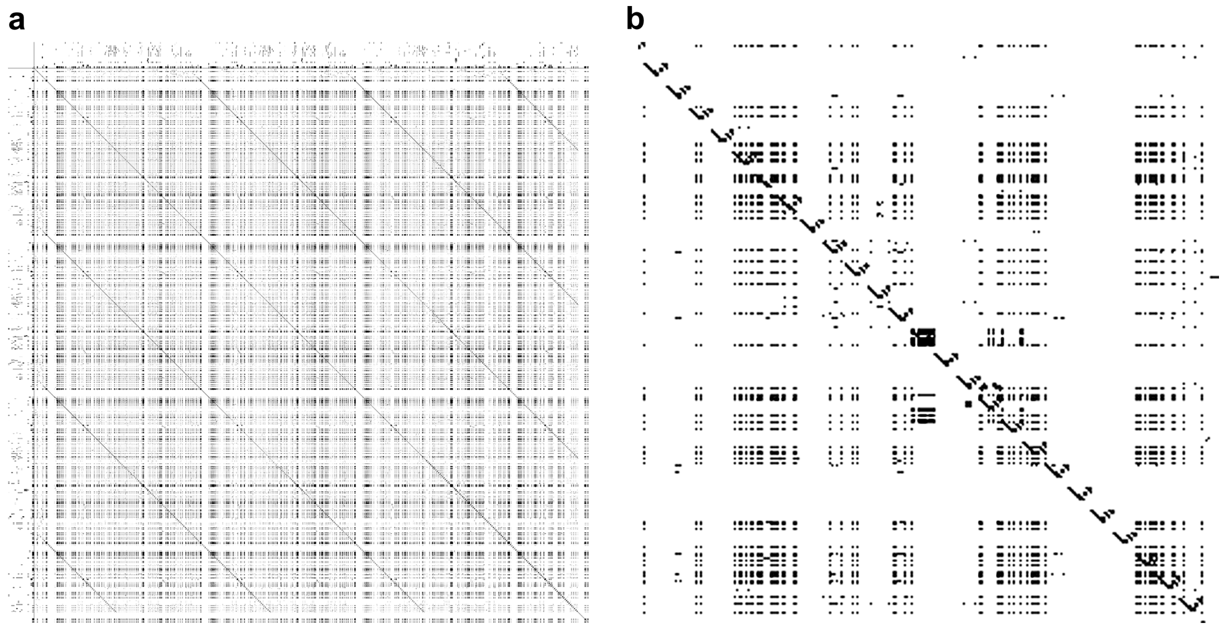

Fig. 7 – Evidence of copying in imaging a system with `dd`. (a) is the complete dotplot; (b) is an excerpt of the top right. Because OS X's file system cache expands to all of inactive memory, large copies can result in many copies of one region being imaged (palette is inverted).

This creates a 'v' shape, seen repeatedly in the diagonal from top left to bottom right. We did not see this behavior when copying over the network, nor did we see it with `win32dd` or `mdd`, which both copy to the local disk.

3.5. `/dev/mem` *on Intel Mac OS X 10.4*

Imaging on OS X can be broken in two different ways. In the earlier section, we described how file caching can interfere with the imaging. But `/dev/mem` is inherently broken without certain boot flags, and is always broken on Intel-based Macs running 10.4. We created an image using Apple's `/dev/mem` and used netcat to save it over the network on an Intel-based iMac running OS X 10.4. Fig. 4 shows its behavior. A single page is repeated for large spans of memory. This is then replaced with another repeated page. The number of pages in each span seems somewhat random. This is consistent with the algorithm in the kernel.[8] For each `read` call from an application to `/dev/mem`, the system randomly allocates a memory page, and then copies the contents of the page to the read buffer without initializing it. It then releases the page. Thus, it can be quite random, although the same page seems to be repeatedly selected for many megabytes, and then is often selected again after being replaced by another page.

3.6. *Visualizing string injection*

In Section 2.1 we described using string injection as a method for testing our `/dev/mem` implementation. Injecting a single string into user space does not lead to interesting patterns in dotplots. A pattern in a dotplot must span several pixels, and must therefore be extremely large. Furthermore, pages in user space (a virtual address space) may be randomly distributed in physical address space, so that only random dots would result from repeated insertions.

It may be worthwhile, however, to see if we can repeatedly inject a large string into memory to test completeness. If a process repeatedly injects a string until memory is exhausted, the string's similarity should appear as low level of noise in the dotplot, making the dotplot a solid color.

We wrote a program that repeatedly injected the complete works of Shakespeare from the Gutenburg Project into memory. This is one string of approximately 5.3 MB. The dotplot in Fig. 8(a) is the result. Without magnification, it appears that our original hypothesis is correct. Most of memory is filled with pages of Shakespeare. Only a small portion, typical from dotplots we have already made, stands out as mostly unique pages. Under magnification, however, we can determine that page allocation is not random. Fig. 8(b) shows that page allocation is sequential, although it may be both forwards (diagonals from upper left to lower right) or backwards (diagonals from upper right to lower left).

4. Related work

Related work to this paper falls into related visualization methods and forensics testing. There is no work that we know of that uses visualization to test digital forensic tools.

Dotplots have previously been used in computer science to find code duplication in large software projects. This was first suggested by Church and Helfman in 1992 (Church and Helfman, 1993). More recently, Conti suggested using dotplots

[8] The kernel source is publicly available at opensource.apple.com.

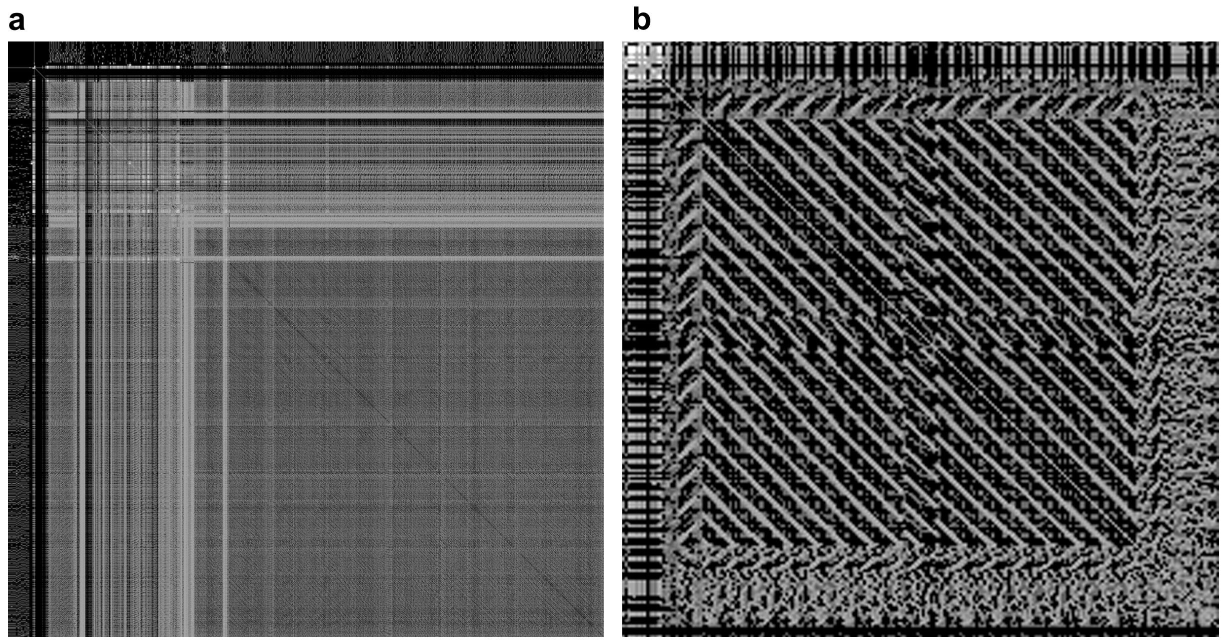

Fig. 8 – A dotplot (a) of the physical memory of an OS X 10.4 machine after a 5.3 MB string was repeatedly injected into its memory. In (b), a magnified version of a small portion of (a) provides evidence that pages are allocated sequentially both forward and backward to user-space applications.

for reverse engineering of binary files (Conti et al., 2008), based on a talk by Kaminsky at Chaos Computer Congress in 2006.

Our application of dotplots differs from Church, Helfman, and Conti's work in that images of physical memory are much larger than their datasets. We also believe that our use of density plots of the most frequent pages has not been used before as a visualization.

In terms of forensics, there is little related work in the area of volatile memory forensics testing. Lyle's work on disk imagers comes the closest (Lyle, 2003; Lyle and Wozar, 2007). His paper at DFRWS in 2010 discusses testing of forensic tools in general. His relevant conclusions to this paper are that "no general error rate exists," "errors that occur in some critical forensic activities are systematic in nature," and "human factors are important" (Lyle, 2010).

In investigating various memory imagers and in engineering and testing our own, we found four criteria for evaluating them, systemic errors which completely corrupted the images, and that an enormous amount of knowledge is required for the investigator to use the tools properly.

5. Discussion and conclusion

This paper recounts the testing efforts we made after implementing a physical memory imager for recent versions of Mac OS X. We concluded that four metrics should be used to evaluate imagers: completeness, correctness, speed (consistency), and interference. Testing a tool by these criteria is difficult when no ground truth exists for comparison.

We made several interesting observations during our tests. First, it is difficult to get a good physical memory image. On Windows, different tools record different sized images for the same machine. On Linux, without knowing the kernel version and configuration, one cannot be confident of the image at all.

We turned to dotplots, a visualization technique borrowed from bioinformatics, and page density plots, a novel visualization method, to test our tool. We found that on OS X, many pages were the zero- or one-pages, and that filtering them out led to better plots. We found that on older versions of OS X, an image must be copied over the network; otherwise it is contaminated with file cache activity. We also found that, on any recent versions of OS X, the hibernation file is not a good proxy for the physical memory image. By saving only memory that is in use, it loses much forensically valuable information. Finally, we found that dotplot visualization can provide information about the underlying implementation and behavior of the system it images—although physical pages do not have to be allocated sequentially, we found that they often are, leading to distinctive visual patterns in the dotplots.

Memory forensics is a rapidly evolving field. Because of the difficulty of obtaining "ground truth" images, tools must use a variety of testing and validation methods to ensure the tools meet the evaluation metrics we defined. We note there are few published case studies that evaluate memory forensic tools. We encourage tool developers and users to document the testing of the tools, so that investigators have greater confidence in them.

Acknowledgements

This project was supported in part by Award No. 2009-FD-CX-K002 awarded by National Institute of Justice, Office of Justice Programs, U.S. Department of Justice. The opinions, findings,

and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the Department of Justice.

REFERENCES

Anderson D. Crash. Technical Report. Red Hat Software, Inc., http://people.redhat.com/anderson/crash_whitepaper/; 2008.

Apple Corporation. Kernel debug kit. http://developer.apple.com/hardwaredrivers/download/kerneldebugkits.html.

Church KW, Helfman JI. DotplXot: a program for exploring self-similarity in millions of lines of text and code. The Journal of Computational and Graphical Statistics 1993;2(2):153–74.

Conti G, Dean E, Sinda M, Sangster B. Visual reverse engineering of binary and data files. In: Workshop on visualization for computer security (VizSEC); 2008.

Corbet J. Who needs /dev/kmem?, http://lwn.net/Articles/147901/; 2005.

Corbet J, Rubini A, Kroah-Hartman G. Linux device drivers. 3rd ed. O'Reilly and Associates; 2005.

Cyber Security Technologies. Online digital forensic suite, http://www.cyberstc.com/; 2003.

DFRWS. DFRWS 2005 forensics challenge, http://www.dfrws.org/2005/challenge/; 2005.

Dornseif M. Owned by an ipod, http://pi1.informatik.uni-mannheim.de/filepool/presentations/0wned-by-an-ipod-hacking-by-firewire.pdf; 2004.

Halderman JA, Schoen SD, Heninger N, Clarkson W, Paul W, Calandrino JA, et al. Lest we remember: cold-boot attacks on encryption keys. Commun ACM 2009;52(5):91–8.

Kollar I. Forensic RAM dump image analyzer. Master's thesis, Charles University in Prague; 2010.

Lyle JR. NIST CFTT: testing disk imaging tools. International Journal of Digital Evidence 2003;1(4).

Lyle JR. If error rate is such a simple concept, why don't I have one for my forensic tool yet? In: The proceedings of the tenth annual DFRWS conference, vol. 7; 2010. p. S135–9.

Lyle JR, Wozar M. Issues with imaging drives containing faulty sectors. In: The proceedings of the 7th annual DFRWS conference; 2007.

ManTech International Corporation. mdd, http://sourceforge.net/projects/mdd; 2009.

Russinovich M. Pushing the limits of windows: physical memory, http://blogs.technet.com/b/markrussinovich/archive/2008/07/21/3092070.aspx; 2008.

Schuster A. PTFinder version 0.3.05, http://www.computer.forensikblog.de/en; 2007.

Singh A. Mac OS X internals: a systems approach. Addison-Wesley; 2007.

Suiche M. Mac OS X physical memory analysis, www.blackhat.com/presentations/bh-dc-10/Suiche_Matthieu/BlackHat-DC-2010-Advanced-Mac-OS-X-Physical-Memory-Analysis-slides.pdf; 2010a.

Suiche M. Moonsols windows memory toolkit, http://www.moonsols.com/windows-memory-toolkit; 2010b.

Waters A. The volatility framework: volatile memory artifact extraction utility framework, https://www.volatilesystems.com/default/volatility; 2007.

Hajime Inoue is a Principal Scientist at ATC-NY. His research interests include computer security, machine learning, and programming language implementation. He received his doctorate in computer science from the University of New Mexico in 2005.

Frank Adelstein is a Senior Staff Scientist at ATC-NY, providing oversight and guidance to projects relating to computer forensics and security. His areas of expertise include digital forensics, intrusion detection, networking, and wireless systems. He has co-authored a book on mobile and pervasive computing. He received his GIAC Certified Forensic Analyst certification in 2004. Dr. Adelstein is the vice-chair of the Digital Forensics Research Workshop (DFRWS).

Several projects in which he has been the principal investigator have resulted in commercial tools, including LiveWire Investigator/OnLine Digital Forensic Suite and P2P Marshal. He has also co-created and conducted training in P2P forensics, live forensics, and reverse engineering.

Robert A. Joyce is the Technical Director for Information Management at ATC-NY. His research interests include distributed information storage and transformation, computer forensics, image and video processing, network and media security, visualization and design, and human–computer interaction. Since joining ATC-NY in 2002, he has led or contributed to over 25 research and development efforts in the areas of information management and computer forensics. He is currently the Principal Investigator of the R&D effort behind Mac Marshal, a computer forensic tool for Apple Macs. Dr. Joyce was a substantial contributor to the development of the OnLine Digital Forensic Suite, a live forensics tool. He also has significant experience in the fields of video and audio signal processing and in systems programming and administration.