Gaining insight into virtualised host disk activity

Christopher R A Hicks
932135
Supervisor: Dr Behzad Bordbar

Submitted in conformity with the requirements
for the degree of MEng Electronic & Software Engineering
with Industrial Placement Year
School of Computer Science
University of Birmingham

Acknowledgments

I would like to offer my gratitude to my project supervisor Dr Behzad Bordbar who was always available to discuss ideas. I would also like to thank Keith Harrison, former Master Technologist at HP Labs, for his valuable input.

I am thankful to Evan Broder from MIT CSAIL for his advice regarding QEMU.

Abstract

Gaining insight into virtualised host disk activity

Christopher R A Hicks

Computer malware is both more prolific and more sophisticated than ever before. There is considerable economic incentive for both attackers and defenders to leverage the latest techniques in order to outdo one another. In virtualised cloud environments a technique for placing security mechanisms in a different domain to the host under protection has emerged. Virtual Machine Introspection is an architecture in which the hardware-level memory state of a virtual machines is inspected from outside the host to look for signs of infection. This project further develops this idea and an architecture for introspecting NTFS disk activity is specified. A proof of concept implementation is developed which is able to recreate NTFS disk activity from outside the host under protection.

Contents

Chapter 1

Introduction

To Secure Cyberspace is one of the Grand Challenges for Engineering defined by the National Academy of Engineering as one of the most pressing and significant challenges of our time. The scope of the problem faced is monumental. Every day there are over 390,000 new malware variants found, a number which has increased almost fourfold since 2012 [AV-TEST, 2015]. Ponemon Institute [2013] identified as many as 138,000 potentially malicious activities on any single day within just one enterprise network, these events correlated to an average of 37 new device infections each day in Q3 of 2014. Trustwave [2014] observed that for the 691 data breaches that they investigated, the median time taken to contain an intrusion by malware was 114 days - nearly four months. A wider study which considered 234 multinational companies found that the mean annual cost of cyber crime was $7.2 million per business, a 30% increase on the previous year [Ponemon Institute, 2013]. Malware, viruses, worms and trojans which successfully infiltrated an organisations network were experienced by 98% or more of the companies which participated.

The UK National Audit Office estimate that cybercrime costs the UK between 18 and 27 billion each year. Some notable victims of recent malware intrusions have included K-Mart (a US supermarket) who BBC News [2014] reported had all of their 1,200+ stores point of sale devices infected for over a month with an undetectable malware variant, the cost of which has not been disclosed and has likely not yet been realised. The 'Cryptolocker' ransomware which encrypts users files before demanding payment in anonymous digital currency infected over 155,000 machines in a single month at its peak, and netted the authors more than 3 million over a 9 month period [Bijl, 2014].

In broad terms malware can be detected by observing a change in system state which is different from the normative patterns of system behaviour. The earliest methods of detecting an intrusion are Host-based Intrusion Detection Systems (HIDS) which were pioneered by Denning [1986]. HIDS have an excellent view of the host state because they share the same resources and operate using the same semantics as the system which they protect; this is also their shortfall as they are able to be subverted and deceived by malware on the host. In corporate networks another common defence is to implement Network-based Intrusion Detection Systems (NIDS). NIDS offer greater resilience and continue to function in spite of host compromise, however, they only have a superficial view of host state and perform much more poorly at detecting malware.

A third variety of Intrusion Detection System (IDS) first proposed by Garfinkel et al. [2003] is based on a technique called Virtual Machine Introspection (VMI). VMI leverages Virtual Machine Monitor (VMM) technology to inspect the hardware-level state of a host's memory. This technique boasts both isolation from the host under protection and good visibility into its internal state. VMI has yet to be widely adopted as its current implementations rely upon the homogenisation of target platforms to make their development effort worthwhile.

The aim of this project is to develop a method which leverages VMM technology to provide intrusion detection capabilities by monitoring the disk access of virtualised hosts. A proof of concept application 'NineDot' (named after the nine-dots-puzzle which popularised the term 'thinking outside the box') is developed. NineDot consists of a POSIX C Linux application which by interfacing with a modified VMM I/O proxy offers the unique ability to extract files from a New Technology File System (NTFS) volume as they are written to disk by a virtualised host.

NTFS is used by default on every consumer-oriented release of the Microsoft Windows operating system since Windows XP (2001) and is likely utilised by over 91% of all desktop computers [NetMarketShare, 2015]. Despite its ubiquity, the inner workings of NTFS are not precisely known as it remains the proprietary property of Microsoft. The reverse-engineering efforts of the forensic analysis community combined with previous work on leveraging VMM technology has enabled the creation of NineDot.

This report begins with an introduction to malware and the methods which have been developed to counter the threat they pose. The goals and specification of the project are outlined before the Xen virtualisation environment and NTFS are considered in more detail. The design and implementation of NineDot are given appropriate attention, as is the testing and evaluation of the system. Finally this report covers the management, software engineering considerations and conclusion of this project.

Chapter 2

Background Research

2.1 Malware

Malicious Software (malware) is a broad term used to classify any computer program which intentionally acts against the requirements of the user. Malware encompasses computer viruses, worms, adware, spyware, ransomware, backdoor applications and any other code which can be considered hostile or intrusive to the user. Malware is big business; large organised criminal gangs seeking financial gain and even state funded organisations seeking political gain [Rosenbach et al., 2015] are behind some of the most prevalent and damaging examples.

Malware has a considerable economic impact on both organisations and individuals. A study by Gantz et al. [2014] estimates that in 2014 enterprises globally spent $491 billion dealing with the security issues and data breaches caused by malware linked to pirated software. Another study by Ponemon Institute [2013] put the figure at $7.2 million per business, per year, a 30% year-on-year increase. The UK Cabinet Office estimates the most likely total cost of cyber crime to the UK economy, including that which results from malware infection, is 27 billion per year [Detica, 2013].

Malware is increasingly prevalent. AV-TEST [2015] an independent IT-Security institute now register and classify more than 390,000 new malicious programs every day, four times as many as during 2012. The Cisco [2014] annual security report claims that 100 percent of the corporate networks which they analysed showed evidence of compromise or misuse.

Malware has evolved from relatively easy to detect user-space programs, to complicated component based rootkits which can subvert and deceive the security systems of a host. Analysis of the Agobot backdoor malware found it is capable of detecting and subverting 105 different anti-virus processes [Jiang et al., 2010]. Rootkit components are traded online and make it possible for even inexperienced programmers to create their own malware variants.

2.2 Resisting attack

As malware has evolved, so have the defences put in place to protect against it. Traditionally these defences have been either host or network-based systems which monitor the state of a system for activity which deviates from the normative pattern of expected behaviour.

The earliest methods of detecting an intrusion are Host-based Intrusion Detection Systems (HIDS) which were pioneered by Denning [1986]. HIDS reside on the system which they protect which gives them an excellent view of the internal state of a system. They use this view to monitor operating system and application audit data for signs of malicious activity. HIDS now include anti-virus and anti-malware software which commonly monitor file system events for signs of infection. HIDS reply upon the integrity of low-level kernel events such as system calls and file system modifications, for this reason they are vulnerable to deception by malware which is able to undermine the kernel [Vigna and Kruegel, 2006].

Network-based Intrusion Detection Systems (NIDS) are another mechanism of detecting an intrusion. NIDS were first introduced by Heberlein et al. [1990] who developed a method of profiling network usage to create historical profiles of behaviour which can be compared probabilistically to current usage. Unlike host-based systems, NIDS are isolated from the host under protection and continue to operate in-spite of host compromise. However, they also have a considerably more limited view of the internal state of the systems which they protect. NIDS which analyse network traffic have to perform a considerable amount of parsing, reassembly and interpretation in order to identify application level behaviours inside a host. A host which encrypts is network traffic poses an even greater challenge [Vigna and Kruegel, 2006].

The resurgence of interest in virtualisation technology driven by Moore's law [Moore, 1965] and the desire to move infrastructure away from single points of failure and into the cloud has created the opportunity for a third type of intrusion detection system. The next section of this report gives a preliminary discussion of virtualisation technology which puts into context the new intrusion detection architecture which follows.

2.3 Virtualisation

Virtualisation is a term which originally described a method of logically subdividing mainframe computer system resources between simultaneously running applications [Graziano, 2011]. The term now refers to any technology which creates a virtual system component and presents it with the same logical interface as its physical hardware counterpart.

Virtualisation techniques are not limited to single system components, but can present entire virtualised computer systems (virtual machines) as though they exist in real hardware [IBM Global Education, 2007]. Modern computers are sufficiently powerful that they can use hardware virtualisation to present the illusion of many virtual machines. Each virtual machine can run its own operating system, and is unaware of its coexistence as part of a larger system [Barham et al., 2003].

Common virtualisation terminology is that the system which provides virtualisation is called the host system. Virtual machine instances running on a host are called guests. The software which creates a virtual machine on a host is called a virtual machine monitor. The term hypervisor is equivalent in meaning to virtual machine monitor.

Hardware virtualisation technologies have introduced both challenges and solutions in the field of computer security. A challenging security problem with hardware virtualisation is ensuring proper isolation between individual guest virtual machines and ensuring isolation between guest virtual machines and their host [Ormandy, 2007]. Despite these challenges hardware virtualisation has also provided new mechanisms for enhancing computer security. Garfinkel et al. [2003] first introduced a technique which leverages virtual machine monitor technology to provide a new mechanism for intrusion detection.

2.4 VMI

The Livewire system developed by Garfinkel et al. [2003] presents a new architecture for intrusion detection which utilises a virtual machine monitors ability to access hardware-level state information about its guests. This approach of inspecting a virtual machine from the host, for the purpose of analysing its internal state, is termed virtual machine introspection (VMI).

Recent research which leverages VMI techniques has presented more efficient architectures which look for symptoms rather than causes of infection [Bordbar et al., 2012, Shaw et al., 2014], and maintain better isolation between host and guest, by performing VMI from privileged guests rather than the virtual machine monitor.

VMI architectures retain the advantages of both host and network-based intrusion detection systems. VMI has both the high visibility into internal state associated with host-based systems and the high degree of isolation offered by networked-based solutions. High visibility into the internal state of a guest virtual machine is desirable for an intrusion detection system because it reduces the likelihood that an intruder can operate unnoticed by the system. A high degree of isolation reduces the likelihood of an intruder being able to disable or otherwise tamper with the intrusion detection system.

VMI is challenging to implement because of the difficulty in interpreting the hardware-level state of memory in the context of a modern operating system. Dolan-Gavitt et al. [2011] defined this obstacle as the semantic gap and developed techniques for automatically generating programs which perform VMI for security.

2.5 Disk VMI

Leveraging virtualisation to introspect upon the disk activity of guest virtual machines has only been sparsely considered in the existing research.

Computer memory can be considered from a hierarchical perspective in which the fastest, most expensive memory (i.e. CPU registers) is at the top, and the slowest but least expensive memory (i.e. hard disk drives) is at the bottom. Owing to the vastly higher cost of memory at the top of the hierarchy, memories at the bottom of the hierarchy are present in much greater capacities [Silberschatz et al., 2010].

For the foreseeable future main memory will be too small to contain all of the programs and data which are required on a computer system. The size of memory has increased by orders of magnitude, but so has the size of the programs and data which are stored on computers [Tanenbaum, 2005]. The volatility of CPU registers and main memory are another reason that they are not suitable storing certain types of data.

Given that computer systems rely so heavily on hard disk drives to contain their programs and data, disk access is a worthwhile indicator of system state from which to detect system intrusion. This is the principle with which conventional host-based anti-virus and anti-malware software operates.

Just as introspecting the contents of memory requires an intimate knowledge of the inner workings of the target operating system, introspecting disk activity requires an intimate knowledge of the inner workings of the target file system. Fortunately, file systems are vastly more structured than the in-memory state of an operating system.

Previous work on introspecting disk activity includes the work of Broder [2010] and Jiang et al. [2010]. Both authors extracted high-level semantic information about the files and directories on a volume by casting guest OS file system structures onto the raw volume data accessible from a virtual machine monitor. Neither author succeeded in applying their techniques to an NTFS volume.

Chapter 3

Analysis and Specification

3.1 Problem Analysis

The aim of this project was to identify and develop a new technique or capability within the remit of virtual machine security architectures. The first task was to read the current literature and look for a relatively underdeveloped area of research. Much of the existing work was found to focus on either the analysis of memory or network traffic. Developing a solution which made use of disk activity to provide insight into host-state was decided as the research topic. The goal of this insight is to facilitate the detection of malicious activity occurring on virtual machine platforms.

Based on research of existing hard disk activity introspection techniques, introspecting Microsoft's New Technology File System (NTFS) was identified as something which had yet to be achieved and was therefore selected as the aim of this project.

The requirements for this solution are based on the need for efficient, scalable architectures for providing security in virtualised environments [Bordbar et al., 2012]. Computational power is one of the main commodities provided by cloud infrastructure, therefore it is important that the computational overhead of any security mechanism is minimal. Other key considerations are scalability and security.

3.2 Functional Requirements

Functional requirements define specific behaviour or functions of the software to be developed. These functional requirements are placed in order of dependency. NineDot was developed with the following functional requirements:

1. The software should be able to parse the physical layout of a hard disk drive.
2. The software should be able to locate NTFS partitions on a physical drive.
3. The software should be able to parse an NTFS partition and locate file records.
4. The software should be able to parse file records and determine appropriate metadata about each file on the partition.
5. The software should be able to pinpoint the absolute location of a file on disk given a file name or file record index.
6. The software should be able to extract files stored on an NTFS volume into a non-native environment.
7. The software should be able to provide a notification of virtual machine disk activity.
8. The software should be able to resolve notifications of virtual machine disk activity and identify which file was written or modified by a virtual machine.
9. The software should be able to extract files which have been modified or created by a virtual machine, in real-time, into a privileged external domain.
10. The software should be able to check extracted files for signs of malicious activity using conventional anti-virus or anti-malware software.

3.3 Non-functional requirements

Non-functional requirements specify criteria which can be used to judge the resulting system. NineDot was developed with the following non-functional requirements:

1. The software should be transparent to a guest virtual machine.
2. The software should not significantly impact the performance of a guest virtual machine.
3. The software should have minimal impact on the resources of the system into which it extracts and inspects files.
4. The software should not jeopardise isolation between guest virtual machines.
5. The software should not jeopardise isolation between guest virtual machine and the virtual machine monitor.
6. The software should not introduce new attack vectors.
7. The software should be scalable to protecting multiple guest virtual machines.

Chapter 4

Xen Hypervisor

4.1 Preliminaries

4.1.1 Hypervisors

A hypervisor (or virtual machine monitor) is a piece of software or firmware which creates virtual machines on a host system. Virtual machines are entire computing platforms which rather than running on physical hardware run on an imitation logical interface created by a hypervisor.

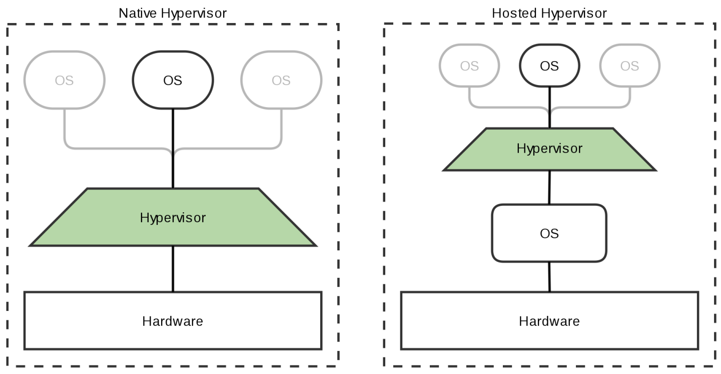


Figure 4.1: Two types of hypervisor.

Hypervisors can be placed into one of two types shown in Figure 4.1.

Native hypervisors, also called type I or bare-metal hypervisors, run directly on a systems hardware and are highly specialised for the task of creating virtual machines. Xen and Hyper-V are two mainstream bare-metal hypervisors.

Hosted or type II hypervisors run on top of a general purpose operating system, VirtualBox, VMware Workstation and QEMU are some widely used examples.

4.2 About Xen

The architecture of a typical Xen hypervisor virtualisation environment is shown in figure 4.2.

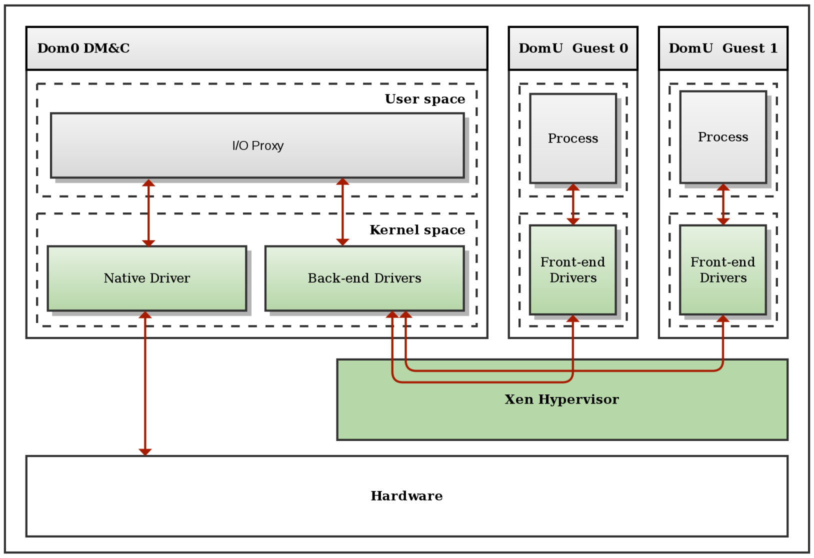


Figure 4.2: A typical Xen hypervisor architecture

The Xen environment architecture has three components: the Xen hypervisor itself, a privileged domain zero (Dom0) virtual machine used for domain monitoring and control (DM&C) and unprivileged guest virtual machines referred to as domain Us (DomUs).

The Xen hypervisor is a layer of software which runs directly on the hardware, below any operating systems. The hypervisor is responsible not only for providing hardware abstractions, but also for scheduling the CPU and appropriating memory between the guest virtual machines running above it. The Xen hypervisor is not concerned with managing access to I/O devices such as network cards, hard disk drives or video displays [Xen Project, 2009].

Dom0 is a privileged virtual machine running above the Xen hypervisor. Dom0 starts before any other virtual machine on the system and manages the DomU unprivileged domains.

Dom0 is typically a modified Linux kernel and runs a Xen management toolstack. Dom0 provides virtual disk and network access to the DomU guests and multiplexes access to the hardware for these devices. Dom0 runs special back-end drivers through which DomU guests make requests for access to the disk or network adapter. The driver queues requests from DomUs and relays them to a native device driver.

DomUs are unprivileged virtual machines which typically have no direct access to physical system hardware. The DomUs in figure 4.2 are paravirtualised (PV guests) as they have been modified to include Xen front-end drivers. Xen front-end drivers are aware that they do not have direct access to the hardware, although they present themselves to the guest OS just like a native driver. Xen supports running completely unmodified operating systems as guest virtual machines, these are called hardware virtual machine guests (HVM guests) [Xen Project, 2009].

4.3 QEMU

Quick Emulator (QEMU) is a hosted (type II) hypervisor. QEMU can be used to create entire virtual machines capable of running unmodified operating systems. QEMU provides several components through which it can be integrated with Xen to provide full hardware virtual machines capable of running unmodified operating systems.

As the Xen hypervisor does not concern itself with I/O devices, domain zero must provide an interface through which unprivileged domU virtual machines can access disk and network resources. QEMU enables guests to run unmodified operating systems with regular hardware device drivers by providing a device-model daemon which runs on dom0 and takes care of I/O access. Each HVM guest requires its own QEMU daemon which handles all of the network and disk requests made by the guest [Xen Project, 2009].

Chapter 5

NTFS

Researching the inner workings of NTFS took considerable project time. NTFS is proprietary Microsoft software and the specification is not available. The most accomplished open source implementation is NTFS-3G [Tuxera, 2015]. Although substantial, NTFS-3G is an incomplete implementation and it is incompatible with VMM introspection of disk activity.

5.1 Preliminaries

5.1.1 Hard drive structure

Traditional hard disk drives (HDD) (See figure 5.1) are typically comprised of several flat, metal, magnetically coated disks (platters). Data is stored and retrieved from the platters by a series of magnetic heads which are moved by an actuator across the surface of the disks. An outdated method of specifying the location of data on a hard disk is using cylinder-head-sector (CHS) addressing. CHS addresses used to identify the physical location of data on a hard disk, every cylinder had an equal number of sectors and therefore a linear relationship between CHS address and position on disk was possible. This method of arranging data is highly inefficient because the circumference of the disk is much larger on the outermost tracks. Hard drives now vary the number of sectors per cylinder according to its location on the disk and consequently CHS addresses no longer correspond to the physical layout of data on a hard drive. CHS addresses are converted by the drive to the actual address for a specific hardware configuration [Tanenbaum, 2005, pp 87].

Traditionally the sector size of hard disk drives is 512 bytes, although recent industry pressure is pushing to increase this to 4096 bytes. Sector size is a hardware defined component of a hard disks construction.

Clusters are a software abstraction which are used to reduce the overhead of managing on-disk data structures. A very commonly used cluster size is 4096 bytes (8 sectors).

Partitions are another type of software abstraction, disk partitioning divides a disk drive into multiple logical spaces and is common even when only one operating system is installed.

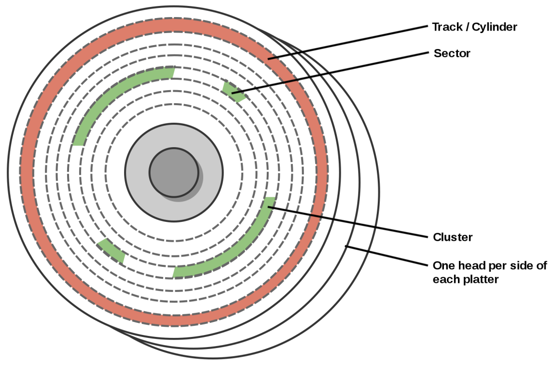


Figure 5.1: A traditional hard disk drive model

5.1.2 Master Boot Record

Most commonly the very first sector on a computer HDD contains a boot sector which is copied by the BIOS into memory and used to facilitate the boot process. The boot sector of a partitioned hard disk is known as the master boot record (MBR), the structure of an MBR is shown in Figure 5.2.

The bootstrap code contains machine code instructions which ultimately serve to transfer control of the system to the chosen primary partition. The primary partition table contains sufficient information about the first four partitions on a hard disk drive for the bootstrap code to transfer control to the boot sector of that partition. CHS addressing is still used here to specify the boundaries of each partition.

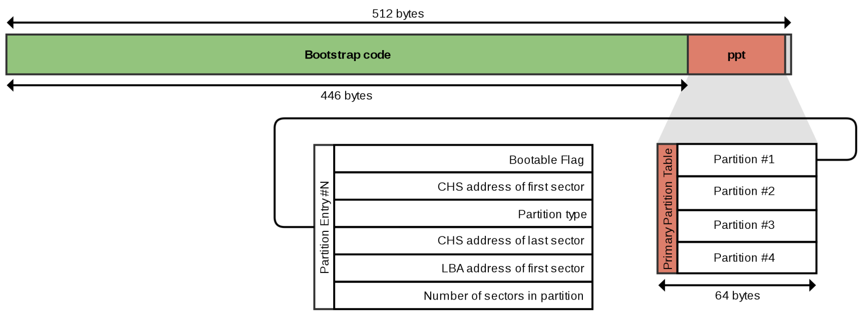


Figure 5.2: Master Boot Record structure

5.1.3 File systems

A file system is a specification for arranging data on a disk and a means of navigating that structure. Typically this is achieved by creating metadata to describe the data. Each group of structured data is called a file.

There are a considerable number of different file systems, many are tailored to specific mediums or to specific goals such as maximum speed or largest capacity. Owing to the popularity of the Microsoft Windows operating system, the most common file system is NTFS.

5.2 NTFS

5.2.1 Why NTFS?

NTFS is used by default on every consumer-oriented release of the Microsoft Windows operating system since Windows XP (2001), and is therefore utilised by approximately 91% of all desktop computers [NetMarketShare, 2015]. A factor limiting the uptake of VMI technology is the fact that there is no x in ‘file systems are to disk drives what x is to memory’. The structure of memory is highly dependent upon the host platform. Fortunately hard disk storage is much more ordered. Targeting the most popular file system provides the greatest possible potential interest and uptake in this project.

5.2.2 NTFS overview

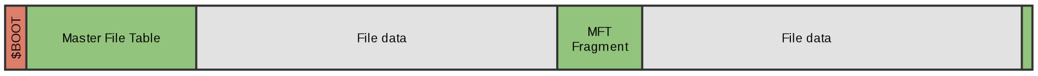


Figure 5.3: A simplified NTFS partition overview.

The overall layout of a typical NTFS partition is shown in figure 5.3. There are only 3 types of region which make up the volume: an NTFS boot sector, regions of master file table, and regions of file data. Despite its simplicity NTFS is a highly advanced file system with many features.

The boot sector is always the first sector on an NTFS partition and it is critically important for understanding the remaining partition. The boot sector contains a structure called the BIOS Parameter Block (BPB) which is always located at offset 0x0B [Microsoft TechNet, 2015]. The BPB contains the information required to understand the NTFS partition structure with which a disk has been formatted. An abbreviated diagram showing the most important sections is shown in figure 5.4. The BPB structure is documented by [Wilkinson, 2012].

The bytes per sector field allows NTFS to operate on drives which deviate from the traditional 512 byte sector size. The number of sectors per cluster defines the minimum unit of disk space which is allocated for file storage by NTFS. The default NTFS sector size is 4096 bytes, 8 regular sectors per cluster.

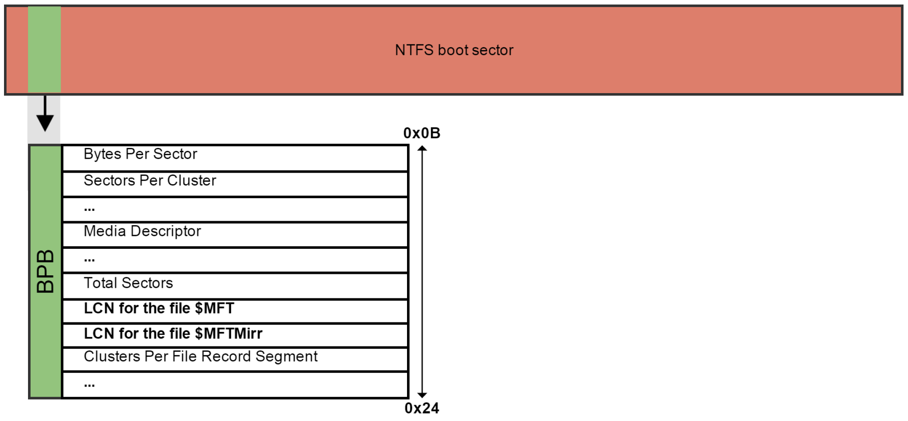


Figure 5.4: An abbreviated NTFS Boot Sector BPB layout.

Clusters on NTFS volumes are sequentially numbered from the beginning of the partition and given a logical cluster number (LCN). The BPB provides the logical cluster number for the master file table.

5.2.3 Master File Table

The main data structure in any NTFS volume is the Master File Table (MFT) [Tanenbaum, 2005]. All objects in NTFS have a record in the MFT, which is similar in structure to a database [Microsoft TechNet, 2003]. Everything is a file under NTFS, so directories, the boot sector, symbolic links, a reference to the MFT itself all have records in the MFT.

From a structural perspective the MFT consists of units called file records. The size of each file record is specified in the BPB, the default size is 1024 bytes.

When a volume is first formatted with NTFS, 12.5% of the available clusters are reserved for MFT file records and cannot be used for file data [Schwarz, 2013]. As the system is used file data is placed in the region beyond the MFT. Depending on system usage and volume size, the reserved space may become insufficient. In this scenario NTFS will define another region of the disk in which to store new file records which will not fit in the first continuous region. When this happens the MFT is said to be fragmented. The MFT in figure 5.3 has two fragments.

The first 16 file records in the MFT are reserved for metadata files which contain information about the MFT itself. The most important of these files within the scope of this project is record zero, which specifies the allocation information for the MFT itself. One reason why this is necessary is that the MFT may become fragmented and cannot be assumed to be continuous on disk.

5.2.4 File records

The master file table consists of sections of contiguous file records. Each file record contains metadata about the file or directory for which it has been created. If there is more metadata than will fit in one record then there may be more than one file record for the same object Wilkinson [2012]. A typical file record structure is shown in figure 5.5.

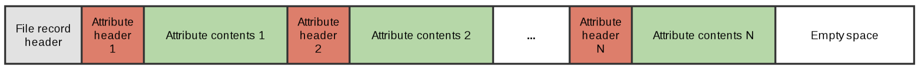


Figure 5.5: A typical file record structure.

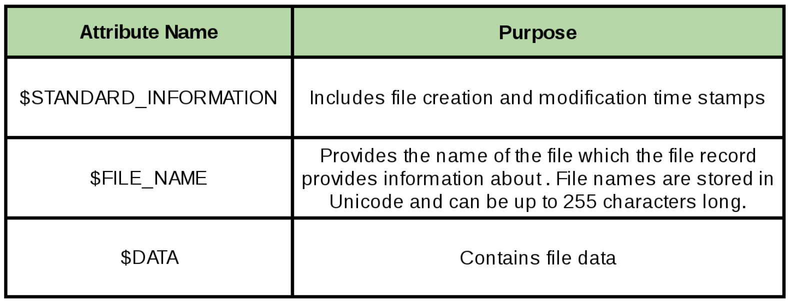
At the beginning of every file record is a small 48 byte header. The header identifies the structure as a file record, maintains consistency checking information and identifies whether the record is for a file, a directory, or has been deleted. The file record header also contains the offset and type of the first file attribute.

5.2.5 File attributes

File metadata is contained in structures called attributes. There are 18 different NTFS file attributes. Most files do not have all possible attributes and some have more than one of the same type of attribute. Each file attribute has a header which specifies its type, size and whether the attribute data is contained within the file record or is large enough to warrant being stored elsewhere on disk [Russon and Fledel, 2008, pp 7].

Both meta file records (the first 16 on an NTFS volume) and file attributes have names which begin with '$'. To avoid confusion file records have names which are $CamelCased whereas file attribute names are written in $UPPERCASE.

The most important file attributes for this project are shown in figure 5.6.



Attribute Name	Purpose
$STANDARD_INFORMATION	Includes file creation and modification time stamps
$FILE_NAME	Provides the name of the file which the file record provides information about. File names are stored in Unicode and can be up to 255 characters long.
$DATA	Contains file data

Figure 5.6: Noteworthy NTFS file attributes

5.2.6 File data

The presence of file data is indicated by the inclusion of a $DATA file attribute in a file record. The location of file data is determined by its size. Files which are smaller than the difference between the size of their metadata and the size of a file record (typically 900 bytes) are called resident files. Resident files have their file data inside the MFT file record [Microsoft TechNet] which is highly efficient as it is not necessary to read another location from the disk.

Files which cannot be contained within a file record are allocated one or more clusters of disk space and are termed nonresident. Nonresident files have a slightly different $DATA attribute header, which rather than specifying the offset to the attribute content gives the offset to a structure called a run list.

The file record attribute property of residency is not confined to the $DATA attribute. Many other file attributes may also have their contents stored outside of a file record and have disk clusters allocated for this purpose. All file attribute headers include a flag which indicates whether the attribute is resident or nonresident [Russon and Fledel, 2008].

Runlists

Nonresident attributes have their content stored outside of a file record, the attribute header instead points to a structure called a runlist.

The purpose of a runlist is to identify which clusters on disk contain nonresident attribute data. Ideally the data would be stored in one continuous block of clusters, however it is possible for data to become fragmented across multiple sections of disk clusters. Each continuous interval of clusters is recorded in a structure called a data run. A runlist contains one or more data runs [Russon, 2014].

Figure 5.7 shows a runlist of several data runs.

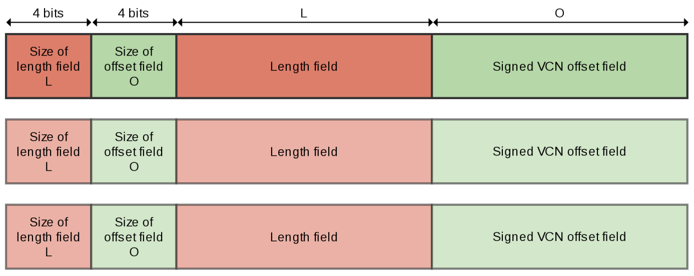


Figure 5.7: NTFS run list structure comprising of three data runs

Each data run contains the location and size of one continuous sequence of data clusters. The fields which contain this information are variable in size between one and eight bytes, each field is only as long as is required to hold the correct value. For this reason there are an additional two fields at the beginning of each data run which specifies the length of the fields which follow. The final element in a runlist has an offset value of zero.

The offset value of a data run is a signed number and yields a virtual cluster number (VCN). A virtual cluster number is an offset relative to the starting logical cluster number (LCN) of the previous data run. If there is no previous data run, as this is the first element in the runlist, then the LCN is equal to the VCN. The reason for offset being a signed value is that it is possible for logically subsequent data clusters to be located physically before the current cluster position.

5.3 Final Word

It should be emphasised that this is just a summary of the pertinent NTFS components which were required for this project, there are many more features and complexities not mentioned here or which are shown abbreviated.

The most useful resource for understanding NTFS was the documentation written by Russon and Fledel [2008] which originally accompanied the Linux NTFS Driver Project. The computer forensics course notes on NTFS by Schwarz [2013] were also very valuable.

Chapter 6

Design

6.1 Design influences

The design of NineDot is based on the work of Broder [2010] who first combined disk-based intrusion detection with virtualisation techniques. The architecture Broder [2010] developed is split into four parts which are discussed below.

Firstly a mechanism for providing a notification when a virtual machine writes to its disk is required. The notification mechanism must indicate how much was written and to which part of the virtual disk. The next step is to identify what was written to disk by appropriately inspecting, reconstructing and parsing the raw file system data belonging to the DomU which caused the notification.

Once a write notification has been semantically resolved to a particular file, it will be extracted into the privileged domain where it can be checked using regular anti-virus software. The mapping between raw disk sectors and the DomU file system is maintained by predicting the behaviour of the file system given an action such as a file increasing beyond its current size or being moved from one directory to another.

Broder [2010] developed a proof of concept implementation of this architecture which introspected the FAT32 file system and leveraged KVM virtual machine monitor technology.

6.2 System design

NineDot is designed around a scenario in which the Xen hypervisor is hosting one or more Microsoft Windows hardware virtual machine (HVM) instances which are using NTFS partitions for disk storage. The guest virtual machines are using virtualised disks provided by the QEMU disk I/O proxy running in Xen domain zero (Dom0). Dom0 is running a modified Xen-kernel Linux distribution such as Debian. NineDot works on the principle of a trusted hypervisor which has not been compromised, this assumption is typical for this type of system [Jiang et al., 2010, Shaw et al., 2014].

The high-level process with which NineDot monitors a guest virtual machine for signs of intrusion is shown in figure 6.1

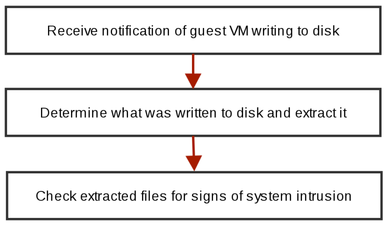

Receive notification of guest VM writing to disk
Determine what was written to disk and extract it
Check extracted files for signs of system intrusion

Figure 6.1: High level system process.

6.2.1 Notification of disk activity

Hardware virtual machines (HVMs) running on a Xen hypervisor reply upon QEMU running in domain zero to proxy their disk and network device interactions. QEMU therefore was the target application through which HVM DomU disk activity metadata was obtained.

6.2.2 Resolving DomU disk activity and extracting files

Metadata about the disk activity of a HVM DomU can be used to determine the state of the guest file system. NineDot is equipped with appropriate knowledge of NTFS internal structures which are cast upon raw disk data to reconstruct the DomU semantic view of the file system (i.e. files and directories).

With information about the location and structure of data in the guest file system it is possible to extract files which have been created or modified into the privileged domain.

6.2.3 Inspecting extracted files for signs of malicious activity

Files extracted into the privileged domain can be inspected or checked with traditional anti-virus or anti-malware software as easily as any other file on the host.

Chapter 7

Implementation

The NineDot system architecture is shown in figure 7.1. It should be compared to figure 4.2 which shows the architecture of a typical Xen virtualisation environment.

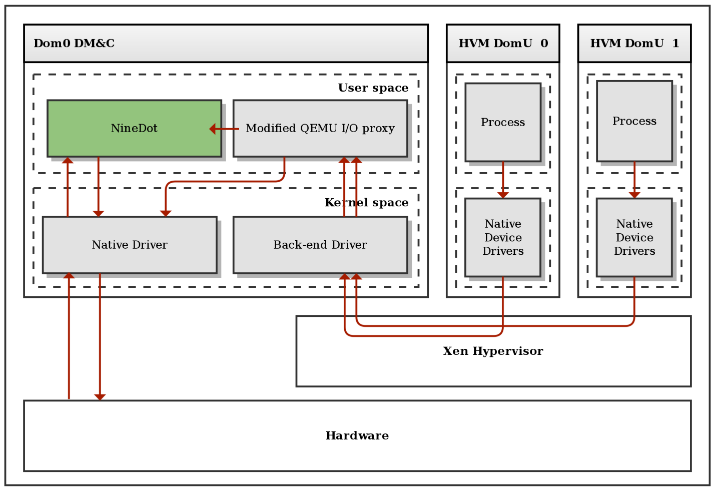


Figure 7.1: System architecture overview.

The NineDot architecture has three distinct components, but the majority of its implementation is contained within the NineDot server component highlighted green in figure 7.1. The three components of the NineDot architecture each have a functionality which corresponds to one of the three elements of the high-level system process shown in figure 6.1.

7.1 Preliminaries

NineDot is written in C (C99) and developed for Linux. It uses the C POSIX library and the POSIX threads library.

7.1.1 POSIX

Portable Operating System Interface (POSIX) is a standard which specifies application programming interfaces (APIs) with the goal of maintaining compatibility between operating systems [The Open Group, 2013]. The C POSIX library adds additional features to those included in ANSI C including time and date functions, Unix domain sockets, file information and functions for manipulating strings. The POSIX threads library is a way of managing concurrency which is portable across POSIX compatible operating systems.

7.2 Notification of DomU Disk Activity

Xen HVM DomU guests each require a QEMU device model daemon running on domain zero to proxy their unmodified device driver I/O requests for disk access. This makes QEMU a very well positioned component of the Xen architecture from which to gain insight into the disk activity of guest virtual machines. Another viable approach would have been to use conventional virtual machine memory introspection (VMI) to introspect the ntfs.sys NT file system driver process, however, this would have both complicated the solution and made it less flexible.

The QEMU project source code is written in the C programming language, spans over 2900 files and exceeds 1.2 million lines of code. Modifying QEMU with the goal of accessing disk activity metadata had to be accomplished without compromising the speed or stability of the device model.

To achieve the required effect QEMU was modified to include two new features. Firstly a Unix domain socket (UDS) client was added. Unix domain sockets are a method of interprocess communication (IPC) which are quick and easy to implement. The UDS client runs in its own thread so that it does not block any other part of the QEMU system. The client is self-forming and self-healing so it will connect to its server counterpart without any user interaction as the QEMU device model daemon is launched. The client thread seeks to restore any broken or uninitialised connections which occur. The second feature is that the virtual disk device drivers were modified so that each time a guest writes to its virtual disk, metadata about the disk activity is also written to the UDS socket.

The QEMU device model main() method can be found in *vl.c*. The block device driver used

for 'raw' files (physical hard disk volumes) is *block/raw-posix.c*
The features added to QEMU are contained within these two files.

The UDS client functionality is located in *vl.c* because the thread which creates and maintains the UDS socket should be created as soon as the device model is launched. Interrupted connections are handled by registering a new signal handler for the POSIX SIGPIPE signal which is sent from the Dom0 kernel to QEMU should the UDS client try to write to a socket which has no end point. The client thread sleeps for 15 seconds and retries should it be unable to connect to a server.

Disk activity metadata is accessed from the *block/raw-posix.c* driver which includes two methods that are essential for gaining insight into host disk activity. The *raw_aio_writev* and *raw_aio_readv* methods are called any time a DomU connected to the QEMU daemon writes or reads respectively to its virtual disk.

The *raw_aio_writev* method is modified so that if the UDS client is connected, the sector number (data location) and number of sectors (data size) of the disk write is written to the UDS. The metadata is written in binary as a 96-bit 2-tuple consisting of a 64-bit sector number and a 32 bit number of sectors. If the UDS client is not connected then the metadata is simply not written to the socket.

The modified version of QEMU created for this project can be found at `https://github.com/hicksc/qemu`. The latest master branch at the time of forking the master branch was version 2.2.50

7.3 Semantic reconstruction and extraction of DomU Disk Activity

A more detailed overview of the NineDot architecture is shown in figure 7.2. This section focusses on the implementation of the NineDot raw extraction engine highlighted green.

The NineDot raw extraction engine primarily serves to reconstruct file system events (i.e. file creation or modification) occurring inside a HVM DomU guest. NineDot uses semantic awareness of the low-level inner workings of NTFS to cast raw data from the guest disk into files which can be extracted into and inspected from the privileged domain.

NTFS Forensics: A Programmers View of Raw Filesystem Data Extraction written by Medeiros [2008] was a useful resource during the development of this component.

The implementation of NineDot can be understood by looking at each of its features. The term offline here is used to refer to a disk volume which is not in use by an unprivileged virtual machine.

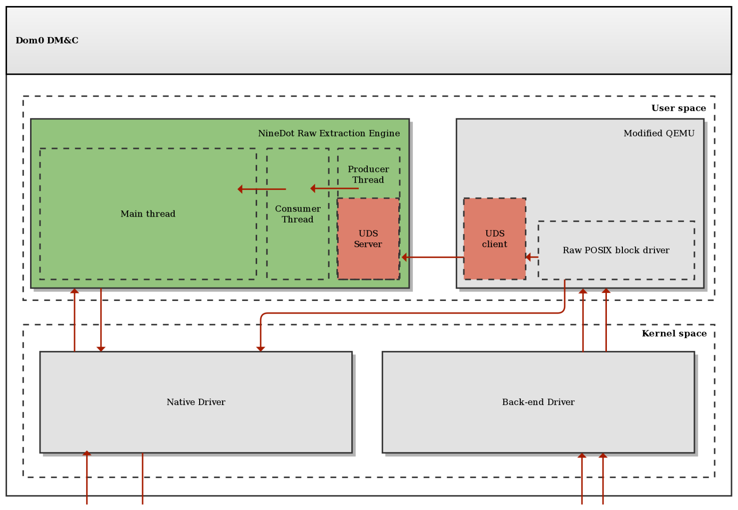


Figure 7.2: Detailed system architecture overview.

7.3.1 Understanding NTFS volumes: Extracting the MFT

NineDot can read and understand the contents of an NTFS volume. The Master File Table (MFT) contains a record of every file and directory on the file system and is therefore essential for understanding an NTFS volume. The process of locating and extracting NTFS master files tables from a disk is shown in figure 7.3.

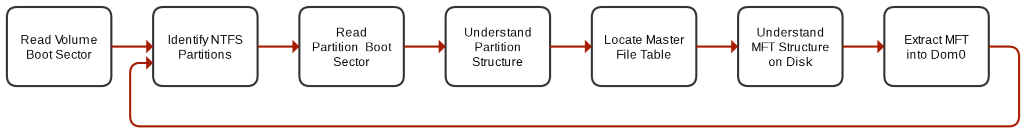


Figure 7.3: Master File Table extraction process

The first stage of understanding a disk volume is a process of reproducing many of the stages of the boot process which occur when a computer (real or virtual) is powered on. When a computer is first booted control of the CPU is handed to the BIOS, a small piece of firmware which aims to get the computer started and hand over control to an operating system. The BIOS looks at each of the disk drives to see if they contain a boot sector. A valid boot sector will include a primary partition table (see figure 5.2 on page 19) which lists the first four partitions on the disk. Each of the primary partition table entries identifies the type of partition, and whether or not it is bootable.

When NineDot is first launched it opens the target volume as a read-only device (to prevent the possibility of corruption) and looks for a primary partition table in the volume boot sector. If a primary partition table is available then it looks at each table entry to see if there are any NTFS partitions.

For each NTFS partition on the disk, NineDot attempts to locate and extract the master file table (see figure 5.3 on page 20). Firstly it is necessary to understand the structure of the partition. The number of bytes per cluster is derived and used to determine the absolute position of the master file table on disk.

The first file record in the master file table always contains file metadata about the master file table itself. The reason for this is that the master file table can become fragmented and has other properties just like any other file on the file system.

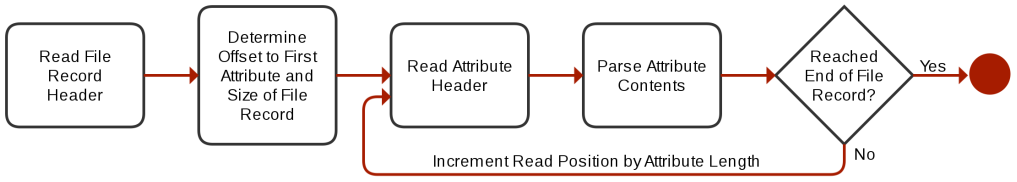

Read File Record Header
Determine Offset to First Attribute and Size of File Record
Read Attribute Header
Parse Attribute Contents
Reached End of File Record?
Yes
No
Increment Read Position by Attribute Length

Figure 7.4: File record parsing process

The structure of the master file table file on disk can be understood by parsing the file record and its attributes using the process shown in figure 7.4. Each file record contains multiple attributes which describe the structure of the file or directory for which the record was created. File record attributes for the $MFT file include its $DATA attribute which specifies where the file data is stored. The process used to understand a $DATA file attribute is shown in figure 7.5.

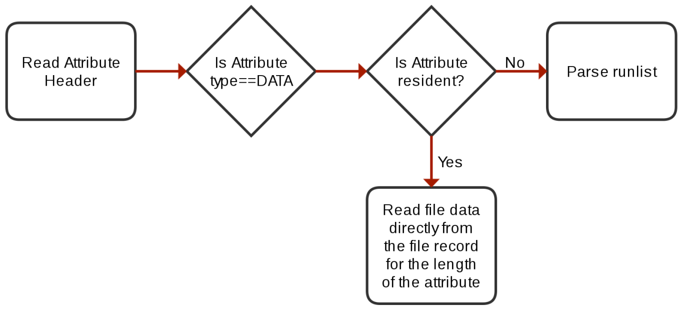


Figure 7.5: DATA attribute parsing

Resident files are those which are small enough to have their file data included in inside a file record. The master file table will never be this small and will always have its location specified by one or more data runs which make up its runlist. Each data run specifies a sequence of contiguous disk clusters which contain data belonging to an attribute (in this case $DATA).

NineDot adds each of these data runs to an internal linked list structure which is later traversed to extract the master file table from disk. For each NTFS partition on the disk, NineDot creates a local file in the privileged domain into which it extracts all of the data runs belonging to the master file table for that partition. The master file table may have just one data run, in which case it is not fragmented and the local file copy is an exact copy of contiguous data clusters from the guest file system.

If there is more than one data run then NineDot reconstructs the fragmented master file table as a single file in the privileged domain. Each fragment copied into the file is preceded by a special fragment record which identifies the original absolute position on disk of the fragment of the master file table which follows.

```
Extracting MFT from partition 0
        $MFT meta file found.
        Writing DATA attribute to local $MFT0 file
        Size of MFT extracted from partition 0: 262144 bytes

Extracting MFT from partition 1
        $MFT meta file found.
        $MFT is fragmented on disk, located 3 fragments.
        Writing DATA attribute to local $MFT1 file
        Size of MFT extracted from partition 1: 74973184 bytes
```

Figure 7.6: NineDot MFT extraction.

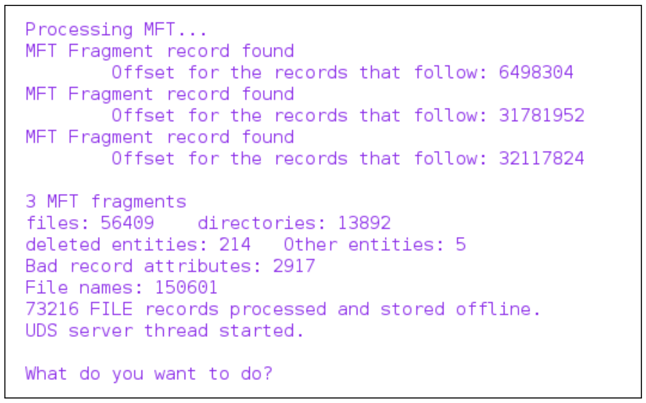
Figure 7.6 shows the output of NineDot as it extracts two master file tables from a typical disk drive containing two NTFS partitions. The second partition contains a fragmented master file table.

Master file table copies written to file in the privileged domain are referred to as offline copies as they can no longer be modified by the virtual machines from which they originate.

7.3.2 Offline MFT parsing and file extraction

The offline master file table copies are parsed to construct a linked list structure in memory. Each element in the list contains a file name, a sector offset to the file record, the size of the file data and the master file table record number.

The master file table is parsed by reading it in file record sized chunks, and for each file record performing the file record parsing process shown in figure 7.4 on page 31.



```
Processing MFT...
MFT Fragment record found
        Offset for the records that follow: 6498304
MFT Fragment record found
        Offset for the records that follow: 31781952
MFT Fragment record found
        Offset for the records that follow: 32117824

3 MFT fragments
files: 56409    directories: 13892
deleted entities: 214   Other entities: 5
Bad record attributes: 2917
File names: 150601
73216 FILE records processed and stored offline.
UDS server thread started.

What do you want to do?
```

Figure 7.7: NineDot output as the files list is built.

Figure 7.7 shows the standard verbosity NineDot output as the list of files is built in memory from the offline master file table copies. NineDot recognises directories and deleted files but presently only adds regular files to its file list for further processing.

NineDot now presents a user interface which is predominantly intended for development, testing and demonstration of the architecture. The first and most basic feature is to print out a list of the offline files. This simply traverses the files list and prints each element to the command line. More advanced features permit searching for specific files and extracting specific files.

Searching for files is accomplished by traversing the files list and checking the respective element member for equivalence to the search term. All matching files have their metadata printed to the screen. Extracting files is accomplished by first performing a search. The returned file metadata is used to perform extraction of file data as shown in figure 7.5 on page 31. Small resident files which have their file data inside a file record are copied directly from the offline master file table copy. For large nonresident files the file data is copied from live disk clusters.

7.3.3 UDS Server

The NineDot UDS server listens for notification of DomU guest disk activity. Each time a DomU writes to its virtual disk QEMU attempts to write metadata containing the sector number and number of sectors for the write activity to the UDS server. UDS server functionality is contained in the *UDSServer.h* project file.

Notification of guest disk activity is handled using a producer-consumer design pattern. A producer-consumer design pattern is a solution to the producer-consumer problem (see figure 7.8 on page 34) in which two (or more) asynchronous threads share a fixed size buffer or queue. The producer generates requests to be processed by putting items (in this case disk write metadata) into the buffer, and the consumer acts upon those requests (in this case by seeking to extract files) [Oaks and Wong, 2004].

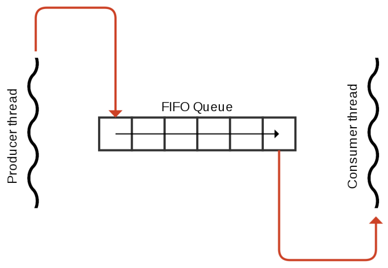


Figure 7.8: Producer-consumer architecture.

The UDS server functionality runs in its own thread and is self-forming and self-healing. The current implementation is a single thread so only one QEMU daemon may connect to it at any given time. The UDS server thread is also the producer thread, it reads sector offset and number tuples from the client and places them into the buffer. Each element in the buffer contains both the sector offset and the number of sectors written for a single write operation. The buffer operates with a first-in first-out (FIFO) behaviour which ensures that disk activity is reconstructed in the order in which it occurs.

The FIFO buffer is a shared resource which must not be concurrently accessed by more than one thread of execution. POSIX thread library mutex locks are used to ensure mutual exclusion of the buffer and prevent race hazards.

7.3.4 Online real-time file extraction

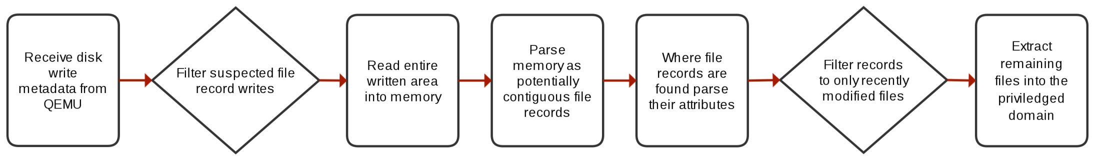

Receive disk write metadata from QEMU
Filter suspected file record writes
Read entire written area into memory
Parse memory as potentially contiguous file records
Where file records are found parse their attributes
Filter records to only recently modified files
Extract remaining files into the priviledged domain

Figure 7.9: Online file extraction process.

Performing online real-time file extraction is a process of carefully piecing together the previous NineDot features and components. Online file extraction is performed by the consumer thread component of the producer-consumer design pattern shown in figure 7.8. The process with which the consumer thread extracts files is shown in figure 7.9.

Each time a file is written or modified under NTFS there are potentially several different writes to disk. One scenario is that a large fragmented file is modified. In this scenario there will be at least one write to the file record in the MFT which corresponds to the file. There will also be one or more writes to clusters on disk which contain the file data.

NineDot receives information about every single disk activity made by a guest DomU, including those which occur at boot before the NT file system driver has even loaded. Once the NT file system driver had loaded it was experimentally observed that NTFS will never write less than one cluster of data to the disk. File records are typically two sectors long, and there are typically eight sectors to a cluster, therefore a resolution issue emerges in which it is not possible to determine intuitively which file record was changed and therefore which file has been written to disk. There are potentially four file records identified for every cluster written to the master file table region of disk.

NineDot makes an attempt to focus its efforts by firstly omitting disk writes which experimentally were found not to be destined for the master file table. Master file table writes span from one to eight clusters in length. Where file records are successfully located further filtering is performed on the file's modification date. Selecting files modified within two hours of the current system time was found to be an effective way of eliminating nearby file records unrelated to the guest disk activity.

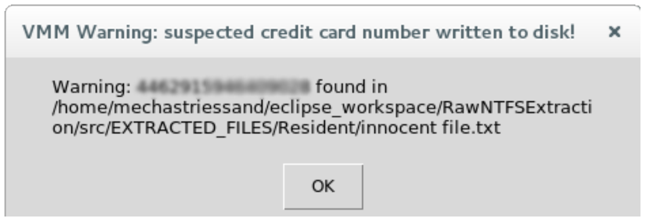
The file records which are identified and fulfil the above criteria have their file data extracted into the privileged domain, where they are given the same file name as they had in the guest file system.

7.4 Identification of Malicious Activity

incrontab is a Linux utility which permits running tasks in response to file system events. It is possible to configure *incrontab* so that every time a file is extracted by NineDot into the privileged domain, some action is taken upon that file.

For the purposes of demonstration and testing, a Python script which implements the Luhn [1960] checksum formula was implemented. The Luhn algorithm can be used to verify whether or not a 16 digit number is potentially a credit card number. It does not guarantee the validity of the number but is a reasonable method of checking if suspicious numbers are being written to file.

incrontab was then configured to run the script on each file which is extracted into Dom0 by NineDot. If one or more numbers which satisfy the Luhn algorithm are extracted to the privileged domain then a warning as shown in figure 7.10 is triggered.



VMM Warning: suspected credit card number written to disk!

Warning: [redacted] found in /home/mechastriessand/eclipse_workspace/RawNTFSExtraction/src/EXTRACTED_FILES/Resident/innocent file.txt

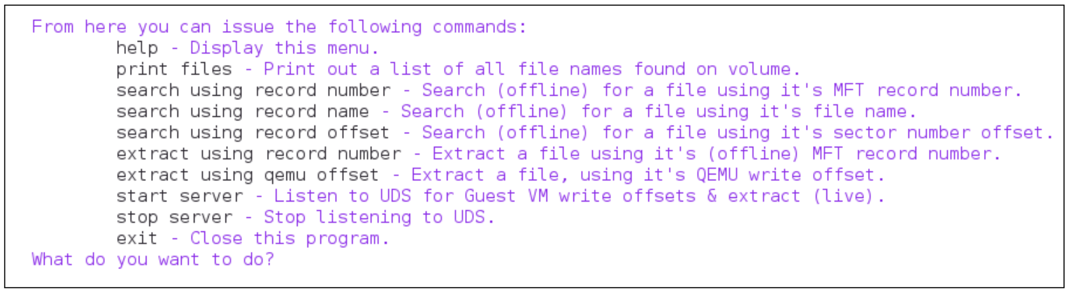
Figure 7.10: Luhn algorithm warning

This component of the project was developed to only a minor degree and was intended to prove and demonstrate the correct extraction of files created within a guest file system rather than provide real-world intrusion detection.

7.5 User Interface

NineDot is primarily intended to run transparently as a background service, with minimal impact on the performance of either virtual machine guests or the privileged domain. Domain zero will most often run without a desktop environment, so a graphical user interface is not appropriate.

For development, testing and demonstration purposes a command line interface (CLI) for NineDot was developed. The NineDot CLI is shown in figure 7.11 below.



```
From here you can issue the following commands:
        help - Display this menu.
        print files - Print out a list of all file names found on volume.
        search using record number - Search (offline) for a file using it's MFT record number.
        search using record name - Search (offline) for a file using it's file name.
        search using record offset - Search (offline) for a file using it's sector number offset.
        extract using record number - Extract a file using it's (offline) MFT record number.
        extract using qemu offset - Extract a file, using it's QEMU write offset.
        start server - Listen to UDS for Guest VM write offsets & extract (live).
        stop server - Stop listening to UDS.
        exit - Close this program.
What do you want to do?
```

Figure 7.11: NineDot command line interface, adapted for a white background

The user interface has several features which ensure its usability. A high usability was judged using the Queensbery [2004] 5Es approach. The 5Es are how effective, efficient, engaging, error tolerant and easy to learn an interface is.

The NineDot CLI is easy to learn and does not require reading any manual pages. The design is error tolerant and helpfully informs the user of any input which is not understood. The interface makes use of colours which engage the user and draw attention to the commands which can be understood by the interface. Commands are plain English and are accompanied by descriptions which allow the user to be effective. Each command has one purpose and is efficient in performing its task. The different *search* and *extract* methods all loop round to accept multiple searches and extractions until the user exits that layer of the interface.

Chapter 8

Testing and Debugging

Testing is a very complex activity which can be difficult to do well. This project had a considerable research component so was inherently less well specified at the beginning than a straightforward software development exercise.

Testing and debugging are different activities. Effective debugging is prerequisite to testing so that the quality of the software is sufficiently high enough to enable rigorous testing [Hambling et al., 2006].

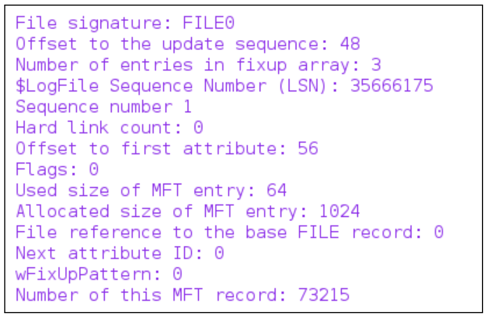
8.1 Debugging

The most basic debugging was accomplished by using static code analysis and extended compiler code warnings. These methods test software without actually executing the code. The Eclipse integrated development environment (IDE) was used to perform static code analysis. Static code analysis is a way of checking for certain programming problems such as missing return values or uninitialised variables. The GNU C Compiler (GCC) was used to compile the program and the *-Wall* and *-pedantic* compiler flags were used at all times. *-Wall* outputs warnings about any questionable code construction and *-pedantic* issues all warnings demanded by strict ISO C.

The inner-workings of NTFS are not well published. Many of the finer details have been reverse engineered by the forensic analysis community. A considerable amount of debugging and testing took the form of causing an event inside a guest virtual machine and manually observing that effect from the privileged domain. A good portion of the NineDot architecture and codebase serves to facilitate the manual inspection and debugging of the inner workings of NTFS.

An example of this is that the master file table (MFT) extracted from a guest domain is first written to disk, then parsed from disk into memory where the offline copy can be analysed. The MFT could have been copied straight from the guest file system into memory, however, this approach permits analysing the MFT file using a hex editor to check that there is no corruption at the fragment boundaries.

Another example is that the *Debug.h* project header file defines the verbosity with which NineDot operates. During development, each stage of the file extraction process was verified by comparing file attributes parsed in the privileged domain with those seen natively inside the guest. File data was printed to the terminal and compared to the data created inside a DomU.



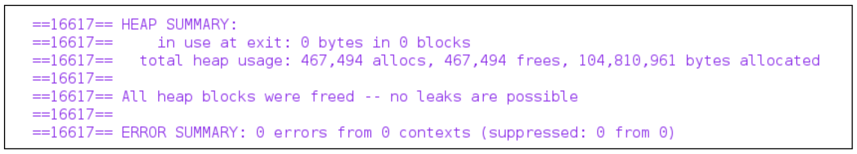
```
File signature: FILE0
Offset to the update sequence: 48
Number of entries in fixup array: 3
$LogFile Sequence Number (LSN): 35666175
Sequence number 1
Hard link count: 0
Offset to first attribute: 56
Flags: 0
Used size of MFT entry: 64
Allocated size of MFT entry: 1024
File reference to the base FILE record: 0
Next attribute ID: 0
wFixUpPattern: 0
Number of this MFT record: 73215
```

Figure 8.1: MFT file attributes recovered from raw guest disk data

For example figure 8.1 shows high verbosity output from NineDot as it parses an MFT file record.

RATS is a static analysis tool which scans source code for common security related programming errors such as buffer overflows and race conditions. RATS is used to highlight sections of code which warrant scrutiny to ensure they are not vulnerable to exploitation. RATS was used to check NineDot and ensure that proper precautions were taken where user input is manipulated.

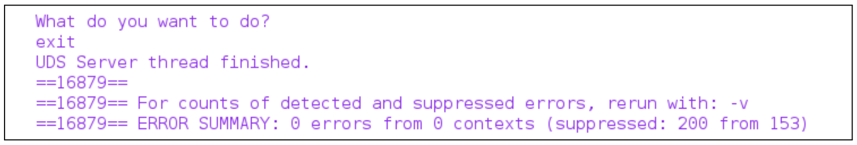
More advanced debugging was accomplished using Valgrind. Valgrind is a framework which includes tools which can dynamically perform memory and thread error detection on code which is running.



```
==16617== HEAP SUMMARY:
==16617==     in use at exit: 0 bytes in 0 blocks
==16617==   total heap usage: 467,494 allocs, 467,494 frees, 104,810,961 bytes allocated
==16617==
==16617== All heap blocks were freed -- no leaks are possible
==16617==
==16617== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 8.2: Valgrind memcheck showing no memory leaks.

The Valgrind *memcheck* tool was used to check for memory errors. Memory errors are a very common issue when writing C programs as memory has to be manually allocated and deallocated by the programmer. A particular concern is memory leaks. Memory leaks cause the memory footprint of an application to grow over time and occur when memory is allocated by the programmer and not deallocated afterwards. Figure 8.2 shows an end-of-development Valgrind *memcheck* output confirming that there are no memory leaks.



```
What do you want to do?
exit
UDS Server thread finished.
==16879==
==16879== For counts of detected and suppressed errors, rerun with: -v
==16879== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 200 from 153)
```

Figure 8.3: Valgrind DRD showing no thread errors.

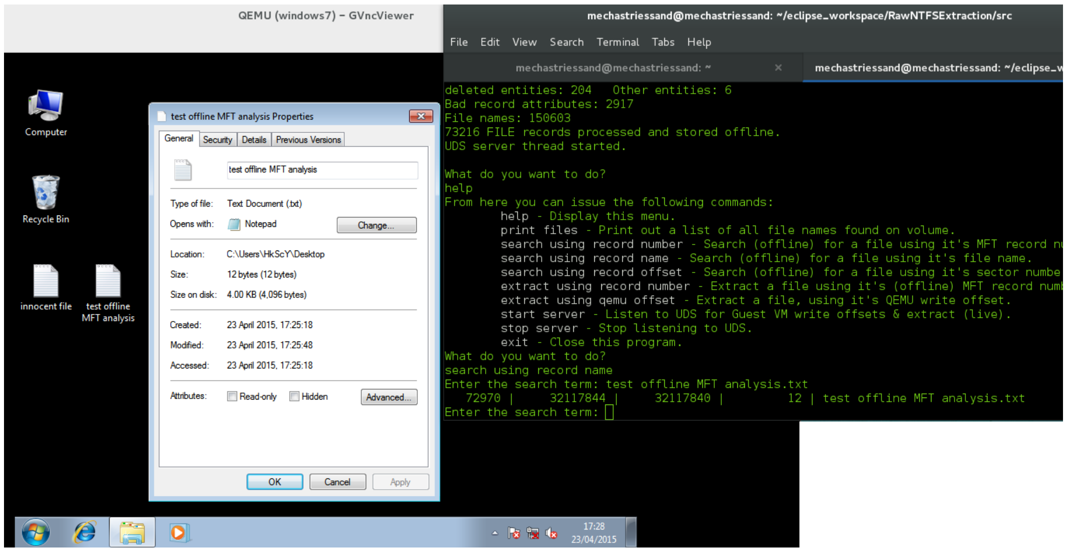
The Valgrind *DRD* tool was was used to check for thread errors. NineDot has three threads: a main thread, a consumer thread and a producer server thread. Thread errors are very common in applications like NineDot which implement concurrency. Data races are a particular concern as they can cause unexpected program behaviour. Data races occur when the same memory location is accessed by more than one thread without sufficient locking. Figure 8.3 shows an end-of-development Valgrind *DRD* output showing that there are no thread errors.

8.2 Black-box testing

Black-box testing examines the functionality of an application without any knowledge of its inner workings. Black-box test cases are derived directly from the specification of a proposed system [Hambling et al., 2006]. The main goal of NineDot is to understand and extract files from an NTFS partition as they are written to disk by a virtual machine. This goal formed the basis of the test scenarios used to confirm the functionality of the software.

Black-box testing was a manual task as it was not possible to easily automate any of the test cases. There are four main test cases which were used to verify the functionality of NineDot.

The first test verifies the offline MFT analysis capabilities of NineDot. A file is first created in a guest virtual machine. In figure 8.4 the file ‘test offline MFT analysis.txt’ is created on the desktop. Next NineDot is run on the target file system and a search is performed for the file which has been created. The command line output shown in figure 8.4 demonstrates that the file record was found in the MFT. From the file record, attributes containing the file name, location and size of the file are recovered.


QEMU (windows7) – GVncViewer
Computer
Recycle Bin
innocent file
test offline MFT analysis
mechastriessand@mechastriessand: ~/eclipse_workspace/RawNTFSExtraction/src
File Edit View Search Terminal Tabs Help
mechastriessand@mechastriessand: ~
mechastriessand@mechastriessand: ~/eclipse_w
General Security Details Previous Versions
Change...
Advanced...
OK Cancel Apply
17:28
23/04/2015

test offline MFT analysis Properties

	test offline MFT analysis
Type of file:	Text Document (.txt)
Opens with:	Notepad
Location:	C:\Users\HkScY\Desktop
Size:	12 bytes (12 bytes)
Size on disk:	4.00 KB (4,096 bytes)
Created:	23 April 2015, 17:25:18
Modified:	23 April 2015, 17:25:48
Accessed:	23 April 2015, 17:25:18
Attributes:	☐ Read-only ☐ Hidden

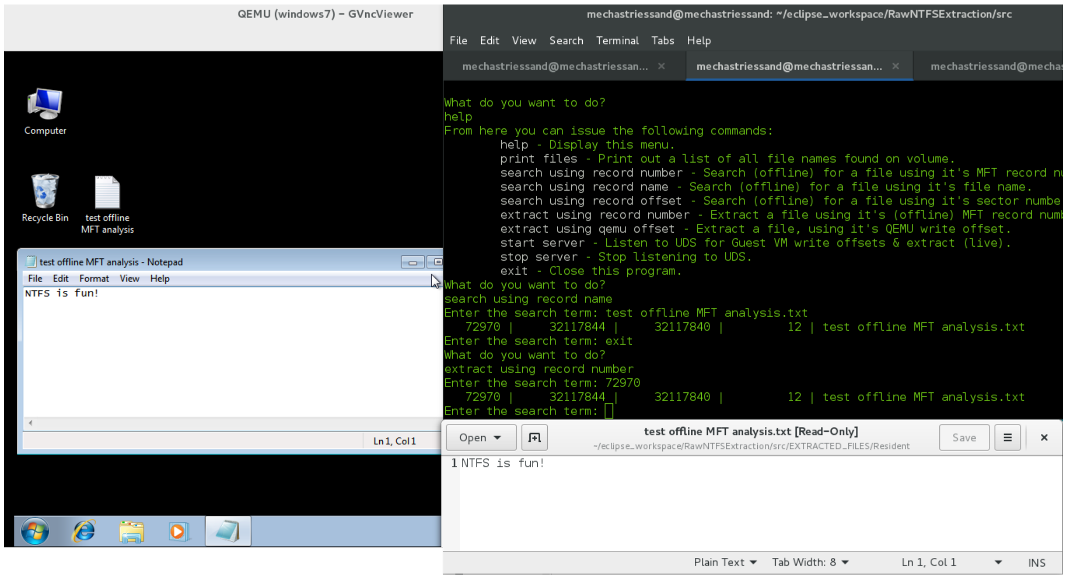
```
deleted entities: 204   Other entities: 6
Bad record attributes: 2917
File names: 150603
73216 FILE records processed and stored offline.
UDS server thread started.

What do you want to do?
help
From here you can issue the following commands:
        help - Display this menu.
        print files - Print out a list of all file names found on volume.
        search using record number - Search (offline) for a file using it's MFT record nu
        search using record name - Search (offline) for a file using it's file name.
        search using record offset - Search (offline) for a file using it's sector numbe
        extract using record number - Extract a file using it's (offline) MFT record numb
        extract using qemu offset - Extract a file, using it's QEMU write offset.
        start server - Listen to UDS for Guest VM write offsets & extract (live).
        stop server - Stop listening to UDS.
        exit - Close this program.
What do you want to do?
search using record name
Enter the search term: test offline MFT analysis.txt
   72970 |     32117844 |     32117840 |          12 | test offline MFT analysis.txt
Enter the search term: 
```

Figure 8.4: Black-box testing MFT file record creation.

The second test scenario checks whether the software can perform offline file extraction on a small resident file. For this test the file created for the last test is opened in the guest virtual machine and a small amount of text is input as file data. Extraction of resident file data from the offline MFT copy is confirmed by instructing NineDot to extract the file as shown in figure 8.5. The file is opened in the privileged domain to verify that the file content is identical to the guests view of the file.

The third test scenario is similar to the second, however, a large nonresident file is created on the guest file system. Files greater than the size of a file record are stored on disk clusters in a vacant part of the disk. The file record keeps track of which clusters are allocated by parsing structures called run lists (see figure 5.7 on page 23). This scenario tests that NineDot correctly parses runlists and is able to recover file data from outside the MFT. Figure 8.6 shows the creation of a large bitmap file, and its successful extraction into the privileged domain.

The fourth and final black-box test scenario checks whether NineDot can extract files in real-time as they are written by a guest virtual machine. Files are created and modified inside the guest file system and they are extracted into the privileged domain without any user input. NineDot passes this test with a latency of between 1 and 30 seconds between the creation of files inside the guest and their subsequent extraction into the privileged domain. There is a video demonstrating this scenario titled *demo.ogv* inside the submission archive for this project.

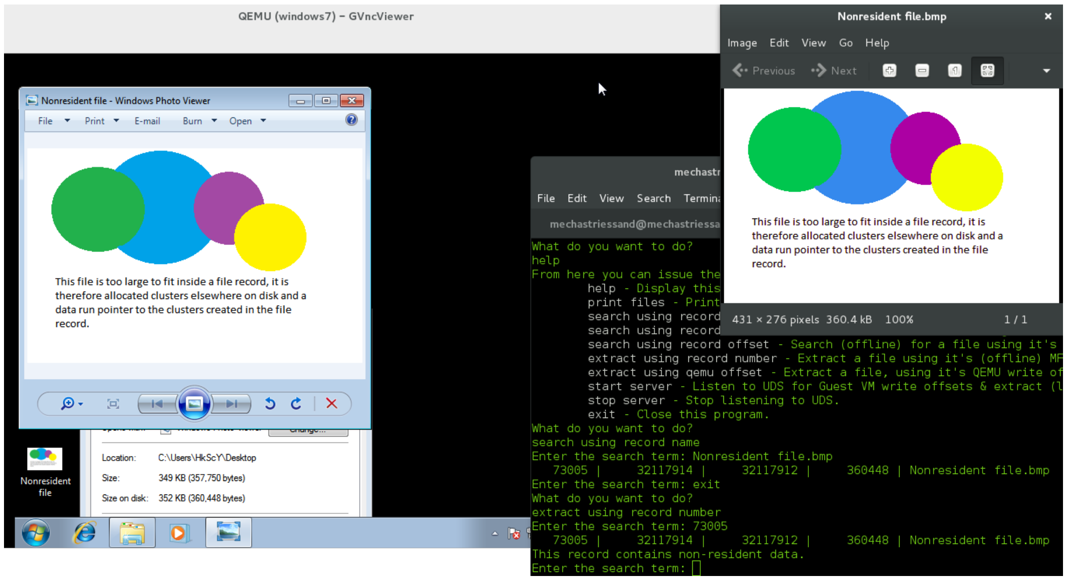

QEMU (windows7) – GVncViewer
Computer
Recycle Bin
test offline MFT analysis
test offline MFT analysis - Notepad
File Edit Format View Help
Ln 1, Col 1
mechastriessand@mechastriessand: ~/eclipse_workspace/RawNTFSExtraction/src
File Edit View Search Terminal Tabs Help
Open
Save
test offline MFT analysis.txt [Read–Only]
~/eclipse_workspace/RawNTFSExtraction/src/EXTRACTED_FILES/Resident
Plain Text
Tab Width: 8
Ln 1, Col 1
INS

```
NTFS is fun!
```

```
What do you want to do?
help
From here you can issue the following commands:
        help - Display this menu.
        print files - Print out a list of all file names found on volume.
        search using record number - Search (offline) for a file using it's MFT record n
        search using record name - Search (offline) for a file using it's file name.
        search using record offset - Search (offline) for a file using it's sector numbe
        extract using record number - Extract a file using it's (offline) MFT record num
        extract using qemu offset - Extract a file, using it's QEMU write offset.
        start server - Listen to UDS for Guest VM write offsets & extract (live).
        stop server - Stop listening to UDS.
        exit - Close this program.
What do you want to do?
search using record name
Enter the search term: test offline MFT analysis.txt
   72970 |     32117844 |     32117840 |          12 | test offline MFT analysis.txt
Enter the search term: exit
What do you want to do?
extract using record number
Enter the search term: 72970
   72970 |     32117844 |     32117840 |          12 | test offline MFT analysis.txt
Enter the search term:
```

```
1 NTFS is fun!
```

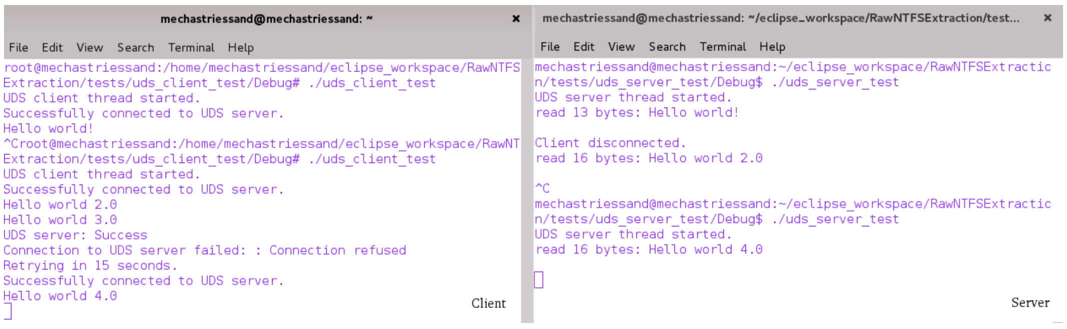
Figure 8.5: Black-box testing resident file extraction.


QEMU (windows7) – GVncViewer
Nonresident file - Windows Photo Viewer
File Print E-mail Burn Open
This file is too large to fit inside a file record, it is therefore allocated clusters elsewhere on disk and a data run pointer to the clusters created in the file record.
Location: C:\Users\HkScY\Desktop
Size: 349 KB (357,750 bytes)
Size on disk: 352 KB (360,448 bytes)
Nonresident file
Nonresident file.bmp
Image Edit View Go Help
Previous Next
This file is too large to fit inside a file record, it is therefore allocated clusters elsewhere on disk and a data run pointer to the clusters created in the file record.
431 × 276 pixels 360.4 kB 100% 1 / 1
File Edit View Search Termin

```
What do you want to do?
help
From here you can issue the
        help - Display this
        print files - Print
        search using record
        search using record
        search using record offset - Search (offline) for a file using it's
        extract using record number - Extract a file using it's (offline) MF
        extract using qemu offset - Extract a file, using it's QEMU write of
        start server - Listen to UDS for Guest VM write offsets & extract (l
        stop server - Stop listening to UDS.
        exit - Close this program.
What do you want to do?
search using record name
Enter the search term: Nonresident file.bmp
   73005 |     32117914 |     32117912 |      360448 | Nonresident file.bmp
Enter the search term: exit
What do you want to do?
extract using record number
Enter the search term: 73005
   73005 |     32117914 |     32117912 |      360448 | Nonresident file.bmp
This record contains non-resident data.
Enter the search term:
```

Figure 8.6: Black-box testing nonresident file extraction.

The Unix domain socket (UDS) client-server functionality between QEMU and NineDot was also black-box tested. These components were first developed outside of the NineDot architecture as separate applications. Client and server functionalities were confirmed as shown in figure 8.7 before they were integrated into QEMU and NineDot respectively. The testing shows the self-forming and self-healing of both the client and server.


mechastriessand@mechastriessand: ~
File Edit View Search Terminal Help
mechastriessand@mechastriessand: ~/eclipse_workspace/RawNTFSExtraction/test...
File Edit View Search Terminal Help
Client
Server

```
root@mechastriessand:/home/mechastriessand/eclipse_workspace/RawNTFS
Extraction/tests/uds_client_test/Debug# ./uds_client_test
UDS client thread started.
Successfully connected to UDS server.
Hello world!
^Croot@mechastriessand:/home/mechastriessand/eclipse_workspace/RawNT
Extraction/tests/uds_client_test/Debug# ./uds_client_test
UDS client thread started.
Successfully connected to UDS server.
Hello world 2.0
Hello world 3.0
UDS server: Success
Connection to UDS server failed: : Connection refused
Retrying in 15 seconds.
Successfully connected to UDS server.
Hello world 4.0
```

```
mechastriessand@mechastriessand:~/eclipse_workspace/RawNTFSExtractic
n/tests/uds_server_test/Debug$ ./uds_server_test
UDS server thread started.
read 13 bytes: Hello world!

Client disconnected.
read 16 bytes: Hello world 2.0

^C
mechastriessand@mechastriessand:~/eclipse_workspace/RawNTFSExtractic
n/tests/uds_server_test/Debug$ ./uds_server_test
UDS server thread started.
read 16 bytes: Hello world 4.0
```

Figure 8.7: UDS client-server black-box testing.

8.3 White-box testing

White-box testing examines the internal structures of software rather than its functionality. White-box tests are derived from source code rather than feature specifications [Hambling et al., 2006]. White-box testing was accomplished using unit testing.

8.3.1 Unit testing

The check unit test framework was used for this project. The unit tests were integrated into the CMake build system so that tests are compiled alongside the project source code and can be executed by calling *ctest* from the project root directory.

Presently seven different unit tests have been written for NineDot. Current test coverage includes all offline file list functionality, testing of file attribute parsing and testing of file record parsing. The unit test code coverage has not been precisely measured but is approximately 50%.

Testing file record and attribute parsing was achieved by extracting several file records from disk using a hex editor and writing them into a C header file as string literals. The string literals are then passed into the various methods which cast low level NTFS structures upon the raw data.

Chapter 9

Evaluation/Appraisal

This project was an attempt to achieve something unique through research and experimentation in the field of virtualised security architectures. NineDot complements existing techniques and enhances visibility into the internal state of guest virtual machines running NTFS beyond what was previously possible. NineDot is able to detect file creation and modification activities occurring inside a virtual machine, recreate those activities in a different domain and finally detect malicious activity by checking the newly extracted files for unexpected contents or by scanning them with conventional anti-virus or anti-malware software. NineDot does not leak memory, and has proven to be both stable and effective during testing.

Whilst NineDot is an effective proof of concept which demonstrates the viability of the architecture, it really only scratches the surface of what is possible. The current implementation can only introspect upon the disk activity of one guest virtual machine at a time. This limitation arises because the NineDot server is not multithreaded and can only serve one client at a time. The architecture would need to be developed further so that disk write metadata was associated with the virtual machine instance from which it arrives. One approach would be to move the entire architecture into QEMU so that the process of extraction into the privileged domain is part of the virtual disk device model which is created for each guest.

A major limitation of the current implementation is that encrypting the contents of the disk (i.e. using BitLocker) would render the level of insight into disk activity somewhere ranging from very limited to non-existent, depending upon whether the encryption is at the file system or disk level respectively. A design decision was made at an early stage to make the assumption that the contents of the disk are unencrypted. There are a number of solutions to this shortcoming. One solution is to use memory introspection to look for the cryptographic keys [Shamir and van Someren, 1999] which decrypt the disk. Another solution is to focus rather than on the contents of what is written to disk, on the patterns of disk behaviour. NineDot could be enhanced with the ability to build historical models of guest disk activity, and then compare current usage probabilistically for signs of intrusion. This solution has many advantages including compatibility with encrypted drives, causing less disk I/O and being file system agnostic.

The current file extraction process is an approximation of the disk activity of a guest virtual machine. A file which has been newly written to disk or had its contents modified will always be extracted by NineDot, however, any files which coincidentally happen to have master file table entries within a few sectors of the culprit may also be extracted if they have been modified recently enough. This resolution issue is caused by the fact that NTFS will write an entire cluster (usually 8 sectors) of data to disk, even if it is only making changes to one file record (usually 2 sectors) within that cluster. NineDot currently works around this by only extracting files which have been modified within two hours of the write notification occurring. This presents an opportunity for malicious files on a compromised system to hide themselves by writing incorrect modification times to their file records.

NineDot is able to parse a number of NTFS file attributes. The $STANDARD_INFORMATION, $FILE_NAME and $DATA attributes are fully implemented. There are many more file attributes defined for NTFS, 18 in total. All of the available file attributes for any given NTFS implementation are defined in the $AttrDef metadata file record. An improvement to NineDot would be to parse the $AttrDef file and use its contents to dynamically parse all of the file attributes in each file record. The $INDEX_ROOT and $INDEX_ALLOCATION attributes would give insight into the directory structure of an NTFS volume. Dynamically parsing $AttrDef would ensure compatibility with future revisions of NTFS.

The development of NineDot was quite feature-driven. With hindsight more time would have been spent planning the architecture of the system so that its placement in a virtualised environment lent itself to maximum insight into guest disk activity with minimum overhead. One architectural inspiration is the Forensic Virtual Machine (FVM) model implemented by Shaw et al. [2014]. The FVM model grants introspection privileges to small, specialised virtual machines which navigate the cloud to inspect regular unprivileged virtual machines for signs of infection. NineDot could be placed into a specialised and privileged virtual machine of its own and used to check for very specific disk activities. Writing to the boot sector, for example, might be a reasonably strong indication of a rootkit infection occurring.

Greater use of the existing Linux-NTFS drivers (although they cannot do what NineDot can) may have shortened the development time of the extraction engine and allowed more testing and comparison to be done upon the actual detection of malicious activity.

NineDot is a reliable and effective demonstration that introspecting NTFS disk activity is a feasible and worthwhile endeavour.

Chapter 10

Project Management

The NineDot project was an approximately 50/50 split between research and development, where development time includes debugging and testing the end product. The beginning of the project was particularly research heavy whilst the end was very development intensive. The natural process of researching new topics, exploring new ideas, building prototypes and refining them into working software lent itself to an agile development methodology. Bi-monthly supervisions were used to discuss ideas with both the project supervisor and sometimes other figures from academia and industry. Goals for the project were reviewed where appropriate, often needing to be curtailed to the time available. Version control was managed using a Github repository, the commit time line is shown in figure 10.1.

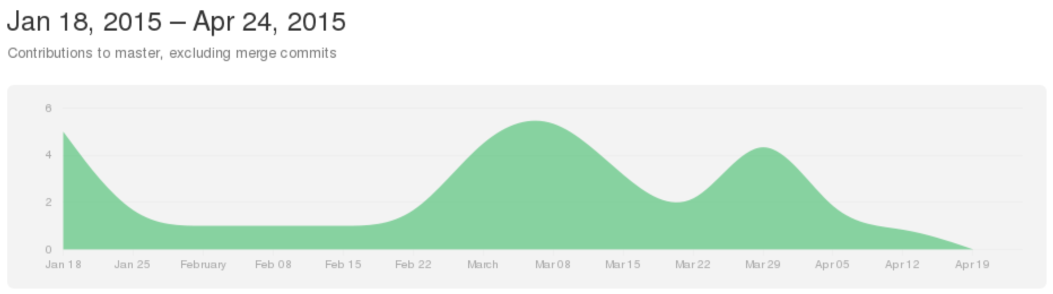


Figure 10.1: Github contributions to NineDot repository.

Early project developments were mostly Python prototypes and modifications to the QEMU virtual disk device driver which probed the feasibility of getting access to metadata about virtual machine disk activity. By December the majority of the project research into virtualisation had been completed and a prototype implementation which displayed disk activity metadata inside a Python server was demonstrated.

Later project developments include deciding upon NTFS as a target platform. This was a risky decision owing the the lack of formal documentation and the fact that nobody had done it before. NineDot was first committed to Github in early January and then developed consistently alongside research into the NTFS platform for the remainder of the project.

The agile development technique adopted for this project was highly effective and permitted the freedom necessary for developing experimental software which spans several different platforms (Xen, QEMU and NTFS).

The end product would have benefited from greater use of software engineering practices to design and specify architectures before implementing any of the software components.

Chapter 11

Conclusion

NineDot has succeeded in achieving a unique development within the field of virtualised intrusion detection architectures. NineDot leverages virtualisation technology to gain insight into the NTFS disk activity of virtual machines. NTFS is the most widely used file system in current usage, and cloud infrastructure as a service is a rapidly growing commodity. The NineDot architecture has the potential to be widely adopted as a technique for providing security to that infrastructure.

Like other introspection architectures, NineDot is isolated from the hosts it protects and therefore can continue to operate in spite of host compromise. NineDot has a good visibility into the internal state of guest virtual machines as main memory is too small to retain all of the programs and data used by a system (they are stored on disk).

NineDot satisfies all of the functional requirements which were defined during the earlier stages of the project. Most of the non-functional requirements have been satisfied, although further testing is necessary to ensure that new attack vectors have not been introduced. NineDot requires further development to enable its scalability to protecting multiple hosts.

NineDot is able to perform offline NTFS disk analysis. It can locate all of the files and directories on a volume, and the files can be extracted from NTFS into a non-native file system from which NineDot is running. The virtual disk driver was successfully modified to provide robust notification of the disk activity (read and write operations) occurring inside a virtual machine. NineDot is able to combine these notifications with knowledge of the low-level structure of NTFS and extract files as they are created or modified by a virtual machine.

There is considerable potential for further development of the NineDot architecture. One of the most rewarding developments would be to use the disk activity notifications to build historical models of expected disk activity which are tailored to each virtual machine. Current usage could then be probabilistically compared to the expected behaviour and unusual activity flagged for attention. This development would enhance the current capabilities which could then be used more selectively where a problem is indicated.

Further works needs to be completed to determine the exact performance impact of NineDot and to ensure that there are no new attack vectors which have been introduced. Overall NineDot is an effective and interesting proof of concept which will hopefully invite further research.

Chapter 12

Bibliography

AV-TEST. Av-test the independent it-security institute, 03 2015. URL `http://www.av-test.org/en/statistics/malware/`.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, and Rolf Neugebauer. Xen and the art of virtualization. *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03*, 2003. doi: 10.1145/945461.945462.

BBC News. Kmart shops hit by payment card hack attack. *BBC Technology*, 10 2014. URL `http://www.bbc.co.uk/news/technology-29595214`.

Joost Bijl. Cryptolocker ransomware intelligence report |fox-it international blog on wordpress.com. 08 2014. URL `http://blog.fox-it.com/2014/08/06/cryptolocker-ransomware-intelligence-report/`.

Behzad Bordbar, Keith Harrison, Syed T.T. Ali, Chris I. Dalton, and Andrew Norman. A framework for detecting malware in cloud by identifying symptoms. *2012 IEEE 16th International Enterprise Distributed Object Computing Conference*, 2012. doi: 10.1109/edoc.2012.27.

Evan Broder. Transparent detection of computer malware using virtualization. Technical report, (Web), 05 2010. URL `http://web.mit.edu/broder/Public/6.uap-report.pdf`.

Cisco. *Cisco Annual Security Report (ASR)*. Cisco Systems, Inc., 01 2014. URL `http://www.cisco.com/web/offer/gist_ty2_asset/Cisco_2014_ASR.pdf`.

Dorothy E. Denning. An intrusion-detection model. *1986 IEEE Symposium on Security and Privacy*, 1986. doi: 10.1109/sp.1986.10010.

Detica. The cost of cyber crime. Technical report, 01 2013. URL `https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/60943/the-cost-of-cyber-crime-full-report.pdf`.

Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 297–312. IEEE, 2011.

John F Gantz, Richard Lee, Alejandro Florean, Victor Lim, Biplab Sikdar, Logesh Madhavan, Mangalam Nagappan, and Sravana Kumar Sristi Lakshmi. *The Link between Pirated Software and Cybersecurity Breaches - How Malware in Pirated Software Is Costing the World Billions (White Paper).* Microsoft Press , U.S., A Joint Study by National University of Singapore and IDC, 03 2014. URL `http://news.microsoft.com/download/presskits/dcu/docs/idc_031814.pdf`.

Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, volume 3, pages 191–206, 2003.

Charles David Graziano. A performance analyis of xen and kvm hypervisors for hosting the xen worlds projec t, 2011. URL `http://lib.dr.iastate.edu/cgi/viewcontent.cgi?article=3243`.

Brian Hambling, Peter Morgan, and Angelina Samaroo. *Software Testing: An ISEB Foundation.* British Computer Society, United Kingdom, 11 2006. ISBN 9781902505794.

L.T. Heberlein, G.V. Dias, K.N. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A network security monitor. 1990. doi: 10.2172/6223037.

IBM Global Education. Virtualization in education (white paper). Technical report, (Web), 10 2007. URL `http://www-07.ibm.com/solutions/in/education/download/VirtualizationinEducation.pdf`.

Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection and monitoring through vmm-based out-of-the-box semantic view reconstruction. *ACM Transactions on Information and System Security*, 13(2):1–28, 2010. doi: 10.1145/1698750.1698752.

Hans. P. Luhn. Computer for verifying numbers, 7 1960. URL `http://www.google.com/patents/US2950048`. US Patent 2,950,048.

Jason Medeiros. *NTFS Forensics: A Programmers View of Raw Filesystem Data Extraction.* Grayscale research, (Web), 06 2008. URL `http://grayscale-research.org/new/pdfs/NTFSforensics.pdf`.

Microsoft TechNet. Microsoft technet library. URL `https://technet.microsoft.com/en-us/library/cc976808.aspx`.

Microsoft TechNet. How ntfs works: Local file systems, 03 2003. URL `https://technet.microsoft.com/en-us/library/cc781134`.

Microsoft TechNet. Disk concepts and troubleshooting. *Microsoft TechNet Library*, 04 2015. URL `https://technet.microsoft.com/en-us/library/cc977221.aspx`.

Gordon E Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, pages 114–117, 04 1965.

NetMarketShare. Market share for mobile, browsers, operating systems and search engines, 04 2015. URL `http://www.netmarketshare.com/operating-system-market-share.aspx`.

Scott Oaks and Henry Wong. *Java threads*. O'Reilly Media, Inc, USA, United States, 3 edition, 09 2004. ISBN 9780596007829.

Tavis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments, 2007.

Ponemon Institute. 2013 cost of cybercrime study: Global report, 10 2013. URL `http://paramountassure.com/wp-content/uploads/2014/04/2013-Cost-of-Cyber-Crime-Global.pdf`.

Whitney Queensbery. Balanc ing the 5es: Usability, 02 2004.

Marcel Rosenbach, Hilmar Schmundt, and Christian Stöcker. Regin malware unmasked as nsa tool after spiegel publishes source code. 01 2015. URL `http://www.spiegel.de/international/world/regin-malware-unmasked-as-nsa-tool-after-spiegel-publishes-source-code-a-1015255.html`.

Richard Russon. Data runs - concept - ntfs documentation, 06 2014. URL `http://0cch.net/ntfsdoc/concepts/data_runs.html`.

Richard Russon and Yuval Fledel. *NTFS DOCUMENTATION*. (Web), 02 2008. URL `https://www.scribd.com/doc/2187280/NTFS-Documentation`.

Thomas Schwarz. 152 ntfs, 01 2013. URL `http://www.cse.scu.edu/~tschwarz/coen252_07Fall/Lectures/NTFS.html`.

Adi Shamir and Nicko van Someren. Playing hide and seek with stored keys, 1999.

Adrian L. Shaw, Behzad Bordbar, John Saxon, Keith Harrison, and Chris I. Dalton. Forensic virtual machines: Dynamic defence in the cloud via introspection. *2014 IEEE International Conference on Cloud Engineering*, 2014. doi: 10.1109/ic2e.2014.59.

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts*. Wiley, John & Sons, United States, 8 edition, 2010. ISBN 9788126520510.

Andrew S. Tanenbaum. *Structured computer organization.* Pearson Prentice Hall, United States, 5 edition, 06 2005. ISBN 9780131485211.

The Open Group. Posix faq, 10 2013. URL `http://www.opengroup.org/austin/papers/posix_faq.html`.

Trustwave. Trustwave global security report, 01 2014. URL `http://www2.trustwave.com/rs/trustwave/images/2014_Trustwave_Global_Security_Report.pdf`.

Tuxera. Open source: Ntfs-3g, 2015. URL `http://www.tuxera.com/community/open-source-ntfs-3g/`.

Giovanni Vigna and Christopher Kruegel. *Host-based intrusion detection.* Technical University Vienna, 2006.

Michael Wilkinson. *NTFS Reference Sheet.* (Web), 01 2012. URL `http://www.writeblocked.org/resources/NTFS_CHEAT_SHEETS.pdf`.

Xen Project. How does xen work? Technical report, 12 2009. URL `http://www-archive.xenproject.org/files/Marketing/HowDoesXenWork.pdf`.

Appendices

Appendix A

Structure of the Project Code

All of the project code is contained in the project archive file. Its immediate contents are shown in figure A.1. Starting with the files in the root directory, *bugs_list* is a simple text file which contains outstanding project bugs which are known at the time of submission. *CMakeLists.txt* controls the CMake build system and an identically named file it is found in each directory which CMake interacts with when creating project Makefiles.

```
RawNTFSExtraction
├── bugs_list
├── CMakeLists.txt
├── CMakeModules
├── Misc
├── src
│   └── EXTRACTED_FILES
│       ├── NonResident
│       └── Resident
└── tests
    ├── test_FileList
    ├── test_NTFSAttributes
    ├── test_NTFSStruct
    ├── uds_client_test
    │   └── src
    └── uds_server_test
        └── src
```

Figure A.1: Directory structure of the project

Figure A.2 shows the *CMakeModules* subdirectory which contains a single file *FindCheck.cmake*. *FindCheck.cmake* enables the CMake build system to find and properly link the check unit test framework when building the project.

```
CMakeModules
└── FindCheck.cmake
```

Figure A.2: Directory structure of the CMakeModules subdirectory

```
Misc
├── ccCheck.py
├── incrontab
├── monitorGuestVM.sh
└── windows7.cfg
```

Figure A.3: Directory structure of the Misc subdirectory

Figure A.3 shows he *Misc* project subfolder which contains supplementary files which helped support the project demonstration. *ccCheck.py* is a Python script which uses the Luhn algorithm to check input files for potential credit card numbers. The *incrontab* file is an incrontab table entry which illustrates of how to configure incrontab to monitor the extracted resident files directory for new files. It reacts to file creation by executing *monitorGuestVM.sh* which is a simple shell script which executes *ccCheck.py* before removing the file which caused the notification. *windows7.cfg* is a Xen management tool configuration script which tells the management tool stack which parameters to start a guest VM instance with. i.e. *xl create windows7.cfg*

The *src* subdirectory shown in Figure A.4 contains the main project code which yields the RawNTFSExtraction (NineDot) executable application upon compilation. The main method for the program is contained in *RawNTFSExtraction.h*, as is the majority of the application logic.

```
src
├── CMakeLists.txt
│   └── Debug.h
├── EXTRACTED_FILES
│   ├── NonResident
│   └── Resident
├── FileList.h
├── NTFSAttributes.h
├── RawNTFSExtraction.c
├── RunList.h
├── UDSServer.h
└── UserInterface.h
```

Figure A.4: Directory structure of the src subdirectory

The *src* directory also contains the following files:

- *Debug.h* controls the verbosity of the programs operation.
- *FileList.h* contains the abstractions necessary to create and manipulate linked lists of NTFS files.
- *NTFSAttributes.h* defines many proprietary NTFS abstractions.
- *RunList.h* contains the abstractions necessary to describe and manipulate NTFS run-lists.
- *UDSServer.h* defines the UDS server thread and the producer consumer design pattern which it feeds.
- *UserInterface.h* defines user interfaces elements and methods.

The *tests* subdirectory shown in figire A.5 contains the check framework unit tests and a prototype UDS client-server application.

```
tests
├── CMakeLists.txt
├── test_FileList
│   ├── CmakeLists.txt
│   └── test_FileList.c
├── test_NTFSAttributes
│   ├── CMakeLists.txt
│   ├── test_NTFSAttributes.c
│   └── test_NTFSAttributes.h
├── test_NTFSStruct
│   ├── CMakeLists.txt
│   ├── FILE_Record.h
│   └── test_NTFSStruct.c
├── uds_client_test
│   └── src
│       └── client.c
└── uds_server_test
    └── src
        └── uds_test.c
```

Figure A.5: Directory structure of the tests subdirectory

Appendix B

Running the system

B.1 Requirements

The software has been written, developed and tested upon the Xen 4.4.1 hypervisor platform, running Debian Linux 3.16.7-2 (Linux kernel version 3.16.0) as domain zero. The device model used is a fork of the latest stable QEMU release available at the time of release (version 2.2.50) which is available from *github.com/hicksc/qemu*. The project makes use of the C POSIX library including POSIX threads and they are required for compiling and running the program. GCC version 4.9.1 has been used to generate executable code. The check unit test framework version 0.9.14 is required for compiling the source files in the tests subdirectory.

Python 2.7.8 is used for running *ccCheck.py*, the GUI element of the script is dependent upon the library PyMsgBox version 1.0.3.

The target NTFS volume used for development was a 30GB Linux Volume Manager (LVM) partition, formatted to NTFS version 3.1 as is the default for Windows XP versions and above.

B.2 Process

The CMake build system has been implemented so that after extracting the contents of *RawNTFSExtraction*, *cmake* can be run within the resulting directory to configure the build system for your specific platform. The resulting makefile will enable the program executable to be built within the *src* subdirectory. Unit tests can be launched using *ctest*.

You must manually create the directories defined by *RESEXTFILESDIR* and *NONRESEXTFILESDIR* in *RawNTFSExtraction.c* within the root of the program executable. The path to the block device containing the NTFS volume for inspection is defined by the constant *BLOCK_DEVICE*, you will need to change this to suit your configuration. After setting an appropriate value for *BLOCK_DEVICE* and compiling the application, it can be launched from the terminal at which point the NTFS volume specified will be parsed and the user interface presented. Verbosity of the programs operation can be configured by setting the value of *DEBUG* and *VERBOSE* constants within *Debug.h*

The UDS server used to listen for disk write metadata can be started and stopped using the *start* and *stop server* commands respectively. Both QEMU and RawNTFSExtraction write to the system debug log to report upon their connectedness. Responding to the files which have been extracted can be accomplished using the *inotify* system to monitor the directory which files are extracted to. An example is provided in */Misc/incrontab* which configures *incron* to run *ccCheck.py* on each extracted resident file and then remove it from the local file system. Instances of plaintext credit card numbers are reported to the system administrator.

Appendix C

Log Book

Term 1, Week 0

I Had my first meeting with my supervisor, we discussed the potential direction of study for my final year project. We left things very open and I went away to think about the types of project that I might like to work on. I agreed to bring a list of three to four project ideas to the next meeting in which we will go through the list and select the most appropriate concept. Behzad and I collaboratively decided that the following areas would be a good basis for investigation: Virtual Machine Introspection, memristor technology, Looking in memory for cryptographic keys, big data.

Term 1, Week 1

I researched the topics which I discussed last week with my supervisor.

Term 1, Week 2

I further investigated the topics of by browsing Google Scholar and the School library for previous students work. I Had a meeting with my supervisor and Keith Harrison who used to work for HP Labs as 'Master Technologist' to discuss and develop the four main ideas.

Term 1, Week 3

I decided that I would study within the remit of Virtual Machine Introspection and prepared my project proposal for submission.

Term 1, Week 4

Mainly researched around the topic of Virtual Machine Introspection (VMI). My aim for the week was to gain a greater understanding into the work completed already on this topic, both at the University of Birmingham and by the scientific community at large. I read papers which provided me with a basic understanding of the technologies and libraries used in recent VMI projects, the existing accomplishments and the current shortcomings.

Reflecting upon my reading so far I think that Intrusion Detection Systems (IDS) using VMI has already been well covered, and that it may be more beneficial (although likely more challenging) to look at disk access rather than network access. Disk access could facilitate identifying malware signatures in much the same way as conventional anti-virus software, whilst retaining the advantage of isolation from the host which motivated VMI.

For next week my goal is to read more papers to gain a better understanding of the topic, and to bring several ideas to Behzad (and possibly John or Keith) to narrow down the scope of the project.

Term 1, Week 5

I read many more papers on the topic of virtualisation security mechanisms and architectures.

I discovered several existing VMI projects: Livewire, LycosID, uDanali, Lares, AntFarm, IntroVirt and ReVirt. I learnt of a project which allows memory events to be replayed for forensic analysis purposes (ReVirt).

I met with Chris and Keith from HP, along with Behzad and one of his PhD students Adrian to talk through the ideas which I have had for the project already: Using disk data analysis to complement existing VMI techniques, VMI IDS, Aggregating multiple threat detection sensors to form more complete protection software.

During the meeting we came to the conclusion than looking at disk data as it streams from VM to hardware will be a good subject for my project.

The project will be split into two parts: Acquiring the disk data and Analysing the data.

This can be used to complement existing malware and intrusion detection efforts, further reducing the opportunities for malware to remain hidden.

I will make the simplifying assumption that the disk is not encrypted.

I will investigate 'XOM' who looked at memory pattern analysis on encrypted memory. The analysis might take the form of time-series analysis.

Term 1, Week 6

Studied 'Transparent Detection of Computer Malware using Virtualization' paper by Evan Broder of MIT CSAIL. Learnt how access to disk writes via QEMU was achieved.

I set up Xen on my own system. I set up LVM on my own system. I considered which filesystem might be a suitable target for introspection.

Term 1, Week 7

Completed setting up Xen on my own system. I researched NTFS, it is documented mainly by reverse engineering and not officially by Microsoft. I got Windows 7 running on Xen as a hardware virtual machine. I tried out para-virtualised disk drivers. Had quite a few issues bridging network access to the guest VM.

Term 1, Week 8

I solidified my knowledge of Xen through study and practical experience. I considered the architecture with which accessing disk write metadata might be possible. I considered the performance of the various proposed architectures. I developed a plan of action with my supervisor. We emailed a few industry contacts for opinions. I recovered from a hard drive failure which wiped out much of the progress with setting up and configuring Xen last week.

Term 1, Week 9

I received correspondence from E.Broder of MIT regarding the QEMU modifications he made for his project. They allowed me to identify which source code files are of interest. I compiled QEMU from source. I configured Xen to use my custom compiled QEMU binary.

I started manually patching QEMU as the patch file is unusable due to significant changes to QEMU since version 0.12.3 I had to learn various parts of the C language. I learnt about Unit domain sockets. I learnt about chmod, tail, and generally became much more proficient at using a Linux terminal.

I debugged and Xen and QEMU to try to get the two components to mesh together.

Managed to get custom device model QEMU to talk to Python using UDS Datagram socket. Device model in C to Python user mode program running as UDS server.

Term 1, Week 10

Continued learning C and developing QEMU to provide access to disk activity metadata. I learnt about build systems, GCC, the linker, diff and patch. I had to use BB archives to track down historical QEMU changes.

Spent considerable time debugging QEMU log writes. I identified methods in the block device driver which are responsible for disk activity: *bdrv_aio_writev* and *bdrv_co_writev_em*.

I need to modify the server so that the format of the messages is more clear. All messages are 16 bytes which makes sense since two 64 bit numbers and being sent.

I met with my supervisor and went through presentation and report techniques and we discussed my solution so far.

It would be very interesting to collect data on the frequency, number and size of disk writes to see if this could be useful too in identifying malware or system intrusion.

I considered if this technique might this have some use in defending against 'Crypto Ransomware', perhaps the encryption of files on disk could be identified and prevented.

I spent some time modifying QEMU so that it recreates the UDS connection if it drops, I need to put the function or at least its declaration inside a C header. I spent time broadening my knowledge of the C programming language including include guards.

Term 1, Week 11

This final week of term was predominantly spent writing the 5000 word technical report in the style of a journal article, which I based around an IEEE Enterprise Distributed Object Computing article. I also had to prepare and give a 15 minute presentation summarising my progress so far.

I liaised with some industry experts from HP via my supervisor and I also spent some time debugging and preparing my solution which currently outputs the offset and length of disk writes performed by guests. This has been achieved by modifying QEMU 2.1 to write to a UDS every time a guest VM writes to its disk drive.

Term 2, Week 1

This week I mainly attended to other project deadlines, however I re-read the literature review submitted last term so that work can begin promptly and efficiently next week.

Term 2, Week 2

I read-into the next step of raw file system data extraction which is necessary to locate and scan the files which are being modified by the guest OS. I learnt about B+ trees which are used by NTFS to maintain its directory hierarchy.

I found and began studying from a technical document titled 'NTFS Forensics: A programmers View of Raw Filesystem Extraction'. The document is written using Windows API code examples. I am creating similar methods as part of a POSIX C program to accomplish the task of NTFS file extraction.

Work on this component is hosted at `https://github.com/hicksc/RawNTFSFileExtraction`

Term 2, Week 3

Continued to work on extracting NTFS information from a volume.

I found the following website helpful in this endeavour:
`http://www.cse.scu.edu/~tschwarz/coen252_07Fall/Lectures/NTFS.html`

Term 2, Week 4

I met with tutor to discuss progress and objectives. I continued to develop my raw NTFS extraction engine which will join in union with modified QEMU to extract files written by the guest OS into the Virtual Machine Monitor.

Term 2, Week 5

I continued to develop my raw NTFS extraction engine which will join in union with modified qemu to extract files written by the guest OS.

Extraction of resident NTFS MFT attributes is now working (i.e. files that areless than 900B and fit within an MFT entry are extractable).

I learnt about data-runs which are intervals of clusters on the disk which contain non-resident attribute data. I found a great resource for understanding the intricacies of NTFS:
`http://www.scribd.com/doc/2187280/NTFS-Documentation`

Term 2, Week 6

I met with my supervisor and Keith Harrison (from HP) to discuss progress and scope. My end product is likely to be a system which displays the folder/file structure of the NTFS drive and also shows the file writes and lengths which are happening, and points towards how the two can be pulled together.

I spent time debugging the parsing of non-resident file attributes, as well as the reporting of meta data via UDS.

I delved deeper into the inner workings of NTFS.

Term 2, Week 7

I developed software which parses cluster runs (run lists) into a linked-list solution so that they can be iterated through and the file or attribute data extracted.

Term 2, Week 8

My main objective is to extract the $DATA attribute from the $MFT file record. Method:

1. Backup blk_offset which contains the current file pointer.
2. Move file pointer to the first data run for the $MFT data attribute.
   For the first run this is relative sector offset + Virtual Cluster Numbe (VCN).
   Subsequent data runs are relative to the preceding run, i.e. previous VCN + VCN.

3. Write each data run to file for offline analysis
4. Restore blk_offset

My code now extracts $MFT records from all NTFS partitions which appear in the primary partition table. I can now extract file names from arbitrary MFT file records, a complication is that most records contain more than one file name. There is usually a shortened 8 character file name alongside the full name. I can identify and recover deleted files by paying attention to the STANDARD_INFORMATION attribute of files.

I create an architecture which allows searching through the MFT files in an offline capacity, indexed by either file name, MFT record number or file record location on disk.

Term 2, Week 9

I developed my NTFS Raw extractions engine/driver to the stage that it can identify and search for files on an NTFS volume using either the file name, the MFT record number, or the offset on the disk in bytes. It it possible to extract file content from the volume too, although not as well developed as the search and locate features.

Returning my attention to QEMU I need to join the two components together. The disk offsets given by QEMU do not match the disk offsets calculated by my driver. I now need to work out the differences and get everything lined up.

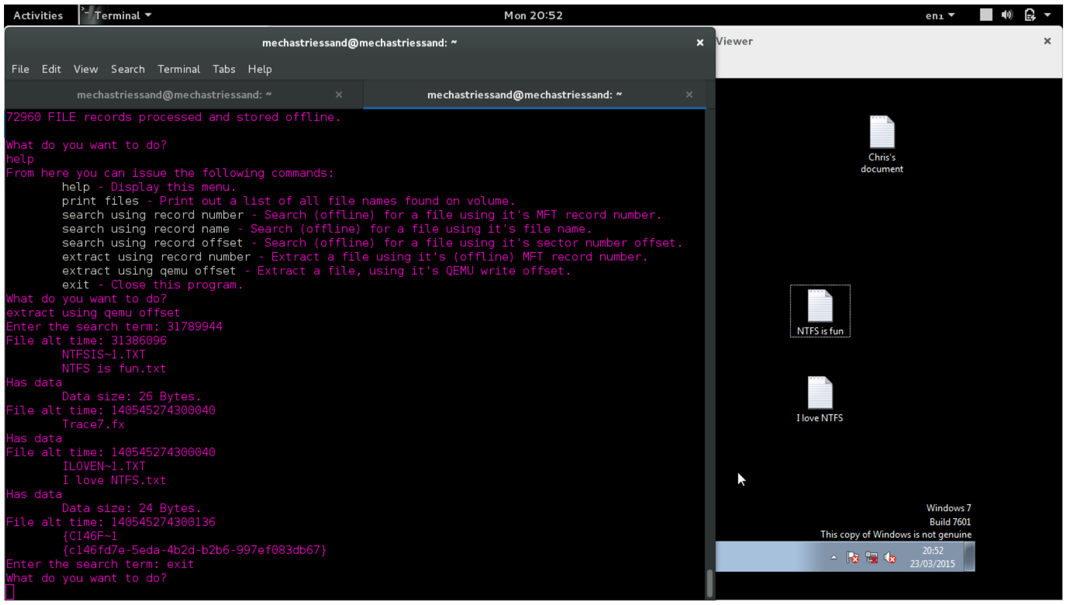
I observed in excess of 2500 disk write operations occurring when simply starting up a Windows 7 guest VM, creating a file, and shutting down again. 95% of these reads are for small files less than 512 Bytes (one sector) in size. I have started analysing the source code for QEMU and made a fork to my own github: `https://github.com/hicksc/qemu`

The important component of the source for accessing disk reads and wirtes is *raw-posix.c*, specifically the functions *raw_aio_readv* and *raw_aio_writev* expose the sector number and offset of the disk activity of the guest. The sector number is a signed 64 bit int, the length is a signed 32 bit int.

Term 2, Week 10

Continued development. Have aligned my driver to the QEMU writes, although they are to the nearest cluster, so it's possible for up to four files to be identified by a single write operation.

i.e. Where I'm writing that 'Chris's third document.txt' is at disk offset 6635910, it's actually being written to from 6635904 which is the bottom of the cluster, it's the fourth record in that cluster i.e. read from cluster 11646.75 to 11647. MFT records are two sectors long, 8 sectors to a cluster. Therefore there are potentially records at sectors 6635904 6635906 6635908 6635910.


Activities
Terminal
Mon 20:52
mechastriessand@mechastriessand: ~
File Edit View Search Terminal Tabs Help
Viewer
Chris's document
NTFS is fun
I love NTFS
Windows 7
Build 7601
This copy of Windows is not genuine
20:52
23/03/2015

```
72960 FILE records processed and stored offline.

What do you want to do?
help
From here you can issue the following commands:
        help - Display this menu.
        print files - Print out a list of all file names found on volume.
        search using record number - Search (offline) for a file using it's MFT record number.
        search using record name - Search (offline) for a file using it's file name.
        search using record offset - Search (offline) for a file using it's sector number offset.
        extract using record number - Extract a file using it's (offline) MFT record number.
        extract using qemu offset - Extract a file, using it's QEMU write offset.
        exit - Close this program.
What do you want to do?
extract using qemu offset
Enter the search term: 31789944
File alt time: 31386096
        NTFSIS~1.TXT
        NTFS is fun.txt
Has data
        Data size: 26 Bytes.
File alt time: 140545274300040
        Trace7.fx
Has data
File alt time: 140545274300040
        ILOVEN~1.TXT
        I love NTFS.txt
Has data
        Data size: 24 Bytes.
File alt time: 140545274300136
        {C146F~1
        {c146fd7e-5eda-4b2d-b2b6-997ef083db67}
Enter the search term: exit
What do you want to do?
```

Figure C.2: Shown here are two files on the desktop of a Windows 7 virtual machine using an NTFS partition for it's non-volatile storage. My driver has located the files using the write cluster offset on disk recovered from QEMU and is showing the file alteration time and size. I am able to recover the contents of the files into the VMM host file system.

File times are stored as a 64-bit value which represents the time which has elapsed since 12:00 AM 01/01/1601 (UTC). One idea which I have is to use the file time to help decide which files should be extracted and checked for a given file write, since there are up to four for each offset recoverable.

I met my supervisor and Keith Harrison to discuss progress and scope for my demonstration and thesis.

Term 2, Week 11

I introduced threads to my architecture and a producer consumer design pattern to facilitate the buffering and processing of disk write metadata as it's received from QEMU. I ported my original Python UDS server to C and incorporated it into my raw NTFS extraction program. I run both the server and client components in their own threads which enable the connection to be lost and recreated without affecting the rest of the architecture.

I spent extensive time creating and working through bug lists for my code, these were created by performing white box user testing on my code.

I eliminated memory leaks and race hazards from my code using the memcheck and drd tools from the valgrind suite.

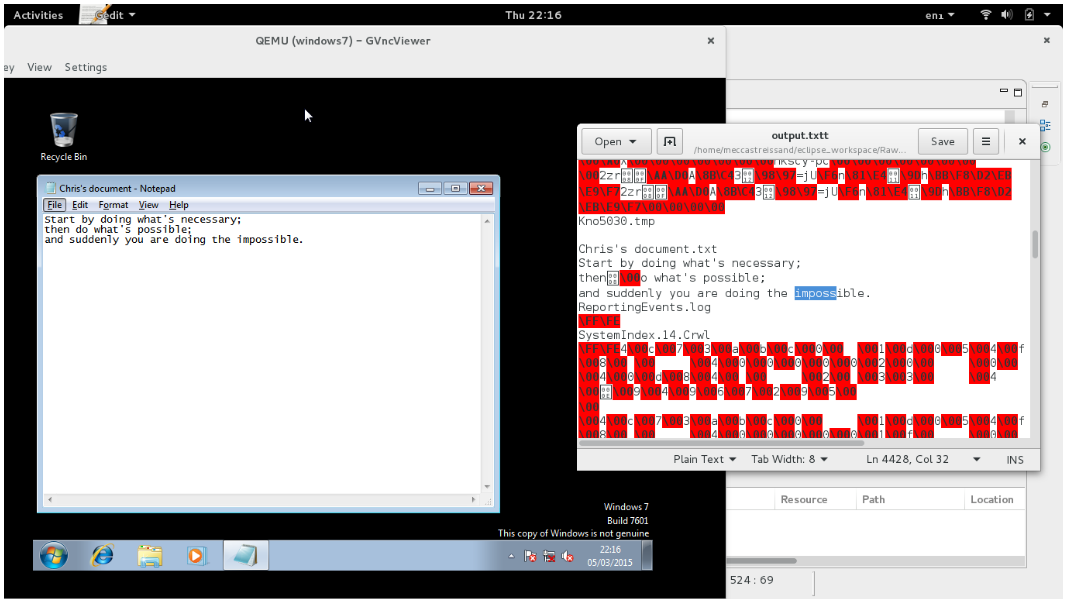
I wrote C unit tests for my code by utilising the check unit test framework.

I set up the CMake build system which automatically creates make files for the target platform the project is compiled upon.

I measured that on average there are approximately 70,500 disk reads and 2500 disk writes just booting up and shutting down a standard Windows 7 install.

I measured that the time taken for a file to be extracted to the VMM after being written to file by a guest is up to 30 seconds. This is attributable to the thread sleeping mechanisms used when the consumer thread is empty, and to the caching of disk writes which Windows 7 performs.

I wrote a shell script program which takes a file name as a parameter, and then calls a ccCheck.py Python program passing in the file name parameter. This checks the file to see if it contains any credit card numbers. The shell script then removes the file.



```
Start by doing what's necessary;
then do what's possible;
and suddenly you are doing the impossible.
```

```
\00\A0x\00\00\00\00\00\00\00nkscy-pc\00\00\00\00\00\00\00
\002zr□□\AA\D0A\8B\C43□\98\97=jU\F6n\81\E4□\9Dh\BB\F8\D2\EB
\E9\F72zr□□\AA\D0A\8B\C43□\98\97=jU\F6n\81\E4□\9Dh\BB\F8\D2
\EB\E9\F7\00\00\00\00
Kno5030.tmp

Chris's document.txt
Start by doing what's necessary;
then□\00o what's possible;
and suddenly you are doing the impossible.
ReportingEvents.log
\FF\FE
SystemIndex.14.Crwl
\FF\FE4\00c\007\003\00a\00b\00c\000\00  \001\00d\000\005\004\00f
\008\00 \00     \004\000\000\000\000\000\002\000\00    \000\00
\004\000\00d\008\004\00 \00    \002\00 \003\003\00     \004
\00□\009\004\009\006\007\002\009\005\00
\00
\004\00c\007\003\00a\00b\00c\000\00    \001\00d\000\005\004\00f
\008\00 \00     \004\000\000\000\000\000\001\00f\00    \000\00
```

Figure C.1: Extraction of a resident file stored on a guest VM, from the MFT of the guest OS hard disk.

Declaration

The material contained within this thesis has not previously been submitted for a degree at the University of Birmingham or any other university. The research reported within this thesis has been conducted by the author unless indicated otherwise.

Signed ....