Windows NT pagefile.sys Virtual Memory Analysis

Michael Gruhn
Department Computer Science, IT Security Infrastructures
Friedrich-Alexander University Erlangen-Nürnberg
Erlangen, Germany
michael.gruhn@cs.fau.de

Abstract

As hard disk encryption, RAM disks, persistent data avoidance technology and memory resident malware become more widespread, memory analysis becomes more important. In order to provide more virtual memory than is actually physical present on a system, an operating system may transfer frames of memory to a pagefile on persistent storage. Current memory analysis software does not incorporate such pagefiles and thus misses important information. We therefore present a detailed analysis of Windows NT paging. We use dynamic gray-box analysis, in which we place known data into virtual memory and examine where it is mapped to, in either the physical memory or the pagefile, and cross-reference these findings with the Windows NT Research Kernel source code. We demonstrate how to decode the non-present page table entries, and accurately reconstruct the *complete* virtual memory space, *including* non-present memory pages on Windows NT systems using 32-bit, PAE or IA32e paging. Our analysis approach can be used to analyze other operating systems as well.

Keywords

Digital Forensics; Virtual Memory Analysis; Pagefile Analysis; Windows NT Paging

I. Introduction

With the increased usage of hard disk encryption, the employment of RAM disks, and persistence data avoidance technologies such as private browsing modes, memory analysis becomes more and more important to the computer forensic process. In some cases, such as memory resident malware, or the already mentioned private browsing modes, the volatile memory can become the only source of information in an investigation.

Because the physical memory of a computer system is limited and far smaller than the available persistent storage modern operating systems provide virtual memory. In a virtual memory environment virtual memory addresses are translated to physical memory addresses via an address translation process. To provide more virtual memory as is physically present in form of RAM modules, an operating system may *swap* parts of the virtual memory *out* to persistent storage. Hence, data in the virtual address space may not always be present in the physical main memory of a computer system, but only in the persistent storage of that system. The location of the virtual memory on the persistent storage is known as the pagefile.

A. Motivation

Previous research has already found the pagefile to be of forensic value. Software such as browsers can leave evidence in the pagefile [1]. However, current analysis methods are not adequate. Currently the pagefile is treated as a miscellaneous date file, without considering each non-paged memory page's position in the virtual address space. This approach can still yield valuable information, e.g. by running a keyword search over all non-paged memory pages in the pagefile [2, 4.1 Mozilla Firefox][3, IV. RESULTS]. Searching the pagefile via a filecarver can also successfully discover files [4, 3 Pagefile Analysis: Results and Issues].

However, without context the forensic value of these findings is diminished. One example is a multi-user system. In which case, finding data of interest in the pagefile may not be enough to be helpful to the case if it can not be attributed to a particular user or process within the complete system. Further a lot of data of interest is simply lost because it can not be adequately reconstructed without putting the non-paged memory pages into context, e.g. process structures or picture files.

In general complex data structures spanning multiple memory pages require the paging relation to be reconstructed accurately and with certainty. Additionally to reconstruct also non-paged memory data from the pagefile must be incorporated into the analysis. This is exactly what our work fixes. It provides the paging context of the non-paged memory pages within the pagefile of Windows NT systems to current memory analysis methods.

B. Related Work

Currently memory analysis is already established and there exist a vast amount of works dealing with the acquisition and analysis of paged memory [5]. However, only few [6] consider non-paged virtual memory. Related work exists for the extraction of the pagefile.sys [4]. Many works mention extracting data from the pagefile.sys via crude methods [2].

However, such methods are at the borderline of forensic sanity, because some rely on knowing specific keywords before hand or extract data chunks without putting them into the context of the rest of the system processes.

The incorporation of the pagefile into the memory analyzing process was proposed [7] and preliminary information has been published regarding the connection between the pagefile and non-present pages in the page table of Windows NT system [6]. This knowledge was further supplemented by information about mapped files [8] and the Windows Virtual Address Descriptor (VAD) tree [9].

Volatility, the leading project in open source memory analysis, strives to add pagefile support as part of their project road map[1], but it is not implemented yet. Rekall, a project forked from the Volatility "scudette" branch known as the "Technology Preview" branch, currently developed by Google, has very recently, during preparation of this work, added preliminary code to their 1.2.1 release[2], in preparation for pagefile analysis. However, no publication nor information about this implementation, nor any evaluation with regard to its correctness exists.

C. *Contribution*

Our contributions to the field of memory forensics can be summarized as follows:

- Previous publications regarding Windows NT pagefile analysis are already aging, and therefore partially out dated. They are also often incomplete. We update and summarize the knowledge about Windows NT pagefile analysis.
- We further provide a grey-box analysis method to verify virtual address translation implementations. This method can be used to verify the correctness of virtual memory analysis software as well as analyze virtual address translation mappings of unknown systems. We evaluate the practicability of our method by analysing the partially known Windows NT paging behaviour, with a distinct focus on the pagefile.
- We provide the source code of our prototype tools for virtual address space reconstruction of Windows NT versions 7, 8.1 and 10 for both x86 systems, with 32-Bit or PAE paging, as well as x64 systems, with IA32e paging.
- With this work, we, last but not least, provide a reference for Windows NT virtual address translation, by summarizing our verification of previous research updated with our own findings.

The source code and documentation of our research can be accessed at https://www1.cs.fau.de/virma.

D. *Outline*

This paper is structured as follows: In section II we present our grey-box analysis scheme and the tools we used to conduct this analysis. In section III we provide an overview over paging on Windows NT systems, the content of which was ascertained and verified via our grey-box analysis scheme. In section IV we give a brief overview over current memory and pagefile acquisition techniques for Windows NT. And in section V we give a brief overview over memory analysis that can be performed on the virtual address space as reconstructed by our method. Finally in section VI we evaluate the new combined virtual memory analysis technique incorporating the pagefile. To this end, we compare it to current analysis methods. Last but not least, we conclude our work in section VII.

II. Grey-Box Virtual Address Translation Analysis

Even though, work related to pagefile incorporation into Windows NT virtual memory analysis exists, as already outlined in subsection I-B, we developed a grey-box virtual address translation analysis method, which can be used to verify, disprove, and/or updated current knowledge. This grey-box scheme can further be used to analyse yet unknown systems.

A. *Scheme*

The virtual address translation analysis scheme targets address translation via paging. However, it can be adopted to other translation methods, such as segmentation. The basic scheme can be seen in Figure 1. The components consist of the "Virtual Memory", the "Physical Memory", the "Pagefile" and the "Virtual Address Translation".

The "Virtual Address Translation" component is the target of the analysis. The goal is to infer knowledge about it. To this end, the "Virtual Memory" is filled with known pages. We use sequential numbers to fill the pages. We call these known pages *crib* pages (cf. word usage of crib in cryptology). The number pattern within each page allows us to uniquely identify each page and to find the corresponding physical frame in either the "Physical Memory" component or the "Pagefile".

The available knowledge about the basic workings of the "Virtual Address Translation" and the availability of at least the binary code of the operating system, residing in physical memory, makes this a grey-box analysis. The goal of this analysis is to provide for any given virtual page address the corresponding physical frame address or physical frame offset, if the physical frame is in the pagefile.

[1]https://code.google.com/p/volatility/wiki/VolatilityRoadmap
[2]https://github.com/google/rekall/releases/tag/v1.2.1

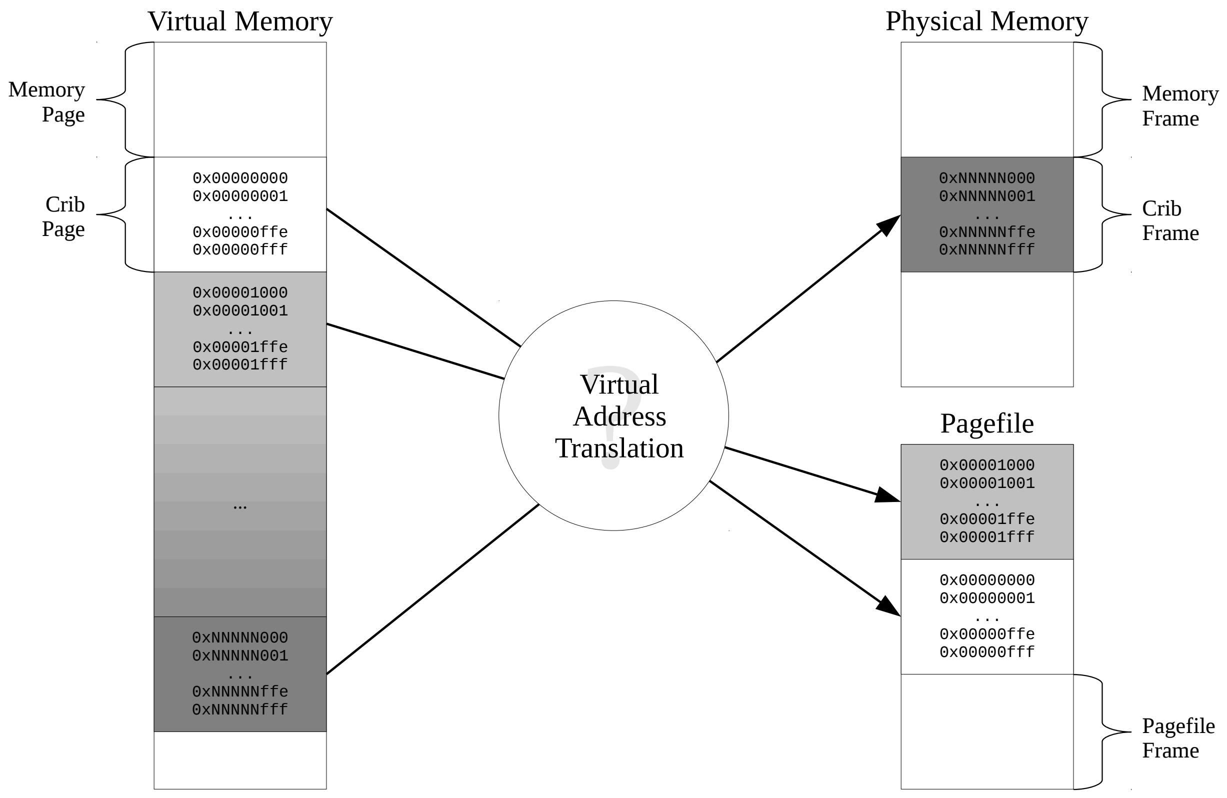


Figure 1: Grey-Box Virtual Address Translation Analysis Scheme

B. *Test Data Generation*

The data needed for our grey-box analysis is created by our `ramwrite` tool. It is platform independent, portable and only relies on ISO C89 features. `ramwrite` allocates a specific amount of memory via `malloc()`. It allocates one page more than is requested. This way the crib data can be aligned to a page boundary. The memory space from the address returned by `malloc` to the next page boundary is filled with `0xc001babe` to distinguish it from the beginning of the crib data and also potentially from any other data.

The `ramwrite` process stays open until user input is provided. This way the memory allocation can persist as long as needed. `ramwrite` supports different patterns it can write. The most useful are:

`addr`: Writes a sequence of 32-bit numbers into the allocated space, starting with 0 at the start of the allocation.
`zero`: Overwrites the allocation with zero bits; useful for "cleaning" the memory before tests.
`one`: Like zero, but writes all one bits.
`deadbeef`: Fills the allocation with the 32-bit value `0xdeadbeef`; useful as a second distinguishable pattern.

`ramwrite` is duplicated and renamed `swapforcer`. `swapforcer` is used to force the memory allocation of `ramwrite` out of physical memory into the pagefile by making a large allocation. `swapforcer` fills its memory allocation with `0xdeadbeef`. Depending on how much of the `ramwrite` memory content should be written to the pagefile, `swapforcer` can also allocate the complete physical memory.

With these two programs the needed crib data can be placed into physical memory and the pagefile as follows:

- `ramwrite` is executed and fills its memory allocation with sequential 32-bit numbers (`addr` pattern).
- `swapforcer` is executed and fills its memory allocation with a distinguishable pattern (`0xdeadbeef` pattern).
- By varying the size of the memory allocation by the `swapforcer`, the crib data distribution between physical memory

and pagefile can be controlled. The more memory swapforcer allocates, the more frames of the ramwrite process are swapped into the pagefile.

C. Inferring the Virtual Address Translation

Once ramwrite and swapforcer are executed on a system, the physical memory and pagefile can be acquired. In section IV we will go into detail how this is can be done on Windows NT. The analysis then consist of

1) figuring out the mapping of virtual crib page addresses to physical crib frame addresses and offsets, and
2) inferring how the operating system "remembers" this mapping.

The first is trivial. A simple program find_addr <offset> <addr> was created that searches the input at the specified offset <offset> for a crib page starting with the 32-bit value given by the <addr> parameter.

The second step is more complicated and has, in some parts, to be done via manual reverse engineering:

1) The hardware dependent address translation must be interpreted.
2) The software dependent address translation must be interpreted.

The hardware dependent address translation is specified by the hardware manufacturer. Address translation is done via paging tables. For this the operating system has to store the base address of the root table somewhere. In subsection V-A we will detail how this base address can be found on Windows NT systems. Once this base address is extracted the paging table tree can be traversed.

After the hardware dependent address translation part is implemented in the memory analysis process, it is recommended to test it with memory dumps of the ramwrite program *without* any crib data being forced into the pagefile. The memory analysis process should be able to perfectly reconstruct the crib data, i.e., bring each crib frame into correct order.

For the software dependent part a trial-and-error process is used. We currently assume the correlation between virtual addresses and offset in the pagefile can be inferred from the page table entry itself. At least this is true for Windows NT and Linux (cf. swp_entry_t and pte_to_swp_entry() in the Linux source code[3]). If a different system does not use the page table entries to store this information, the relevant data structure must be found within the operating system data structures. Here the location, i.e., addresses and offsets, of the crib pages and crib frames can be used to verify found data structures. Even though, theoretically an exhaustive search for crib frame addresses and offsets could be used the amount of possible candidates for operating system data structures could be overwhelming. Our findings so far, however, indicate that the crib frame offset into the pagefile, can be determined, by appropriately shifting and masking the page table entry value.

III. Windows NT x86 and x64 Virtual Memory Overview

In this section we give an overview over virtual memory translation for Windows NT x86 and x64 with 32 Bit, PAE and IA32e Paging. The information herein was derived via the techniques described in the previous section and was further extended and verified via cross-referencing the Windows NT Research Kernel source code [10].

A. pagefile.sys

Before the paging overview we briefly explain the pagefile of Windows NT. The default pagefile on Windows NT systems is called pagefile.sys. On default it is stored in the root directory of the file system the operating system is installed on, i.e., %SystemRoot%\pagefile.sys. However, up to 16 pagefiles can exist. Their name, location, minimum and maximum size in MiB are specified in the registry strings stored in the following entry:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles
```

With each string entry having the following form:

```
<path to pagefile> <minimum size in MiB> <maximum size in MiB>
```

The pagefiles are numbered, starting from 0 to 15 beginning with the first entry in the PagingFiles registry entry. At runtime the Windows NT kernel keeps a _MMPAGING_FILE structure for each pagefile. The most relevant fields are

```
lkd> dt nt!_MMPAGING_FILE
   +0x018 File             : Ptr32 _FILE_OBJECT
   +0x024 PageFileName     : _UNICODE_STRING
   +0x040 PageFileNumber   : Pos 0, 4 Bits
   +0x044 FileHandle       : Ptr32 Void
```

[3]https://www.kernel.org/

with the offsets referring to a Windows 7 x86 kernel version 7600. This way the pagefile number and corresponding path and name of the file can be derived from the physical memory of a system. However, in order to be of any use, the corresponding pagefiles must have been acquired alongside the physical memory, at which point the pagefile numbers, paths and names should already be known to the analyst who acquired the pagefiles in the first place. In any case multiple pagefiles are rare, because they must be explicitly configured and are, to our knowledge, not generated by Windows NT on default settings.

On a live system the pagefile itself is locked by the kernel. However, in subsection IV-B we will detail how the file can still be acquired on a live system.

The pagefile's content is unstructured. Raw memory frames are stored in the file as if it was physical memory. Each frame is referenced by an offset. The next section explains the address translation between virtual memory page addresses and the pagefile frame offsets.

B. Page Table Entries

Windows NT systems running on x86 based hardware will use PAE Paging [11, 4.4. PAE PAGING] if available. If it is not available they will use 32 Bit Paging [11, 4.3. 32 BIT PAGING]. Windows NT systems running on x64 based hardware will use IA32e Paging [11, 4.5. IA32e PAGING].

In the following sections we will give an overview over the page table entries (PTEs). PTEs can be divided into two categories: hardware PTEs and software PTEs. Hardware PTEs are all PTEs which have the present bit set. If this bit is set the interpretation of the values within the PTE is strictly done by hardware. In case the PTEs are marked as invalid, they are software PTEs. Software PTEs are interpreted strictly by software. In this case the Windows NT kernel. Later we will go into detail how Windows NT interprets the PTEs and translates the virtual addresses. We will now give an overview of the bit layout of possible PTEs.

1) 32 Bit Paging:

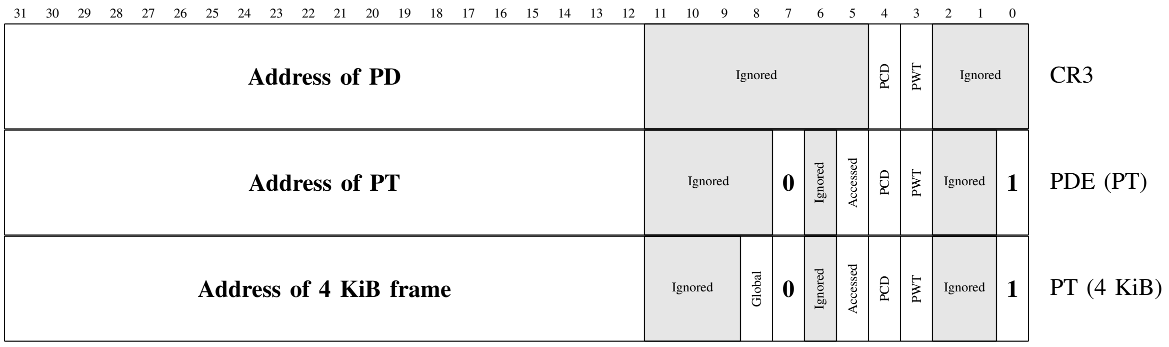


Figure 2: 32-Bit Paging PTE structures

Figure 2 lists the hardware PTEs that we encountered during our analysis of Windows NT running on x86 based hardware without PAE Paging present. Their definition is given strictly as per the Intel specification [11, Figure 4-4.]. Ignored fields are grayed out. They are not interpreted by the hardware and can thus be considered software fields. The bits at offset 12 to 31 give the 4 KiB address of either further tables or the start of the physical memory frame.

Windows NT does not seem to use 2 MiB frames. We neither encountered such frames during our analysis nor makes the Windows Research Kernel source code any references to 2 MiB frames for 32 Bit Paging [10, `mi386.h`].

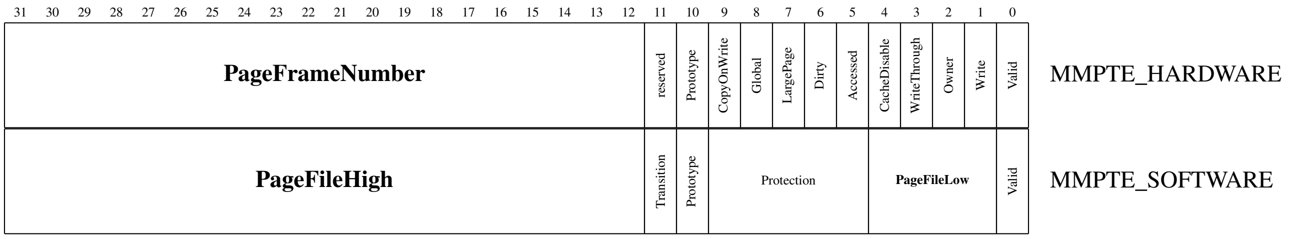


Figure 3: Windows NT 32 Bit Paging structures

Figure 3 lists the software PTEs of Windows NT with regard to 32 Bit Paging. The definition of the MMPTE_SOFTWARE PTE was taken from the Windows Research Kernel source code [10, `mi386.h:2446`] as was the MMPTE_HARDWARE PTE definition [10, `mi386.h:2508`].

The PageFileLow field gives the pagefile number. As discussed earlier there can be up to 16 pagefiles. The PageFileHigh field gives the offset into the pagefile. Like the hardware PTEs this offset is a 4 KiB offset. Hence the byte offset can be obtained by simply masking the last 12 bits of the PTE. Not only memory frames can reside in the pagefile, but also PTEs themselves can reside in the pagefile. These findings are the same as presented in other research [6, Figure 3].

2) PAE Paging:

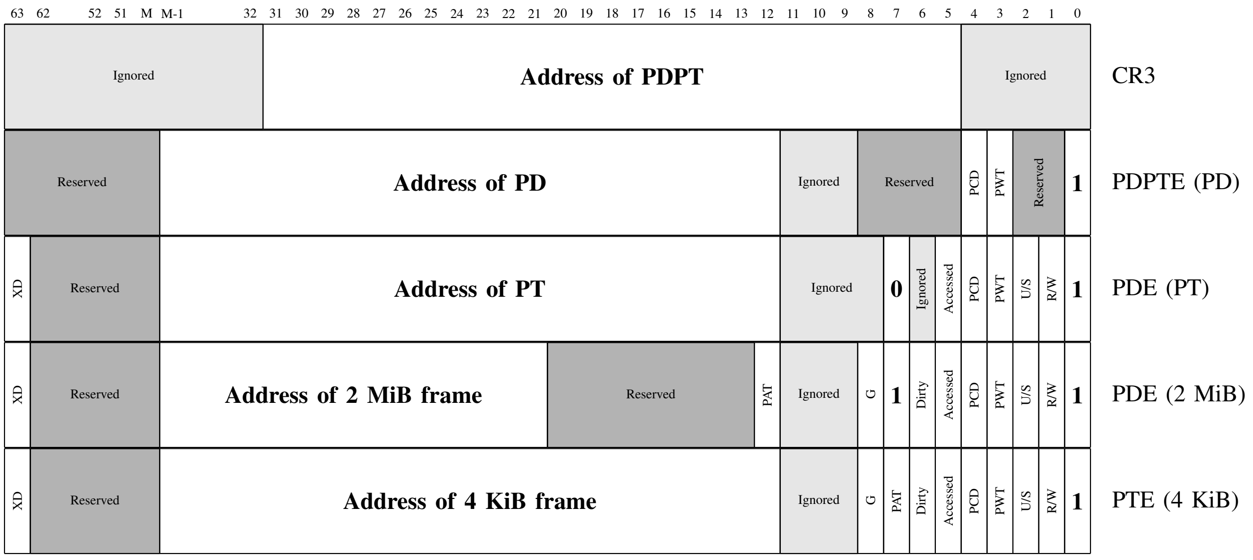


Figure 4: x86 PAE Paging structures

Figure 4 lists the hardware PTEs that we encountered during our analysis of Windows NT running on x86 based hardware with PAE Paging enabled. Their definition is given strictly as per the Intel specification [11, Figure 4-7.]. Ignored fields are again grayed out and can be considered software fields. Further we also grayed out reserved fields, which according to Intel's specification have to be zeroed [11, p. 4-18]. For all but the Page Directory Entry pointing to a 2 MiB frame (PDE (2 MiB)) the bits at offset 12 to 62 give the 4 KiB address of either a further table or the start of the physical memory frame. For the PDE (2 MiB) bits 21 to 62 give the 2 MiB address of the physical 2 MiB frame.

Again Windows NT does not seem to use 2 MiB frames, at least we did not encountered such frames during our analysis. However, the Windows Research Kernel source code references 2 MiB frames for PAE and IA32e Paging [10, `miamd.h:2685`]. The software PTEs for PAE Paging seem to be virtual identical to the IA32e ones, which we detail in the next section.

3) IA32e Paging:

Figure 5 lists the hardware PTEs that we encountered during our analysis of Windows NT running on x64 based hardware using IA32e Paging. Their definition is given strictly as per the Intel specification [11, Figure 4-11.]. The IA32e PTEs are virtually identical to the PAE PTEs. The only difference is that bits at offset 62 to 52 are ignored.

Again Windows NT does not seem to utilize neither the 1 GiB nor the 2 MiB frames. As state before, however, code for 2 MiB frames is defined in the Windows Research Kernel as the MMPTE_HARDWARE_LARGEPAGE software PTE structure [10, `miamd.h:2685`].

Figure 6 lists the software PTEs of Windows NT with regard to IA32e Paging and, as already explained above, PAE Paging. The definition of the MMPTE_SOFTWARE, MMPTE_HARDWARE and MMPTE_HARDWARE_LARGEPAGE PTE was taken from the Windows Research Kernel source code [10, `miamd.h`].

As before the PageFileLow field gives the pagefile number. The PageFileHigh field gives the offset into the pagefile. Like the hardware PTEs this offset is a 4 KiB offset. Hence the byte offset can be obtained by shifting the value of the PTE by 21 bits and then simply masking the last 12 bits. As before not only memory frames can reside in the pagefile, but also PTEs themselve.

C. Virtual Address Translation

After the overview of available PTEs we now explain how they are used in Windows NT to translate virtual to physical addresses.

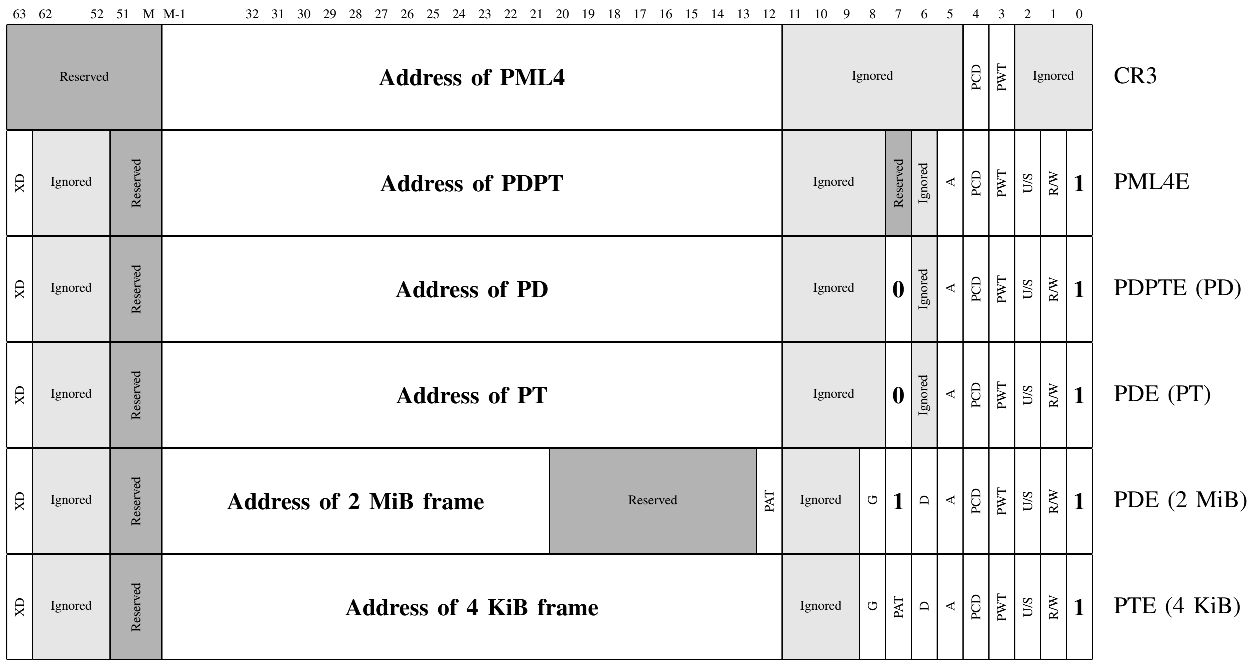


Figure 5: x64 IA32e paging structures

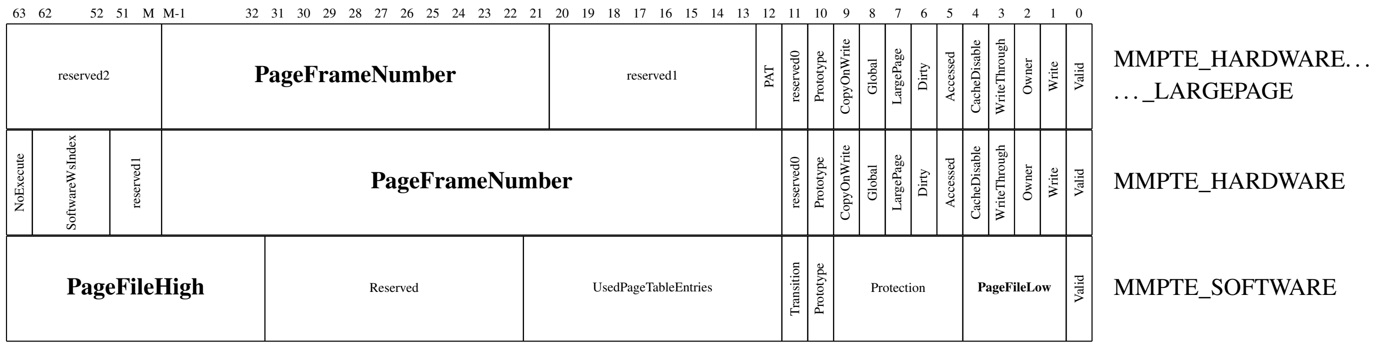


Figure 6: Windows NT x64 paging structures

1) Hardware: All hardware PTEs, i.e., PTEs that have the valid bit set, are interpreted by the hardware. For a reference on how this is done we recommend the Intel specification [11] which details the 32 Bit Paging [11, Figure 4-2.], PAE Paging [11, Figure 4-5.] and IA32e Paging [11, Figure 4-8.] address translation process. In any case, if the valid bit is set, the interpretation *must* strictly follow the hardware specification.

2) Software: The software PTEs can be divided into different types: Demand Zero, Pagefile, Transition, and Prototype PTEs. All Software PTEs are hardware and paging mode independent. Only their corresponding fields, namely PageFileHigh, may reside at different bit offsets within the PTE. However, they are interpreted the same for every paging mode, as the memory management code of the NT kernel responsible for these software PTEs is already abstracted from the hardware PTEs [10, `pagfault.c`].

Demand Zero:

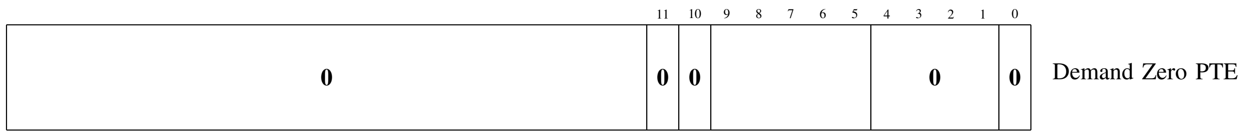


Figure 7: Windows NT Demand Zero PTE

A PTE with the Valid, Transition, Prototype, PageFileLow and PageFileHigh fields set to zero [10, `miamd.h:2385`][10, `mi386.h:2225`] is a so called Demand Zero PTE. It can be seen in Figure 7. Any request to this virtual address should be satisfied by a memory frame that is filled with zeros [10, `MiResolveDemandZeroFault()`].

Pagefile:

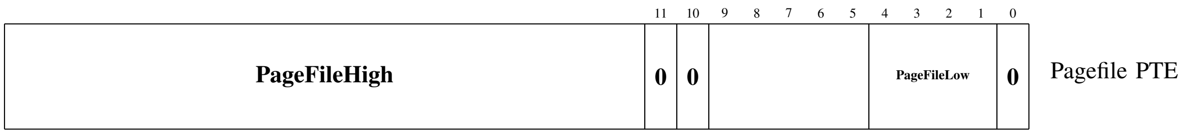


Figure 8: Windows NT Pagefile PTE

The Pagefile PTEs, as seen in Figure 8, or MMPTE_SOFTWARE PTEs, as they are referred to in the Windows Research Kernel source code, are the most interesting to this work. If the Valid, Prototype and Transition bits are zero and the PageFileHigh field is not zero the PTE references the pagefile [10, `MiResolvePageFileFault()`] given by PageFileLow. The offset into the pagefile is given by PageFileHigh. Pagefile PTEs can also reference other PTEs, in which case the PTE that is referenced is loaded from the pagefile. If a Pagefile PTE is encountered during analysis in place where a PDPTE (PDE), PDE (PT) or PTE (4 KiB) page table is expected the PageFileHigh field gives the offset to the relevant paging table structure within the pagefile. The index in this paging structure is taken from the virtual address bits the same way as for the hardware address translation. Note that because Windows NT does not use 1 GiB nor 2 MiB paging structures the 7th bit of the Pagefile PTE must be ignored. If bit 7 is one this does not indicate a large page. A PTE loaded from the pagefile can then be interpreted like a PTE loaded from a physical memory frame.

Transition:

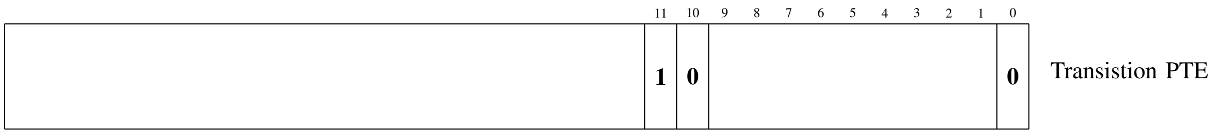


Figure 9: Windows NT Transition PTE

A Transition PTE, as seen in Figure 9, is a PTE with the Valid and Prototype field set to zero, but the Transition field set to one [10, `MiResolveTransitionFault()`]. This PTE is used to mark a page as being in transition, i.e., being in the process of being paged out to the pagefile. A PTE marked as being in transition, hence, still resides at the physical memory frame it points to. As with Pagefile PTEs also Transition PTEs can also reference other PTEs, which can be interpreted normally.

Prototype:

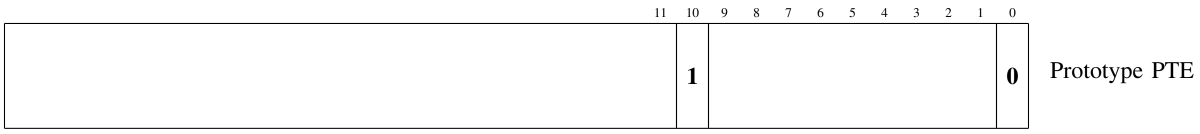


Figure 10: Windows NT Prototype PTE

Prototype PTEs, as depicted in Figure 10, are used to facilitate shared memory and mapped files [10, `MiResolve-MappedFileFault()`]. They have the Valid bit set to zero and the Prototype bit set to one [10, `MiResolveProto-PteFault`]. Even though these Prototype PTEs also constitute parts of the virtual address space we currently do not resolve them as we focus on reconstructing the pagefile.

IV. Acquisition

In this section we provide a brief overview over the available acquisition methods with regard to physical memory and pagefile. We will do so by summarizing existing research. We start with physical memory acquisition.

A. Memory

While classical memory acquisition techniques could be divided into hardware- and software-based solutions this no longer holds true because many acquisition techniques use both hard- and software [5, 3. Acquisition of volatile memory]. We hence divide the acquisition methods into: DMA attacks, coldboot attacks, core dumps, software acquisition and virtualization.

1) DMA Attacks: Memory can be acquired via a DMA attack. This attack uses a system bus such as PCI [12], PCIe[4][5] or IEEE1394 [13] to perform direct memory access (DMA) on the target machine. Firewire is restricted to the lower 4 GiB of physical memory and requires a software driver on the target system to be present. PCI is not hot-plugable, making it a solution that needs to be pre-installed, i.e., the target system must be made forensic ready before memory can be acquired. Because DMA has to transfer individual memory pages while the system is running, which potentially changes the memory contents, the atomicity measure, as proposed by Vömel and Freiling [14], of this method is only moderate [5, Fig. 5].

2) Coldboot Attacks: The physical memory of a system can be acquired via a cold boot attack [15]. This provides a very high degree of atomicity [5, Fig. 5] and cold boot attacks do not depend on any traget operating system administrative privileges to run. However, they may need hardware administrative privileges, if only a simple reset attack, instead of a full blown memory transplantation attack [16, C. RAM Transplantation Attacks] should be performed. However, in 2013, researchers evaluated the practicability of the cold boot attack against DDR3 RAM modules [16] and found that DDR3 is not exploitable anymore, due to data scrambling preformed by the memory controller [17]. Hence, this acquisition method may not be available.

3) Crash Dumps: Crash dumps are a good way to obtain the system memory. They can be, appropriate system configuration provided, triggered via keyboard input[6] or by crashing the Windows NT kernel. As of writing, a bug in Windows 7 allows the system to be crashed via a GPT partitioned storage medium with the number of possible partition entries in the GPT header set to zero. This causes a division by zero in the kernel and, on default system configuration, a core dump is written to disk. Because the operating system is stopped during the crash dump procedure the method has high atomicity [5, Fig. 5].

The problem with this method, however, is the location on disk the core dump is written to. It is written to the pagefile, destroying it as possible information source for virtual memory analysis. Another problem is caused by encryption. If the filesystem is encrypted the crash dump used to be dumped to disk in cleartext. However, starting from Windows Vista with Service Pack 1 (SP1) and Windows Server 2008, a disk encryption software can implement a Crash Dump Filter Driver[7] which allows to encrypt the contents of crash dumps, as well as the hibernation file. Thus, rendering this method useless if software disk encryption is used.

4) Software: A very popular, because simple, method are software memory dump tools, such as: WinPMEM[8], DumpIt[9], Belkasoft Live RAM Caputer[10], Memoryze[11], or FTK Imager[12], to name a few. They allow to conveniently dump the physical memory to either a removable storage medium or transmitting it over the network.

The problem of software tools is their trustworthiness with regard to anti-forensics [18], their requirement for target operating system administrative privileges, and their low atomicity [5, Fig. 5].

5) Virtualization: One method we extensively applied during this work is virtualization. For this method the operating system is executed in a virtual machine. We used VirtualBox[13]. From VirtualBox the memory of a paused VM can be dumped via the `debugvm dumpguestcore` command. Because the virtual processor can be paused at any time, the atomicity of this method is very high [5, Fig. 5].

B. pagefile.sys

In this section we outline how the pagefile of Windows NT can be acquired either on a live system or via dead analysis.

[4]http://www.windowsscope.com/
[5]https://github.com/NSAPlayset/SLOTSCREAMER
[6]https://msdn.microsoft.com/en-us/library/windows/hardware/ff545499%28v=vs.85%29.aspx
[7]https://msdn.microsoft.com/en-us/library/windows/hardware/ff552475%28v=vs.85%29.aspx
[8]http://www.rekall-forensic.com/downloads.html
[9]http://www.moonsols.com/windows-memory-toolkit/
[10]http://forensic.belkasoft.com/en/ram-capturer
[11]https://www.mandiant.com/resources/download/memoryze
[12]http://accessdata.com/support/adownloads#FTKImager
[13]https://www.virtualbox.org/

1) Dead Acquisition: Acquiring the pagefile.sys via dead acquisition is trivial. If no disk encryption is used we can simply shut down the computer or even just unplug the storage device the pagefile.sys is stored on from the target system. Next we can mount the filesystem on the storage device and copy the pagefile.sys.

In case disk encryption is used we must first obtain the encryption keys via either a cold boot, DMA attack or any other available means. Note that crash dumps can not be used for key recovery as outlined in subsubsection IV-A3. Also key recovery via software executed on the target system should be avoided, because in such cases the pagefile.sys can be acquired live, as explained in the next section. The recovered encryption keys can be used to circumvent the disk encryption [15][19]. In case self encrypting disks (SEDs) are used a warm-replug attack [20] can be used. In case, disk encryption is used, and a DMA attack is not available, or SEDs are used, care must be taken to not loose the pagefile.sys, because neither the warm-replug attack on SEDs nor the cold boot attack are revertible. Hence, in such cases, a life acquisition is preferable, if available.

2) Live Acquisition: Acquiring the pagefile on a live system is more complicated because, as already outlined in subsection III-A, the Windwos NT pagefile is locked by the kernel against any ordinary access during runtime. However, the pagefile can be acquired from the Win32 Device Namespace, e.g. the extraction process of the pagefile.sys onto the removable drive `E:` via the ifind and icat tools from the Sleuthkit[14] is as follows

```
C:\>ifind.exe -n /pagefile.sys \\.\c:
11163
C:>icat.exe \\.\c: 11163 > E:\pagefile.sys
```

with 11163 being the $MFT entry number of the pagefile.sys.

Tools which are able to automatically copy the pagefile of a running system are: Disk Explorer[15], Forensic Toolkit[16], WinHex[17], Pagefile Collection Toolkit (PCT) [4], ntfscopy, icat[18], or FGET (Forenisc Get by HBGary Inc.)[19], to name a few.

V. Analysis

Once a physical memory dump and corresponding pagefile are acquired the process of reconstruction can begin. In order to reconstruct a virtual address space of a process the process structures, namely the Directory Table Base, i.e., the pointer to the process' page table root must be found. After this the virtual address space can be reconstructed by implementing the virtual address translations as outlined in subsection III-C. To this end, we implemented two kinds of tools. First a tool that finds the EPROCESS structures and extracting the DirectoryTableBase from them by carving the physcial memory dump. Second a tool that, given a physical memory dump, a pagefile copy and a DirectoryTableBase value, reconstructs the virtual address space defined by the paging structure.

A. Finding DirectoryTableBase

The address of the root table of the paging structure governing the virtual address space of a process is stored in the processes EPROCESS structure in a variable called `DirectoryTableBase`. To find this variable the EPROCESS structures of the various Windows NT kernel versions can be carved from physical memory via a signature. EPROCESS structures are, to our knowledge, not swapped out to the pagefile. We found the following eight fields[20] to be enough to build a robust signature

```
lkd> dt nt!_EPROCESS
   +0x000 Pcb                : _KPROCESS
          +0x000 Header            : _DISPATCHER_HEADER
                 +0x000 Type             : UChar
                 +0x002 Size             : UChar
                 +0x003 Reserved2        : Pos 2, 4 Bits
          +0x028 DirectoryTableBase : Uint8B
          +0x030 ThreadListHead   : _LIST_ENTRY
                 +0x000 Flink            : Ptr64 _LIST_ENTRY
                 +0x008 Blink            : Ptr64 _LIST_ENTRY
   +0x180 UniqueProcessId  : Ptr64 Void
   +0x2e0 ImageFileName    : [15] UChar
```

[14]http://www.sleuthkit.org/sleuthkit/
[15]https://www.runtime.org/diskexplorer.htm
[16]http://accessdata.com/solutions/digital-forensics/forensic-toolkit-ftk
[17]http://www.x-ways.net/winhex/index-d.html
[18]http://www.sleuthkit.org/sleuthkit/man/icat.html
[19]http://opensecurityresearch.com/files/FGET.zip
[20]The presented field names and output were obtained via the WinDbg kernel debugger.

with the following constraints

```
ThreadListHead.Flink >= KERNEL_ADDRESS
ThreadListHead.Blink >= KERNEL_ADDRESS
DirectoryTableBase != 0
DirectoryTableBase % DTB_ALIGMENT == 0
Type == 0x03
Size == SIZE
Flags == 0x00
Reserved2 == 0x00
IS_PRINTABLE_ASCII_STRING(ImageFileName)
```

where `IS_PRINTABLE_ASCII_STRING()` denotes a function that tests whether the string is composed of only printable ASCII characters. And with `KERNEL_ADDRESS` denoting the start of the kernel space. This is `0x80000000` for 32-bit and `0x80000000000` for 64-bit systems. Further `DTB_ALIGNMENT` is `0x1000`, except for PAE systems, there it is `0x20`. The value for `SIZE` is Windows NT version and build dependent. Size values for some systems are given in Table I.

Windows NT Version	`SIZE`
Windows 7 x86 7600	0x26
Windows 7 x64 7600	0x58
Windows 8.1 x86 9600	0x28
Windows 8.1 x64 9600	0xb2
Windows 10 x64 9841	0xb4

Table I: Size value for signature.

The offsets of the quoted fields can be obtained via WinDbg[21] by executing

```
.sympath srv*
.reload
dt nt!_EPROCESS
dt nt!_KPROCESS
dt nt!_DISPATCHER_HEADER
dt nt!_LIST_ENTRY
```

Please not that this signature may not be anti-forensic resistant, i.e., it may rely on values such as the `Size` field, which is not used by the operating system and hence can be overwritten with other values by a rootkit or malware [21]. We only used these signature for our evaluation.

B. Reconstructing the Virtual Address Space

Once the root of the paging table structure is found the virtual address space spanned by that paging structure can be reconstructed. To this end, the memory range from zero to the highest memory address can be iterated and any present physical frame can be written out to a new file in order to create a dump of the process' virtual memory space. The address translation is performed as per subsection III-C.

C. Analyzing the Virtual Address Space

Once the virtual address space is reconstructed current methods to analyze the now flat process memory space can be used. These methods include, but are not limited to, the following:

- File carving via Foremost[22]
- Keyword search in the process space via strings[23]
- Encryption key search via aeskeyfind [24] or Interrogate [25]
- Disassemble the process' code and data segments

While some of them may still not be very forensic savvy and accurate, such as file carving and strings, they can now be used to extract full files and texts from editors, emails, messengers or visited websites, even if such process memory was not paged or physically fragmented. Further can any findings by these tools now be linked to the process they were found in, which can be linked to the user owning the process. All of which was not possible previously.

[21]https://msdn.microsoft.com/en-us/windows/hardware/hh852365
[22]http://foremost.sourceforge.net/
[23]https://www.gnu.org/software/binutils/
[24]https://citp.princeton.edu/research/memory/code/
[25]https://github.com/carmaa/interrogate

VI. Evaluation

We developed, tested and verified the correct working of our virtual address space extraction tool against the following versions of Windows NT:

- Windows 7 7600 x86 (with 32 Bit Paging and/or PAE Paging)[26]
- Windows 7 7600 x64 (with IA32e Paging)[27]
- Windows 8.1 9600 x86 (with 32-Bit Paging and/or PAE Paging)
- Windows 8.1 9600 x64 (with IA32e Paging)
- Windows 10 9841 x86 (with PAE Paging)[28]
- Windows 10 9841 x64 (with IA32e Paging)[29]

A. Problem Cases of Virtual Memory Analysis

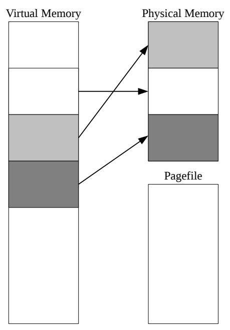
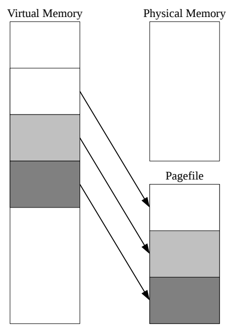
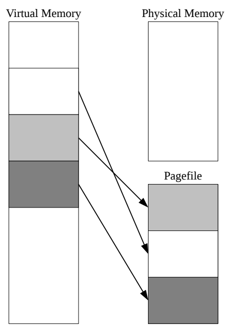
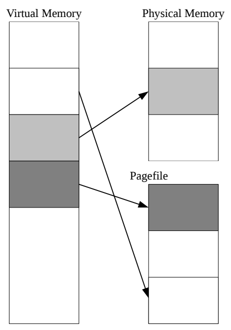
	Problem Case 1	Problem Case 2	Problem Case 3	Problem Case 4
				
Memory Carving	Maybe	No	No	Maybe
Pagefile Carving	No	**Yes**	Maybe	Maybe
Virtual Memory	**Yes**	No	No	Maybe
Virtual Memory + Pagefile	**Yes**	**Yes**	**Yes**	**Yes**

Table II: Problem cases of virtual memory analysis: Only one case can be perfectly solved by current memory analysis techniques.

For evaluation we consider four problem cases with regard to virtual memory analysis, as depicted in Table II:

1) Physical memory frames are fragmented but no frames are in the pagefile, as illustrated in Table II:
   - Naive carving of the physical memory may not yield any results due to the fragmentation.
   - Memory analysis only considering physical memory can complete reconstruct the virtual address space.
   - File carving applied to the pagefile can not yield any results, because no virtual page is mapped to a pagefile frame.
2) All virtual memory pages are sequentially mapped into the pagefile, as shown in Table II:
   - Any physical memory analysis will not yield any results, because all virtual pages are mapped to pagefile frames.
   - File carving applied to the pagefile will retrieve the content, because it is available sequentially.
3) Physical memory frames only reside in the pagefile and are fragmented, as shown in Table II:
   - Any physical memory analysis will not yield any results, because all virtual pages are mapped to pagefile frames.
   - File carving applied to the pagefile may not yield any results due to the fragmentation.
4) Physical memory frames are scattered over physical memory and the pagefile. This is illustrated in Table II:
   - Any method besides virtual memory analysis incorporating the pagefile may not yield results, with regard to an investigation, because neither the physical memory not the pagefile contain the complete mapping of the virtual pages. Hence, only a combination of physical memory and pagefile analysis can provide a perfect reconstruction.

Only the presented virtual memory analysis approach incorporating the pagefile is able to perfectly reconstruct the virtual address space in all of the four problem cases. In reality we have found problem cases 1 and 4 to be most prevailing. We

[26]http://msft-dnl.digitalrivercontent.net/msvista/pub/X15-65740/X15-65740.iso
[27]http://msft-dnl.digitalrivercontent.net/msvista/pub/X15-65741/X15-65741.iso
[28]Windows Technical Preview: http://go.microsoft.com/fwlink/?LinkId=510226
[29]Windows Technical Preview: http://go.microsoft.com/fwlink/?LinkId=510225

have only rarely encountered case 3. We have never encountered case 2. However, close matches, where almost all virtual memory pages where sequentially mapped to pagefile frames, have been observed.

B. Synthetic Data

	Kernel	Allocation	Memory Carving [KiB]			Pagefile Carving [KiB]		Virtual Memory [KiB]			Virtual Memory + Pagefile [KiB]		
	[Build (Paging)]	[KiB]	Start		Longest	Start	Longest	Start		Longest	Start		Longest
1	7600 x64 (IA32e)	0x4000	0x8	=	0x8	-	-	0x4000	=	0x4000	0x4000	=	0x4000
2	7600 x86 (PAE)	0x4000	0x4		0x8	-	0xd	0x4000	=	0x4000	0x4000	=	0x4000
3	9841 x64 (IA32e)	0x40000	-		0xc	0x10	0x28	-		0x3afbc	0x40000	=	0x40000
4	7600 x86 (32 Bit)	0x8000	-		0x8	0x18	0x38f8	-		0x2114	0x8000	=	0x8000
5	9600 x64 (IA32e)	0x10000	-		0x1c	0x1fe8	0x2800	-		0x8c	0x10000	=	0x10000
6	9600 x86 (PAE)	0x10000	-		0x4	0x74c	0x4000	-		0x24	0x10000	=	0x10000
7	7600 x64 (IA32e)	0x10000	-		0x4	0x13c	0x2c00	-		-	0x10000	=	0x10000

Table III: Results of reconstructing synthetic data

First we evaluated our virtual address reconstruction with synthetic data. This data consists of an allocation of crib pages, as outlined in subsection II-B. After the `ramwrite` and `swapforcer` processes were started `sync.exe` from the Sysinternals Tools[30] was started to increase atomicity with regard to the pagefile written to disk. The physical RAM and pagefile were acquired via VirtualBox as outlined in subsubsection IV-A5, i.e., the VM was first paused, then the RAM was acquired via the `debugvm dumpguestcore` command, after which the VM state was saved, the hard disk cloned and the pagefile extracted via `icat`[31].

Selected results can be seen in Table III. The table lists seven different data sets obtained from different kernels with different paging. The different kernels and paging modes demonstrate the correct functionality of our implementation for the various different Windows NT systems. The table further lists the amount of crib data allocated. It then details how much sequential crib data could be reconstructed with the four methods: physical memory carving, pagefile carving, classical memory analysis, i.e., reconstructing the virtual address space without incorporating the pagefile, and last but not least our proposed virtual memory analysis method incorporating the pagefile. For each method the length of reconstructable crib data at the start of the memory allocation and the longest overall crib data reconstructable was listed. An equal sign ("=") before the longest reconstructable crib data listing denotes that the longest match was found at the start of the memory allocation. This is important if file carving methods are considered, as these methods rely on file headers which are usually at the beginning of a file. For a perfect reconstruction both the reconstructable crib data at the start and the longest overall match should coincide with the allocation size and the longest overall match must be at the start of the allocation. A dash ("-") denotes that no specific crib frame could be found.

Data set 1 was obtained from a system exhibiting problem case 1, i.e., all crib data was mapped into physical memory. This can be seen from the fact that no crib frame was present in the pagefile at all. All crib data can be reconstructed only from physical memory.

Data set 2 provides the transition state, where pages have started to be swapped out to the page file. However, because the pages were still in transition, refer section III-C2, their content was still present in physical memory and hence all crib data could be reconstructed using only physical memory.

Data sets 3 to 6 provide problem cases of type 4, i.e., crib data is placed in both the physical memory and the pagefile. The various techniques yield unpredictable results depending on the current fragmentation and scattering of the crib data. Only the proposed virtual memory analysis approach is able to perfectly reconstruct the crib data for all data sets. Data sets 3 to 6 provide the transition into problem cases 2 and 3, with the data that can be obtained via classical memory analysis not making use of the pagefile gradually declining from data set 3 to 6.

Last, data set 7 is the extreme problem case 2, in which classical memory analysis can not reconstruct any data at all, because all crib pages have been swapped out to the pagefile. The fact that the naive memory carving method was still able to extract one 4 KiB frame, can be attributed to the fact that one particular physical memory frame, which was already allocated to a different process, was not overwritten by the other process yet. We verified this hypothesis by tracing that particular found frame back to being part of the `swapforcer` process' virtual address space. As stated earlier, we used the `swapforcer` process to swap as many crib pages as possible into the pagefile.

[30]https://technet.microsoft.com/en-us/sysinternals/bb897438.aspx
[31]http://www.sleuthkit.org/sleuthkit/man/icat.html

C. Real Life Data

Besides the evaluation on synthetic data using crib pages, we also evaluated our approach against real life data. To this end, we acquired the memory and pagefile of a Windows 8.1 x64 system with 1024 MiB RAM running on x64 based hardware, while an instance of Firefox in private browsing mode was running. Firefox was used to open a prepared HTML web page with 200 JPEG images embedded. The 200 JPEG images total over 1.2 GiB. Each image was a high quality 6000×4000 pixel image from a DSLR. The data was acquired via WinPMEM[32] and fcat[33]. We then used our virtual memory analyzer to extract the virtual address space of the `firefox.exe` process. We then carved this address space with the Foremost file carver[34]. We further repeated the carving process on the raw physical memory image, the pagefile and the virtual memory address space only reconstructed from physical memory.

Physical Memory			Pagefile			Virtual Memory			Pagefile and Virtual Memory			Virtual Memory and Pagefile		
Full	Viewable	Total	Full	Viewable	Total	Full	Viewable	Total	Full	Viewable	Total	Full	Viewable	Total
0	0	269	0	≈ 33	113	0	≈ 135	454	0	≈ 135	567	200	200	600

Table IV: Images carved out of the raw memory, pagefile or different memory address space reconstructions.

The number of images each procedure could recover is listed in Table IV. The Full column lists the image files that could be fully recovered, i.e., 6000×4000 pixel images that are not corrupted in any way. The Viewable column, on the other hand, lists *distinct* images that were recognizable by visual inspection, these include corrupted and only partially recovered images, as well as embedded thumbnail images. Because the JPEGs used had two smaller thumbnail images embedded the total number of JPEGs in the Firefox process is 600. Our method was able to recover all of them. No broken images were recovered. Because no other JPEG files were opened by Firefox, no additional images were recovered as well. The other methods could only recover the smaller embedded thumbnail images but no full 6000×4000 pixel image.

This evaluation using real life data underlines the practicability of our results. The fact that current memory analysis techniques were only able to retrieve viewable content for 135 of the 200 images and were unable to recover the full 6000×4000 pixel images, while we are able to reconstruct all images, further punctuates the practical importance of our results.

VII. Conclusion

In all evaluated cases, we performed better or at least on par with current memory analysis techniques. Further has our grey-box analysis approach quickly exposed critical errors during the development of our tools. Hence we expect this technique to be able to successively improve current and future memory analysis software with regard to correctness and completeness, as well.

A. Future Work

As with all research this work only presents a small part of what can be done. Future research will be needed on the following fields.

1) Increase Operating Support: While the Windows NT family of operating system is most prevailing, other operating systems need to be considered as well. Our analysis approach can, as already stated in item II-C, be used on the Linux operating system as well, but also other operating systems such as Apple's Mac OS or the BSD family of operating systems should be evaluated.

2) Improve Combined Acquisition Methods: To develop correct reconstruction we used virtual machines to acquire the physical memory and corresponding pagefiles of systems with ultimate atomicity. While this provides, without doubt, the best results, it is not always possible. Hence, better acquisition methods acquiring both physical memory and the pagefile at the same time with high atomicity must be investigated. Also the effects of virtual address reconstruction using inconsistent physical memory and pagefile dumps must be researched. Our preliminary findings indicate that even though a virtual address space can be reconstructed from inconsistent physical memory and pagefile dumps, the reconstructed address space inevitably contains incorrect, i.e., outdated, memory pages. But the impact of these reconstruction errors is not known yet.

[32]http://www.rekall-forensic.com/downloads.html
[33]http://www.sleuthkit.org/sleuthkit/man/fcat.html
[34]http://foremost.sourceforge.net/

3) Extend Virtual Memory Analysis: The virtual memory analysis needs to be extended beyond the pagefile. To this end, Prototype PTEs of shared memory and mapped files [8] should be considered. Relevant code can be found in the Windows NT Research Kernel source code [10]. The relevant parts are the code handling software page faults due to the Prototype PTEs [10, `MiResolveProtoPteFault()`][10, `MiResolveMappedFileFault()`].

4) Compressed and Encrypted Pagefiles: As shown in other research [22], pagefiles can be compressed and also encrypted. Hence, decompression and decryption procedures may need to be integrated into future virtual memory analysis procedures that want to incorporate compressed and/or encrypted pagefiles.

5) Best Practice: Last but not least, best practice approaches with regard to combined pagefile and memory acquisition must be researched. For our tests we always synced the disks of the system, because we assume it yields higher atomicity. However, we have not evaluated what consequences this could have, e.g. old data, which may contain relevant evidence, could be overwritten by the sync operation.

B. Summary

Summarizing our work it can be said that even though virtual memory analysis is an important field, it is, as outlined by our future work list, still an emerging field, that still has problems. We solved one of these problems, namely incorporating the pagefile. The generic research method we used is expected to be deployed against other operating systems as well as being used to evaluate the correctness of other memory forensic tools.

References

[1] D. Ohana and N. Shashidhar, “Do private and portable web browsers leave incriminating evidence?: a forensic analysis of residual artifacts from private and portable web browsing sessions,” *EURASIP Journal on Information Security*, vol. 2013, no. 1, 2013. [Online]. Available: http://dx.doi.org/10.1186/1687-417X-2013-6

[2] H. Said, N. Al Mutawa, I. Al Awadhi, and M. Guimaraes, “Forensic analysis of private browsing artifacts,” in *Innovations in information technology (IIT), 2011 International conference on*. IEEE, 2011, pp. 197–202.

[3] N. Al Mutawa, I. Al Awadhi, I. Baggili, and A. Marrington, “Forensic artifacts of Facebook’s instant messaging service,” in *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for*. IEEE, 2011, pp. 771–776.

[4] S. Lee, A. Savoldi, S. Lee, and J. Lim, “Windows pagefile collection and analysis for a live forensics context,” in *Future Generation Communication and Networking (FGCN 2007)*, vol. 2. IEEE, 2007, pp. 97–101.

[5] S. Vömel and F. C. Freiling, “A survey of main memory acquisition and analysis techniques for the windows operating system,” *Digital Investigation*, vol. 8, no. 1, pp. 3–22, 2011.

[6] J. D. Kornblum, “Using every part of the buffalo in Windows memory analysis,” *Digital Investigation*, vol. 4, no. 1, pp. 24–29, 2007.

[7] N. L. Petroni Jr, A. Walters, T. Fraser, and W. A. Arbaugh, “FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory,” *Digital Investigation*, vol. 3, no. 4, pp. 197–210, 2006.

[8] R. Van Baar, W. Alink, and A. Van Ballegooij, “Forensic memory analysis: Files mapped in memory,” *digital investigation*, vol. 5, pp. S52–S57, 2008.

[9] B. Dolan-Gavitt, “The VAD tree: A process-eye view of physical memory,” *digital investigation*, vol. 4, pp. 62–64, 2007.

[10] Microsoft Corporation, *Windows Research Kernel v1.2*, 2006. [Online]. Available: http://www.microsoft.com/education/facultyconnection/articles/articledetails.aspx?cid=2416

[11] *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide, Part 1*, Intel, Santa Clara, CA, USA, Jun. 2010.

[12] B. D. Carrier and J. Grand, “A hardware-based memory acquisition procedure for digital investigations,” *Digital Investigation*, vol. 1, no. 1, pp. 50–60, 2004.

[13] M. Becher, M. Dornseif, and C. N. Klein, “FireWire: all your memory are belong to us,” *Proceedings of CanSecWest*, 2005.

[14] S. Vömel and F. C. Freiling, “Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition,” *Digital Investigation*, vol. 9, no. 2, pp. 125–137, 2012.

[15] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, “Lest we remember: cold-boot attacks on encryption keys,” *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[16] M. Gruhn and T. Müller, “On the practicability of cold boot attacks,” in *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*. IEEE, 2013, pp. 390–397.

[17] M. Falconer, C. Mozak, and A. Norman, “Suppressing power supply noise using data scrambling in double data rate memory systems,” Aug. 6 2013, uS Patent 8,503,678. [Online]. Available: https://www.google.com/patents/US8503678

[18] J. Stüttgen and M. Cohen, “Anti-forensic resilient memory acquisition,” *Digital Investigation*, vol. 10, pp. S105–S115, 2013.

[19] C. Maartmann-Moe, S. E. Thorkildsen, and A. Årnes, “The persistence of memory: Forensic identification and extraction of cryptographic keys,” *digital investigation*, vol. 6, pp. S132–S140, 2009.

[20] T. Müller, T. Latzo, and F. Freiling, “Hardware-based Full Disk Encryption (In) Security Survey,” Tech. rep., Friedrich-Alexander University of Erlangen-Nuremberg, Technical Report (September 2012), Tech. Rep., 2012.

[21] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, “Robust Signatures for Kernel Data Structures,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS ’09. New York, NY, USA: ACM, 2009, pp. 566–577. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653730

[22] G. G. Richard and A. Case, “In lieu of swap: Analyzing compressed RAM in Mac OS X and Linux,” *Digital Investigation*, vol. 11, pp. S3–S12, 2014.