A Lightweight Software Write-blocker for Virtual Machine Forensics

Patrick Tobin, Nhien-An Le-Khac, M-Tahar Kechadi,
University College Dublin,
Ireland

pat.tobin@ucdconnect.ie; {an.lekhac, tahar.kechadi}@ucd.ie

***Abstract*—The integrity of any original evidence is fundamental to a forensic examination. Preserving the integrity of digital evidence is vitally important as changing just one bit among perhaps gigabits of data, will irrevocably alter that data and cast doubt on any evidence extracted. In traditional digital forensics write-blockers are used to preserve the integrity of that evidence and prevent changes from occurring, but virtual machine forensics presents more difficult challenges to address. Access to the digital storage device will probably not be possible, typically the only accessible storage will be a virtual hard disk drive. This will have the same integrity issues as those of a real device, but with the added complication that it is not possible to use a hardware write-blocker to prevent changes to those data. For this reason it is important to explore how to implement write-blocking mechanisms on a virtual device. In this paper we present an implementation of a software write-blocker and show how we can use it to be compliant with the 2nd ACPO principle on digital evidence.**

***Index Terms*—Digital Forensics; Digital Evidence; Write blocker; Virtual Machine**

I. Introduction

During the acquisition and analysis of a hard drive there is a possibility that the evidence device may be written to. A write-blocker prevents this and is an essential element of a digital investigators tool kit. They are a physical device that interfaces between the device to be imaged and the workstation making that image. They prevent a host operating system from issuing commands or writing data to the storage device that will alter its contents or data [1].

Proving data has not been altered is a very important step in digital forensics. This can be achieved through taking an md5 hash of the original evidence [2]. This hash is then used to prove that subsequent copies of those data are bit for bit images of that original evidence, by recalculating hashes of those images and comparing them against the original hash. If they differ then we can conclude that the evidence has been changed. Causing even one bit to change, fundamentally alters that evidence and will result in a completely different new hash value for the device - the avalanche effect. If the hashes differ then the forensic credibility of those copies is questionable and by association the trustworthiness of evidence extracted from those compromised device images is also weakened. Preserving the original evidence is therefore vital in proving it in Court, any alteration of those data will have profound and overwhelming consequences for that evidence, and it will be declared as inadmissible. A second hash, the SHA1 hash, is usually used in conjunction with md5, to support the md5 hash of the image.

A write-blocker is used with physical storage devices, however, virtual machines (VMs), and virtual hard disk drives (vHDDs) present very different scenarios to investigators. By definition a virtual environment is an instruction-by-instruction representation of a computer on a different system [3], recreated rather than having its own physical working environment provisioned through hardware. Current technologies allow many VMs to be installed and co-exist with each other, each sharing the same hardware and each operating independently of one another.

To all intents and purposes a VM is real to the user, there should be nothing to suggest that what they are using is anything other than a true desktop machine, with its own operating system (OS), hardware, and network interfaces. They should respond as real devices should and should have, more or less, the same performance issues. This virtual environment poses an enigma for investigators, a vHDD is a virtual representation of a true HDD, yet it is not possible to forensically retrieve data from it in the same manner as a real HDD is forensically examined.

Memory management in a virtual machine monitor (VMM) does not necessarily allocate memory as a contiguous block [4], it is more likely to be fragmented, with the distribution managed by the VMM, this has consequences when investigating a VM. To take possession of all possible devices that may contain data and image those devices would be very difficult, and identifying those data of interest to an investigation would also be very difficult. Copying those data directly from the vHDD is perhaps the most feasible and practical way of capturing evidence, but there also exists a risk of writing data to a vHDD.

In this paper we address the issue of write-blocking a VM. In the first section we will examine current VM technology, in the second section we will look at digital forensics and compare that against current VM forensics, and in the third

section we will introduce our approach, we will evaluate it and show how we can comply with the Association of Chief Police Officers (ACPO) guidelines [5] and NIJ Report [6] and finally we will conclude and outline future research in Section 6.

II. Virtual Machines

VMs and cloud computing are fundamentally and inextricably coupled. They provide the platform from which services such as SaaS[1], IaaS, and PaaS are furnished. VMs allow efficient use of hardware and share these resources among their users. VMs are used by many companies to standardise the services they provide to their customers, and to make the best use of hardware and software resources.

The ubiquitous nature of VMs also brings with it challenges for Law Enforcement and cyber-crime investigators. The volatility of VMs is the most pressing issue to overcome, the ease with which a VM can be created and configured is matched by the simplicity of deleting one. Other challenges include VM standarisation and VM remote access.

Deleting a VM results in the loss of all data that VM contains. To overcome that problem Tobin and Kechadi [7] proposed using kernel code injection and virtual machine introspection (VMI) as a means of capturing data before they were deleted. The approach they propose is a very feasible one, however it carries with it some problems to be overcome. To extract data from a VM with injected code, any data that the code identifies as evidence, or has the qualities of evidence, must be collected by VMI, or somehow be flagged for extraction. This will invariably mean that a file or folder will be created to hold those data, or some method of flagging VMI will be used to identify which data are for extraction. The implication for this is that data will have to be written to the vHDD, possibly over writing evidence or deleted files [8]. Writing data to an evidence drive must be avoided to preserve that drive in its original form and allow for the recovery of any deleted data, and by preserving the drive in its original state will ensure that the md5 hash of the data will not change thereby proving the evidence is unaltered.

What Tobin and Kechadi essentially proposed is the live forensic examination of a computer system, this area is still developing with many proposed approaches already. Triaging a live system is one approach, where data are assessed for their evidence value [9, 10], outside the laboratory, prior to the recovery of a HDD and its forensic examination. The process of triage avoids data being written to the HDD through the use of external storage, typically a USB memory device, to contain the executable triage code and the resulting data recovered. This approach is a reasonable one to take with a VM, but only if that VM is still running. In cases where it is not possible to capture a live VM then this approach will obviously fail.

[1]Software as a Service, Infrastructure as a Service, Platform as a Service

Triaging will also fail where direct access to the VM is not possible. For example if the VM is in the cloud, or is running on a computer, physical access to the VM will not be possible, therefore alternative methods must be explored, this is what Tobin and Kechadi have proposed - the remote real-time examination of a live VM [7].

III. Digital Forensics

Digital forensics was first introduced by Parker in his 1976 book - Crime by Computer [11]. Computing has developed in many ways since then, particularly with the widespread introduction of virtualisation and cloud computing. Digital forensics has itself developed through what Parker described as *pre-history, infancy, childhood and adolescence*. He advocated that we are now in the adolescent phase.

The re-emergence of virtual computing in recent years has facilitated the development of a new type of crimes. The ability to create a VM instance in a foreign country has allowed a crime scene to be effectively put beyond the jurisdiction of most investigators. The tools and techniques that have been developed for a traditional computer crime scene are largely ineffective on VMs, particularly those that are in the cloud, or on distributed networks. This has arisen primarily because of the shared resources used, the distribution of, and access to those resources.

A very important element of any forensic analysis is that the methodologies used are sound and repeatable [17]. By following the same procedures, and by using the same tools and the same evidence, the same results should be obtained as those obtained in the forensic laboratory, failure to adhere to this assumption will weaken the evidence.

IV. Virtual Machine Forensics

Recently cloud computing [23, 24, 25] forensics, mobile cloud forensics [26] and VM forensics have attracted many forensic research scientists. Some VM forensics challenges are similar to those of traditional digital forensics. Once data have been captured the same analysis techniques are used, but data capture is the most pressing challenge. Unlike standard computer systems where tools and techniques exist to capture data from a standard hard drive, similarly effective tools and methods have to be developed for a vHDD where access to it is restricted. Just like in traditional digital forensics capturing the host log and data movements, are equally as important in VM forensics. These too, should not be overlooked when preserving data on a VM.

In digital forensics, securing data on a hard drive involves capturing the suspect computer, and seizing and removing the hard drive, however seizing a vHDD is less straightforward. If the VM is operating in the cloud through a service provider, accessing the hard drive is much more difficult, if possible at

all. Apart from obtaining Court orders to authorise this, it is likely to take time and may expose other users data, causing privacy concerns. Combined with this, there are very few tools to assist in investigating vHDDs, apart from LibVMI [12].

Virtual Machine Introspection (VMI), as described by Garfinkel and Rosenblum in their 2003 paper [18], is a very important development in the inspection of VMs. This describes a framework whereby examination of virtual systems can be carried out from within the VMM. VMI allows aspects of a VMs current status to be viewed from the VMM, without interference with or the knowledge of, the VM and without impacting the performance of the virtual system. It offers the ability to view a VM's memory pages, processes, register values, and network settings, and provides a structure within which data can be removed from a VM. The LibVMI tool implements this proposal in a forensic manner and when used with The Volatility tool [13], it can be used to recover data from live systems. Volatility provides a set of terminal commands that can be used to recover and parse data from a variety of format types, including ELF files, VirtualBox snapshots, core dumps, QEMU VM memory dumps, Windows Hibernation files and Crash Dumps, plus more. A very wide range of operating systems (OS) which include Windows XP to Windows 10, Linux Kernels, from 2.6.11 to 4.2.3 and Mac OSes from Leopard (10.5) to El Capitan (10.11) are also supported.

Despite this, VMI is limited by the semantic gap that exists between those data on a vHDD and the natural language representation of those data. The issues that the semantic gap presents are dependent on several variants which must be accounted for prior to parsing the raw data captured. These involve factors which alone are difficult to overcome, but when combined, amplify that difficulty. Windows operating systems represent data differently to Linux which in turn differ to Unix and Mac OSes, additionally some processors use little endian data structures while others use big endian. Further complications can be introduced with each update or new release of the various operating systems, which can introduce changes to data structures, thus altering the semantics of an operating system and further complicate conversion to a natural language representation. Without having knowledge of the operating system, version and release and any changes to data structures, available to, or known by, investigators, the process of giving meaning to these raw data is made considerably more difficult.

In their paper Bahram et al. [19] assert this expectation and express that the fundamental assumptions of VMI are that the VM being examined uses data in a prescribed manner by following data structure templates and that the target VM adheres to certain semantic rules and behaviours. Both of these principles can be replicated to interpret data, but by compromising these assumptions, through the modification of the kernels use of any data structure, VMI tools are defeated. These modifications can be made very simply, through changing the syntax and semantics of data structures. This type of obfuscation of data may be unexpected, and is likely to be a very considerable obstacle for investigators.

It is very likely that parsing those compromised data will be computationally complex and expensive to resolve. To overcome these problems one solution, offered by Tobin and Kechadi's approach, is to inject forensically sound code into a VM to recover data. Their approach can by-pass the obfuscation methods described by Braham et al., by using the VM operating system semantics.

A. Virtual Machine Forensic Tools

VMI, and the tools built upon it, are currently the primary method used to recover data from a VM. The VMI process alone will not interpret raw data into its natural language counterpart, tools are required to do this. Two such tools are Virtuoso [20] and VMST [21] which can be used to bridge the semantic gap. Both of these tools rely on the semantics of the operating system being known. Both these tools generate code that will replicate the behaviour of those in-guest programs, using them to bridge the semantic gap. but can be defeated by using the obfuscation methods presented by Bahram et al.

V. Write-blocking a VM

A. Platform

We used a Sabayon Linux (Gentoo) platform, Kernel 4.4.0, with an AMD eight core processor and 16 GiB RAM and installed KVM/QEMU v1.3.2 on it. We then created a similar Sabayon Linux VM as the host system, allocating it with 30 GiB SSD, 1024 MiB RAM, 2 vCPUs and defaulted to the remaining settings and used this as our test platform. We wrote our code in C and used gcc version 4.9.3 (Gentoo Hardened 4.9.3 p1.1, pie-0.6.2) to compile the code. The test data we used was the */home* directory contents from our host system which we copied into the guest system, this contained 95,473 files in 7729 sub-folders, and was 7.8 GiB in size.

We disabled network access to ensure that we had a platform that would remain relatively unchanged as we carried out our experiments. We did this to ensure that little modification took place to any files, data and settings that would normally be dependent on a web connection. This helped maintain a comparatively unchanged environment for our experiments.

Before we describe our proposal further we must first describe certain prerequisites we have assumed. We have complete access to the VM under investigation, we have root privileges and the VM was used only for lawful purposes. We also certify that all data are lawful, were created by us for exclusive use in our experiments, that all data accessed were owned by us, that we have unrestricted access to and use of those data, that no breach of copyright was committed and that we did not exceed the licence or conditions of use, of any third-party software used.

B. Software Write-blocking

Software write blocking can be enabled on Linux systems, but with limits as to its effectiveness. One option that allows a file system (FS) to be mounted read-only (RO) is the Linux command *mount* when used with the switch *ro*. This mounts an FS RO, but if the FS is *ext3* and *ext4*, both will replay the journal if the FS is 'dirty' [22]. This behaviour may allow write access to the FS, but is dependent on a number of factors - the FS type and state, and the kernel behaviour, however this information may not be readily available. Another option that can be used with the *ro* switch is the switch *noload*, but this will block any writes to the device, irrespective of their source, and if not unmounted cleanly can lead to further problems. The *noload* switch is compatible with *ext4* and backwardly compatible with *ext3* and *ext2* [22], but will not work on a partition that is already mounted and working.

The Linux command *blockdev* [15] is a Linux system call to set and check block device attributes. Using this command's *setro* switch the block device can be set to read only, however setting this attribute affects access to the block device system-wide, with the possibility of flagging that a fatal error has occurred [16]. Both of these commands will block any writes to a FS, irrespective of the source.

This effectively means that a VM that is network connected is blocked from reading from the network, we do not want this behaviour to happen. The *mount* and *blockdev* behaviours described are unwanted in the context of what we are aiming to do. Mounting an FS as RO will not prevent system updates to the system, it will only prevent user updates to it, or using *blockdev* to make it RO to prevent writes by injected software will affect system access to the devices, whereas we want the system to operate normally, but prevent writes to the vHDD caused by the investigator. The Linux commands and switches we have just described, render them inconsistent with our software on virtual devices in the context of use as a forensic write-blocker.

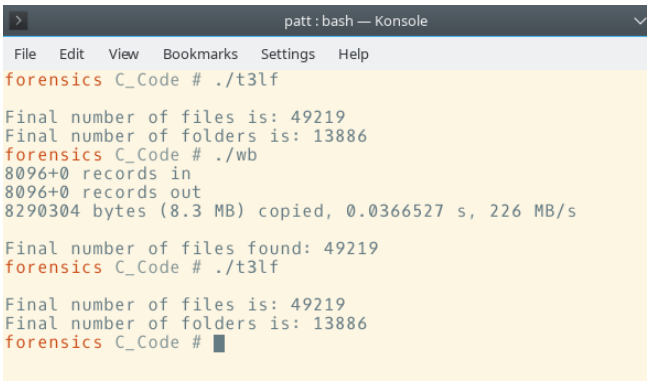
C. Our proposal

We have developed a simple, but effective, search engine to search a vHDD. We have built this search function to recover all files on a vHDD that are text or ASCII text files only, and have used the Linux command '*file*' to identify these files. We then calculated an md5 hash for each of these files and saved that signature to a file for extraction from the VM using LibVMI [12], this would normally cause a file to be written to the vHDD.

Writing to the vHDD would cause the md5 hash of that drive to be changed., however as we are accessing a live VM we expect that some data on the vHDD will not remain static. Logs will change and files will update, this is unavoidable with a live system, but one big danger in altering the vHDD lies in accidentally overwriting data that may have been deleted. These updates may arise through an action of the investigator, either deliberate, accidental or non-intentional. We want to avoid any data writes to the vHDD arising from any action of the investigator.

With this in mind we create a RAM drive in the VM's memory space. This allows any data being saved to be saved to the RAM drive, thus preventing any writes to the vHDD. This effectively blocks our application from writing to the vHDD, implementing a software write-blocker. We expect that log files will record execution of our data, and therefore some data will be written to the vHDD, but since we can explain this change we can satisfy Principle 2 of the ACPO principles. Furthermore a RAM drive is a protected region of RAM, processes cannot use this RAM, access to it is made only through specifically addressing this memory space, effectively write-blocking it too.

We need to show that nothing is written by us to the vHDD, and for the purposes of verifying this and the proper operation of our tool, we took the following course of action. We executed a search of the data we copied to the VM, and enumerated the number of files in that space, a total of 49,219 files, saving the data to */tmp*, thus isolating it from the test space. This is shown in *Image 1* as the first command executed. We then renamed the text file produced to save it for further examination. We executed our file list program, again saving the results to */tmp*, this is also shown in *Image 1* as the third command executed. This shows that the number of files is unchanged from the first run of our tool, confirming that the vHDD has not been written to.



, *Image 1.*

We executed our forensic tool on those same data. This tool uses Linux command *mount* [22][22]to create and mount a *tmpfs* RAM drive in the */mnt* directory using terminal commands executed with the C function *system()*. We have set this drive to 8 MiB, and to prevent any messages being displayed to terminal we redirected any output from the terminal command to */dev/null*. We chose 8 GiB as an arbitrary value, and then hard-code this into the tool prior to compilation and injection. We hard-coded it for the simple reason that there is therefore no ambiguity and no possibility of uncatalogued interaction with the tool or system, the tool is then completely independent and autonomous.

Next we use the Linux *dd* command with */dev/zero* to zero the drive. Creating the RAM drive will wipe the RAM drive,

but to be able to guarantee that the RAM drive is certifiably empty and free from any data residue, we zero the drive. The tool then searches the data space using the criteria we have code into it, building strings to be used by the C library function *system()* to calculate the md5 and outputting results to a file in the */mnt/rdisk* drive, called *md5.txt*, a sample of this output is shown in *Image 2*. All the files found and shown, have the description 'text' in their file-type strings returned by the command *file*. Finally we search the data space again and compare the results of the first data space search with the results of the second search with the Linux command *diff*.

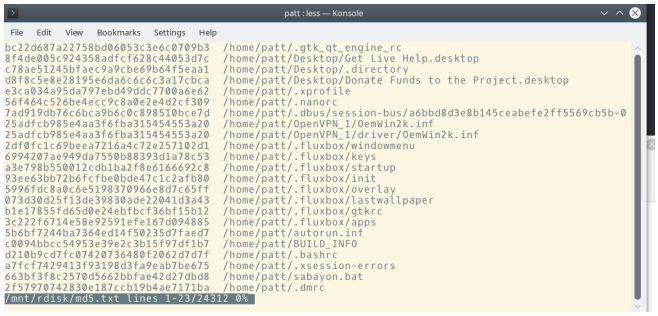


Image 2.

Comparison of these two files shows that there has been no files written to the virtual disk, and that our tool is effectively write-blocking the vHDD. This is shown in *Image 3*, where there is no result from the *diff* command showing that the file system is unchanged and has not had any files added after we run our tool, compared with it before we run our tool,. This confirms that we have maintained the integrity of the vHDD, confirming that our write-blocker is working correctly. We retest the vHDD by re-running our tool multiple times, each returning the same number of files found as the first run - 49,219.

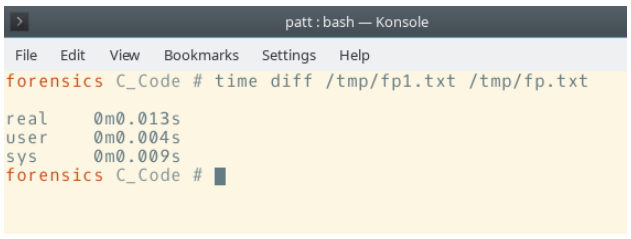


Image 3.

In order to measure the vCPU usage and memory usage we ran the Linux utility *ps* using the *-o* switch with the program's process id, formatting the output with the *%cpu* and *%mem* arguments at 5 second intervals. This showed that vCPU use by our program was consistently within the range 1.7% to 2.0% of the CPU processing time, and memory use was approx. 0.1% throughout the program run. The ps process expresses CPU usage as the percentage of time spent running during the entire lifetime of a process, it does not approximate the amount of CPU used, in terms of processing carried out. These levels of hardware use in the VM we created show the low impact our program would have on a system being investigated. If we scale to much larger systems the proportion of vCPU and memory that would commit to our tool should be proportionately less, and therefore should be less likely to flag as suspicious.

We progressively expanded our directory tree by creating sub-sets of our test data. Table 1 illustrates the number of files found, system execution times, real execution time, CPU and memory consumed. All times are approximated over 10 runs of the tool.

Files/GiB	sys time used	total exe time	CPU time	Memory
49,659/5.6	152.4 secs	773.4 secs	1.6 - 1.9%	0.1%
98,259/10.7	290 secs	796.5secs	1.9%	0.1%
146,583/14.8	410.3 secs	1163.7 secs	1.8 - 2.0%	0.1%

Table 1.

An interesting outcome of these tests is that the time taken for both the system execution times and real times are within very tight ranges, averaging 0.00815 ± 0.00002 secs and 0.0028 ± .000015 secs respectively. Using this it is possible to approximate the expected execution time of the software, given the data set used, and it should be possible to schedule execution during slack time. These estimates are specific to the set of tests we ran on our data set, and using our computer system, the outcomes will vary with different data sets and different hardware, virtual or otherwise. Memory use measured refers only to that used by the process, it does not include the RAM disk created as a write-blocker.

VI. Conclusion

The ACPO guidelines [5] recognise that when investigating live operating systems, alterations and changes occur to stored data without a users knowledge and interaction. These principles require that the impact of any tool used be known and explanation be given of any artefacts left by the tool. They also require those persons who analyse data to be able to explain their actions and the relevance of their actions, but do not exclude data that has been altered due to the investigation process being admissible, subject to satisfactory explanation. This may be relevant to our process, as there is a possibility that some log data may be written to the *system.journal* log file, thus causing a change that will require explanation.

We have designed our write-blocker to be used in conjunction with code we have written to forensically search a VM, and have designed and implemented it to prevent that code from writing to the vHDD. It is an instance specific write-blocker, working only within the code instance that invokes it, but it is an simple and effective way of solving the issue of write-blocking a VM. It will allow a vHDD operate as normal without impairing its interaction with the system and network. It offers a solution that is effective, auditable, simple and easy, this could be very important in demonstrating its forensic effectiveness.

We are currently carrying out a deep analysis of a vHDD to determine what, if any, log changes are made by OS

processes activated by this tool. We have not designed this as a standalone tool but as one to be embedded into forensic software and invoked with that forensic tool. We expect this research to fully support our hypothesis that our tool prevents writing to a VM when used in conjunction with our other tools.

The minimal impact our tool had on the suspect VM is illustrated by the Linux ps process in Table 1, this low level of vCPU usage is less likely to flag a user of unusual activity. Our tool does not prevent writes to log files, by preventing *system.journal* being written to could have the unwanted consequence where a question may be asked as to what else it blocks from that file. From a forensic standpoint it is important that we show the small footprint that our tool has on the suspect system, preventing writes to log files would be counter-intuitive.

The execution times are unnecessarily long and we are working towards reducing these by optimising our code. We are also examining a number of ways of extracting our data from the VM, one method being explored is by opening an *ssh* tunnel and exporting data collected through it. We also look at the feasibility of extending our approach to memory forensics of mobile devices [27].

REFERENCES

[1] Woods, K. and Lee, C.A., 2012, January. Acquisition and Processing of Disk Images to further archival goals. In Archiving Conference (Vol. 2012, No. 1, pp. 147-152). Society for Imaging Science and Technology

[2] Payne, B.D., De Carbone, M.D.P. and Lee, W., 2007, December. Secure and flexible monitoring of virtual machines. In Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual (pp. 385-397). IEEE

[3] Carrier, B. and Spafford, E.H., 2003. Getting physical with the digital investigation process. International Journal of digital evidence, 2(2), pp.1-20

[4] Goldberg, R.P., 1974. Survey of virtual machine research. Computer, 7(6), pp.34-45

[5] http://www.digital-detective.net/digital-forensics-documents/ACPO_Good_Practice_Guide_for_Digital_Evidence_v5.pdf

[6] https://www.ncjrs.gov/pdffiles1/nij/199408.pdf

[7] Tobin, P. and Kechadi, T., 2014, March. Virtual machine forensics by means of introspection and kernel code injection. In Proceedings of the 9th International Conference on Cyber Warfare & Security: ICCWS 2014 (p. 294)

[8] Adelstein, F., 2006. Live forensics: diagnosing your system without killing it first. Communications of the ACM, 49(2), pp.63-66

[9] Roussev, V., Quates, C. and Martell, R., 2013. Real-time digital forensics and triage. Digital Investigation, 10(2)

[10] Mislan, R.P., Casey, E. and Kessler, G.C., 2010. The growing need for on-scene triage of mobile devices. Digital Investigation, 6(3), pp.112-124

[11] Parker, D. B., & Parker, D. B. (1976). Crime by computer (pp. xii-xii). New York: Scribner

[12] http://libvmi.com/ - accessed 26.2.2016

[13] http://www.volatilityfoundation.org/#!about/cmf3

[14] http://man7.org/linux/man-pages/man8/mount.8.html - accessed, 26.2.2016

[15] http://man7.org/linux/man-pages/man8/blockdev.8.html - accessed, 26.2.2016

[16] https://github.com/msuhanov/Linux-write-blocker/blob/master/README.md, accessed 26.2.2016

[17] Barrett, D. and Kipper, G., 2010. Virtualization and forensics: A digital forensic investigator's guide to virtual environments. Syngress

[18] Garfinkel, T. and Rosenblum, M., 2003, February. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In NDSS (Vol. 3, pp. 191-206)

[19] Bahram, S., Jiang, X., Wang, Z., Grace, M., Li, J., Srinivasan, D., Rhee, J. and Xu, D., 2010, October. Dksm: Subverting virtual machine introspection for fun and profit. In Reliable Distributed Systems, 2010 29th IEEE Symposium on (pp. 82-91). IEEE

[20] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J. and Lee, W., 2011, May. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In Security and Privacy (SP), 2011 IEEE Symposium on (pp. 297-312). IEEE

[21] Fu, Y. and Lin, Z., 2012, May. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In Security and Privacy (SP), 2012 IEEE Symposium on (pp. 586-600). IEEE

[22] http://linux.die.net/man/8/mount accessed - 4.3.2016

[23] Ruan, Keyun; Carthy, Joe; Kechadi, Tahar; Survey on cloud forensics and critical criteria for cloud forensic capability: A preliminary analysis Proceedings of the Conference on Digital Forensics, Security and Law, Vol.55, 2011, Association of Digital Forensics, Security and Law

[24] Ruan, K; Carthy, J; Kechadi, Tahar; Cloud Forensics-: Key terms for Service Level Agreements. Advances in Digital Forensics VIII, Springer

[25] Plunkett, James; Le-Khac, Nhien-An; Kechadi, Tahar; Digital Forensic Investigations in the Cloud: A Proposed Approach for Irish Law Enforcement 11th Annual IFIP WG 11.9 International Conference on Digital Forensics, (IFIP119 2015)

[26] Kechadi, Tahar; Faheem, Muhammad; Le-Khac, Nhien-An; The State of the Art Forensic Techniques in Mobile Cloud Environment: A Survey, Challenges and Current Trends, International Journal of Digital Crime and Forensics Vol.7 No.2, Jan-19 2015, IGI Global

[27] Witteman, Rob; Meijer, Arjen; Nhien-An Le-Khac; Kechadi, M-Tahar; Toward a new tool to extract the Evidence from a Memory Card of Mobile phones 2016 4th IEEE International Symposium on Digital Forensic and Security (ISDFS) p.143-147, April 2016