



Computer Forensics:
The Persistence of Data in Physical Memory

Jason Michael Solomon
Bachelor of Computer Science (Honours)

Thesis Supervisor: Dr. Ewa Huebner

October 2006

School of Computing and Mathematics

Table of Contents

Table of Contents	i
Table of Figures.....	iii
Table of Tables	v
Abstract.....	vi
Statement of Authentication.....	vii
Acknowledgements.....	viii
Chapter 1 - Introduction	1
1.1 - Research Questions.....	1
1.2 - Significance of Research.....	2
1.3 - Research Methodology	3
1.4 - Thesis Overview	3
Chapter 2 - Computer Forensics	5
2.1 - Conventional Analysis	6
2.2 - Live Analysis	7
2.3 - Analysis of RAM.....	9
2.4 - Summary	10
Chapter 3 - Memory Management	11
3.1 - Memory Management System	11
3.2 - Virtual Memory	12
3.3 - User Address Space	15
3.4 - Copy-On-Write	16
3.5 - Block Device Cache.....	16
3.6 - Paging.....	18
3.7 - Page Fault Handling	18
3.8 - Summary	20
Chapter 4 - Methodology	21
4.1 - Description of Experiments	21
4.1.1 - Obtaining Memory Snapshots	22
4.1.2 - Hardware Used	23
4.1.3 - Existing Software Tools Used	24
4.1.4 - Additional Software Developed.....	25
4.2 - Data Collection.....	26

4.2.1 - Experiment Parameters	26
4.2.2 - Interval Versus Amount of Memory.....	27
4.2.3 - Storing Data	28
4.3 - Artificial Load	30
4.3.1 - Justification.....	30
4.3.2 - Creating Artificial Load.....	30
4.3.3 - Levels of Load	31
4.4 - Statistics Gathered.....	31
4.4.1 - Metrics Used.....	31
4.4.2 - Analysis of Page Hash Values.....	32
4.4.3 - Analysis of Final Memory Dump.....	33
4.5 - Summary	34
Chapter 5 - Evaluation of Results.....	35
5.1 - Empty Pages	36
5.2 - Unchanged Pages	39
5.3 - Total Changes.....	43
5.3.1 - Total Changes Within Memory	43
5.3.2 - Total Changes Per Page	46
5.4 - Intervals to Initial Change	51
5.5 - Intervals Between Changes.....	54
5.6 - Final Age of Data	57
5.6.1 - Experiment Statistics	58
5.6.2 - Analysis of Memory Snapshots	61
5.6.3 - Frequency Distribution of Types of Pages	64
5.6.4 - Analysis of Timestamps	65
5.7 - Summary	69
Chapter 6 - Conclusion	70
Future Work.....	73
References	74
Appendix – Source Code	76

Table of Figures

Figure 3-1: Common kernel space	13
Figure 3-2: Two-level page table	14
Figure 4-3: Two systems connected through a switch.....	28
Figure 4-4: Two systems directly connected	29
Figure 4-5: The difference in results when not connected to the external network.....	29
Figure 5-6: Empty pages in the first portion of physical memory	36
Figure 5-7: Empty pages in the second portion of physical memory	37
Figure 5-8: Empty pages in the third portion of physical memory	38
Figure 5-9: Empty pages average across all three portions of physical memory.....	39
Figure 5-10: Unchanged pages in the first portion of physical memory.....	40
Figure 5-11: Unchanged pages in the second portion of physical memory	41
Figure 5-12: Unchanged pages in the third portion of physical memory	41
Figure 5-13: Unchanged pages average across all three portions of physical memory ..	42
Figure 5-14: Total changes in the first portion of physical memory.....	44
Figure 5-15: Total changes in the second portion of physical memory.....	44
Figure 5-16: Total changes in the third portion of physical memory	45
Figure 5-17: Total changes average across all three portions of physical memory	46
Figure 5-18: Total changes per page in the first portion of physical memory	47
Figure 5-19: Total changes per page in the second portion of physical memory	48
Figure 5-20: Total changes per page in the third portion of physical memory.....	49
Figure 5-21: Total changes per page average across all three portions of physical memory	50
Figure 5-22: Intervals to initial changes in the first portion of physical memory.....	51
Figure 5-23: Intervals to initial changes in the second portion of physical memory	52
Figure 5-24: Intervals to initial changes in the third portion of physical memory	52
Figure 5-25: Intervals to initial changes average across all three portions of physical memory	53
Figure 5-26: Intervals between changes in the first portion of physical memory.....	54
Figure 5-27: Intervals between changes in the second portion of physical memory	55
Figure 5-28: Intervals between changes in the third portion of physical memory.....	55
Figure 5-29: Intervals between changes average across all three portions of physical memory	56

Figure 5-30: Final age of data in the first portion of physical memory	58
Figure 5-31: Final age of data in the second portion of physical memory	59
Figure 5-32: Final age of data in the third portion of physical memory	60
Figure 5-33: Final age of data average across all three portions of physical memory....	60
Figure 5-34: An example of a block device cache page	62
Figure 5-35: An example of a data segment page.....	62
Figure 5-36: An example of a text segment page	62
Figure 5-37: An example of an unidentified page related to file blocks.....	63
Figure 6-38: Counting memory page changes every hour over 402 hours (16.75 days) using MD5 hashes of memory pages (Red Hat Linux 6.1.) (Farmer and Venema, 2004)	71
Figure 6-39: Total changes observed per page	71

Table of Tables

Table 2-1: The expected lifespan of data (Order of Volatility)	7
Table 5-2: Frequency distribution of recognised pages in the second portion of physical memory	64
Table 5-3: Frequency distribution of recognised pages in the third portion of physical memory	64
Table 5-4: Age of block device cache data in the second portion of physical memory .	66
Table 5-5: Age of block device cache data in the third portion of physical memory	66
Table 5-6: Age of undefined file block pages in the second portion of physical memory	67
Table 5-7: Age of undefined file block pages in the third portion of physical memory .	67
Table 5-8: Age of data segment pages in the second portion of physical memory	68
Table 5-9: Age of data segment pages in the third portion of physical memory	68

Abstract

Computer Forensics is concerned with the use of computer investigation and analysis techniques in order to determine and collect potential legal evidence suitable for presentation in court.

The purpose of the research presented in this thesis is to gain an insight into how long data persists within physical memory and to draw a conclusion as to whether it is worthwhile developing a methodology and dedicated tools for the forensic investigation of physical memory.

Under the current method of incident response the contents of the volatile physical memory of a computer system are overlooked. Experiments were designed and conducted to provide an insight into the underlying behaviour of physical memory and the persistence of resident data. Looking at the results of these experiments, it can be said that a substantial amount of data does persist for a long time. The experiments however, are not concerned with the actual contents of memory. When the actual contents of memory were analysed, it was found that the majority of pages were not very old.

Although there was little forensically relevant data regarding files found within physical memory, it was statistically shown that a substantial amount of memory resident data does persist for quite some time. Further research is needed to study what this data is and its potential use forensically. The results presented in this thesis only hold true under Suse Linux 10.0, as this was the chosen Operating System for experimentation. More relevant data may be found under different operating systems.

Statement of Authentication

The work presented in this thesis is, to the best of my knowledge and belief, original except as acknowledged in the text. I hereby declare that I have not submitted this material, either in full or part, for a degree at this or any other institution.

.....

(Jason M. Solomon)

Acknowledgements

I would like to thank Dr. Ewa Huebner for being a fantastic mentor and friend. I would not have been able to see this thesis to completion without her guidance and support.

I would also like to thank my family for all their love and support and also for putting up with me through this very stressful year.

Finally, I would like to thank Timothy Boyce. The year was so much easier with a good friend going through it by my side.

Chapter 1 - Introduction

Computer Forensics is concerned with the use of computer investigation and analysis techniques in order to determine and collect potential legal evidence suitable for presentation in court. Evidence might be sought for during and after the occurrence of computer crimes or misuses, such as theft of trade secrets, theft or destruction of intellectual property, and fraud. Such evidence may be found in both logical places (files, caches, buffers, etc.), and physical places (hard disks, CDs/DVDs, removable media, main memory, etc.) within a computer system.

1.1 - Research Questions

The purpose of the research presented in this thesis is to gain an insight into how long data persists within physical memory and to draw a conclusion as to whether it is worthwhile developing a methodology and dedicated tools for the forensic investigation of physical memory. There are many places where evidence might be found within a computer system. Computer Forensics is a relatively new area of study and there are still some areas of computer systems that are being overlooked in investigations. The current method of incident response is to power off the computer system in question, and to take it to a computer forensics lab for analysis. However, when the computer system loses power, the contents of the volatile physical memory of the computer system are lost. There may be data resident in physical memory which is useful in a forensic investigation and by the current methodology, it is being overlooked.

To find out how long data will persist in physical memory, research was conducted on how the memory management system of an operating system works. With an understanding of the management of memory, experiments were designed and conducted to provide an insight into the underlying behaviour of physical memory and persistence of resident data.

A simplified, high-level explanation of the way a computer system uses physical memory is as follows. When a process is started, it is allocated a portion of physical memory which is needed for that process to run. After the process has completed and terminated, the memory which was used by this process is de-allocated. This de-allocation of memory simply means that the memory is no longer in use, however, the data which was stored in this portion of memory by the process, may still exist for some time after the process has terminated. The data will remain in memory until the memory is re-allocated to, and used by another process.

Another important and relevant function of physical memory is to serve as a cache for the file system. Any file opened and used by an active process has some of its contents copied from disk into physical memory, and all operations on files are performed on the copy in memory. The interesting side effect of this is that files which are encrypted or compressed on disk appear in clear text (or decrypted) in physical memory. The length of time that the data persists in physical memory varies depending on a number of factors, such as the level of activity / load on the computer system, the available size of physical memory, etc.

1.2 - Significance of Research

Currently, the only data gathered from physical memory is information about the state of the computer system. This data is gathered in a similar way to a crash dump analysis though the use of specialised tools.

Memory can potentially tell us more than just information regarding the current state of the computer system. When pages of memory are used by a process and the process

terminates, these pages are marked as free, but the data is not overwritten until the pages need to be reused by the system. There are no specialised software tools or techniques that have been developed which can be used to assist in collecting and analysing the data contained in these pages in such a way that is admissible in a court of law. It is simply overlooked. The data in these pages is invisible to crash dump analysis of physical memory because logically, the data no longer exists. This research will serve as a feasibility study for the further development of the forensic investigation of the physical memory of live computer systems which is the only way of recovering this data.

1.3 - Research Methodology

There are very few existing studies in this area of research. As such, the results of this research will be compared to the existing studies where possible, but for the majority, will be based on experiments conducted on a live test system.

The experiments will consist of the tracking of changes within the physical memory of the test system under various levels of system load. The tracking of these changes will show the rate of page decay within physical memory. The program used to create load on the system will place a known value into the physical memory of the test system so that it can be easily recognised in analysis.

1.4 - Thesis Overview

This thesis is composed as follows.

This chapter outlines the research questions, the significance of the research and the research methodology. It also provides a brief background on the current methodology of Computer Forensics Incident Response and the physical memory of a computer system.

Chapter 1 - Introduction

Chapter 2 provides a more detailed background of Computer Forensics and the conventional methodology used for incident response. It aims to show the value of developing methods of collecting the contents of physical memory for an investigation.

Chapter 3 provides a detailed background of physical memory and the memory management system of a computer, outlining the important mechanisms and how they would affect an investigation of the physical memory of a computer system.

Chapter 4 outlines the methodology used for experimentation. It describes the experiments conducted, the hardware and software (existing and developed) used, and discusses the metrics that were used to analyse the behaviour of physical memory.

In Chapter 5 the results produced by the experiments are presented, discussed and analysed and in Chapter 6 the conclusion which was drawn from the analysis of the results is presented and the possible future work is outlined.

Chapter 2 - Computer Forensics

Computer Forensics can be defined as the identification, analysis, preservation, and presentation of digital evidence in a legally acceptable manner. It deals not only with computer based crimes but with any crime that involves the use of a computer or electronic device in any way or form (Landman, 2002). Evidence might be sought for during and after the occurrence of computer crimes or misuses, such as:

- Theft of trade secrets
- Email spam or harassment
- Unauthorized or unlawful intrusions into computing systems
- Embezzlement
- Possession or dissemination of child pornography
- Denial-of-service (DoS) attacks
- Tortious interference of business relations
- Extortion
- Any unlawful action when the evidence of such action may be stored on computer media such as fraud, threats, and other traditional crimes.

(Mandia et al., 2003)

2.1 - Conventional Analysis

There are many places where digital evidence might be found within a computer system. The generally accepted methodology for incident response is to power off the computer system under investigation and take it to a computer forensics lab.

Once a computer system is in the lab, the system is booted from a removable disk (usually a DOS or Linux CD) to prevent making any changes to the hard disk storage. A write blocker is used to further prevent any inadvertent writing to the hard disks. Write blockers may be implemented using software after the system has been booted; however the safest method is a hardware device which is installed between the hard disk and the mother board of a system before it is booted to block any write commands from getting to the hard disk (Mohay et al., 2003). Once the write blocker is in place, a forensically sound copy of the hard disk(s) is made and analysis is carried out using the image(s) rather than the disk(s).

A forensically sound copy is an exact copy which does not alter the contents of the source disk in any way when created (Nelson et al., 2004). This is so that in a case where a second opinion is required by a court of law, another computer forensics team can make their own forensically sound copy of the disk, and obtain the same results. However, by shutting down the computer, the contents of volatile storage such as physical memory are lost.

Hard disks are getting bigger, and the increasing amount of data that can be stored on hard disk makes forensic investigation a lot more time consuming. A forensic investigation needs to be carried out fairly quickly, because the evidence is usually needed for a court case. If the evidence is not found before the case, then it is too late and the evidence is useless (Clark and Diliberto, 1996). The contents of physical memory may be able to provide investigators with an insight into where important evidence may be found within a hard disk storage and thus cutting down the time needed for the investigation.

2.2 - Live Analysis

There are several places within a computer system other than hard disk storage where data may be found. This data may contain important evidence. Certain types of data are more persistent than others. This is known as the order of volatility (OOV).

Registers, peripheral memory, caches, etc.	Nanoseconds
Main Memory	Nanoseconds
Network state	Milliseconds
Running processes	Seconds
Disk	Minutes
Floppies, backup media, etc.	Years
CD-ROMs, printouts, etc.	Tens of years

Table 2-1: The expected lifespan of data (Order of Volatility)

(Farmer and Venema, 2004)

With conventional analysis (powering off and seizing the system for analysis), the first four types of data in the OOV table are lost. These types of data may be important to a forensic investigation. Not only is data lost, but if the computer is under attack at the time of investigation, then powering off the system may shift the attacker elsewhere. For these reasons, live analysis was introduced.

The difference with live analysis as compared to conventional analysis is that there is no way to collect data without altering it. Anything done on the live system changes the contents of memory and various system logs.

Another issue in live system analysis is that local system utilities may be compromised, so for an investigation of a live system, only trusted software tools should be used (on a portable form of media such as floppy disk or CD). To circumvent the need to write to the hard disk and possibly disturb evidence, all live

data collected should be stored on portable media or on a remote system. Netcat is a specialised tool designed to send data across a network with as little memory footprint as possible (Wysopal, 2004).

In a live investigation, the following information is generally collected first and in the following order:

1. The current date and time (in order to establish the time frame of the live investigation)
2. Cache tables (arp, routing, etc.)
3. Network state (i.e. current, pending and open TCP/UDP ports)

(Burdach, 2004)

This information needs to be collected first as it is very high in the OOV. Once this information is collected, the investigator can move on to collect the following:

4. List of active processes
5. System configuration (hardware and software)
6. List of logged on users
7. MAC (Modification, Access, Creation) times of files on all drives
8. The current date and time (in order to establish the time frame of the live investigation)

(Prosise, 2003)

It is also important to document the exact commands or actions performed on the system under investigation. The investigator may be required to show this documentation in court to prove that their actions did not destroy evidence or create artifacts on the system.

While it is not possible to collect live data without disturbing the system, forensic tools are designed to reduce the impact on the system under investigation. At the time of incident response, there is no way to create a forensically sound copy of live data because, as was stated earlier, every activity carried out on a computer system will alter the contents of physical memory. As the live data collected is not forensically sound, and there is no standard set for live data collection, the data may not be

admissible as evidence in court. However, the data may be just as useful to aid investigators with conventional analysis.

Modern computer systems, especially servers, are rarely shut down and the time between restarts is substantial. This means that the physical memory of computer systems may contain a lot of valuable information and potential forensic evidence, which by the current methods of forensic investigation, are being overlooked.

2.3 - Analysis of RAM

Up until recently, the only investigation performed on physical memory was similar to the analysis performed by system administrators on crash dumps to find out the cause for a crash. Theoretically, the contents of physical memory can show everything about the state of the system at the time the memory dump was taken (open files, running processes, network state, logged on users, etc...). This information can be found through conventional crash dump analysis. However, the analysis of the complete image of physical memory is able to go beyond this information. When a process terminates, the pages of memory which were allocated to it are marked as free by the memory management system, but the actual contents of these pages are not overwritten until the memory is needed again. This means that this portion of memory actually contains data which is logically no longer part of the system. This data may prove invaluable to a forensic investigation as it will be totally invisible to conventional crash dump analysis and the only way to recover it, is by means of an in depth forensic investigation of physical memory.

As it was stated earlier, at the time of incident response there is no way to create a forensically sound copy of live data, however, Brian Carrier and Joe Grand (2004) created a hardware solution (known as TRIBBLE) which is capable of making a forensically sound copy of the physical memory of a computer system. The only problem with TRIBBLE is that it needs to be installed in the system before it can

capture the contents of physical memory. This means that at the time of incident response, it is too late to use TRIBBLE.

Previous studies into the persistence of data in physical memory through statistical analysis of page decay were conducted by Dan Farmer and Weiste Veneema (2004). These studies show that substantial data does persist within physical memory which may be useful in a forensic investigation.

2.4 - Summary

Computer Forensics is a relatively new area of study, and there is a lot of room for development. Hard disk storage is continually growing, and the amount of data that can be stored on a single hard disk makes investigation very time consuming, especially when the system under investigation contains more than one hard disk. The contents of physical memory may be able to provide forensic investigators with an insight into where there may be relevant data stored within the hard disk storage of the computer system under investigation. There may even be data which is relevant to the investigation resident within physical memory which under the current method of investigation is being overlooked.

Chapter 3 - Memory Management

For the Computer Forensic Analyst, understanding how operating systems manage physical memory or Random Access Memory (RAM) is very important. For example, the 'SQL Slammer Worm' (Symantec Corporation, 2004), resides solely in the physical memory of the computer it is attacking and never writes anything to disk. This means that without an understanding of physical memory, the analysis of a compromised machine would show no evidence of the worm. Computer Forensic Analysts are also concerned with activities which do not breach computer security, but are still against the law. An example of this might be someone who commits tax fraud, and keeps records on a computer. In this case, understanding RAM is equally as important, as the analysis of hard disk is very time consuming due to the volume of data. If the analyst could see what was in RAM, this may provide an insight to where to begin analysis on the hard disk. Understanding RAM may also prove useful as files which are encrypted on disk will appear in plain text (decrypted) form in RAM.

3.1 - Memory Management System

Each operating system differs in the way they manage the physical memory of a computer, but for most modern operating systems, the principle is the same.

Computers have a limited amount of physical memory which is usually not enough to satisfy the memory requirements of all running processes. To overcome this, processes do not have direct access to physical memory, they each have a byte addressable virtual address space and the use of physical memory is managed by the Memory Management Subsystem (MMS) of the operating system. This virtual

address space is divided into equally sized pages. Under the Intel32 (IA32) architecture, this size is 4KB, however, page sizes from 512 bytes to 64KB have been used in real systems (Tanenbaum, 2001). The physical memory is divided into page frames, which are the same size as the pages of virtual memory (Stallings, 2005). All access to memory is in units of pages. Each page of virtual memory which is in use by a process must be backed by some form of physical storage. As there is not enough physical memory to back all used pages of virtual memory for all running processes, the memory management system uses a portion of hard disk storage known as swap space. Swap space is broken into blocks of equal size to the pages of virtual memory and the page frames of physical memory. Each page of a processes virtual address space is backed by a page frame of physical memory or a block of swap space. This means that it may not be possible to contain all data belonging to all running processes in physical memory. From a forensics point of view, this means that at the time of data collection, there may be important data which is not resident in physical memory.

3.2 - Virtual Memory

The virtual address space of a process is split into two separate address spaces, a kernel space and a user space. Instructions and data relating to the operating system kernel are stored in the kernel space while instructions and data relating to the process are stored in the user space. While the user space of a virtual address space is different for each process, the kernel space is the same for all running processes as shown in Figure 3-1.

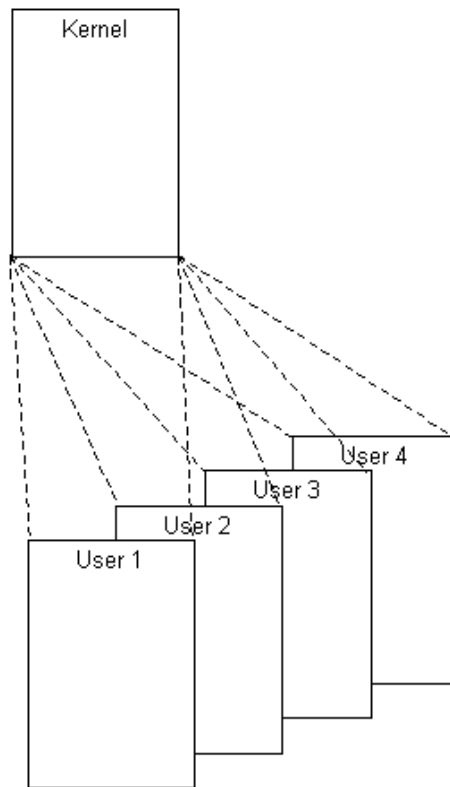


Figure 3-1: Common kernel space

Under a 32-bit architecture, virtual addresses are 32 bits in length. A 32-bit address means that the total virtual address space for each process is limited to 4GB. However, as was already stated, some of this space (usually 1GB) belongs to the kernel. The virtual addresses must be mapped into physical addresses. A page table for each process and for the kernel is used to store mappings from virtual to physical addresses. Each mapping corresponds to one page table entry (PTE). The difficulty with using a linear mapping is that there is generally a 3GB virtual address space for each process (4GB if you count the kernel space) which means that for each process, assuming that a page size of 4KB is used, there can be up to 1,048,576 pages of virtual memory. This means that the page table for each process would potentially need 1,048,576 entries. For this reason, a multi-level page table is used.

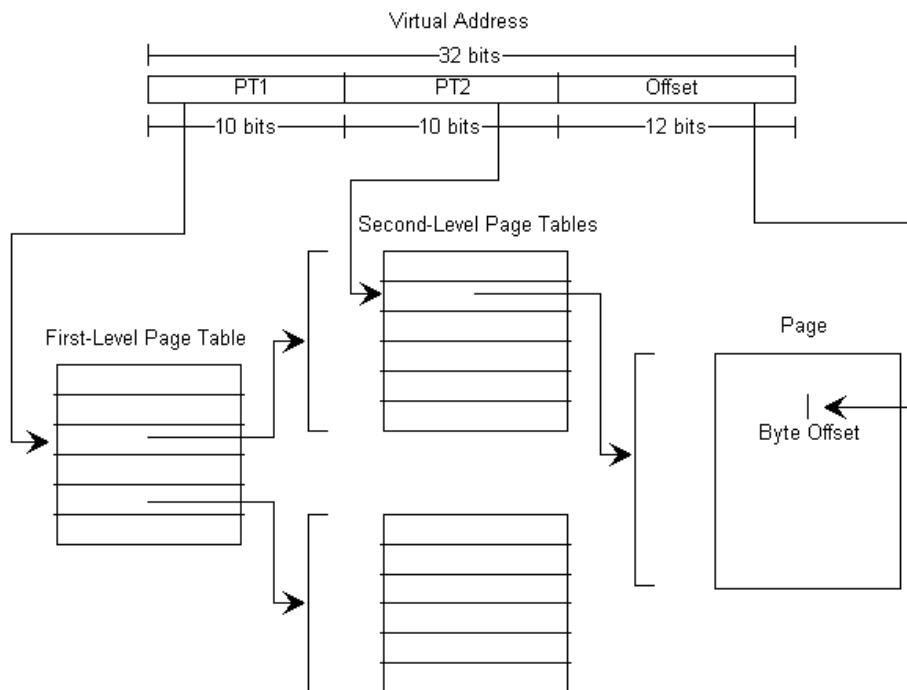


Figure 3-2: Two-level page table

The IA32 architecture uses a two-level page table, as shown in Figure 3-2. To implement a 2-Level Page Table, the linear address is split into 3 parts. The first part is an index in the first-level page table. Each index of the first-level page table points to a second-level page table. The second part of the address is an index within the second-level page table pointed to by the first-level page table. Each index of the second-level page table points to a physical page of memory. The third part of the address is the byte offset within the page in memory pointed to by the second-level page table. In the IA32 architecture, the first 10 bits of the address are the index in the first-level page table, the second 10 bits are the index in the second-level page table and the last 12 bits are the byte offset within the referenced page. As the offsets are 12 bits, page size of $2^{12} = 4\text{KB}$, and there are $2^{20} = 1,048,576$ pages. Depending on the system architecture, different multi-level page tables can be used.

3.3 - User Address Space

A process, when in user mode, only has access to the user space of its virtual address space. The user address space of a program is generally split into three segments.

- The *text segment* contains the instructions and constant variables of a process. It is allocated when the process is started and stays the same for the life of the virtual address space, i.e. it is read only.
- The *data segment* is working storage for the process. It can be pre-allocated and pre-loaded when the process is started and the process can extend or shrink it. The low address end of this segment is fixed.
- The *stack segment* contains a program stack. It grows as the stack of the process grows, but doesn't shrink when the stack shrinks. The high address end of the stack segment is fixed.

(Free Software Foundation Inc., 2006b)

All three segments of the virtual address space of a process may contain forensically relevant data. Pages recovered from the text segment of a process may contain the values of static or constant variables used within the owner process. Pages recovered from the data segment may contain the values of local variables used in the owner process. If the process reads from or writes to a file, the data to be read or written may also be found in pages belonging to the data segment. Pages recovered from the stack segment may contain parameters used in function calls within the owner process. However, if a variable is passed by reference, then the value of the variable will not be found in the stack segment, only a memory address pointing to the value being passed.

3.4 - Copy-On-Write

Quite often, more than one process will require access to the same page, for example the same shared library. It would be a waste of resources to load the same library into memory more than once, so the memory management subsystem loads the library into memory once and simply puts the mapping for this page into the page table of each process which requires it. This is also applicable when more than one process has an identical data segment. If one of the processes tries to modify a page of the shared data segment, the page is copied into a new page frame in memory and the page table entry for the modifying process is changed to reference the new copy. All other processes still reference the original un-changed page. Under the UNIX operating system, there is a single process from which all user processes are created. When a new process is started, it is forked from this single process (Davis and Rajkumar, 2004). This fork means that every process that is created (or spawned), shares the same data segment (and all other segments) as the initial process. They only receive their own data segment when they first write to it. The term used to describe this process is “Copy-on-Write”. Copy-on-Write may make it quite difficult for a forensic investigator to determine which process a page of physical memory belongs to as it may actually belong to more than one process.

3.5 - Block Device Cache

Access to secondary storage disks is a lot slower than access to memory. Secondary storage devices such as disks and tapes are known as block devices as the data is arranged in blocks of equal size. In order to improve system performance and reduce access to physical disk, the operating system uses an area of physical memory known as the block device cache. When there is a request for disk access, the block containing the data is read into the block device cache and then copied (or mapped)

from the cache to the address space of the process which requested it. This way, if the same data is required again, it can be taken from the cache rather than requiring another physical disk read. The block device cache uses physical memory which is not needed by the kernel and the user mode processes. As the demand for primary storage increases, the system will decrease the amount of memory used for the block device cache. The Linux operating system also maps pages of this buffer directly into the virtual memory areas of processes, so that if a process tries to read from a block device, it can directly reference the buffer rather than making unnecessary copies.

The actual device holding the file should only be accessed in two cases:

- If the file is not yet in the buffer, it will be loaded from the disk.
- If the contents in the buffer no longer match the file on disk, it will be written back to disk.

The file may not be written back to the disk immediately after the contents of the buffer are changed, there is usually a delay. This delay is used to improve system performance. Writing to disk is much slower than writing to memory, so blocks are written to disk in batches to maximise throughput. Under UNIX, there is a system call '*sync*' (Figgins et al., 2005), which will force all modified data in the block cache to be immediately written to disk. When the UNIX system is started up, there is usually a process called '*update*' which is started in the background, which continually loops, calling '*sync*' and then sleeping for 30 seconds (Tanenbaum, 2001). The block device cache may prove very useful to forensic investigators, as it contains pages of actual files on the hard disk. If these files are opened and some of the contents are loaded into the block device cache, and then the files are closed and deleted, the partial contents of the files which were loaded into the block device cache will still be resident in physical memory until the system needs to reuse them. Forensic investigators may then be able to recognise these block device cache pages and recover the data they contain.

3.6 - Paging

Paging is the moving of pages between physical memory and backing storage. The Memory Management System keeps a record of page frames in physical memory which are free. When a process needs memory, the Memory Management System allocates some of the free page frames to the process and adds entries to the process' page table to map the page frames to pages in the virtual memory address space of the process.

In the standard Linux system, the kernel space is in the last 1GB of virtual address space (0xC0000000 – 0xFFFFFFFF). This leaves the first 3GB of virtual address space for the user space (0x00000000 – 0xBFFFFFFF) (LinuxMM, 2006). User processes, while in user mode, do not have access to the kernel space of memory, and will trap to the kernel with an access violation if they attempt to access this region. As was stated earlier, process requests for user space memory are handled by the Memory Management System which allocates and frees page frames to each processes virtual address space as they are required. If there are no free page frames, the memory management system needs to move/swap pages to make room to service the request. This is known as demand paging.

3.7 - Page Fault Handling

Whenever a page is loaded into memory, an entry is added into the Translation Look-Aside Buffer (TLB). The TLB is a hardware cache for recently used page table entries (usually between 16 – 64 entries). When the CPU receives a memory access instruction, it checks the TLB for a mapping. If the mapping does not exist in the TLB, the CPU raises a page fault exception.

There are two types of page fault, hard and soft. A hard fault is when the Memory Management System needs to access the disk to read data in order to satisfy the

request while a soft fault is when fault can be satisfied without having to read in the page from disk. There are two cases where a hard fault may occur. One is if the page has not been loaded into memory in the first place, and the other is when the page was previously swapped out of memory. If the page has not been loaded into memory yet, the Memory Management System allocates a new page and, depending on the operating system, may zero the page out for security purposes before passing it back to the process.

When a page is swapped out of memory, the page table entry (PTE) for that page is updated with a reference to the page in swap space. So with this swap address, the system can check whether the referenced page is still resident in memory, and if it is, can replace this swap entry with a map to the physical address of the page. If the requested page is not in memory, then the system will read it in from swap space.

Along with the page requested, the page before and the page after the requested page may also be loaded into memory as these are likely to be the next pages needed (known as spatial locality) and this will usually cut down the number of reads which are needed from swap space. Depending on the operating system, if the swap space is filling up (more than 50% full for example), the system will also try to free up some swap space by checking to see which swap pages are no longer referenced by any processes and releasing the swap pages. This is interesting to note from the point of view of a Computer Forensic Analyst because it may mean that even though no processes reference the data in swap space, it may still remain there until re-allocated or freed by the system.

It is also important to note that swapping is not directly triggered by page faults. When the number of free pages goes below a preset limit, the operating system will begin swapping pages until it reaches another preset limit. Also note that different operating systems use different methods of determining which page to swap out of memory when swapping is necessary. This can have a dramatic effect on the contents of physical memory on a live system depending on the system load and how often page faults occur (so called page fault rate). If the fault can not be serviced, the

process is terminated. This is a precautionary method of making sure that the system does not enter an endless loop of trying to service the same fault.

3.8 - Summary

For the forensic analyst, it is important to have an understanding of the inner workings of the memory management system such as the logic behind the page replacement method used, as it can have a large effect on the results of the investigation. Memory management is a very complex area, and it is almost impossible to predict the exact content of physical memory based on the activity of the system. However, with knowledge of how the memory management system works, it is possible to estimate the most likely occurrences within the contents of memory, and this insight may prove invaluable for an investigation of the physical memory of a live computer system.

Chapter 4 - Methodology

This chapter describes the experiments conducted in the course of this research. It shows the hardware that was used, including specifications of computer systems and network devices used. It also describes the software tools used to conduct the experiments, some existing and some developed for the purpose of the experiments. This chapter also outlines what data was collected and what results and statistics were gathered through experimentation.

4.1 - Description of Experiments

The purpose of the experiments was to analyse the behaviour of physical memory under various loads and with the findings, reach a conclusion about the relevance of physical memory to forensic investigation. If data persists in physical memory for a substantial amount of time, it may prove to be crucial to a forensic investigation. Data present in memory which was used by a process which has already terminated logically no longer exists, and can never be found without examining the contents of physical memory and the swap file. This study is limited to the exploration of physical memory.

4.1.1 - Obtaining Memory Snapshots

The experiments conducted were periodic snapshots of memory contents (so called memory dumps) taken from a test system at a set interval of 30 minutes. One hundred memory dumps were taken in each experiment, making the total experiment length 50 hours. As each memory dump was saved, it was sent to a remote system for storage and processing. This was to reduce the footprint of the experiment programs and data in the physical memory of the test system. On the remote system, an MD5 hash value (Rivest, 1992) was obtained for each page of the memory dump and stored in a separate file. These hash values were then compared with the hash values from the previous memory dump in order to track changes to the contents of each page of memory.

An MD5 hash value is 128 bits long, meaning that there are 2^{128} possible values. The probability that the contents of a page changes and still produces the same hash value is 1 in 2^{128} . 2^{30} is close to one billion, so the chance of a clash is 1 in $2^8 \times 2^{30} \times 2^{30} \times 2^{30} \times 2^{30}$, i.e. approximately 256 billions of billions of billions of billions, which is statistically negligible.

Due to the total size of the memory dumps for each experiment (100 x 117.1875 MB = 11.44 GB), only the final memory dump from each experiment was retained along with the page hash values, and only the page hash values were retained for all the other memory dumps.

4.1.2 - Hardware Used

The following hardware was used to conduct the experiments:

Test System

Processor:	AMD Athlon 700 MHz
Physical Memory:	512 MB (131,072 pages)
Operating System:	Suse Linux 10.0
File System Type:	Ext3
File System Size:	10.2 GB
Network Interface Card:	Intel PRO/100+ Management Adapter (100 Mbps)

Remote system

Processor:	AMD Athlon 64 3200+ 2.2 GHz
Physical Memory:	512 MB
Operating System:	Windows XP Professional SP2
File System Type:	NTFS
File System Size:	38.1 GB
Network Interface Card:	Realtek RTL8169/8110 Family Gigabit Ethernet (Although this NIC is capable of 1 Gbps, it was only run at 100 Mbps due to the limitations of the other hardware used.)

Network

- Netgear Fast Ethernet Switch (100 Mbps) (Connected to the external network)
- Cross-over cable (Direct connection to the remote system)

The use of this hardware may have limited the scope of the experiments as it was found that while obtaining hash values for each page in the memory dumps, AMD

Athlon processors would occasionally hang for almost a minute at a time. This increased the time taken to process each memory dump significantly. As explained in Section 4.2.1, the entire physical memory was not collected in each memory snapshot; only a portion was taken due to the processing time. If a different processor was used, it could have been possible to calculate the hash values faster, and as a result, it could have been possible to process a larger portion of memory within the 30 minute interval.

4.1.3 - Existing Software Tools Used

A number of software tools were used to obtain and process the data produced by each experiment.

***dd* (Free Software Foundation Inc., 2006a)**

dd is used to make an exact bit-by-bit copy of a file. However, if it is applied to the `/dev/mem` file under the Unix system, it can be used to make a copy of the contents of physical memory (privileged access to the computer system is needed in order to access `/dev/mem`).

***Netcat 1.10* (Wysopal, 2004)**

Netcat 1.10 is an updated release of Netcat, a simple Unix utility which reads and writes data across network connections using TCP or UDP protocol. It can be used to send the output of *dd* over the network to the remote system for storage and processing. This reduces the impact of the experiment on the physical memory of the test system.

***md5sum* (Free Software Foundation Inc., 2006a)**

md5sum is used to convert a series of bits, in this case each page of physical memory, into a unique 128-bit string represented as 32 hexadecimal digits. With these strings (hash values), it is possible to track changes to each page in memory.

4.1.4 - Additional Software Developed

A number of additional software tools were developed to set up the experiments and process data.

artificialLoad

A program written in C which is invoked before the experiment starts, to simulate load on the test system. The program loads virtual memory, file cache and disk I/O by first creating a file, writing to it, and then reading it back into memory (a more detailed explanation can be found in Section 4.3.2).

PAL

A program written in C which starts multiple, concurrent instances of *artificialLoad*. Multiple concurrent instances are needed to simulate different levels of load on the test system. This program takes as a parameter the number of instances to start. It simply loops as many times as required, starting an instance of *artificialLoad* in each iteration of the loop.

memDump

A *bash* script to be run on the test system to automate each experiment. The script sets the parameters of the experiment, such as the interval between memory dumps and the amount of memory to collect.

Data Collection Scripts

A set of scripts which run on the remote system to automate the receiving of memory dumps as they are sent by the test system and the conversion of these memory dumps to page hash values.

AnalyseHash

A program written in C which analyses the page hash files produced by the experiments and compiles various statistics.

AnalyseDump

A program written in C which analyses the final memory dump from each experiment. It identifies the three types of pages created by the *artificialLoad* program: block device cache pages, data segment pages, and text segment pages. The block device cache and data segment pages contain timestamps. The program outputs these timestamps so that they can be used to determine the exact age of the page relative to the end of the experiment.

The source code for these programs is available in the Appendix.

4.2 - Data Collection

4.2.1 - Experiment Parameters

The possible variables of the experiments were the length of each experiment, the level of load on the system, the interval between memory dumps, and the amount of memory to process.

The length of each experiment was set to 50 hours as this would give results relevant to forensic investigation. If the experiments were too short, the results would be of lesser value as we would gain less insight into the rate of decay of pages in memory.

The length remained constant for every experiment, while the level of load on the system varied. This variation provided a view of the changes in activity of physical memory under different levels of load.

Due to storage limitations, each memory dump had to be processed and discarded before the next memory dump was received. It became clear that the entire contents of memory took a substantial time to process into page hash files. If this interval was used between memory dumps, it would have allowed for using the entire contents of memory, but would have left too large a gap between each memory dump to give a detailed view of the activity of memory within the test machine. It is expected that a significant number of pages may change within as short a time as 15 minutes. For this reason, a compromise was needed for a more fine-grained view of the activity of physical memory. The amount of memory to be used in the experiment had to be decreased to approximately one fifth of the total size of physical memory.

4.2.2 - Interval Versus Amount of Memory

The collection interval was based on the ratio between the amount of memory to be used and the time it takes to calculate all hash values. Through experimentation, it was found that a 30 minute interval allowed the hashing of 30,000 pages of memory (22.89% of the total memory) and this gave a reasonable view of the activity of memory. If too many pages were used, then the remote system would not complete processing the pages into hash values in time to receive the next memory dump sent by the test system and the memory dump would be lost, or the available storage exhausted.

Another factor to consider was which portion of memory to use. If the first 30,000 pages were processed it might not give an accurate representation of the activity of physical memory as the low addresses of memory are likely to be used by the system rather than the user processes. It was decided that the first 30,000 pages would be skipped, and the second 30,000 pages of memory would be collected for

experimentation. This was easily implemented using the *skip* parameter of *dd*. The *skip* parameter instructs *dd* not to begin input at the start of file, but instead to skip *x* blocks. The block size was set to 4096, so that instructing *dd* to skip 30,000 blocks would equate to 30,000 pages.

4.2.3 - Storing Data

In order to reduce the footprint of the experiment programs in the memory of the test system, the dumps were sent across the network using *Netcat* and all processing was done on a remote computer.

For the experiments, the computers were connected in two different ways. The first of which was to use a network switch which both computers were connected to. The switch, in turn was connected to the external network.

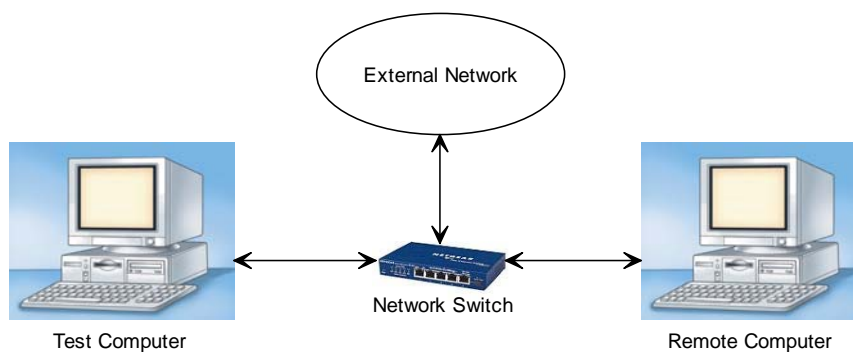


Figure 4-3: Two systems connected through a switch

The second method was to use a cross-over cable, so that the two computers were directly connected, and completely isolated from the external network.

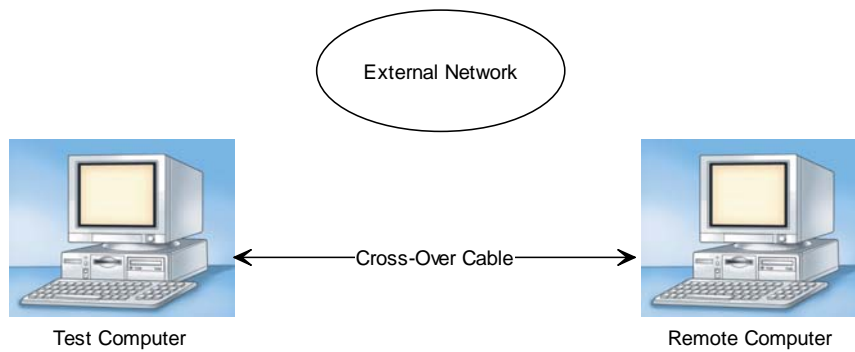


Figure 4-4: Two systems directly connected

The reason for this was to determine whether or not being connected to the external network made a difference to the results of the experiment.

It was found that the external network did not make a substantial difference to the experiment results as shown in Figure 4-5.

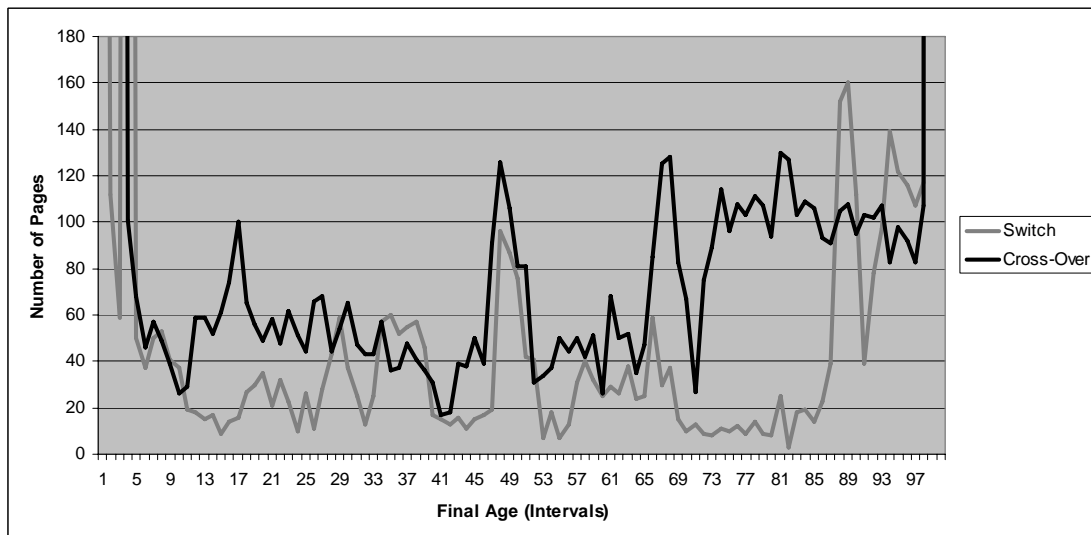


Figure 4-5: The difference in results when not connected to the external network

4.3 - Artificial Load

4.3.1 - Justification

The contents of memory on an idle system will change very little over time as there is no demand for pages, so conducting the experiments on a test system without some kind of load would not give accurate results for real situations. For obvious reasons, we were not able to investigate a live production system. The only aspect of the system to be measured is the frequency and extent of changes to physical memory, which can be achieved through the use of simple programs with predictable memory requirements.

4.3.2 - Creating Artificial Load

Artificial load is measured by the number of concurrent instances of the *artificialLoad* program there are running on the system for the duration of the experiment. The program first creates a file, and then writes a unique string followed by a timestamp. This is so that it will be possible to quickly identify the string within a memory dump and view the timestamp in order to ascertain how long the data actually persisted in physical memory. The file is then padded with 0s until the file size is 4096 bytes (one page). The file is then closed and re-opened. The 4096 bytes are then read back from the file into memory and held there for 30 seconds. The file is then deleted.

The program begins by calling the *fork()* function which will create a second process (child process). The child process creates the load as described above, and then terminates. Once the child process terminates, the original process (parent process) loops to create another child process which carries out the same loading function. This continues for the duration of the experiment. The artificial load processes are

terminated automatically by the data collection scripts upon completion of the experiment.

4.3.3 - Levels of Load

Different levels of load are achieved by starting a different number of instances of *artificialLoad* concurrently. For every concurrent instance of *artificialLoad*, a child process is created. This way there are multiple child processes which all compete for resources at the same time.

The maximum number of instances which were able to run concurrently on the test computer was found to be 2,500. When more instances were started, the system became unresponsive and it was impossible to start the actual experiment program. Each of the instances of the *artificialLoad* program puts demands on all system resources, including memory, CPU and disk I/O. It is obvious that some or all of these resources were exhausted with more than 2,500 instances of the *artificialLoad* program running. The range of different loads which were used for experiments, measured by the number of concurrent processes was: 0, 50, 100, 1000, 2000 and 2500.

4.4 - Statistics Gathered

4.4.1 - Metrics Used

There were six metrics collected in order to investigate the dynamic memory behaviour of the test system:

1. The number of pages which remained empty throughout the whole experiment. It was found that a completely empty page produced the MD5 hash value '620F0B67A91F7F74151BC5BE745B7110', so whenever the *AnalyseHash* program read a hash with this value, the page was marked as empty.
2. The number of non-empty pages which remained unchanged throughout the whole experiment.
3. The total number of changes within each page and the total number of changes which occurred in the selected portion of memory regardless of which page throughout the whole experiment.
4. The length, in intervals, until the first change for each page, excluding the initial loading of pages (i.e. the page contained no data to start with).
5. The length in intervals to all changes including the first change, and excluding the initial loading of pages.
6. The final age of data in each page.

The first five metrics are used to monitor the changes in memory activity when the test system is under various levels of load. The final age of pages is used to show how much data persists in memory while the test system is under different levels of load and for how long.

4.4.2 - Analysis of Page Hash Values

As the memory dumps are processed into page hash values, the page hash files are created and named using the convention *hashx*, where x is the consecutive number of the memory dump. So following this convention, the first memory dump is processed to give a file called hash1. The second will be hash2, all the way up to hash100.

The *AnalyseHash* program reads two hash files at a time, initially hash1 and hash2. It checks the hash value for each page for differences between the two page hash files. If

a page does not change, a counter is incremented to keep track of how long it has been since that page last changed. If the page does change, the number of intervals that the page persisted for is recorded and the counter is reset. After checking all the pages in the hash file (reach the end of file), the program will loop and read in the next two hash files for comparison, which will be hash2 and hash3. This continues until a file which the program tries to open does not exist. In this case, hash101 does not exist as the experiment is only 100 intervals in length. At this point, there are no more page hash files left to process and all the statistics are output to a text file for this experiment.

4.4.3 - Analysis of Final Memory Dump

Along with the statistics gathered, the final memory dump of each experiment was analysed using the *AnalyseDump* program (see Section 4.1.4). The *AnalyseDump* program searches each page of the memory dump looking for the unique string used in the *artificialLoad* program. For each page containing the unique string, the program determines the type of the page. The type of page is defined by the position of the unique string within the page. If the unique string is at offset 0 within the page, then the page is related to the block device cache. If the offset within the page is not 0 and there is no timestamp following the string, then the page is part of the text segment of the *artificialLoad* program. If the offset within the page is not 0 but there is a timestamp following the string, then the page is part of the data segment of the *artificialLoad* program. Once the type is determined, the program reads the timestamp (unless the page is a text segment page). As the end time of the experiment is known, the timestamp shows exactly how old is the data held in the page.

4.5 - Summary

The experiments were based on collecting periodic memory dumps which were processed into page hash files containing a 128-bit hash value for each page of the memory dump. These hash files were used to track changes to each page in the memory dump over time. Through analysis of these page hash files, six metrics were derived from each experiment in order to investigate the dynamic memory behaviour. With these results we can show how long memory contents persist and how much information can be obtained about the past behaviour of the system.

Chapter 5 - Evaluation of Results

This chapter describes and evaluates the results of experiments conducted in the course of this research. All together 27 experiments were run, most for the duration of 50 hours, corresponding to 100 sampling intervals excluding the initial experiments which were mainly used to fine tune the experiment parameters. Due to the fact that they were used to fine tune the experiment parameters and hence the parameters used for these initial experiments differ from the constant parameters which were eventually set for all subsequent experiments, these initial experiments could not be included in the results. There were also some experiments which did not complete due to power failures mid-way through. All together, there were 20 usable experiments conducted.

There were two variable factors in the experiments: the portion of memory analysed and the level of load on the system. Because of the physical limitations discussed in Chapter 4, three equally sized portions of physical memory were used. Initially, the first 30,000 pages were used, however it was decided that it is likely that the operating system allocates these low addresses of memory to the kernel. For this reason, most of the experiments were conducted using either the second or the third 30,000 pages of physical memory.

This chapter will present and analyse the results to attain an insight to the rate of decay of pages in physical memory. A conclusion will then be reached as to how much information on the past usage of a computer system can be derived from a snapshot of the contents of physical memory.

5.1 - Empty Pages

The count of pages which remained empty throughout each experiment shows that the increasing load on the system is affecting the activity of the physical memory of the test system. It is expected that as the level of load on the test system increases, the count of pages which remain empty will decrease as the demand for pages is higher.

Figure 5-6 shows the count of pages which remained empty throughout each experiment of loads 0, 50 and 100 using the first 30,000 pages of physical memory.

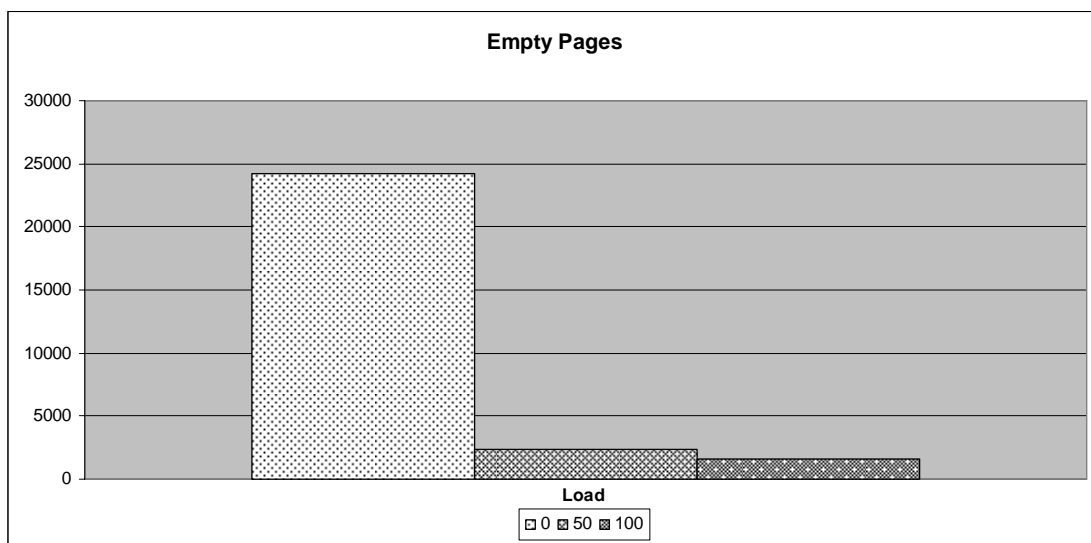


Figure 5-6: Empty pages in the first portion of physical memory

As seen in Figure 5-6 within the first 30,000 pages of memory, the introduction of load has a substantial effect on the count of empty pages. Only the first three levels of load were tested on the first 30,000 pages of memory as it is likely that the operating system kernel uses the low addresses of physical memory and it was decided that a better insight into the behaviour of physical memory would be obtained using the second and third 30,000 page portions of physical memory which are more likely to be allocated to user processes.

Figure 5-7 shows the count of pages which remained empty through experiments with all levels of test load using the second portion of physical memory.

The experiment with a load of 50 was run five times and the experiment with a load of 100 was run twice in order to observe the differences in having the two computers directly connected to each other or connected through the external network. The difference was negligible, so these two columns in Figure 5-7 are the average result over all experiments with the same load.

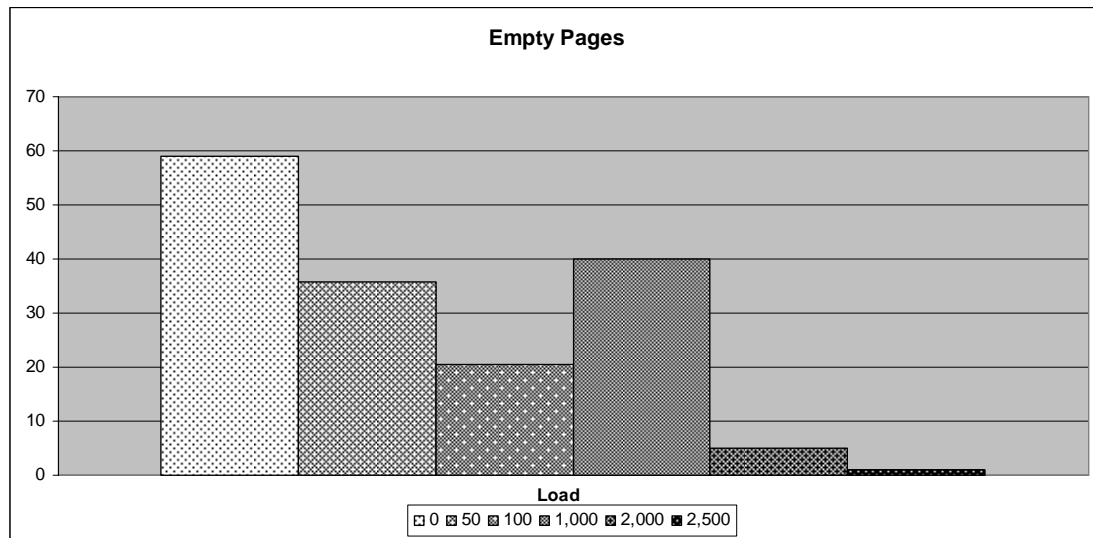


Figure 5-7: Empty pages in the second portion of physical memory

It can be seen in Figure 5-7 that as the load increases, the count of pages which remain empty throughout the whole experiment decreases. However, the activities of physical memory are dynamic and are very difficult to predict. This is also visible in Figure 5-7. The result produced by the experiment run with a load of 1,000 does not follow the trend set by the other five experiments. This may be due to the fact that we are only looking at 30,000 pages and the activity may have moved to another portion of memory after this point. However, this may also be an example of the aberrant behaviour of physical memory. It is not possible to predict what will definitely happen to the contents of physical memory, only what is most likely to happen.

Figure 5-8 shows a graph of results from the same experiments, only this time using the third portion of physical memory.

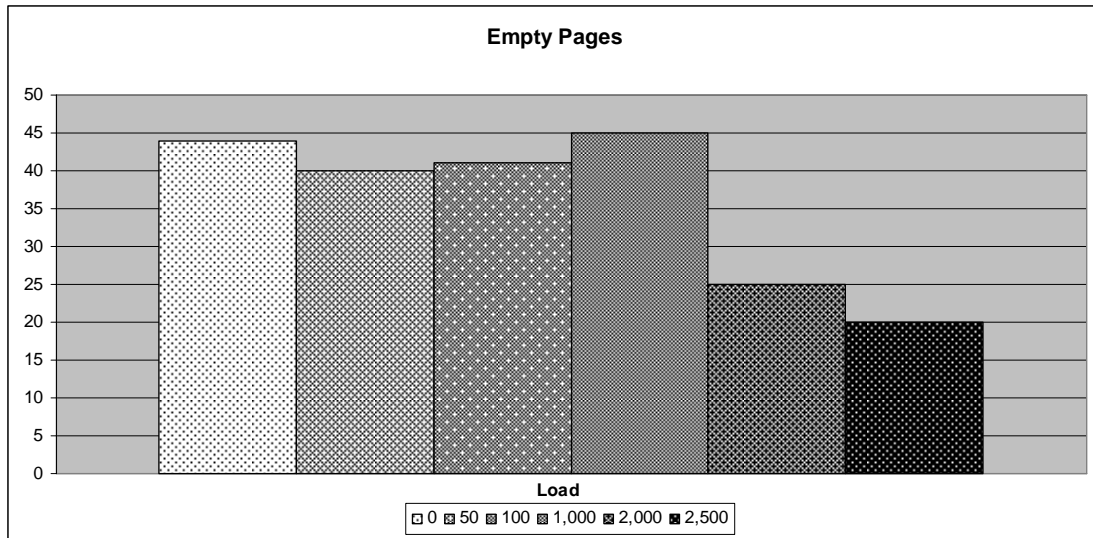


Figure 5-8: Empty pages in the third portion of physical memory

By comparison of the three graphs, it can be seen that the *artificialLoad* program has more effect on the first and second portions of physical memory than it does on the third. The third portion is only really affected once the load gets to 2,000. This would indicate that physical memory begins filling as required from lower addresses, and only when these addresses are filled will the higher addresses of the third portion of memory be used.

For an overall view, Figure 5-9 shows the average results of all experiments with common levels of load over all three portions of physical memory.

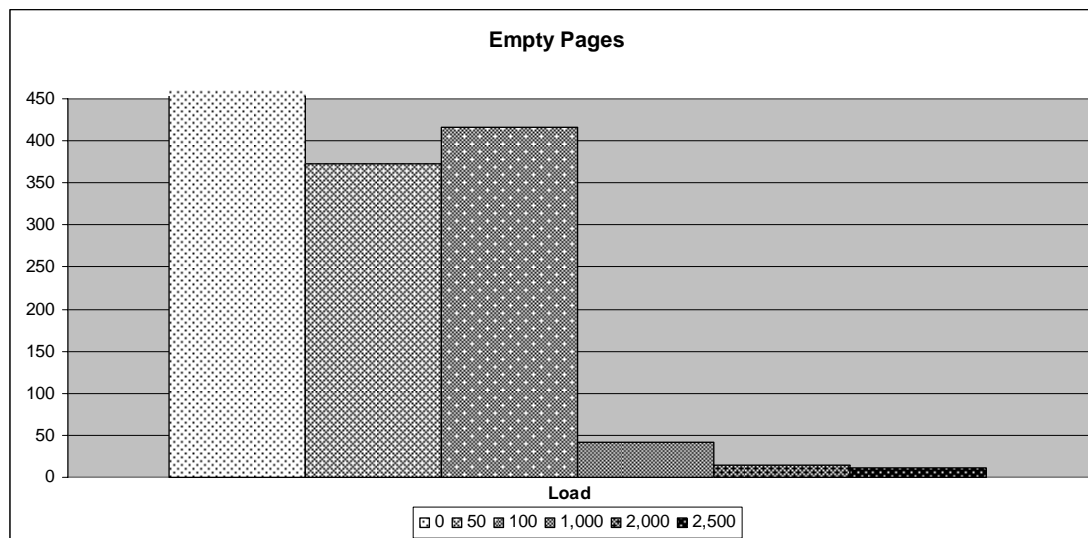


Figure 5-9: Empty pages average across all three portions of physical memory

The result for load 0 was actually 8115 empty pages, but as the count of empty pages is expected to be high for an unloaded system, the graph was cropped to give a more detailed view of the other five results.

As was expected, the count of pages which remain empty throughout each experiment generally decreases when the system is under greater load. This does not reveal much from a forensics point of view; however, it shows the relevance of using the selected portions of memory as the effects of the *artificialLoad* program can clearly be seen.

5.2 - Unchanged Pages

The count of pages which remain unchanged throughout each experiment shows that data does in fact stay resident in memory, however, it is expected that the amount of unchanged pages will decrease as the load on the test system is increased. This is because the demand for pages of physical memory will increase and more pages marked as free will be reused. This creates more changes in physical memory because pages marked as free are not necessarily empty. They may have been removed from resident sets of processes, either still active or terminated. Furthermore, once the

count of free pages drops below a preset limit, the memory management system will need to begin swapping pages out of physical memory to make room for new data.

Figure 5-10 shows the count of pages which remained unchanged throughout each experiment in the first portion of physical memory under loads 0, 50 and 100.

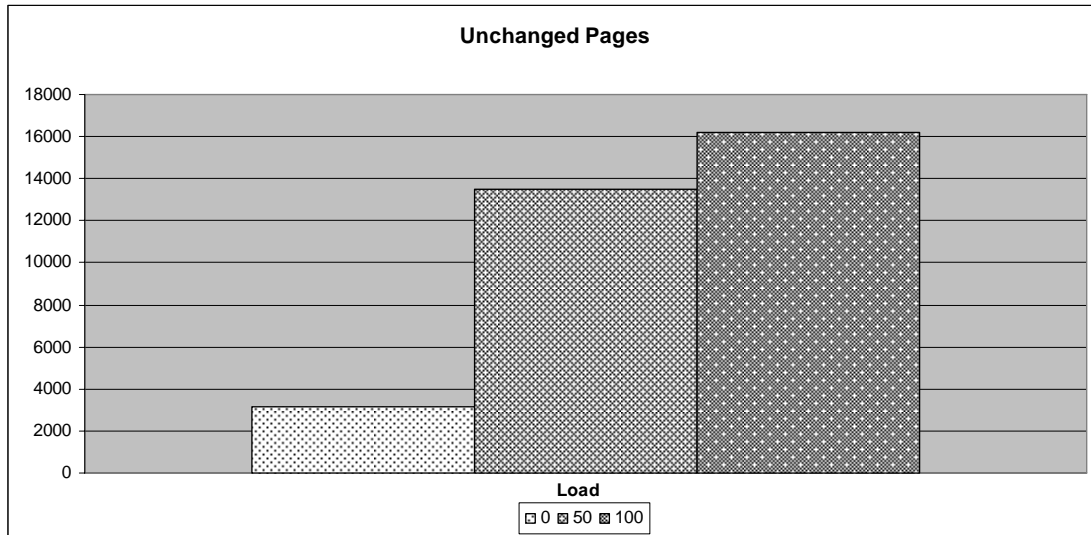


Figure 5-10: Unchanged pages in the first portion of physical memory

It can be seen that the count of unchanged pages actually increases as the load on the test system increases. However, it was seen in the previous section that for an unloaded system, almost 85% of pages in the first portion of memory are empty. This does not leave much to change. As the load increases, the count of empty pages drops, leaving more pages which can potentially change.

Figure 5-11 shows the count of pages which remained unchanged in the second portion of memory under all levels of test load.

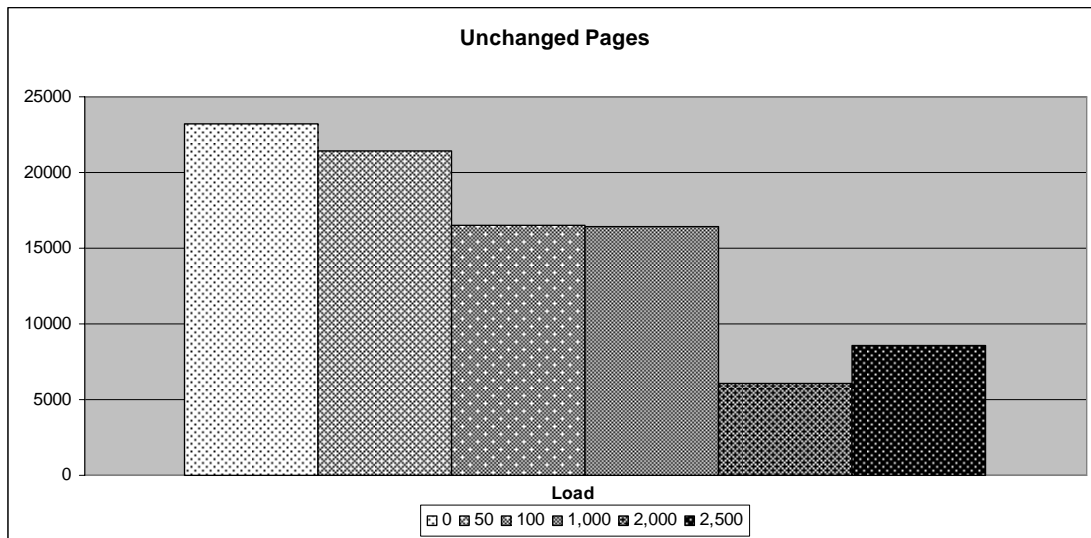


Figure 5-11: Unchanged pages in the second portion of physical memory

Allowing for the nondeterministic behaviour of the operating system, this graph follows the expected trend. As the level of load on the test system increases, the count of pages which remain unchanged throughout each experiment generally decreases.

Figure 5-12 shows the results produced by the same experiments conducted on the third portion of physical memory.

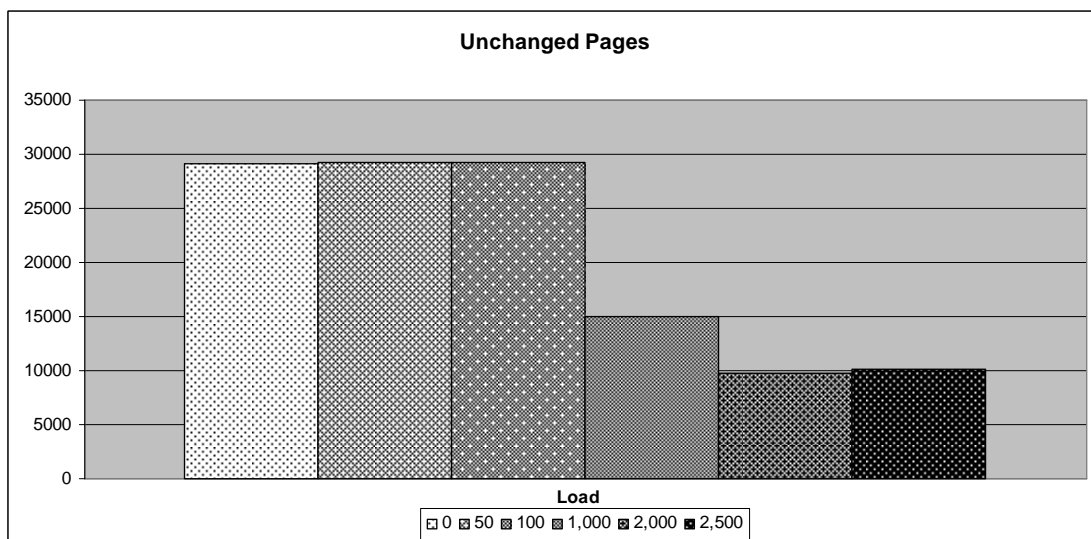


Figure 5-12: Unchanged pages in the third portion of physical memory

The count of unchanged pages in the third portion of physical memory is only really affected for the higher levels of test load. This confirms the results for the count of pages which remained empty within the third portion of memory.

Figure 5-13 shows the average result of all experiments with common levels of load over all three portions of physical memory.

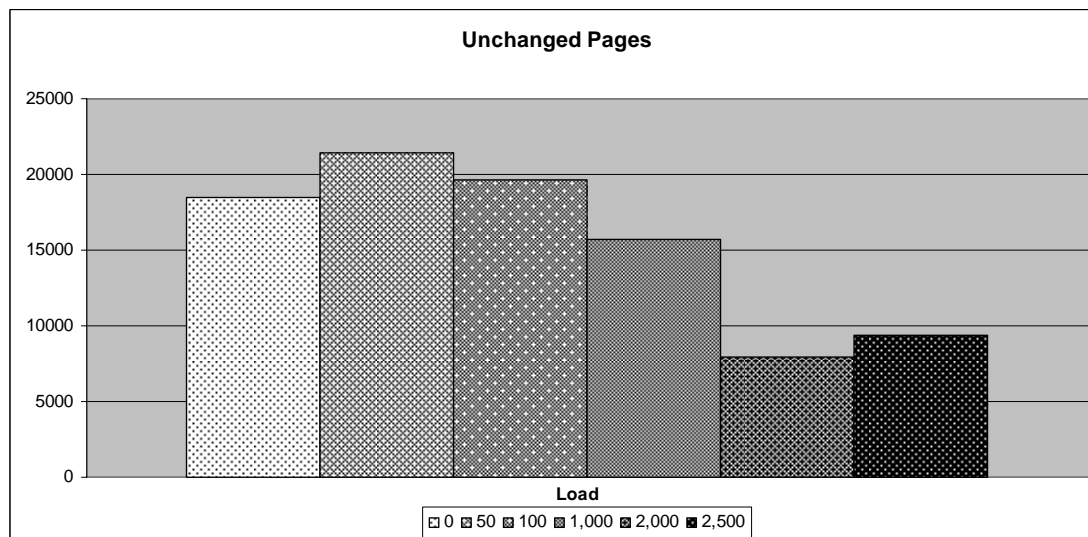


Figure 5-13: Unchanged pages average across all three portions of physical memory

The graph produced by the average results from all three portions of physical memory follows the expected trend of diminishing as the level of test load increases.

The count of pages which remain unchanged throughout each experiment confirms the results shown for the count of pages which remain empty throughout each experiment. From a forensics point of view, this shows that even with the highest level of load, data still persists within physical memory for the entire length of each experiment.

5.3 - Total Changes

The total count of changes within physical memory throughout each experiment will show how likely it is that data will persist. From a forensics point of view, the count of changes is not what is important. It is impossible to say how many changes occurred precisely, because a page may change more than once in a single interval. It is important to note what is implied by a change to a page. This may mean that the data in the page was simply modified by the owner process, but it could also mean that the data has been swapped out or overwritten. So what is important from a forensics point of view is the rate of decay of pages. The more changes there are within physical memory, the more likely it is that the data will be overwritten.

5.3.1 - Total Changes Within Memory

The total count of changes which occur in each portion of memory throughout the experiments gives an overall view of the activity of physical memory. This will be broken down in the next section to give a more detailed view of the rate of page decay.

Figure 5-14 shows the total count of changes which occurred in the first portion of memory under loads 0, 50 and 100.

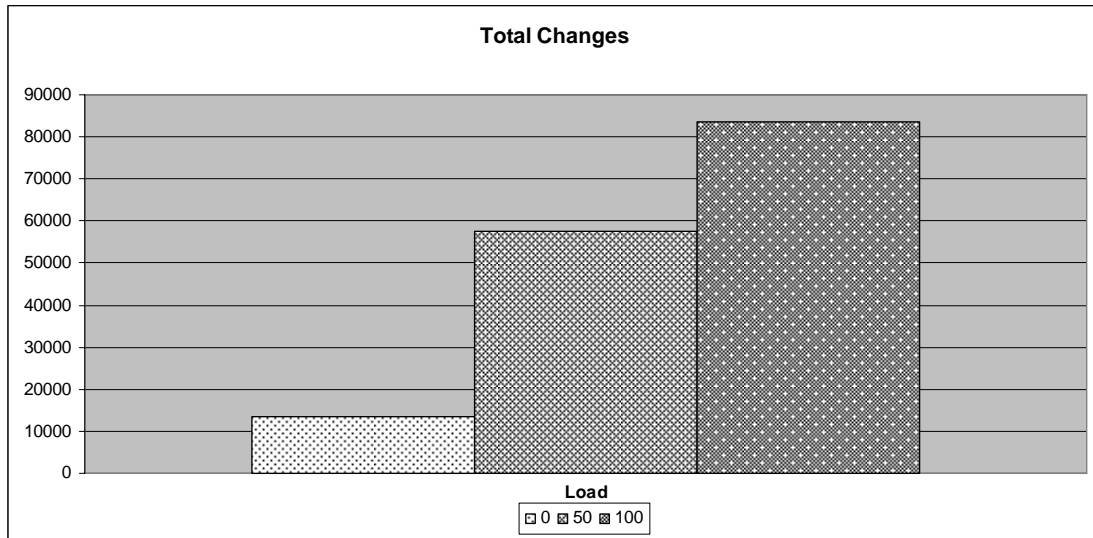


Figure 5-14: Total changes in the first portion of physical memory

Figure 5-15 and Figure 5-16 show the total count of changes under all levels of test load which occur in the second and third portions of memory respectively.

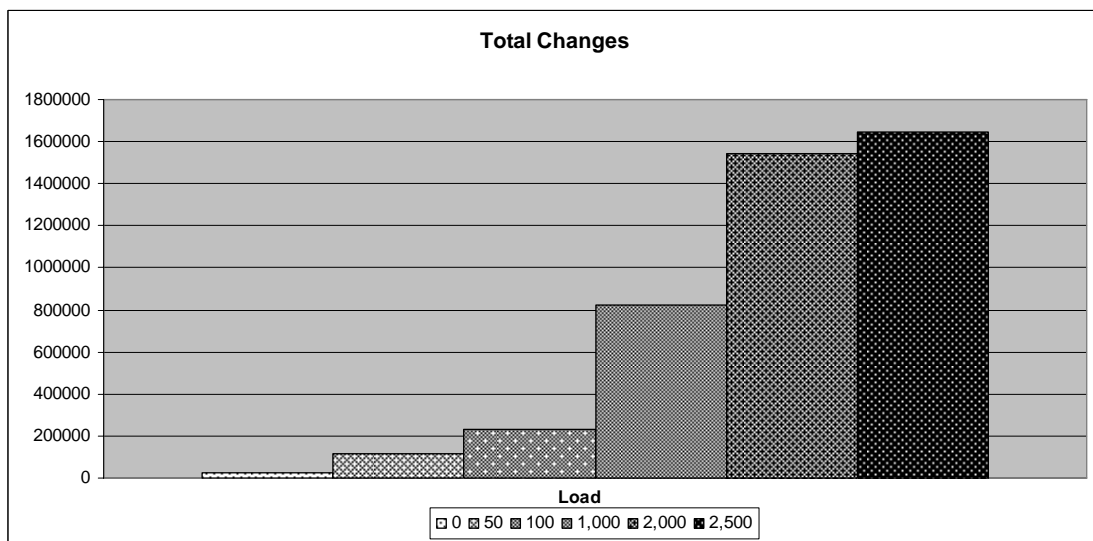


Figure 5-15: Total changes in the second portion of physical memory

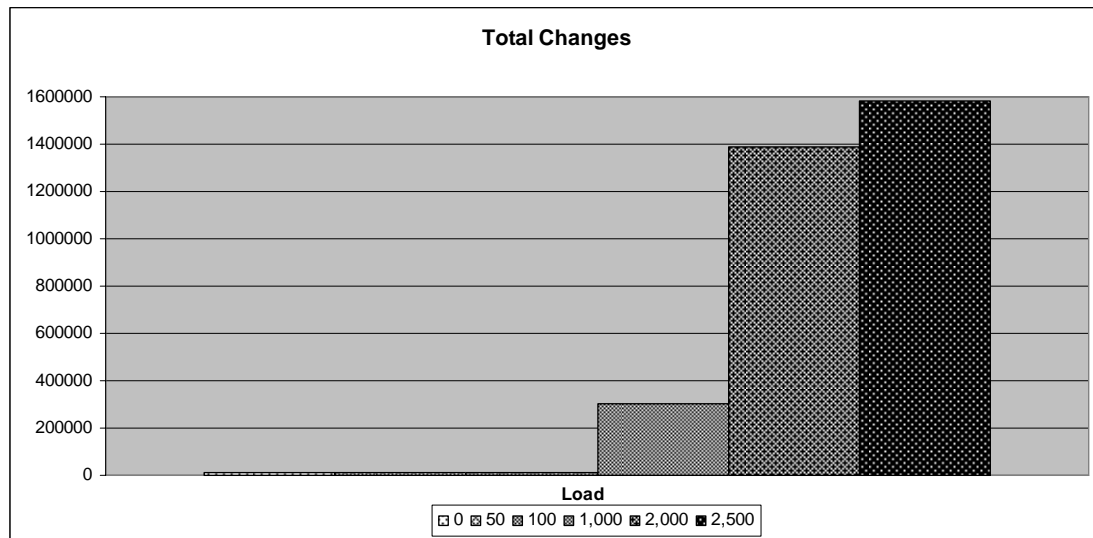


Figure 5-16: Total changes in the third portion of physical memory

As would be expected, the graphs show that when the level of load is increased, more changes occur within physical memory. Figure 5-16 shows that the count of changes which occur in the third portion of memory is not affected until the level of load reaches 1,000, again confirming the hypothesis that the higher addresses of memory are not used until the lower addresses are exhausted.

Figure 5-17 shows the average results from all experiments with common levels of load across all three portions of memory.

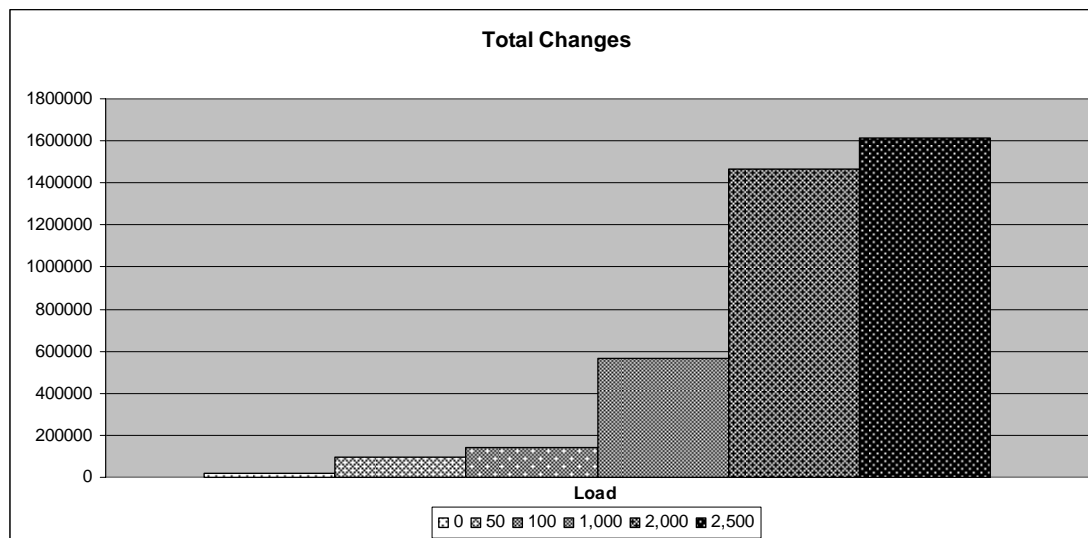


Figure 5-17: Total changes average across all three portions of physical memory

It can be seen in Figure 5-17 that as the load increases, the total count of changes within physical memory also increases.

From a forensics point of view, this gives an idea of how likely it is that data will persist in physical memory. The more changes there are within physical memory, the more likely it is that data will be overwritten sooner. Inversely, this indicates that on a lightly loaded system, it is unlikely that data resident in physical memory will be overwritten for quite some time.

5.3.2 - Total Changes Per Page

The total count of changes per page in physical memory gives a more detailed view of the behaviour and activity of memory.

All of the graphs presented in this section are cropped as the columns representing the pages which change between 1 and 10 times on each graph are so high that if they were shown in full, the other columns would be so small in comparison, that they would not be able to be accurately read. The count of pages which change 99 times also tends to be high, especially under higher levels of load. This is because the value

is a 'catch-all'. It actually represents all pages which changed 99 times or more as it is quite possible that a page changes more than once within an interval.

Figure 5-18 shows the count of changes which occurred per page in the first portion of physical memory under loads 0, 50 and 100.

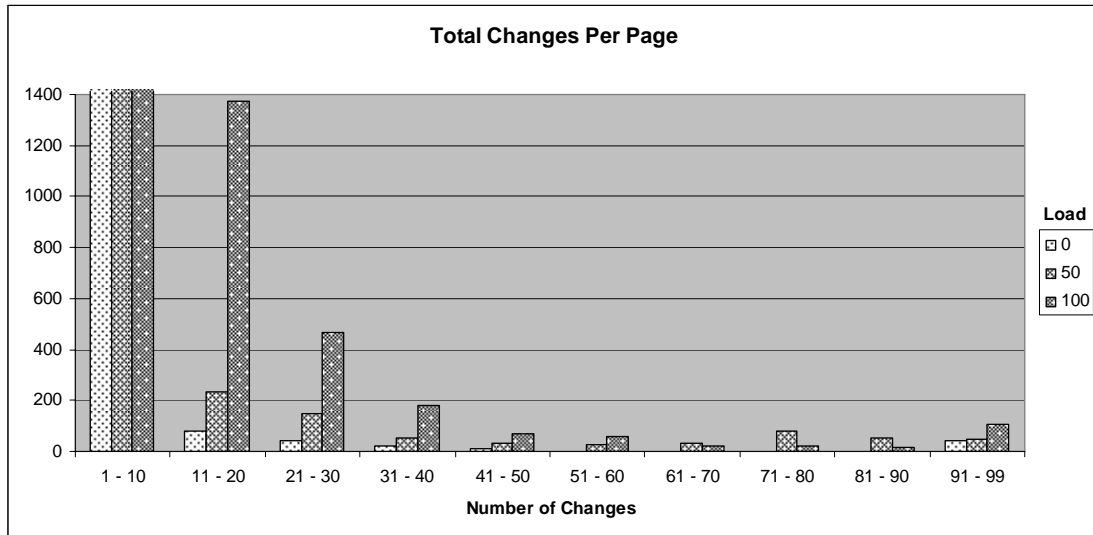


Figure 5-18: Total changes per page in the first portion of physical memory

Figure 5-18 shows that the majority of pages change between 1 and 10 times within 50 hours sampling at half hour intervals. However, from the statistical data produced by the experiments, it was derived that the median count of changes per page in the first portion of physical memory was 40.

Figure 5-19 shows the count of changes which occurred per page in the second portion of physical memory under all levels of test load.

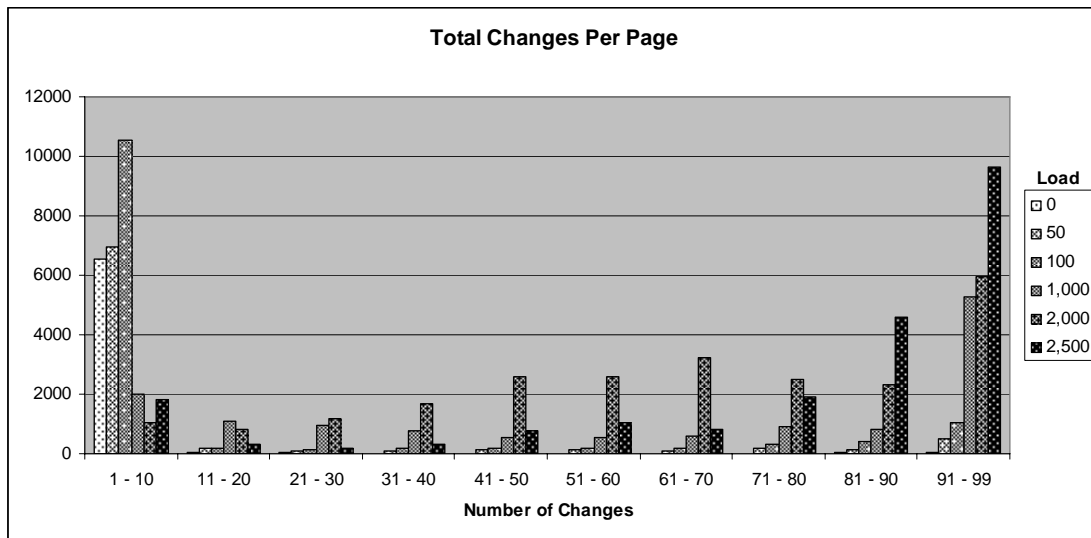


Figure 5-19: Total changes per page in the second portion of physical memory

It can be seen in Figure 5-19 that as the level of load increases, the bulk of pages shift to the higher counts of changes per page. However, there are still a number of pages which still change only few times within the 50 hour experiment. From a forensics view point, this indicates that even under a high level of system load, it is still possible to find relatively old data residing in physical memory.

Figure 5-20 shows the count of changes which occurred per page in the third portion of physical memory under all levels of test load.

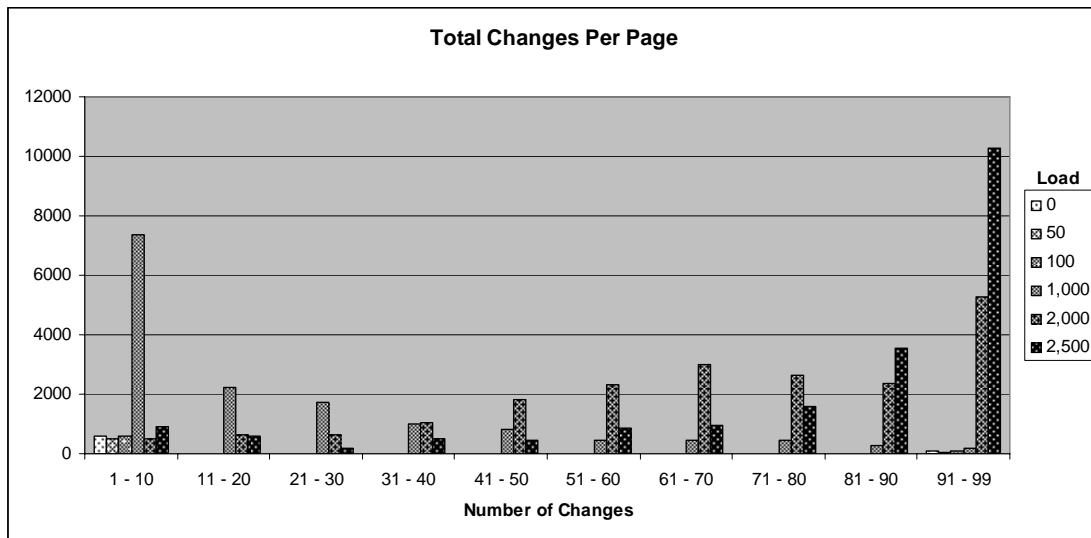


Figure 5-20: Total changes per page in the third portion of physical memory

When Figure 5-19 is compared with Figure 5-20, it can be seen that there is still a shift in the count of pages towards the higher counts of changes as the load increases, but for the third portion of physical memory, this shift occurs at a higher level of load. Additionally, it can be derived from the statistical data produced by the experiments that with a load of 1,000 the majority of pages in the second portion of memory change more than 70 times, whereas in the third portion of memory, the majority of pages change less than 40 times. This further indicates that the lower addresses of physical memory are used before the higher addresses are allocated.

Figure 5-21 shows the average results from all experiments with common levels of load across all three portions of memory.

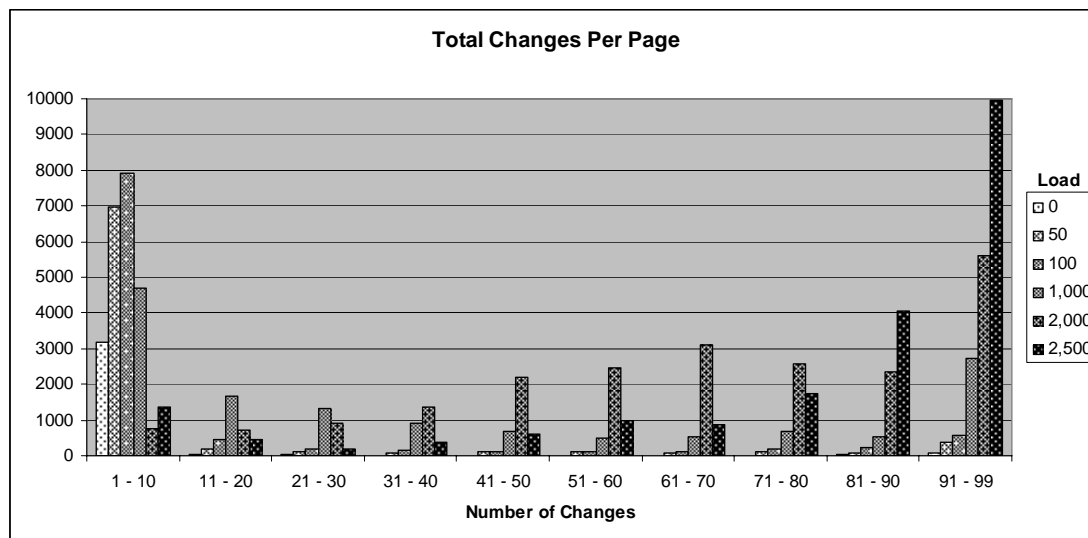


Figure 5-21: Total changes per page average across all three portions of physical memory

The graph shows that as the level of load increases, the count of pages that change frequently increases. However, this still leaves pages which change less than 10 times even under the highest level of load. From a forensics point of view, this shows that there is a very good chance that a reasonable amount of data will still remain in physical memory after a substantial period of time, especially because it is unlikely that a normal system be loaded to the equivalent of the test load level of 2,500.

A similar experiment to this was conducted by Dan Farmer and Wietse Venema (2004). The count of changes per page within physical memory was measured at an interval of 1 hour over 402 hours. The results produced confirm the results shown in this chapter as it was found that in Farmer and Venema's experiment there was a high count of changes early in the experiment, a very low count of changes mid-way through the experiment and finally a spike at the end of the experiment for the 'catch-all' value which follows the general shape produced by the experiments in this chapter. It was also shown that even on a moderately busy web server, there still remained over 2,000 pages of memory which did not change after a two and a half week period.

5.4 - Intervals to Initial Change

The initial change for a page for the purpose of these experiments is defined as the first time that the contents of a page change, not including the initial loading of the page (i.e. the page must not be empty). The count of intervals that it takes for each page to initially change gives an idea for each portion of memory, how long data is likely to persist in physical memory under different levels of load. In other words, for how long each page remains unchanged.

Figure 5-22 shows the count of intervals that it takes each page to initially change in the first portion of memory under loads 0, 50 and 100.

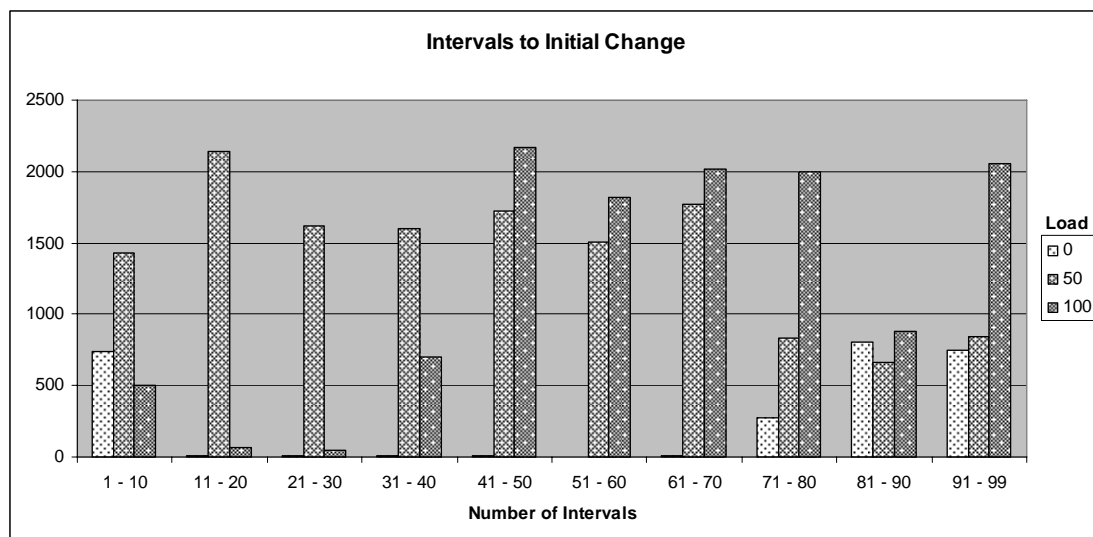


Figure 5-22: Intervals to initial changes in the first portion of physical memory

The graph shows that while there are pages which change quite early in the experiment as would be expected, there is a substantial number of pages which do not change at all until close to the end of the experiment.

Figure 5-23 and Figure 5-24 show the count of intervals that it takes each page to initially change in the second and third portions of memory respectively under all levels of test load.

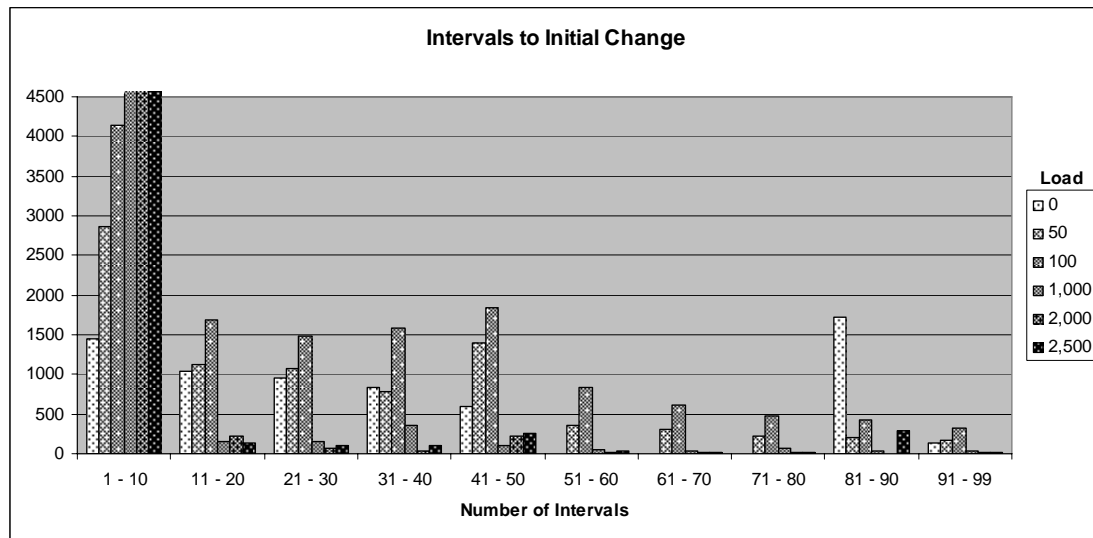


Figure 5-23: Intervals to initial changes in the second portion of physical memory

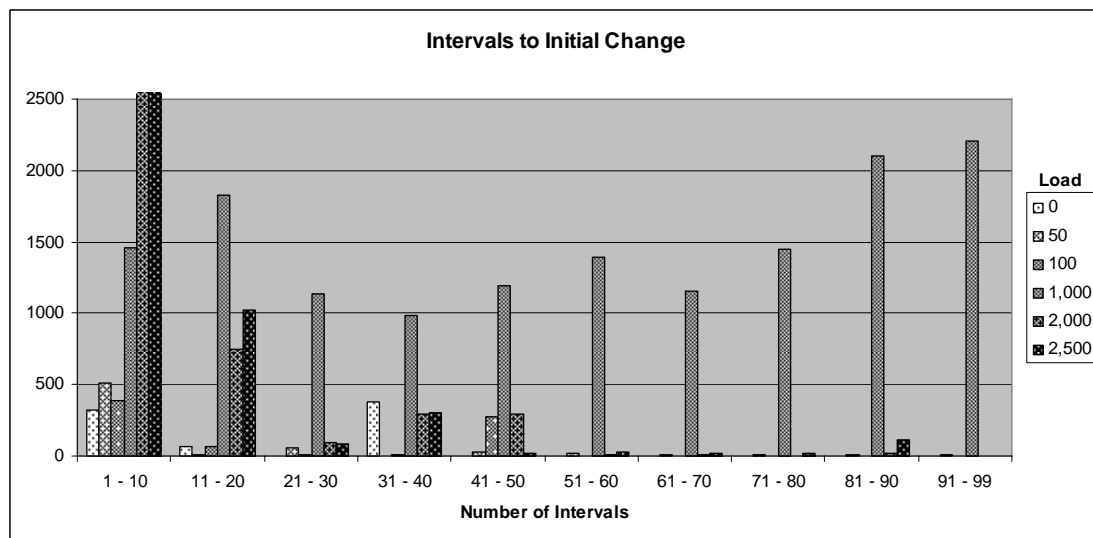


Figure 5-24: Intervals to initial changes in the third portion of physical memory

It can be seen in the two graphs that pages are spread relatively evenly by how many intervals it takes for the page to initially change under a mid-range load such as 1,000. For the higher loads, as would be expected, most of the pages change within the first few intervals. The graphs indicate that under mid-level loads, physical memory is filled completely, and gradually overwritten after memory is full as some pages do not

initially change until very late in the experiment remembering that the initial loading of pages is not counted as a change.

Figure 5-25 shows the average results from all experiments with common levels of load across all three portions of memory.

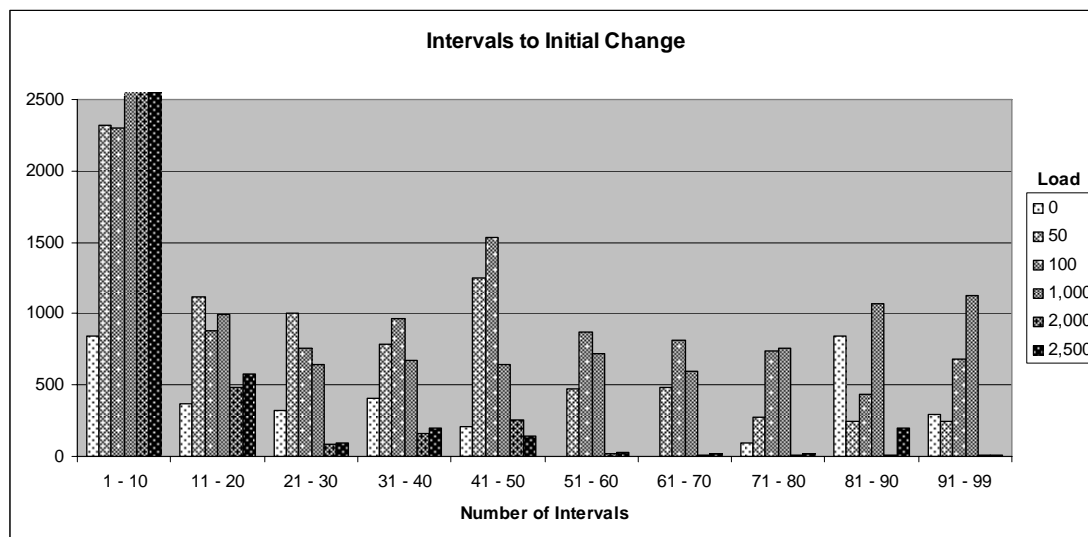


Figure 5-25: Intervals to initial changes average across all three portions of physical memory

The graph shows that while the pages generally initially change within the first 20 intervals for the higher loads, there is an even spread for loads 1,000 and below. This indicates that physical memory is gradually overwritten after it is completely filled across all three portions. From a forensics point of view, this indicates that even if the system to be investigated is under load, data can take up to two days or longer to be overwritten, so depending on how long it takes forensic investigators to begin investigation of the system, physical memory may still contain data which logically ceased to exist two days prior.

5.5 - Intervals Between Changes

The count of intervals between the changes of each page gives an idea of the average time that data will stay in memory for. It is not known whether data is changed because the program which owns the data made modifications to it, the memory management system swapped it out of physical memory to make room for new data or whether the owner process had terminated and the memory was marked as free, but the results are still able to provide a reasonable view of the behaviour of physical memory under different levels of load.

All graphs in the section are cropped, as the counts of changes which occur 1 to 10 intervals apart are so high, that the other values become unreadable.

Figure 5-26 shows the count of intervals between changes of each page in the first portion of physical memory under the load levels 0, 50 and 100.

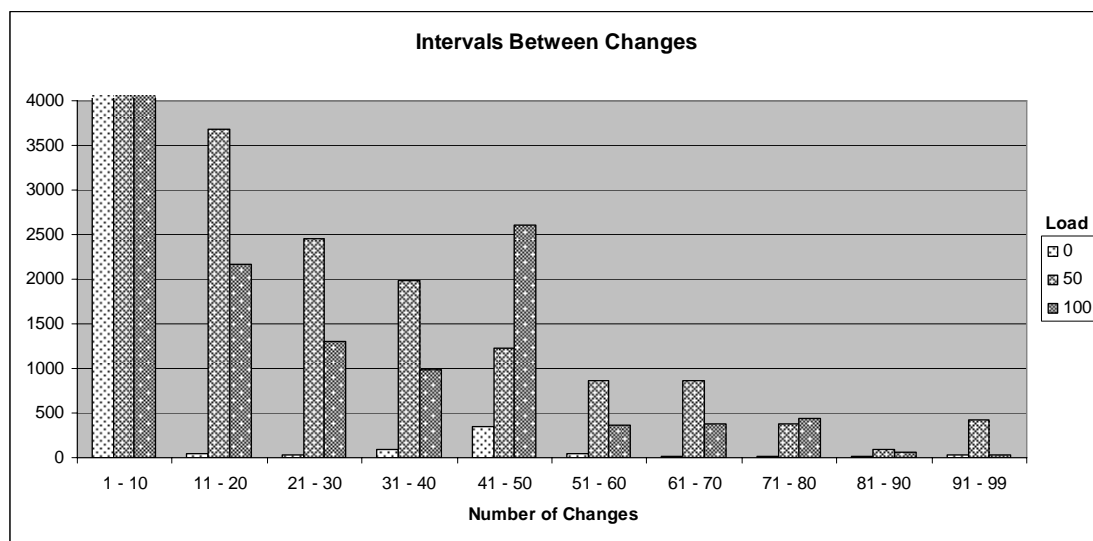


Figure 5-26: Intervals between changes in the first portion of physical memory

The diminishing trend of the graph is expected as the experiments are of a fixed length. If a page takes 50 intervals to change, there is less chance of it changing again than there is for a page which takes only 10 intervals to change because there is more time left for the latter page to change again than there is for the former page.

Figure 5-27 and Figure 5-28 show the count of intervals between changes of each page in the second and third portions of physical memory respectively under all levels of test load.

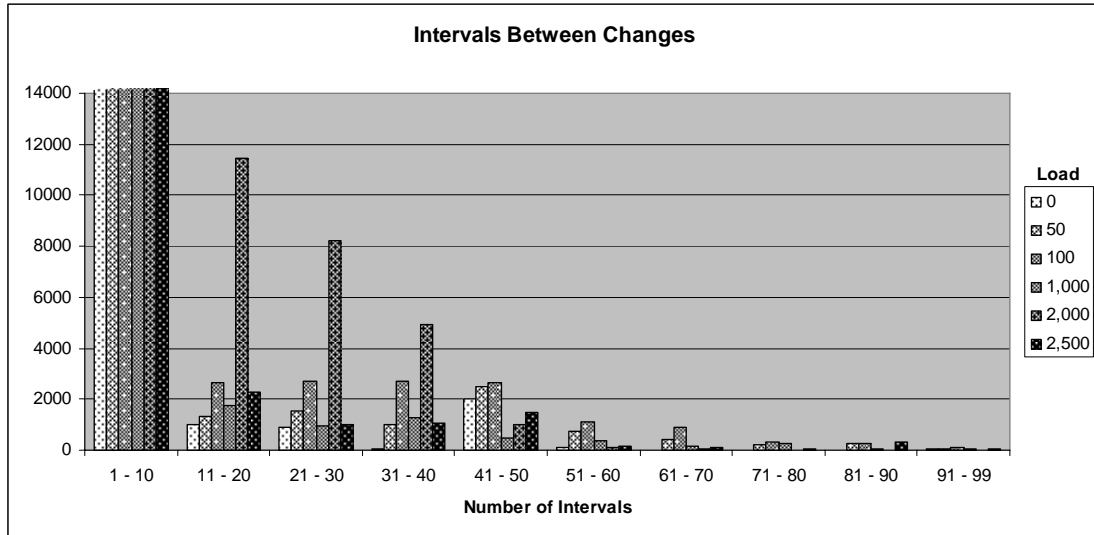


Figure 5-27: Intervals between changes in the second portion of physical memory

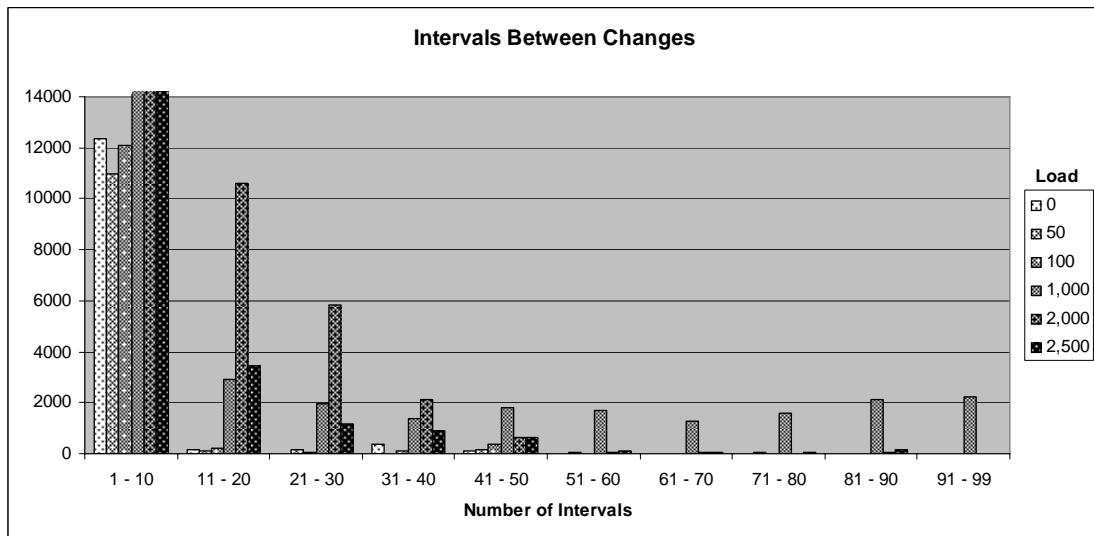


Figure 5-28: Intervals between changes in the third portion of physical memory

The comparison of Figure 5-27 and Figure 5-28, once again confirms the hypothesis that physical memory is allocated starting from the lower addresses moving toward the higher addresses as the majority of page changes in the third portion of memory

do not begin to occur within 10 intervals after the previous change for that page until the level of load reaches 1,000, whereas in the second portion of memory the majority of page changes occur within 10 intervals of the previous change for that page under all levels of load.

Figure 5-29 shows the average results from all experiments with common levels of load across all three portions of memory.

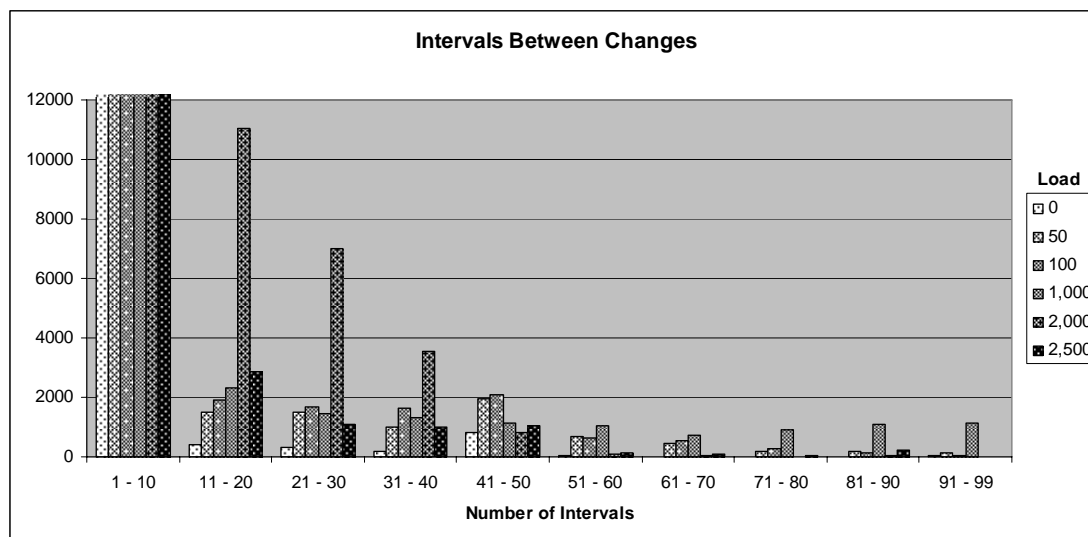


Figure 5-29: Intervals between changes average across all three portions of physical memory

The average results show that under all levels of load, the majority of page changes occur within 10 intervals of the previous change for the page. This is expected, because if a page changes after 60 intervals, assuming that the page changes again, it will have to change within 40 intervals as each experiment has a total duration of 100 intervals. So forensically, the high counts of page changes which occur within 10 intervals of the previous change for the page do not necessarily indicate a fast rate of decay of data within physical memory. There may be several quick changes, but the graphs in this section do not reveal which pages are changing. The changes may represent the same group of pages changing over and over again for the duration of the experiment. There still may be important data which is not being overwritten.

5.6 - Final Age of Data

The final age of data within pages is perhaps the most important metric collected from a forensics point of view. It shows how long the data in memory has been resident for at the completion of each experiment. Along with using the metrics collected from the experiments, the final memory dumps for each experiment were analysed with a hex editor to find the timestamps stored by the *artificialLoad* program. As the ending time of each experiment is known, the timestamps show exactly how long the data has been resident within physical memory where the comparison of hash values can only show the age to a degree of 30 minute intervals. However, these timestamps only relate to the pages used by the block device cache and the data segment of the *artificialLoad* program. The pages belonging to the block device cache are the most important pages from a forensics point of view as they are parts of files which investigators may attempt to interpret in a forensic investigation rather than the data segment or stack of a process what was running on the system.

This section is broken into two sections, the first of which shows the statistics collected from the experiments and the second shows the information found through analysis of the final memory dumps from the experiments. It is important to note that while the experiment statistics relate to all pages, the information found through the analysis of the final memory dumps only relates to the pages used by the block device cache and data segment of the *artificialLoad* program.

5.6.1 - Experiment Statistics

It should be noted that empty pages (i.e. pages filled with zeros) are excluded from the statistical analysis of the final age of pages.

Figure 5-30 shows the age of data at the end of each experiment in the first portion of physical memory under the load levels 0, 50 and 100.

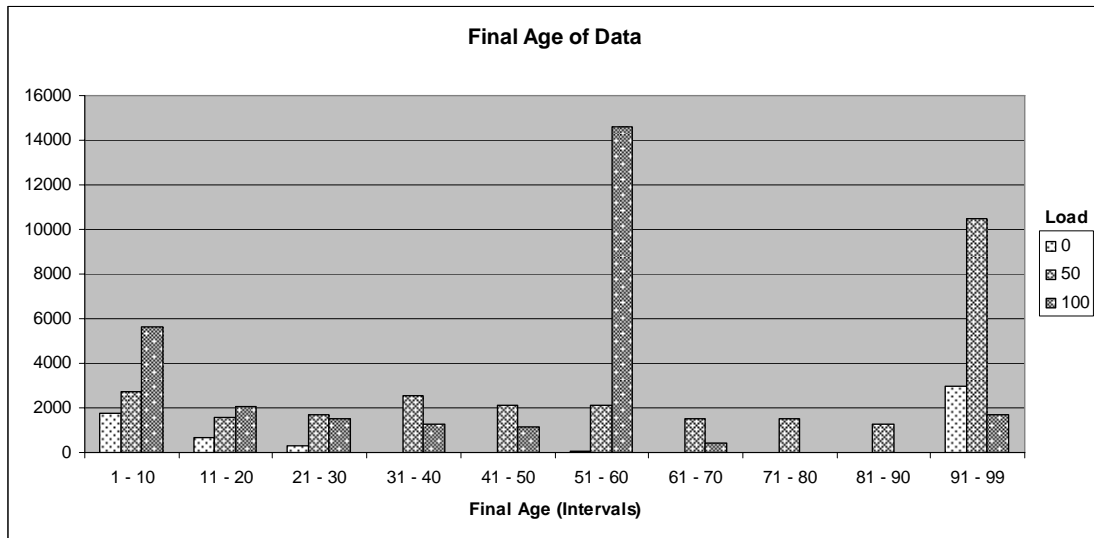


Figure 5-30: Final age of data in the first portion of physical memory

Experiments with a higher level of load were not conducted for the first portion of memory, but already it can be seen in the graph that after the experiment with test load of 100, the majority of pages were unchanged for 51 to 60 intervals. With a slightly lighter load of 50, the majority of pages were between 91 and 99 intervals old which is approximately two days.

Figure 5-31 shows the age of data at the end of each experiment in the second portion of physical memory under all levels of test load.

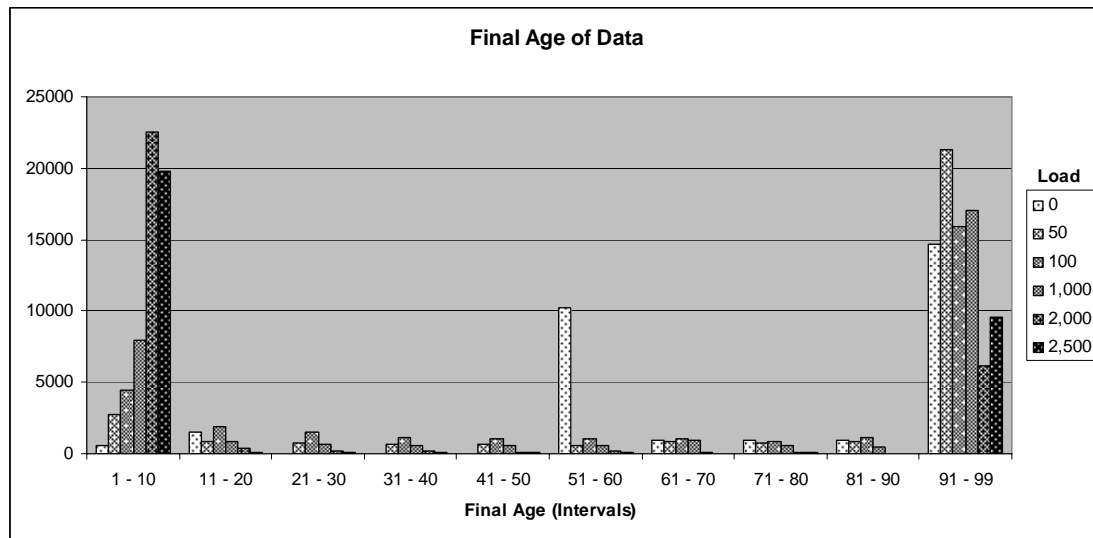


Figure 5-31: Final age of data in the second portion of physical memory

In Figure 5-31, a clear shift can be seen. Under light loads, the majority of pages are between 91 and 99 intervals old at the conclusion of each experiment while under heavy loads, the majority of pages are between 1 and 10 intervals old at the conclusion of each experiment. A spike also shows on the graph for an unloaded system. This is due to 9,794 pages changing after 59 intervals of the experiment. As the system was unloaded throughout the experiment, this can only be attributed to the aberrant behaviour of the operating system.

Figure 5-32 shows the age of data at the end of each experiment in the third portion of physical memory under all levels of test load.

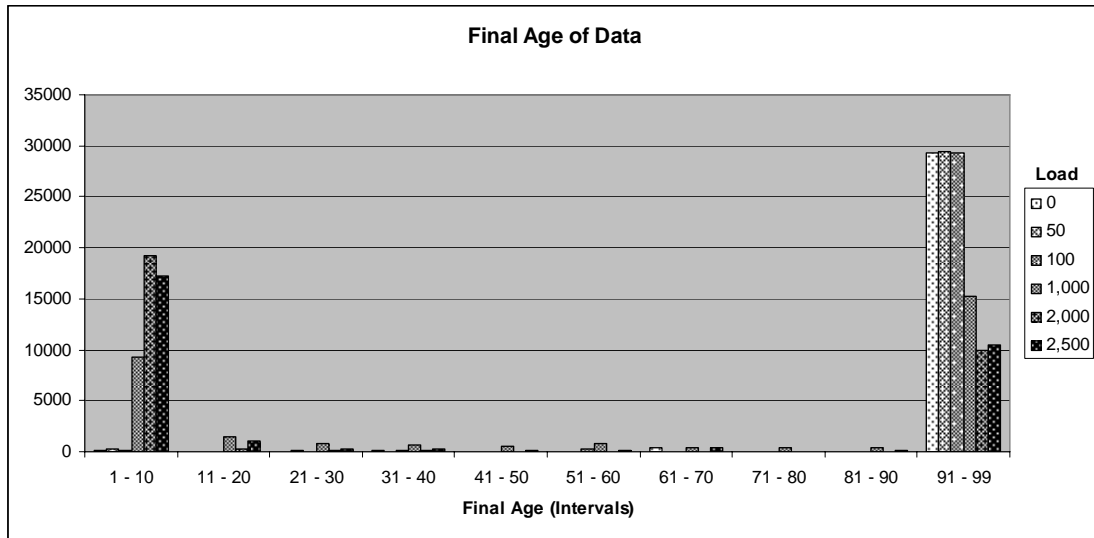


Figure 5-32: Final age of data in the third portion of physical memory

It can be seen in the graph that for experiments with lighter levels of load, almost all 30,000 pages are between 91 and 99 intervals in length at the conclusion of each experiment, whereas once the load reaches over 1,000, only approximately one third of pages are over 91 intervals old at the conclusion of each experiment.

Figure 5-33 shows the average age of data at the conclusion of all experiments with common levels of load across all three portions of memory.

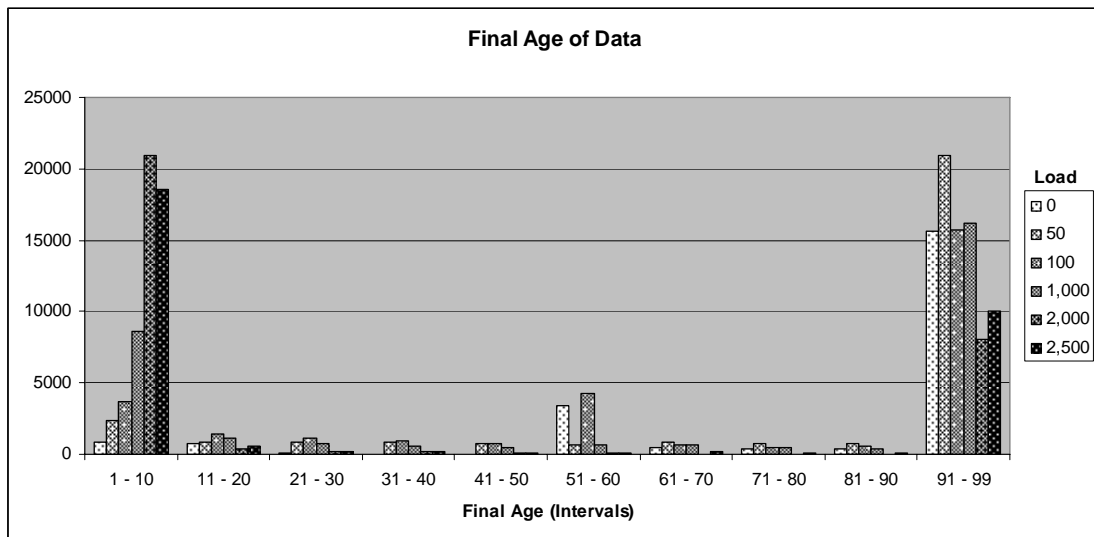


Figure 5-33: Final age of data average across all three portions of physical memory

The average results show that even under the highest levels of load, one third of physical memory is between 91 and 99 intervals or approximately two days old. From these results, the use of physical memory in forensic investigation seems to be a promising area for further study.

5.6.2 - Analysis of Memory Snapshots

So far it has been shown that statistically, pages do persist for long periods of time in physical memory after the owner process has terminated. In this section, the actual final memory dumps of the experiments will be analysed to show how much data from the block device cache and data segment can be found in memory and exactly how old the data is.

Each file written by the *artificialLoad* program contains a unique character string followed by a timestamp. In order to tell the difference between the types of pages, the *analyseDump* program looks first at the byte offset within the page of the unique string. If this offset is 0 (as the page size is 4KB, the beginning of a page has an overall offset ending in three hex 0s), then the string is at the very beginning of the page and the page is related to the block device cache. If the offset is not 0, then the page is either a part of the data segment or text segment of the *artificialLoad* process. This is determined by the presence of a timestamp following the unique string. If there is a timestamp present, the page belongs to the data segment of the *artificialLoad* process, if there is no timestamp (there will be a '%s' in the place of the timestamp), then the page belongs to the text segment of the *artificialLoad* process.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00025000	54	68	69	73	20	69	73	20	74	68	65	20	75	6E	69	71	This is the uniq
00025010	75	65	20	73	74	72	69	6E	67	20	77	69	74	68	20	74	ue string with t
00025020	69	6D	65	3A	20	46	72	69	20	4A	75	6E	20	32	33	20	ime: Fri Jun 23
00025030	31	34	3A	35	35	3A	35	37	20	32	30	30	36	0A	00	00	14:55:57 2006
00025040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00025050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00025060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00025070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00025080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00025090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000250A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000250B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000250C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000250D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000250E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000250F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

Figure 5-34: An example of a block device cache page

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00036D60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00036D70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00036D80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00036D90	00	00	00	00	1E	00	00	00	00	00	00	00	F4	7F	14	40	
00036DA0	00	00	00	00	C0	5C	01	40	08	0E	E3	BF	0C	89	04	08	À\ @ àì
00036DB0	1E	00	00	00	00	10	00	00	01	00	00	00	08	A0	04	08	
00036DC0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00036DD0	00	00	00	00	00	00	00	00	5C	74	9B	44	00	00	00	00	
00036DE0	00	00	00	00	00	00	00	00	00	54	68	69	73	20	69	73	
00036DF0	20	74	68	65	20	75	6E	69	71	75	65	20	73	74	72	69	This is
00036E00	6E	67	20	77	69	74	68	20	74	69	6D	65	3A	20	46	72	the unique stri
00036E10	69	20	4A	75	6E	20	32	33	20	31	34	3A	35	35	3A	35	ng with time: Fr
00036E20	36	20	32	30	30	36	0A	00	00	00	00	00	00	00	00	00	i Jun 23 14:55:5
00036E30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	6 2006
00036E40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00036E50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

Figure 5-35: An example of a data segment page

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0003AAA0	8B	5D	F4	8B	75	F8	8B	7D	FC	89	EC	5D	C3	90	90	90]ó uø }ü i]Ã
0003AAB0	55	89	E5	53	52	A1	58	9B	04	08	83	F8	FF	74	12	BB	U âSR X øyt »
0003AAC0	58	9B	04	08	FF	D0	8B	43	FC	83	EB	04	83	F8	FF	75	X ýD Cü è øyu
0003AAD0	F3	58	5B	5D	C3	90	90	90	00	00	00	00	00	00	00	00	óX[]Ã
0003AAE0	55	89	E5	53	83	EC	04	E8	79	FC	FF	FF	81	C3	4C	11	U âS i èyuüý ÃL
0003AAF0	00	00	E8	79	FC	FF	FF	59	5B	5D	C3	00	03	00	00	00	èyuüýY[]Ã
0003AB00	01	00	02	00	74	66	25	64	00	77	00	00	54	68	69	73	tfzd w This
0003AB10	20	69	73	20	74	68	65	20	75	6E	69	71	75	65	20	73	is the unique s
0003AB20	74	72	69	6E	67	20	77	69	74	68	20	74	69	6D	65	3A	tring with time:
0003AB30	20	25	73	00	72	00	53	50	41	57	4E	45	44	21	21	21	%s r SPAWNED!!!
0003AB40	20	74	66	25	64	0A	00	66	6F	72	6B	20	66	61	69	6C	tfzd fork fail
0003AB50	65	64	00	00	00	00	00	00	FF	FF	FF	FF	00	00	00	00	ed yyyý
0003AB60	FF	FF	FF	FF	00	00	00	00	00	00	00	00	01	00	00	00	yyyý
0003AB70	24	00	00	00	0C	00	00	00	A0	85	04	08	0D	00	00	00	\$
0003AB80	E0	8A	04	08	04	00	00	00	80	81	04	08	05	00	00	00	à
0003AB90	AC	83	04	08	06	00	00	00	2C	82	04	08	0A	00	00	00	- .

Figure 5-36: An example of a text segment page

Some of the block device cache pages (usually quite old) were found to be partially overwritten, these pages still may be useful in a forensic investigation, however they are classified as a separate type (unidentified pages related to file blocks).

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0
00388000	54	68	69	73	20	69	73	20	74	68	65	20	75	6E	69	71	This is the uniq
00388010	75	65	20	73	74	72	69	6E	67	20	77	69	74	68	20	74	ue string with t
00388020	69	6D	65	3A	20	46	72	69	20	4A	75	6E	20	32	33	20	ime: Fri Jun 23
00388030	31	30	3A	30	30	3A	34	37	20	32	30	30	36	0A	00	00	10:00:47 2006
00388040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00388050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00388060	00	30	66	D0	80	80	92	C3	88	00	00	00	88	80	8B	C7	0fD 'Ä Ç
00388070	06	00	00	00	07	00	00	00	01	00	02	00	03	00	04	00	
00388080	05	00	06	00	FF	FF	06	00	00	E8	00	00	01	E8	00	00	yy è è
00388090	02	E8	00	00	03	E8	00	00	04	E8	00	00	05	E8	00	00	è è è è
003880A0	06	E8	00	00	07	E8	00	00	08	E8	00	00	09	E8	00	00	è è è è
003880B0	0A	E8	00	00	0B	E8	00	00	0C	E8	00	00	00	00	00	00	è è è
003880C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
003880D0	00	00	00	00	01	00	00	00	00	00	00	00	00	00	00	00	
003880E0	00	00	00	00	00	00	00	00	E8	80	8B	C7	E8	80	8B	C7	è Çè Ç
003880F0	FF	FF	FF	FF	FF	FF	FF	FF	F8	80	8B	C7	F8	80	8B	C7	yyyyyyyyyè Çè Ç

Figure 5-37: An example of an unidentified page related to file blocks

The results produced by the analysis of the memory dumps only look at the second and third portions of memory under loads 100, 1000, 2000 and 2500. An unloaded system would be completely meaningless as the unique string would not be present in memory without running the *artificialLoad* program.

5.6.3 - Frequency Distribution of Types of Pages

Table 5-2 and Table 5-3 show the frequency distribution of the different types of recognised pages in the second and third portions of memory respectively under different levels of test load.

Load	Distribution				
	Block Device Cache Pages	Unidentified Pages Related to File Blocks	Data Segment Pages	Text Segment Pages	Total Hits
100	17	3	20	118	158
1,000	165	9	168	633	975
2,000	724	36	711	1049	2520
2,500	590	42	593	1312	2537

Table 5-2: Frequency distribution of recognised pages in the second portion of physical memory

Load	Distribution				
	Block Device Cache Pages	Unidentified Pages Related to File Blocks	Data Segment Pages	Text Segment Pages	Total Hits
100	0	0	1	0	1
1,000	344	0	341	340	1025
2,000	551	14	559	889	2013
2,500	599	29	565	1197	2390

Table 5-3: Frequency distribution of recognised pages in the third portion of physical memory

It can be seen in these tables that generally, the most common type of page located is the text segment page. The text segment is of little use for this study as it is the only type of page which does not contain a timestamp, so it is not possible to determine exactly how old the page is. The block device cache pages and the data segment pages are almost of equal frequency and the least common type of page is the unidentified page related to file blocks. Table 5-3 also shows that for lower levels of load, the third portion is used considerably less. This supports the hypothesis that memory is filled from the lower addresses up, and unless needed, the third portion will not be used.

5.6.4 - Analysis of Timestamps

Analysis of the timestamps contained in the recognised pages will show exactly how old the data is. This section will be broken into three subsections to represent the three types of identified pages.

5.6.4.1 - Block Device Cache Pages

Table 5-4 and Table 5-5 show the results produced by the analysis of the timestamps found in the block device cache pages of the second and third portions of memory respectively.

Load	Age	
	Oldest	Newest
100	2min 46s	2min 17s
1,000	2min 29s	2min 12s
2,000	1h 34min However, the next oldest page is: 2min 12s	2min 52s
2,500	4h 58min However, the next oldest page is: 2min 34s	1min 21s

Table 5-4: Age of block device cache data in the second portion of physical memory

Load	Age	
	Oldest	Newest
100	N/A	N/A
1,000	9h 50min However, the next oldest page is: 29s	18s
2,000	42s	1s
2,500	3min 43s	3min 5s

Table 5-5: Age of block device cache data in the third portion of physical memory

5.6.4.2 - Undefined File Block Pages

Table 5-6 and Table 5-7 show the results produced by the analysis of the timestamps found in the undefined pages related to file blocks of the second and third portions of memory respectively.

Load	Age	
	Oldest	Newest
100	27h 20min 51s	27h 20min 42s
1,000	30h 20min 19s	30h 20min 12s
2,000	26h 56min 47s	1h 57min
2,500	50h	4h 56min 38s

Table 5-6: Age of undefined file block pages in the second portion of physical memory

Load	Age	
	Oldest	Newest
100	N/A	N/A
1,000	N/A	N/A
2,000	4h 54min 38s	4h 53min 52s
2,500	50h	6h 28min 57s

Table 5-7: Age of undefined file block pages in the third portion of physical memory

5.6.4.3 - Data Segment Pages

Table 5-8 and Table 5-9 show the results produced by the analysis of the timestamps found in the data segment pages of the second and third portions of memory respectively.

Load	Age	
	Oldest	Newest
100	3min 10s	2min 19s
1,000	2min 29s	2min 19s
2,000	5h 27min 9s However, the next oldest page: 2min 12s	1min 29s
2,500	5min 31s	1min 24s

Table 5-8: Age of data segment pages in the second portion of physical memory

Load	Age	
	Oldest	Newest
100	1min 2s	1min 2s
1,000	28s	19s
2,000	1h 13min 26s However, the next oldest page: 31s	1s
2,500	3min 43s	3min 5s

Table 5-9: Age of data segment pages in the third portion of physical memory

5.6.4.4 - Comparison of Page Types

While in some cases, the age of pages is in hours, these pages were only of the ‘undefined pages related to file blocks’ type. The pages of the other types were seldom older than 2 or 3 minutes. As was shown in Section 5.6.3, the undefined pages related to file blocks are the most uncommon pages recognised, which means that most of the data related to the *artificialLoad* program resident in physical memory is quite new and is not likely to provide as much information about the past state of the

system. Previous studies in this area were based on statistical analysis, and the statistical analysis carried out earlier in this chapter confirms the results of these previous studies (Farmer and Venema, 2004). However, after a deeper look at the contents of the pages, the statistical analysis is not confirmed. It should be noted though, that these results only hold true for Suse Linux 10.0, as this is the operating system used to conduct the experiments. The results may differ significantly under different operating systems due to the complexity of the Memory Management System.

More studies should be conducted using other common operating systems. However, there is reason to believe that there is not as much information which persists within the physical memory of a computer system as was originally anticipated.

5.7 - Summary

The statistical analysis of page decay within physical memory regardless of page content shows that there is a significant number of pages which persist for a long period of time. These pages may be very useful in a forensic investigation. However, after a detailed analysis of the contents of pages which were used by the *artificialLoad* program, it is shown that the vast majority of these pages are less than 2 or 3 minutes old and are not likely to provide as much information about the history of usage of the computer system. There may still be other pages which persist for long periods of time and may well prove useful forensically; however, further studies would be needed to determine this.

Chapter 6 - Conclusion

The research goal of this thesis was to find out how long data can persist within physical memory even after the owner process has terminated by analysing the behaviour of the physical memory of a computer system. Using the existing tools for the investigation of physical memory, little more can be done aside from searching for ASCII strings. Using the results produced by the analysis of physical memory, a conclusion must be drawn as to whether further development of tools for the forensic investigation of physical memory is feasible.

Looking at the results produced by the statistical analysis of the rate of page decay within physical memory, presented in this thesis, and confirmed by other studies, it can be said that a substantial amount of data does persist for a long time. Through comparison of Figure 6-38 and Figure 6-39 it can be seen that the statistical analysis performed by Dan Farmer and Wietse Venema is confirmed by the results obtained through the experiments outlined in this thesis.

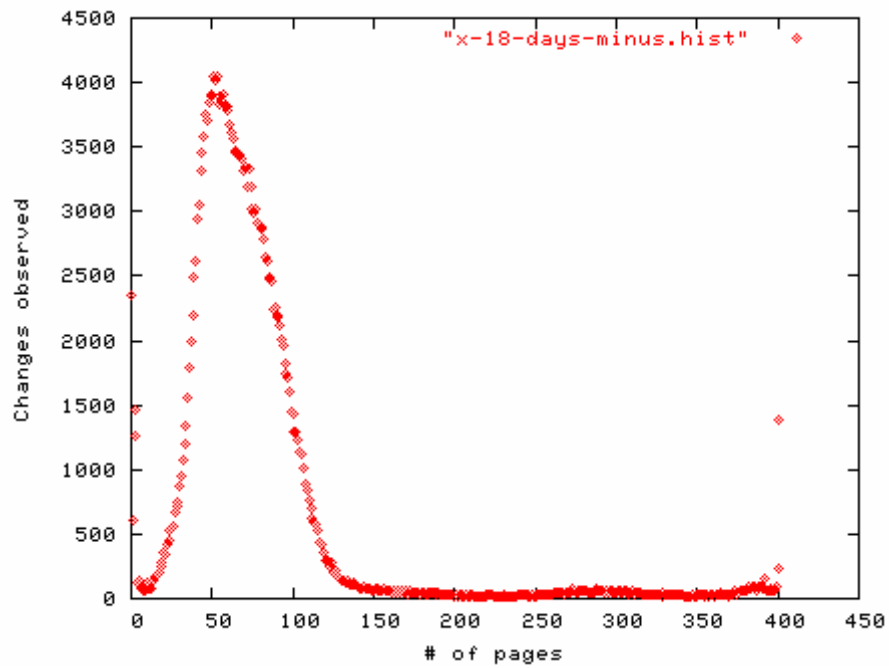


Figure 6-38: Counting memory page changes every hour over 402 hours (16.75 days) using MD5 hashes of memory pages (Red Hat Linux 6.1.) (Farmer and Venema, 2004)

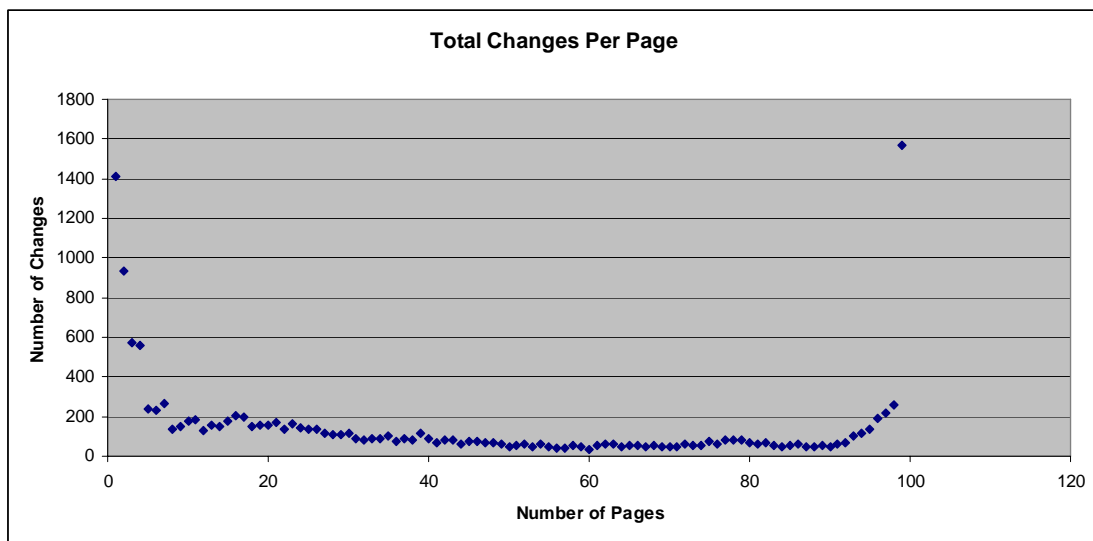


Figure 6-39: Total changes observed per page

The graphs are shown here simply to portray that both sets of results show that a significant amount of data does persist in physical memory statistically.

Both of these sets of results are based solely on statistics and the tracking of changes within physical memory. They are not concerned with the actual contents of these pages. In Section 5.6.2, the actual contents of memory were analysed, and pages were recovered from the block device cache and data segment of physical memory. These pages contained a timestamp showing the exact age of the page relative to the beginning of the experiment for which the memory snapshot was collected. The results derived from this analysis show that the majority of pages were only 2 or 3 minutes old. The only pages which were older were from the block device cache, however, the few of these older block device cache pages that were recovered, seemed to be partially overwritten and not likely to be of much use forensically.

It can therefore be concluded that although there was little forensically relevant data regarding files found within physical memory, it is statistically shown that a substantial amount of memory resident data does persist for quite some time. Further research is needed to study what this data is and its potential use forensically. The persistent data may be beyond the reaches of conventional crash dump analysis as it is possible that pages of physical memory that have been marked as free still contain data which is logically no longer a part of the system and hence can not be recovered any other way. Furthermore, the results presented in this thesis only hold true under Suse Linux 10.0, as this was the chosen Operating System for experimentation. More data relating to the block device cache may be found under different operating systems.

Future Work

This study can be performed under different operating systems to compare the results and determine whether more data can be recovered. It is suggested however, that experiments are run with a shorter duration and a smaller interval between memory dumps (for example, a 1 hour experiment with 5 minute intervals). This will give a much more fine grained view of the behaviour of physical memory.

The following areas are also suggested as extensions to this research:

- The swap space may be investigated in a similar fashion to the research presented in this thesis as this area of hard disk storage contains pages which were swapped out of physical memory at some point. Data is likely to persist for longer in swap space as it is generally only used when there is more demand for physical memory than the system can satisfy. As swap space is hard disk storage, it is non-volatile, which means that data will still persist even when the system is powered off.
- There is a new type of hard disk being developed called the “Hybrid Hard Disk”. These hard disks will have a non-volatile flash memory cache built into them (Samsung Electronics Co. Ltd., 2005). This cache will be between 128MB and 256MB. Research may be conducted on this cache as it is non-volatile and file data may persist in this cache after the system is powered down.

References

- BURDACH, M. (2004) *Forensic Analysis of a Live Linux System, Pt. 1*,
<http://www.securityfocus.com/infocus/1769>.
- CARRIER, B. D. & GRAND, J. (2004) A Hardware-Based Memory Acquisition Procedure for Digital Investigations. *Digital Investigation Journal*.
- CLARK, F. & DILIBERTO, K. (1996) *Investigating Computer Crime*, Boca Raton, USA, CRC Press.
- DAVIS, W. S. & RAJKUMAR, T. M. (2004) *Operating Systems - A Systematic View*, Boston, USA, Pearson Education, Inc.
- FARMER, D. & VENEMA, W. (2004) *Forensic Discovery*, Westford, USA, Addison Wesley Professional.
- FIGGINS, S., LOVE, R., ROBBINS, A., SIEVER, E. & WEBER, A. (2005) *Linux in a Nutshell*, North Sebastopol, USA, O'Reilly.
- FREE SOFTWARE FOUNDATION INC. (2006a) *Coreutils*,
<http://www.gnu.org/software/coreutils/>.
- FREE SOFTWARE FOUNDATION INC. (2006b) *Memory Concepts*,
http://www.gnu.org/software/libc/manual/html_node/Memory-Concepts.html.
- LANDMAN, J. (2002) *Forensic Computing: An Introduction to the Principles and the Practical applications*.
- LINUXMM (2006) *Virtual Memory*, <http://linux-mm.org/VirtualMemory>.

References

- MANDIA, K., PROSISE, C. & PEPE, M. (2003) *Incident Response & Computer Forensics*, New York, USA, The McGraw-Hill Companies, Inc.
- MOHAY, G., ANDERSON, A., COLLIE, B., VEL, O. D. & MCKEMMISH, R. (2003) *Computer and Intrusion Forensics*, Norwood, USA, Artech House.
- NELSON, B., PHILLIPS, A., ENFINGER, F. & STEUART, C. (2004) *Guide to Computer Forensics and Investigations*, Boston, USA, Course Technology.
- PROSISE, C. (2003) *Incident Response & Computer Forensics*, New York, USA, McGraw-Hill/Osborne.
- RIVEST, R. L. (1992) *The MD5 Message-Digest Algorithm*, <http://tools.ietf.org/html/rfc1321>.
- SAMSUNG ELECTRONICS CO. LTD. (2005) *SAMSUNG Teams with Microsoft to Develop First Hybrid HDD with NAND Flash Memory*, http://www.samsung.com/Products/HardDiskDrive/news/HardDiskDrive_20050425_0000117556.htm.
- STALLINGS, W. (2005) *Operating Systems: Internals and Design Principles*, Upper Saddle River, USA, Prentice-Hall.
- SYMANTEC CORPORATION (2004) *W32.SQLExp.Worm - Symantec.com*, http://www.symantec.com/security_response/writeup.jsp?docid=2003-012502-3306-99.
- TANENBAUM, A. S. (2001) *Modern Operating Systems*, Upper Saddle River, USA, Prentice-Hall.
- WYSOPAL, C. (2004) *Netcat*, <http://www.vulnwatch.org/netcat/>.

Appendix – Source Code

ArtificialLoad

```
// Jason Solomon
// 13729123
// B. Computer Science (Honours)
// artificialLoad
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
    int pid;
    int nValue = 1;
    char sTempFileName[10];
    FILE* fdTempFile;
    char sData[4097];
    int i;
    char sArg[9];
    int nStatus;
    time_t now;
    struct timeval *tv;

    if(argc == 2)
    {
        nValue = atoi(argv[1]);
    }

    while(1)
    {
        pid = fork();
        if(pid == 0)
        {
            // Child process
            // Prepare temporary filename
            sprintf(sTempFileName, "tf%d", nValue);

            // Create the temporary file
            fdTempFile = fopen(sTempFileName, "w");
```

Appendix – Source Code

```
// Write to the file
time(&now);
for(i = 0; i < 4097; i++)
    sData[i] = 0;
sprintf(sData, "This is the unique string with time: %s",
        ctime(&now));
fwrite(sData, 4096, 1, fdTempFile);

// Close and re-open temporary file for reading
fclose(fdTempFile);
fdTempFile = fopen(sTempFileName, "r");

// Read the file
fread(sData, 4096, 1, fdTempFile);

sleep(30);

// Close the temporary file
fclose(fdTempFile);

// Delete the temporary file
remove(sTempFileName);

exit(0);
}
else if(pid > 0)
{
    // Parent process
    fprintf(stderr, "SPAWNED!!! tf%d\n", nValue);
    nValue++;
    nValue = nValue % 100000000;
    wait(&nStatus);
}
else
{
    perror("fork failed");
    gettimeofday(tv, NULL);
    srand(tv->tv_usec);
    sleep((rand() % 60) + 1);
}
}
}
```

PAL

```

// Jason Solomon
// 13729123
// B. Computer Science (Honours)
// PAL
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
    int nProcesses;
    int nPid;
    int x;
    int nStartVal;
    char sArg[10];
    struct timeval *tv;

    if(argc != 2)
        return 1;

    // Get number of processes to start
    nProcesses = atoi(argv[1]);

    for(x = 0; x < nProcesses; x++)
    {
        nPid = fork();
        if(nPid == 0)
        {
            // Child process
            // Prepare start value
            nStartVal = x * 20;
            sprintf(sArg, "%d", nStartVal);
            execl("./artificialLoad", "artificialLoad", sArg, (char *)0);
            perror("execl failed");
            exit(1);
        }
        else if(nPid > 0)
        {
            // Parent process
            sleep(3);
        }
        else
        {
            perror("fork failed");
            x--;
            gettimeofday(tv, NULL);
            srand(tv->tv_usec);
            sleep((rand() % nProcesses) + 1);
        }
    }
}

```

```
    return 0;
}
```

MemDump

```
#!/bin/bash
# Jason Solomon
# 13729123
# B. Computer Science (Honours)
# memDump

# Must be run as root
if [ `id -u` != "0" ]; then
    echo "Must be run as root."
    exit 1
fi

COUNTER=0
# Set number of intervals
while [ $COUNTER -lt 100 ]; do

    # Dump physical memory
    # First 'count' pages
    # Send to remote computer
    dd bs=4096 count=30000 skip=30000 if=/dev/mem | netcat
    137.154.151.70 9123

    # Increment counter
    COUNTER=$(expr $COUNTER + 1)

    echo "Sent $COUNTER"

    # Set interval length (in seconds)
    sleep 1800

done

# Shutdown the system
/sbin/shutdown -h now

exit 0
```

Data Collection Scripts

The following is a collection of small scripts and programs which work together to collect and process experiment data.

RunHash.bat

```
@echo off
CScript hashMem.vbs
```

HashMem.vbs

```
'Jason Solomon
'13729123
'B. Computer Science (Honours)
'Data Collection
Dim page
Dim hash
Dim hashNo
Dim returnCode

page = 0
hashNo = 1
x = 0

' Number of pages to hash
hash = 30000

Set WshShell = WScript.CreateObject("WScript.Shell")

' Set number of intervals
do While x < 100
    ' Delete memDump
    returnCode = WshShell.Run("cleanUp.bat", 0, True)
    WScript.Echo("Waiting for next hash...")
    ' Listen for next memory dump
    returnCode = WshShell.Run("nc -l -p 9123 -O memDump", 0, True)
    WScript.Echo("Processing...")
    ' Hash each page
    do while not page >= hash
        returnCode = WshShell.Run("hashCall.bat " & page & " " & hashNo,
            0, True)
        if page mod 500 = 0 then
            ' Output the page number every 500 pages
            WScript.Echo(page)
        end if
        page = page + 1
    loop
    page = 0
    hashNo = hashNo + 1
end do
```

Appendix – Source Code

```
x = x + 1
loop

' Shutdown the computer
returnCode = WshShell.Run("shutdown -s -t 10", 0, True)
```

CleanUp.bat

```
@echo off
del /F /Q memDump
```

HashCall.bat

```
@echo off
dd bs=4096 count=1 if=memDump skip=%1 | md5 | saveHash hash%2 %1
```

SaveHash.c

```
// Jason Solomon
// 13729123
// B. Computer Science (Honours)
// Data Collection
#include <stdio.h>

int main(int argc, char *argv[])
{
    FILE* fdHashFile; // File for saving hashes
    unsigned char sData[4096]; // Buffer to collect input
    unsigned char sHash[33]; // Buffer to hold hash
    int i = 0;
    int j;

    // Check for correct parameters
    if (argc != 3)
    {
        fprintf(stderr, "Usage: saveHash.exe hashfilename pagenumber\n");
        return 1;
    }

    // Find end of hash
    while((sData[i] = getchar()) != '-')
    {
        i++;
    }

    // Point i to beginning of hash
    i = i - 34;
    // Extract hash from input
    for(j = 0; j < 32; j++)
    {
```

Appendix – Source Code

```
sHash[j] = sData[i];
i++;
}
sHash[32] = '\\t';

// Open hash file in append mode
fopen_s(&fdHashFile, argv[1], "a+");
//fdHashFile = fopen(argv[1], "a+"); // If you can't use fopen_s,
// use this line instead

fwrite(sHash, 33, 1, fdHashFile);

j = 0;
while(argv[2][j] != '\\0')
{
    sHash[j] = argv[2][j];
    j++;
}
sHash[j] = '\\n';
j++;

// Add page number to hash
fwrite(sHash, j, 1, fdHashFile);

// Close the hash file
fclose(fdHashFile);

return 0;
}
```

AnalyseHash

```

// Jason Solomon
// 13729123
// B. Computer Science (Honours)
// AnalyseHash
#include <errno.h>
#include <stdio.h>
#include <string.h>

int main()
{
    FILE* fdHashFile1;           // File descriptor for hash file
    char sHashFileName1[10];     // File name of hash file
    FILE* fdHashFile2;
    char sHashFileName2[10];

    int x, y;                   // Counters

    char sTempData1[41];        // Data for comparison
    char sTempData2[41];

    int nCurrPage;              // Page being processed
    int nHashes;                // Number of hashes processed
    short nChanges[131072];     // How many times each page changes
    int nChangedPages[200];     // How many pages changed x times
    short nPageIntervals[131072]; // Intervals of each page
    int nIntervals[200];        // How many pages changed in x
intervals
    short nEmptyPage[131072];   // Which pages are empty
    int nFinalAge[200];         // Final age of pages
    int nTotalChanges = 0;      // Total number of changes
    int nTotalEmpty = 0;       // Total number of empty pages
    int nInitialChanges[200];   // How many pages initially changed
        in x intervals

    // Initialise
    for(x = 0; x < 131072; x++)
    {
        nChanges[x] = 0;
        nPageIntervals[x] = 1;
        nEmptyPage[x] = 1;
    }
    for(x = 0; x < 200; x++)
    {
        nIntervals[x] = 0;
        nFinalAge[x] = 0;
        nInitialChanges[x] = 0;
    }

    // Begin processing
    fprintf(stderr, "Processing hash files...\n");
    for(x = 1; x < 200; x++)
    {

```

Appendix – Source Code

```
fprintf(stderr, "Processing hash file %d...\n", x);
// Prepare hash file names (hashxxxx)
sprintf(sHashFileName1, "hash%d", x);
sprintf(sHashFileName2, "hash%d", x + 1);

// Open hash files
fdHashFile1 = fopen(sHashFileName1, "r");
if(fdHashFile1 == 0)
{
    perror("Opening Hash File");
    return 1;
}
fdHashFile2 = fopen(sHashFileName2, "r");
if(fdHashFile2 == 0)
{
    if(errno == 2)
    {
        // File does not exist
        fclose(fdHashFile1);
        fprintf(stderr, "Processing complete...\n");
        nHashes = x + 1;
        x = 201;
    }
    else
    {
        perror("Opening Hash File");
        return 1;
    }
}

// If processing isn't complete
if(x != 201)
{
    nCurrPage = 0;
    // Process all pages
    fgets(sTempData1, 40, fdHashFile1);
    fgets(sTempData2, 40, fdHashFile2);
    while(feof(fdHashFile1) == 0 && feof(fdHashFile2) == 0)
    {
        if(strcmp(sTempData1, sTempData2) != 0)
        {
            // Page has changed
            // Check if page was empty
            if(strstr(sTempData1, "620F0B67A91F7F74151BC5BE745B7110")
                == NULL)
            {
                // Page wasn't empty
                nEmptyPage[nCurrPage] = 0;
                // Increment change counter for current page
                nChanges[nCurrPage]++;
                if(nChanges[nCurrPage] == 1)
                {
                    nInitialChanges[x]++;
                }
            }
        }
    }
}
```

Appendix – Source Code

```
nTotalChanges++;
// Record persistence and reset counter for current page
nIntervals[nPageIntervals[nCurrPage]]++;
nPageIntervals[nCurrPage] = 1;
}
else {
// Page was empty
if(nEmptyPage[nCurrPage] == 1)
{
// Persistence not counted
nPageIntervals[nCurrPage] = 1;
}
else
{
nEmptyPage[nCurrPage] = 0;
// Increment change counter for current page
nChanges[nCurrPage]++;
nTotalChanges++;
// Record persistence and reset counter for current
page
nIntervals[nPageIntervals[nCurrPage]]++;
nPageIntervals[nCurrPage] = 1;
}
}
}
else
{
// Page was unchanged
// Page persisted for one further interval
nPageIntervals[nCurrPage]++;
if(strstr(sTempData1, "620F0B67A91F7F74151BC5BE745B7110")
== NULL)
{
// Page wasn't empty
nEmptyPage[nCurrPage] = 0;
}
}
}

// Read next page
fgets(sTempData1, 40, fdHashFile1);
fgets(sTempData2, 40, fdHashFile2);
nCurrPage++;
}

// Close hash files
fclose(fdHashFile1);
fclose(fdHashFile2);
}
} // Next hash

// Analysis complete
// nCurrPage now represents how many pages were processed
for(x = 0; x < nCurrPage; x++)
{
```

Appendix – Source Code

```
// If page wasn't empty
if(nEmptyPage[x] == 0)
{
    nChangedPages[nChanges[x]]++;
    nFinalAge[nPageIntervals[x]]++;
}
else
{
    nTotalEmpty++;
}
}

// Output
printf("\n--- Statistics ---\n\n");
printf("%d pages were empty\n", nTotalEmpty);
for(y = 0; y < nHashes; y++)
{
    printf("%d pages changed %d times\n", nChangedPages[y], y);
}
printf("\n-----\n");
printf("\nThe total number of changes:\t%d\n", nTotalChanges);
printf("\n-----\n\n");
for(y = 1; y < nHashes; y++)
{
    printf("%d pages took %d intervals to initially change\n",
        nInitialChanges[y], y);
}
printf("\n-----\n\n");
for(y = 1; y < nHashes; y++)
{
    printf("%d pages took %d intervals to change\n", nIntervals[y],
        y);
}
printf("\n-----\n");
printf("\nFinally\n\n");
for(y = 1; y < nHashes; y++)
{
    printf("%d pages were %d intervals old\n", nFinalAge[y], y);
}
printf("\n-----\n");

return 0;
}
```

AnalyseDump

```

// Jason Solomon
// 13729123
// B. Computer Science (Honours)
// AnalyseDump
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>

// Function to recognise the unique string
char* findStr(char*, char*);

int main(int argc, char* argv[])
{
    int nOut;
    FILE* fdMemDump; // File
        descriptor for hash file
    struct stat *statFile = malloc(sizeof(struct stat)); // Memory
        dump statistics
    time_t tModTime; // Memory
        dump modification time
    char sPage[4096]; // Page
    char *sSearch = "This is the unique string with time:"; // Search
        string
    char sBuf[25]; // Buffer
    char *sPosition; // Address
        of search string
    int nIndex; // Location
        of search string
    int fPageEmpty; // Flag
    int x, y; // Counters
    int nBlockDevCache = 0;
    int nUnid = 0;
    int nDataSeg = 0;
    int nTextSeg = 0;
    int nTotal = 0;

    // Check parameters
    if(argc != 2 || atoi(argv[1]) < 1 || atoi(argv[1]) > 4)
    {
        fprintf(stderr, "Usage: AD x\n\n");
        fprintf(stderr, "Values for x:\n");
        fprintf(stderr, "1:\tBlock Device Cache pages\n");
        fprintf(stderr, "2:\tUnidentified pages related to file
            blocks\n");
        fprintf(stderr, "3:\tData Segment pages\n");
        fprintf(stderr, "4:\tStatistics only\n");
        exit(1);
    }
}

```

Appendix – Source Code

```
// Set desired output from command line parameter
nOut = atoi(argv[1]);

// Read modification time of memory dump
stat("memDump", statFile);
tModTime = statFile->st_mtime;

printf("Memory dump taken:\t%s\n\n", ctime(&tModTime));

switch(nOut)
{
  case 1:
  {
    printf("Block Device Cache pages:\n");
    break;
  }
  case 2:
  {
    printf("Unidentified pages related to file blocks:\n");
    break;
  }
  case 3:
  {
    printf("Data Segment pages:\n");
    break;
  }
  case 4:
  {
    printf("Statistics:\n");
  }
}

// Open memory dump
fdMemDump = fopen("memDump", "rb");
if(fdMemDump == 0)
{
  perror("Opening Memory Dump");
  return 1;
}

// Traverse memory dump page by page
for(x = 0; x < 30000; x++)
{
  // Seek to page and read
  fseek(fdMemDump, 4096 * x, SEEK_SET);
  fread(sPage, 4096, 1, fdMemDump);

  // Locate unique string
  sPosition = findStr(sPage, sSearch);
  if(sPosition != NULL)
  {
    // String found!
    // Increment total count of pages containing the string
    nTotal++;
  }
}
```

Appendix – Source Code

```
// Convert address to index
nIndex = sPage - sPosition;
if(nIndex < 0) {
    nIndex = -nIndex;
}

if(nIndex == 0)
{
    // Possibly a block device cache page
    // Check the rest of the page
    fPageEmpty = 1;
    y = 62;
    while(y < 4096 && fPageEmpty == 1)
    {
        if(sPage[y] != 0x00)
        {
            // Page not empty
            fPageEmpty = 0;
        }
        y++;
    }

    if(fPageEmpty == 1)
    {
        // Block Device Cache page
        nBlockDevCache++;
        if(nOut == 1)
        {
            memcpy(sBuf, sPage + 37, 24);
            sBuf[24] = '\\0';
            printf("\\t\\t\\t%s\\n", sBuf);
        }
    }
    else
    {
        // Unidentified page related to file block
        nUnid++;
        if(nOut == 2)
        {
            memcpy(sBuf, sPage + 37, 24);
            sBuf[24] = '\\0';
            printf("\\t\\t\\t%s\\n", sBuf);
        }
    }
}
else
{
    if(((nIndex + 7) % 16) == 0)
    {
        // Data Segment page
        nDataSeg++;
        if(nOut == 3)
        {
            memcpy(sBuf, sPage + nIndex + 37, 24);
        }
    }
}
```

Appendix – Source Code

```
        sBuf[24] = '\\0';
        printf("\\t\\t\\t%s\\n", sBuf);
    }
}
else
{
    // Text Segment page
    nTextSeg++;
}
}
}

// Close memory dump
fclose(fdMemDump);

// Output statistics
printf("\\nBlock Device Cache pages:\\t\\t\\t%d\\n", nBlockDevCache);
printf("Unidentified pages related to file blocks:\\t%d\\n", nUnid);
printf("Data Segment pages:\\t\\t\\t\\t%d\\n", nDataSeg);
printf("Text Segment pages:\\t\\t\\t\\t%d\\n", nTextSeg);
printf("\\nTotal hits:\\t\\t\\t\\t\\t%d", nTotal);

return 0;
}

/* Locate the unique string within a page */
char* findStr(char* sPage, char* sSearch)
{
    int i = 0;
    int fFound = 0;

    // Traverse page
    while(i < 4096 && fFound == 0)
    {
        if(sPage[i] == sSearch[0])
        {
            // Found the first character of unique string
            // Check the rest of the string
            if(memcmp(sPage + i, sSearch, 36) == 0)
            {
                fFound = 1;
            }
        }
        if(fFound == 0)
        {
            // The string doesn't match, keep searching to end of page
            i++;
        }
    }

    if(fFound == 0)
    {
        // String not found, return null
    }
}
```

Appendix – Source Code

```
    return NULL;
}
else
{
    // String found, return address of string
    return (sPage - i);
}
}
```