electronics

Article

Memory Layout Extraction and Verification Method for Reliable Physical Memory Acquisition

Seungwon Jung [1], Seunghee Seo [1], Yeog Kim [2] and Changhoon Lee [1,*]

[1] Department of Computer Science and Engineering, Seoul National University of Science and Technology, Seoul 01811, Korea; swjung703@seoultech.ac.kr (S.J.); sh.seo@seoultech.ac.kr (S.S.)
[2] Research Institute of Electric and Information Technology, Seoul National University of Science and Technology, Seoul 01811, Korea; kim_yeog@seoultech.ac.kr
* Correspondence: chlee@seoultech.ac.kr

Abstract: Physical memory acquisition is a prerequisite when performing memory forensics, referring to a set of techniques for acquiring and analyzing traces associated with user activity information, malware analysis, cyber incident response, and similar areas when the traces remain in the physical RAM. However, certain types of malware have applied anti-memory forensics techniques to evade memory analysis strategies or to make the acquisition process impossible. To disturb the acquisition process of physical memory, an attacker hooks the kernel API, which returns a map of the physical memory spaces, and modifies the return value of the API, specifically that typically used by memory acquisition tools. Moreover, an attacker modifies the kernel object referenced by the kernel API. This causes the system to crash during the memory acquisition process or causes the memory acquisition tools to incorrectly proceed with the acquisition. Even with a modification of one byte, called a one-byte modification attack, some tools fail to acquire memory. Therefore, specialized countermeasure techniques are needed for these anti-memory forensics techniques. In this paper, we propose a memory layout acquisition method which is robust to kernel API hooking and the one-byte modification attack on *NumberOfRuns*, the kernel object used to construct the memory layout in Windows. The proposed acquisition method directly accesses the memory, extracts the byte array, and parses it in the form of a memory layout. When we access the memory, we extract the *_PHYSICAL_MEMORY_DESCRIPTOR* structure, which is the basis of the memory layout without using the existing memory layout acquisition API. Furthermore, we propose a verification method that selects a reliable memory layout. We realize the verification method by comparing *NumberOfRuns* and the memory layout acquired via the kernel API, the registry, and the proposed method. The proposed verification method guarantees the reliability of the memory layout and helps secure memory image acquisition through a comparative verification with existing memory layout acquisition methods. We also conduct experiments to prove that the proposed method is resistant to anti-memory forensics techniques, confirming that there are no significant differences in time compared to the existing tools.

Keywords: memory forensics; anti-forensics; malware; cyber security; digital forensics; incident response

check for updates

Citation: Jung, S.; Seo, S.; Kim, Y.; Lee, C. Memory Layout Extraction and Verification Method for Reliable Physical Memory Acquisition. *Electronics* **2021**, *10*, 1380. https://doi.org/10.3390/electronics10121380

Academic Editor: Krzysztof Szczypiorski

Received: 27 April 2021
Accepted: 4 June 2021
Published: 9 June 2021

1. Introduction

Traditional forensics focuses on data in the storage media of computers, cell phones, cameras and other devices, but in recent years, the scope of digital forensics has expanded to include data extracted from volatile memory [1]. Memory can contain decrypted data such as passwords, running processes themselves, and recently used commands [2,3]. Based on the remaining data, encrypted data in the file system can be decrypted or additional user activity records can be stored. In other words, memory forensics refers to techniques that acquire and analyze data that can only be verified in an active state. In addition, memory forensics is effective when used to analyze malicious code such as rootkits, as this strategy can verify the traces of the process.

However, memory-resident malware via disk-avoiding is distributed with anti-memory forensics techniques installed to prevent memory forensics tools from acquiring and analyzing data [4]. Anti-forensics has been defined as "any attempt to compromise the availability or usefulness of evidence to the forensics process", and anti-memory forensics is a technique that attempts to disrupt the memory forensics process [5,6]. Anti-memory forensics refers to anti-forensics techniques such as a process hiding by deleting information from the list of running processes and preventing memory dumps by detecting the execution of memory forensics tools or monitoring the kernel application programming interface (API) for these processes. In particular, if the memory acquisition process is disturbed, the basis of the analysis must be invalidated. Therefore, countermeasure studies on anti-memory forensics techniques are required [7].

In the physical memory, the basic input/output system (BIOS), read only memory (ROM), peripheral component interconnect (PCI) resource and direct memory access (DMA) buffer are loaded as well as the operating system, with these loaded into physical memory spaces called the reserved area. When memory forensics tools are acquiring the memory of a running computer, accessing reserved areas can cause a system crash or can corrupt data. In order to prevent these outcomes, a memory layout representing map of the physical address space must be obtained through a BIOS interrupt call (INT 15 h E820 h) [8]. However, Windows works in protected mode supported by CPU architectures such as the Intel Architecture (IA)-32, IA-64, and ARM [9]. Thus, because Windows operates in Protected Mode, it is not possible to request a memory layout directly through the BIOS [10].

Thus, memory acquisition tools use the kernel API to obtain the memory layout and perform memory acquisition based on it. However, malware can interfere with the memory acquisition process by hooking the kernel API or providing false information through a tiny modification of the kernel objects [11,12]. *MmGetPhysicalMemoryRanges()* is a kernel API function that is used to acquire the memory layout, and we have confirmed that with memory acquisition tools such as Winpmem [13–15] and Ftk Imager [16], if the function is hooked and modified to always return as NULL, these tools cannot perform a memory dump because the memory layout cannot be acquired [11].

The one-byte modification attack refers to an attack that utilizes anti-memory forensic techniques by modifying only one byte of a kernel object. This attack interferes with the process listing, OS version identification, and memory acquisition [12]. Some previous versions of memory acquisition tools use the *KdDebugBlock* structure to acquire the kernel base address, which can be found by scanning "KDBG" which is the OwnerTag value. However, because the search is based on a character string, even if only 1 byte is modified, it cannot be scanned, and this causes a problem in the entire memory acquisition process [11].

Additionally, *NumberOfRuns* is a value representing the number of physical address spaces, and *MmGetPhysicalMemoryRanges()* acquires a memory layout based on *NumberOfRuns*. If the value of *NumberOfRuns* is modulated to "0", *MmGetPhysicalMemoryRanges()* returns memory layouts with empty information [12]. In this paper, we modified the value of *NumberOfRuns* and utilized memory acquisition tools and as shown in Section 4.2, we confirmed that memory acquisition is performed abnormally.

We propose a method of acquiring memory layout by directly accessing the memory through the kernel driver in the Windows operating system and a memory layout verification process that securely acquires the data in the physical memory. The verification process selects reliable memory layout through the acquisition method using the proposed kernel object and cross-validation with the two existing methods. Existing memory forensics tools mainly use *MmGetPhysicalMemoryRanges()*, the kernel API function, or they extract and parse from the '*.Translated*' value of the registry to acquire the memory layout. The proposed method was tested and evaluated on a Windows 10 64-bit operating system. The contributions of this paper are as follows.

- We analyzed how existing memory acquisition tools acquire the memory layout, and identified API hooking and one-byte modification attacks that disturb this process.

The proposed memory layout acquisition technique can respond to existing API hooking attacks and to one-byte modification attacks.

- We found that existing memory acquisition tools perform memory acquisition without the verification of the memory layout, and we provide stability to the memory acquisition process by cross-validating the memory layout, which is acquired in three ways.
- The proposed technique was integrated into Winpmem, which is an open source tool, and we confirmed that it responds to aforementioned attacks through experiments. The proposed method took an average of 30s to acquire and verify the memory layout, but it was confirmed that this time can be reduced to 1 s if applying an improvement method.

We propose a memory layout verification method for reliable physical memory acquisition. Section 2 of this paper describes research related to memory acquisition and anti-memory forensics. In Section 3, three methods for acquiring data related to the memory layout and the proposed memory layout verification method are described, and a plan for implementing the proposed method is presented. Section 4 presents experimental results demonstrating the effectiveness of the proposed method. Section 5 concludes the thesis by discussing the meaning and contributions of the proposed method.

2. Related Work

In this section, we discuss research related to hardware-based memory acquisition, software-based acquisition, and anti-memory forensics.

2.1. Hardware-Based Memory Acquisition

Hardware-based memory acquisition methods have been proposed to overcome limitations related to or dependent on the operating system. Because acquisition takes place when the system is stopped, it can be acquired without changing the memory. Carrier et al. [17] suggested a hardware-based memory acquisition technique that does not depend on the operating system, considering that acquiring memory from a damaged computer is often dependent on unreliable code or on the operating system. Their method works by installing the memory acquisition tool through a PCI expansion card before an incident, acquiring the memory through a simple switch operation. It was verified that hardware-based memory acquisition could be successfully performed through a comparison with the memory dump file acquired by software.

However, PCI expansion cards are overpriced and stability is weak because they can trigger the "Blue Screen of Death" (BSoD) state in the Windows system. In addition, memory can only be acquired if the card is installed before an accident. Zhang et al. [18] presented a Windows-based memory acquisition technique using the Firewire protocol. They proposed a hardware-based memory acquisition technique that is compatible with software-based memory acquisition, which is not usable when the unlocked password is unknown and information manipulated by anti-memory forensics malware or information acquisitions may fail. In particular, in order to prevent the BSoD state during the memory acquisition process, the memory layout was acquired by parsing the '*.Translated*' value in the registry, with memory acquisition then performed based on the memory layout [19].

The registry is a system-defined database that stores the information necessary for managing the operating system [20]. *Translated*, the value of the subkey *HKEY_LOCAL_MACHINE/HARDWARE/RESOURCEMAP/System Resources/Physical Memory*, contains the number of memory address spaces and the start address and length information of each memory space. When *.Translated* is extracted and parsed, data related to the memory layout can be acquired. Figure 1 lists the values of *.Translated* in hexadecimal and shows the structure.

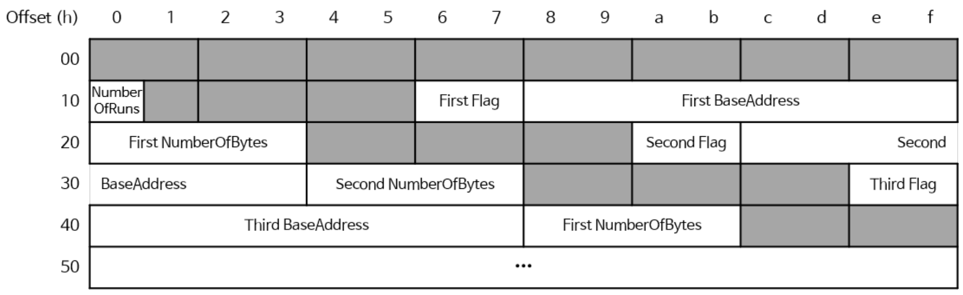

Figure 1. Structure of *.Translated*, the value of the registry key related to memory layout.

NumberOfRuns located at 0x10 is a value representing the number of memory spaces used by the Windows operating system. Flag information is included for each space, representing the data type of the length of the memory space. If the flag is 0x0, the 32-bit value itself indicates the length of the memory space. In the 0x200 case, the upper 32 bits of the 40-bit length value are indicated and the lower 8 bits are considered to have a value of 0. In the 0x400 and 0x800 case, the upper 32 bits of 48 bits and the upper 32 bits of 64 bits are represented, and the lower 16 bits and lower 32 bits, respectively, are considered to be equal to 0 [21,22].

However, when parsing the value of *.Translated*, it is impossible to acquire the correct size memory layout if the *NumberOfRuns* value of *.Translated* is an abnormal value due to an attack. Moreover, the start address and length information of each space may also be attacked.

2.2. Software-Based Memory Acquisition

MmGetPhysicalMemoryRanges() is a kernel API function mainly used by memory acquisition tools, and it returns *_PHYSICAL_MEMORY_RANGES*, which is an array of structures containing information about the memory layout. Figure 2 shows the process by which *MmGetPhysicalMemoryRanges()* returns the memory layout. *_PHYSICAL_MEMORY_RANGES* is created based on the *_PHYSICAL_MEMORY_RUN* structure array of the *_PHYSICAL_MEMORY_DESCRIPTOR* structure. The length of the array is determined by the value of *NumberOfRuns*, a member variable of the *_PHYSICAL_MEMORY_DESCRIPTOR*. The member variable of *_PHYSICAL_MEMORY_RANGES* consists of the base address and NumberOfBytes and stores the start address and length of each area. Additionally, the last member variable of *_PHYSICAL_MEMORY_RANGES* is initialized as 0 to signal the end of the array. The *_PHYSICAL_MEMORY_RUN* array has BasePage and PageCount as member variables, and it stores the start page and the number of pages for each space. *_PHYSICAL_MEMORY_DESCRIPTOR* has *NumberOfRuns*, *NumberOfPages* and Run as member variables, which store the number of memory spaces, the total size of the pages, and the start address of the *_PHYSICAL_MEMORY_RUN*, respectively.

As described above, the *MmGetPhysicalMemoryRanges()* function refers to the number of physical address spaces, and if the value of *NumberOfRuns* is abnormal due to an attack, it is impossible to acquire the correct size of the memory layout. In addition, when the value of the *_PHYSICAL_MEMORY_RUN* structure array is changed, the number of physical address spaces is correct but the start address and length information for each space are incorrect.

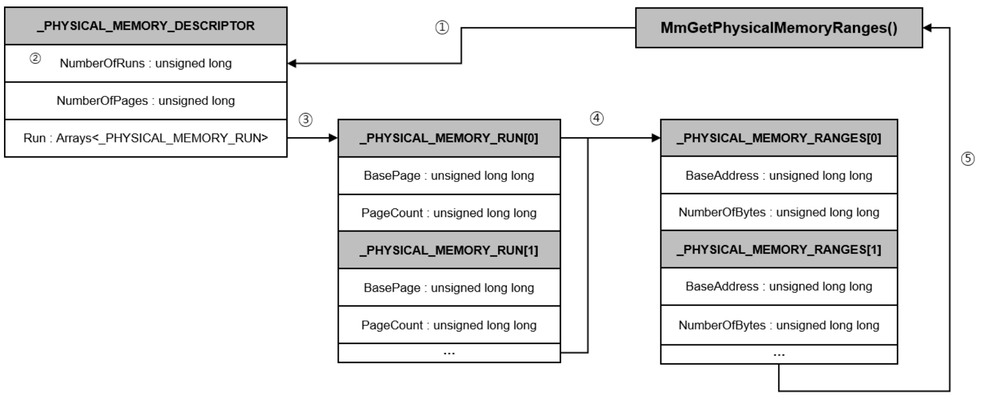
_PHYSICAL_MEMORY_DESCRIPTOR
②
NumberOfRuns : unsigned long
NumberOfPages : unsigned long
Run : Arrays<_PHYSICAL_MEMORY_RUN>
①
MmGetPhysicalMemoryRanges()
③
_PHYSICAL_MEMORY_RUN[0]
BasePage : unsigned long long
PageCount : unsigned long long
_PHYSICAL_MEMORY_RUN[1]
BasePage : unsigned long long
PageCount : unsigned long long
...
④
_PHYSICAL_MEMORY_RANGES[0]
BaseAddress : unsigned long long
NumberOfBytes : unsigned long long
_PHYSICAL_MEMORY_RANGES[1]
BaseAddress : unsigned long long
NumberOfBytes : unsigned long long
...
⑤

Figure 2. Member variables of each structure and process of *MmGetPhysicalMemoryRanges()* function.

2.3. Anti-Memory Forensics

Until recently, various techniques that prevent memory forensics tools from deriving meaningful results have been studied, and memory forensics techniques have developed in the direction of complementing them.

2.3.1. API Hooking

Stüttgen et al. suggested a technique that hinders memory acquisition by hooking the Windows API [11]. Some memory acquisition tools use the *MmGetPhysicalMemoryRanges()* function to acquire physical memory space information. If the physical memory space information is not properly transmitted, the system crashes or incorrect memory is acquired. Therefore, Stüttgen et al. hooked the *MmGetPhysicalMemoryRanges()* function to always return NULL, and it was confirmed that some memory collection tools were working abnormally as a result.

Milković presented Dementia, an anti-memory technology that works through Windows API hooking [23,24]. Dementia hooks NtWriteFile, a Windows API, and manipulates and returns the data used in the buffer when Memoryze, a memory acquisition tool, calls the API. The manipulated data are various data related to processes such as process, thread, object handle, and the virtual address descriptor (VAD), among others. When memory is acquired through data manipulation, the object to be hidden does not exist in the dump file.

2.3.2. One-Byte Modification

Haruyama et al. analyzed the anti-memory forensics technology based on one-byte modification of kernel objects [12]. Certain special kernel objects used for memory forensics have unique identification values that can identify the kernel objects. For *_DBGKD_DEBUG_DATA_HEADER64*, as an example, it has a "KDBG" tag as a unique identification value. If only one byte of the tag is modified to a different value, it has been confirmed that some memory analysis tools do not operate normally.

Google's Rekall project team presented a type of anti-memory forensics technology that works through one-byte modification [25]. Some memory collection tools use the *_PHYSICAL_MEMORY_DESCRIPTOR* structure, an aspect of the kernel objects, to obtain physical memory area information. At this time, if the member variable *NumberOfRuns* of the structure was modified to 0, hence, it was determined that when using this strategy, no physical memory area was acquired, and it was confirmed that the acquisition process of the memory acquisition tool was stopped.

2.4. Consideration

All of these studies showed that software-based memory acquisition tools were unstable, and no clear solution was provided. Furthermore, most of the tools use only one method to acquire the memory layout, and no verification method for the acquired memory layout is included. Hence, the acquired memory image is also unreliable. Latzo et al. confirmed that the lower the execution layer of the acquisition method, the higher the accuracy of the acquired memory image [26]. Therefore, we propose a memory layout acquisition method that can cope with the existing API hooking and one-byte modification attacks. We also propose a memory layout verification method for reliable physical memory acquisition.

3. Proposal of the Memory Layout Verification Method for Acquiring Reliable Physical Memory

This section describes the proposed method of acquiring the memory layout, and explains the method of selecting a reliable memory layout by cross-validating the method using the registry, as described in Section 2.1, and the kernel API, as described in Section 2.2.

3.1. Acquisition Method of the Memory Layout via NumberOfPages

_PHYSICAL_MEMORY_DESCRIPTOR contains a number and a map of the physical address spaces, as noted in Section 2.2. Because direct access to memory is possible, *_PHYSICAL_MEMORY_DESCRIPTOR* can be extracted as a byte array. Figure 3 shows the structure of the byte array acquired by accessing the address of *_PHYSICAL_MEMORY_DESCRIPTOR*.

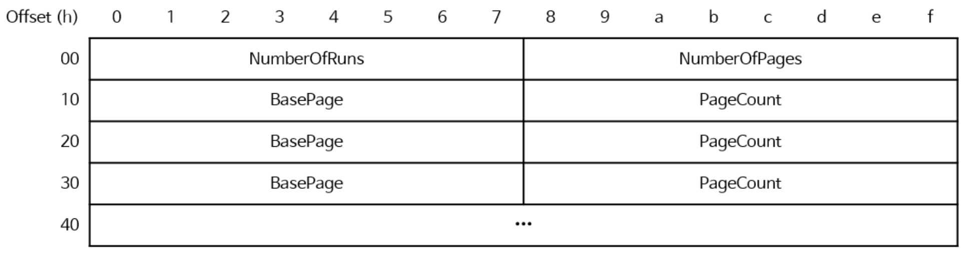

Figure 3. Kernel object's byte array structure representing the memory layout.

NumberOfRuns is vulnerable to the one-byte modification attack; accordingly, we use the alternative method of calculating *NumberOfRuns* using the value of *NumberOfPages*. Given that the value of *NumberOfPages* is identical to the sum of the values of the *PageCount* item, *NumberOfRuns* is calculated according to the sum of *PageCount* and by counting the number of times until the value of *NumberOfPages* and the sum of *PageCount* are equivalent. The relationship between the values inside the byte array is as follows:

$$NumberOfRuns = COUNT(BasePage) \tag{1}$$

$$NumberOfPages = \sum_{k=1}^{n} PageCount(n : NumberOfRuns) \tag{2}$$

Algorithm 1 shows the process of calculating *NumberOfRuns* using these equations. Because the value of *NumberOfRuns* is calculated using the value of *NumberOfPages*, if the value of *NumberOfPages* is attacked or if even one value of PageCount for each area is incorrect, the correct value of *NumberOfRuns* cannot be acquired. Moreover, if the sum of *PageCount* is calculated to be as high as the row count of the byte array but smaller than the size of *NumberOfPages*, a longer byte array is parsed and the same process is repeated.

Algorithm 1: Pseudocode of Counting *NumberOfRuns*

```
Data: $S_{row}$: Number of rows in byte array
Result: $NumberOfRuns$
1  $SumOfPage \leftarrow 0$
2  for $i = 0, 1, \ldots, S_{row} - 1$ do
3      $SumOfPage \leftarrow$ SumOfPage + PageCount[i]
4      if $SumOfPage = NumberOfPages$ then
5          $NumberOfRuns \leftarrow i + 1$
6          return NumberOfRuns
7      else if $SumOfPage > NumberOfPages$ then
8          return false
9      else if $i + 1 = S_{row}$ then
10         if $SumOfPage < NumberOfPages$ then
11             return false
12         end
13     end
14 end
```

3.2. Verification Method of the Memory Layout

Figure 4 shows the overall process of the proposed method which is divided here into three parts. The acquisition step for the data related to the memory layout acquires physical data related to memory layout using three methods: the kernel API, the registry, and the kernel object. The attack detection step through data extraction and verification parses and verifies the acquired data to detect the attack. Finally, the secure memory layout selection step selects a secure memory layout based on the detection result of an attack. The proposed method uses a strategy that counteracts anti-memory forensics techniques that hinder physical memory acquisition.

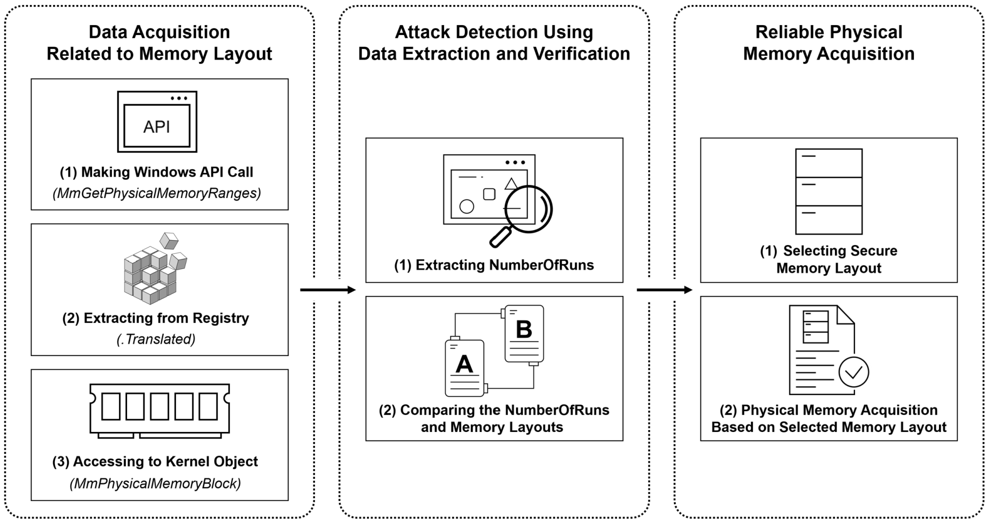
Data Acquisition
Related to Memory Layout
API
(1) Making Windows API Call
(MmGetPhysicalMemoryRanges)
(2) Extracting from Registry
(.Translated)
(3) Accessing to Kernel Object
(MmPhysicalMemoryBlock)
Attack Detection Using
Data Extraction and Verification
(1) Extracting NumberOfRuns
B
A
(2) Comparing the NumberOfRuns
and Memory Layouts
Reliable Physical
Memory Acquisition
(1) Selecting Secure
Memory Layout
(2) Physical Memory Acquisition
Based on Selected Memory Layout

Figure 4. Memory layout data extraction and verification method for reliable physical memory acquisition.

According to the three methods of acquiring the memory layout described above, in order to acquire a memory layout, it is necessary first to identify *NumberOfRuns* in which information about the number of memory spaces is stored. Using this, in order to cross-verify the memory layout acquired from each method, we compare *NumberOfRuns* and classify cases according to the results. We also compare the memory layout for each case. *NumberOfRuns* obtained through the registry, kernel API, and kernel object are expressed as NOR_{Reg}, NOR_{API}, and $NOR_{KrnlObj}$, respectively. Table 1 shows classified cases according to the comparison of *NumberOfRuns*. There are five cases: one with the same values of *NumberOfRuns*, three with only one different value, and one in which all values are different.

Table 1. Classification of cases based on comparison of *NumberOfRuns*.

Case	NOR_{Reg} == NOR_{API}	NOR_{API} == $NOR_{KrnlObj}$	NOR_{Reg} == $NOR_{KrnlObj}$	Attack Target
Case 1	True	True	True	N/A
Case 2	False	True	False	Registry
Case 3	False	False	True	Kernel API
Case 4	True	False	False	Kernel object
Case 5	False	False	False	Two or more

The following shows the details of each case.

- Case 1 : No anti-memory forensics technique has been applied to prevent memory layout acquisition;
- Case 2 : NOR_{Reg} was attacked, and the memory layout could not be obtained by parsing the value of .Translated;
- Case 3 : NOR_{API} was attacked, and the memory layout could not be obtained by the *MmGetPhysicalMemoryRanges()* function;
- Case 4 : $NOR_{KrnlObj}$ was attacked, and memory layout could not be obtained by *NumberOfPages*;
- Case 5 : Two or more values of *NumberOfRuns* were attacked, and a reliable memory layout could not be selected.

Table 2 shows a comparison of the memory layouts for each case and whether a reliable memory layout can be selected. Memory layouts are compared for each case, except when all values are different. The memory layout can be trusted only when *NumberOfRuns* and the memory layout verification process pass, and can be used when acquiring physical memory.

Table 2. Case-specific memory layout comparison and results.

Case	Memory Layout Comparison Target		Results	Remarks
Case 1	$Layout_{Reg}$	$Layout_{API}$	True	Both layouts are secure
			False	Not all layouts are secure
Case 2	$Layout_{API}$	$Layout_{KrnlObj}$	True	Both layouts are secure
			False	Not all layouts are secure
Case 3	$Layout_{Reg}$	$Layout_{KrnlObj}$	True	Both layouts are secure
Case 4	$Layout_{Reg}$	$Layout_{API}$	True	Both layouts are secure
			False	Not all layouts are secure

Case 1, Case 2, and Case 4 can be targeted for API hooking attacks because they are compared with $Layout_{API}$. Therefore, if the two memory layouts are different, it means that $Layout_{API}$ has been altered by an API hooking attack. Case 3 means that it is always secure against one-byte modification and API hooking attacks. Figure 5 shows a flow chart of the verification process.

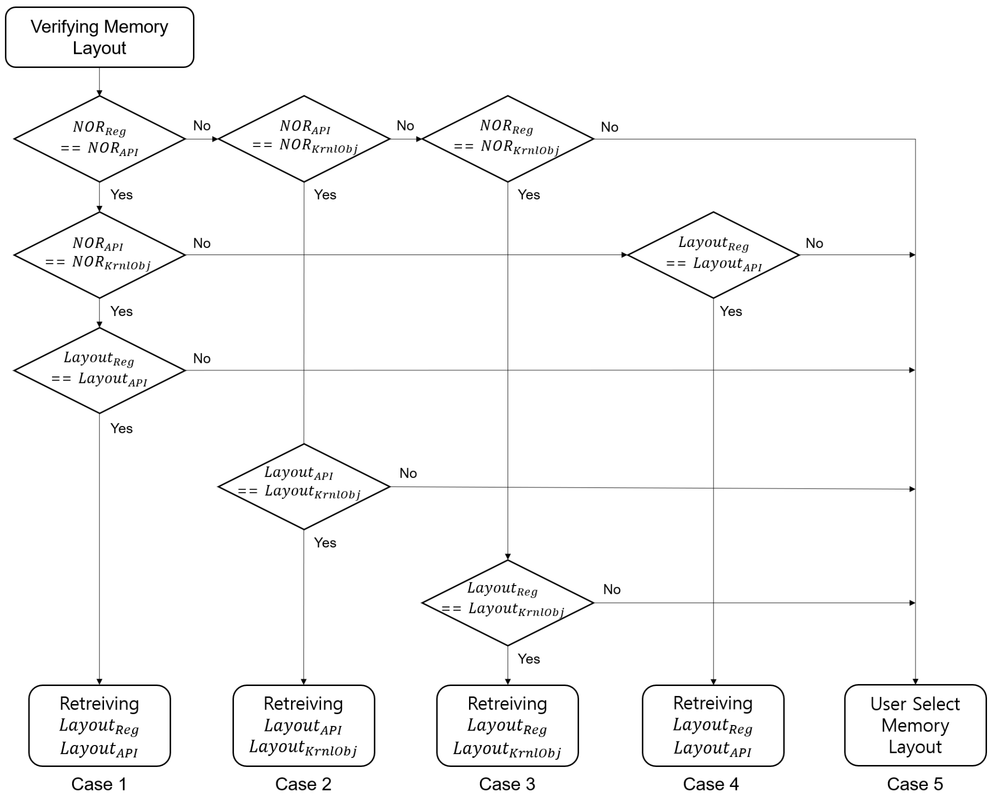
Verifying Memory Layout
$NOR_{Reg} == NOR_{API}$
$NOR_{API} == NOR_{KrnlObj}$
$NOR_{Reg} == NOR_{KrnlObj}$
$NOR_{API} == NOR_{KrnlObj}$
$Layout_{Reg} == Layout_{API}$
$Layout_{Reg} == Layout_{API}$
$Layout_{API} == Layout_{KrnlObj}$
$Layout_{Reg} == Layout_{KrnlObj}$
Yes
No
Retreiving $Layout_{Reg}$ $Layout_{API}$
Case 1
Retreiving $Layout_{API}$ $Layout_{KrnlObj}$
Case 2
Retreiving $Layout_{Reg}$ $Layout_{KrnlObj}$
Case 3
Retreiving $Layout_{Reg}$ $Layout_{API}$
Case 4
User Select Memory Layout
Case 5

Figure 5. Memory layout verification and reliable memory layout selection process.

3.3. Implementation of the Proposed Method

This section describes the process of acquiring physical memory data by selecting a secure memory layout. It is selected by adding a memory layout validation part to modify Winpmem, a Windows memory acquisition tool. It was installed in Rekall, a memory forensics tool, but now it has been converted into a separate project maintained by Velocidex [14,15]. Winpmem uses the method of acquiring the memory layout with the kernel API, as described in Section 2.2. Therefore, we added a process of acquiring data related to the memory layout using the method of the registry and directly accessing the memory, as correspondingly mentioned in Sections 2.1 and 3.1. Furthermore, before performing memory acquisition, we added a process of verifying *NumberOfRuns* and the memory layout. However, in order to access *_PHYSICAL_MEMORY_DESCRIPTOR* from the kernel driver, a process for acquiring the structure address in memory is required.

Program DataBase (PDB) files, called debugging symbols, contain information on how the compiler connects the source code and the machine language [27,28]. Microsoft provides a PDB file of the kernel image for each Windows version, and the address of the kernel object can be acquired by acquiring and parsing the file from Microsoft's Symbol Server [29,30]. Accordingly, *MmPhysicalMemoryBlock*, which is a kernel object in which the address of *_PHYSICAL_MEMORY_DESCRIPTOR* is stored, can be obtained from the PDB, and the byte array of the structure can be obtained from the address. Figure 6 shows the operation process of the proposed memory acquisition tool for reliable physical memory acquisition. The highlighted part shows the added process compared to the existing Winpmem.

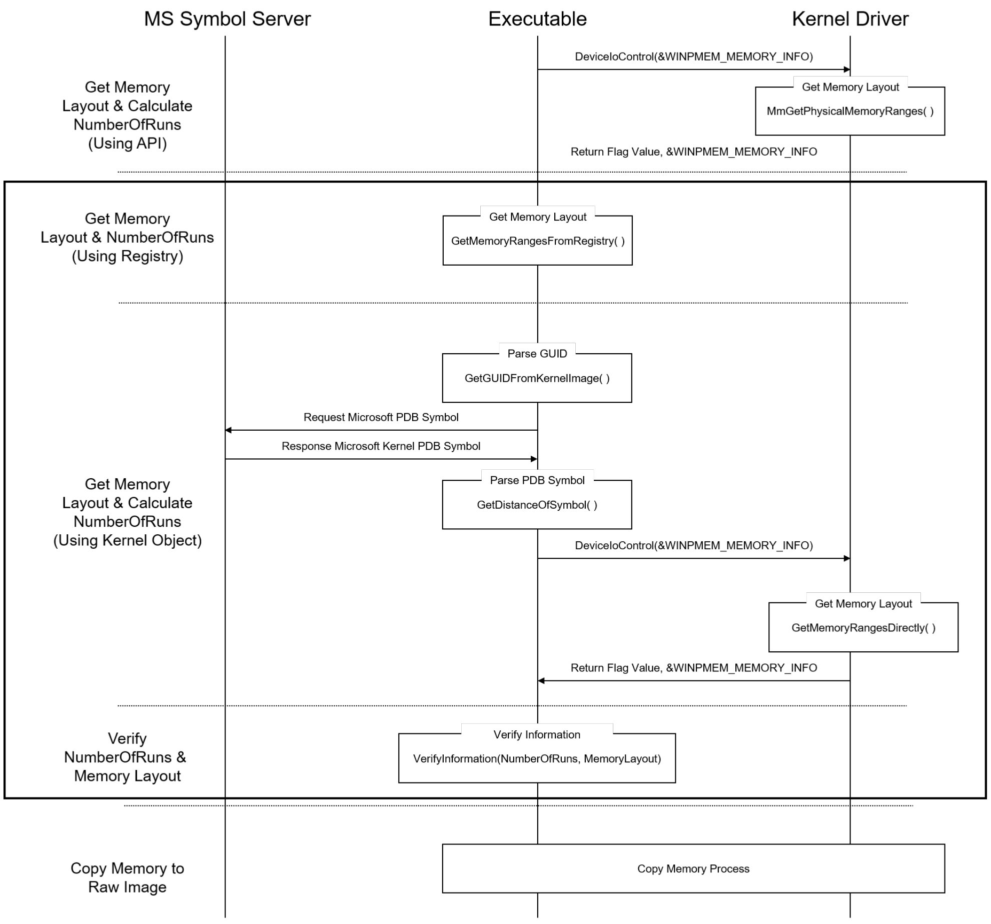
MS Symbol Server
Executable
Kernel Driver
Get Memory Layout & Calculate NumberOfRuns (Using API)
DeviceIoControl(&WINPMEM_MEMORY_INFO)
Get Memory Layout
MmGetPhysicalMemoryRanges( )
Return Flag Value, &WINPMEM_MEMORY_INFO
Get Memory Layout & NumberOfRuns (Using Registry)
Get Memory Layout
GetMemoryRangesFromRegistry( )
Parse GUID
GetGUIDFromKernelImage( )
Request Microsoft PDB Symbol
Response Microsoft Kernel PDB Symbol
Get Memory Layout & Calculate NumberOfRuns (Using Kernel Object)
Parse PDB Symbol
GetDistanceOfSymbol( )
DeviceIoControl(&WINPMEM_MEMORY_INFO)
Get Memory Layout
GetMemoryRangesDirectly( )
Return Flag Value, &WINPMEM_MEMORY_INFO
Verify NumberOfRuns & Memory Layout
Verify Information
VerifyInformation(NumberOfRuns, MemoryLayout)
Copy Memory to Raw Image
Copy Memory Process

Figure 6. Operation process of the proposed memory acquisition tool for reliable physical memory acquisition.

When the memory acquisition process starts, the memory layout is acquired using the kernel API, which is an existing Winpmem process. The *MmGetPhysicalMemoryRanges()* function is a kernel API, meaning that the process must be performed at the kernel driver level. Second, the process parses the value of *.Translated* and obtains *NumberOfRuns* and the memory layout information. Third, it accesses the kernel object *MmPhysicalMemoryBlock* to acquire and parse the byte array of the *_PHYSICAL_MEMORY_DESCRIPTOR* structure. Likewise, this process must be performed by the kernel driver in order to use functions that directly access memory. *NumberOfRuns* and the memory layouts acquired through three methods are compared and verified, and memory acquisition is performed based on a reliable memory layout.

4. Experiments

This section describes the experiments and results to prove the effectiveness and performance of the proposed method. The proposed method acquires reliable physical memory data based on the cross-validation of the memory layout. This experiment checks whether the *NumberOfRuns* attack countermeasure techniques are applied to Winpmem (Rekall, Velocidex) and FTK Imager, which are windows memory acquisition tools, and Custom Winpmem, to which the proposed method is applied.

4.1. Experimental Setup

In this experiment, an environment for performing a memory layout modulation attack among one-byte modification attack techniques is configured. Here, we configure the target computer with VMWare on the host computer, and we configure the debugging environment with WinDbg. The host computer uses the Windows 10 64-bit operating system, the updated version is 2004, and the target computer is the 1909 version of the Windows 10, 64-bit operating system. The CPU of the host computer is an Advanced Micro Deviccs (AMD) Ryzen 7, 3700X (8-core processor) and it has 40 GB of RAM. Each memory acquisition tool performs the process of acquiring the 8, 16, and 32 GB of memory of the target computer. The following shows the versions of the programs used in the experiment:

- Rekall Winpmem : 2.1.post4
- Velocidex Winpmem : 4.0 RC2
- Access Data FTK Imager : 4.5.0.3

Figure 7 shows the configuration of the experimental environment.

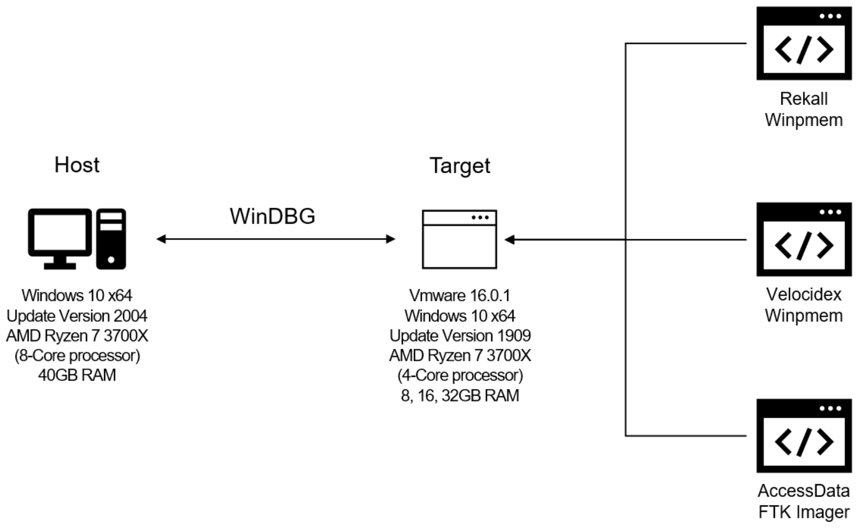

Figure 7. Configuring the memory acquisition tool experiment environment.

4.2. Experimental Results

In this experiment, each attack 2qs performed for the three methods of acquiring the memory layout, and physical memory data were acquired using the memory acquisition tools. Among the one-byte modification attack techniques, the memory layout information modification attack can be realized by modifying the *NumberOfRuns* value of *_PHYSICAL_MEMORY_DESCRIPTOR* [12]. As described in Section 3.1, the *NumberOfRuns* and *NumberOfPages* values of *_PHYSICAL_MEMORY_DESCRIPTOR* can be accessed through the kernel variable *MmPhysicalMemoryBlock*. Moreover, *NumberOfRuns* of *.Translated* can be accessed with the Windows registry editor. The experiment was performed by accessing *NumberOfRuns* using WinDbg and the registry editor and modifying the value to "0." Each test was performed 10 times, and break on access (BA) was set at the *NumberOfRuns* position to check if the memory acquisition tool accesses the *NumberOfRuns* value of *_PHYSICAL_MEMORY_DESCRIPTOR*. If setting BA, the breakpoint is triggered when the memory acquisition tool accesses that address while performing the acquisition process.

4.2.1. Attack Resistance Evaluation

Table 3 shows the experimental results of whether the tools respond to a memory layout modification attack. As a result of the experiment, the memory acquisition tools other than the tools modified by the proposed method did not normally perform memory acquisition when the *NumberOfRuns* value of *_PHYSICAL_MEMORY_DESCRIPTOR* was modified, outputting a result message indicating that the memory acquisition process was performed normally. In the case when the size of the output image file was "0" or some metadata were included in the output, it was confirmed that *NumberOfRuns* was accessed during the memory acquisition process.

Table 3. Version of the memory acquisition tool.

Tools	Case 2	Case 3	Case 4
Custom Winpmem	o	o	o
Rekall Winpmem	x	o	o
Velocidex Winpmem	x	o	o
AccessData FTK Imager	x	o	o

4.2.2. Performance Evaluation

Table 4 shows the memory acquisition time and output file size for each memory acquisition tool in a general situation. The tool implemented by the method proposed in this paper takes an average of 25–30 s to acquire the PDB file and extract *MmPhysicalMemoryBlock*. Therefore, by building a csv file in advance for the GUID of the Windows kernel and the value of *MmPhysicalMemoryBlock* in the key-value format, the time required for PDB acquisition and parsing was reduced. Advanced Custom Winpmem shows the result of testing after applying that function. The file output as a result of time memory acquisition process could be analyzed normally in Rekall, a memory forensics tool, and there is no problem with the memory acquisition function.

Table 4. Version of memory acquisition tool.

Tools	Access NumberOfRuns	Time According to Memory Size		
		8 GB	16 GB	32 GB
Custom Winpmem	o	50.1 s	67.0 s	97.9 s
Advanced Custom Winpmem	o	20.9 s	36.4 s	71.8 s
Rekall Winpmem	o	19.8 s	36.8 s	71.0 s
Velocidex Winpmem	o	19.4 s	35.9 s	71.1 s
AccessData FTK Imager	o	28.0 s	48.1 s	94.3 s

Figure 8 shows the result of measuring the operating times of the memory acquisition tools according to the memory size and calculating the estimated time spent in an environment larger than 32 GB of memory size using a linear regression technique. We confirmed that Advanced Custom Winpmem does not lag behind the performance of the existing Winpmem (Rekall) and Winpmem (Velocidex) methods.

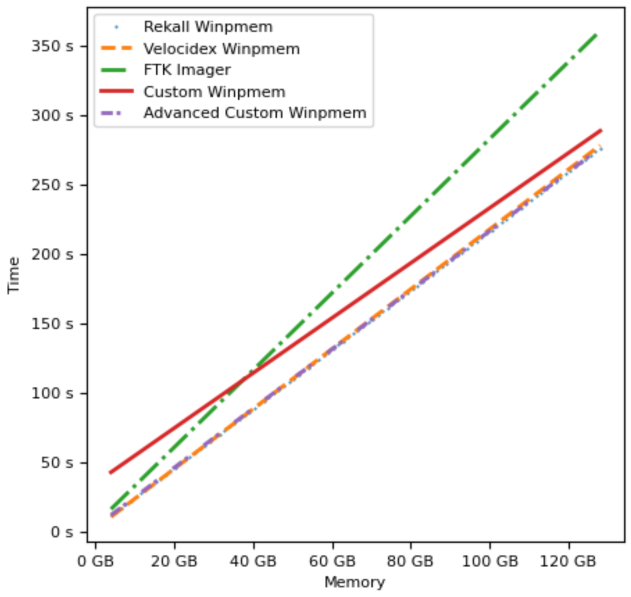

Figure 8. A graph drawn through the linear regression technique of the measured values.

Figures 9–11 show the experimental results expressed as a box plot. Likewise, we confirmed that there is no significant difference from the existing Winpmems tools.

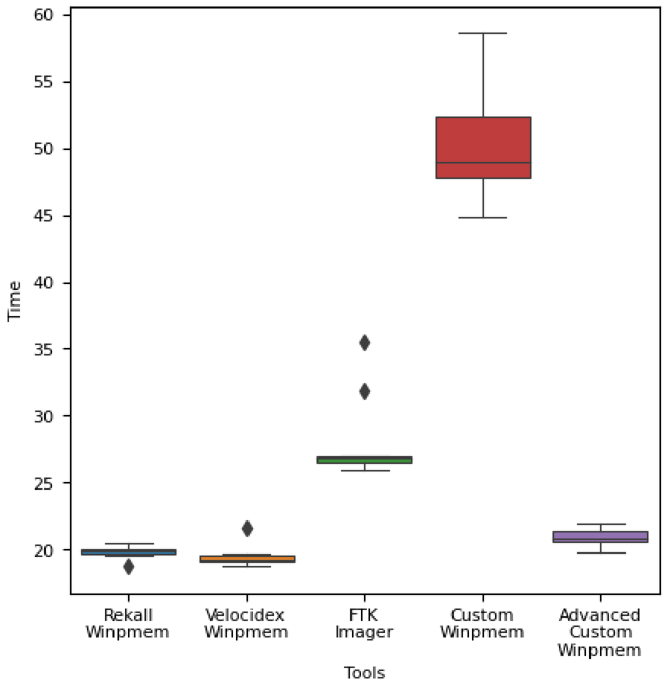

Figure 9. Operation time for each memory acquisition tool in an 8 GB memory environment.

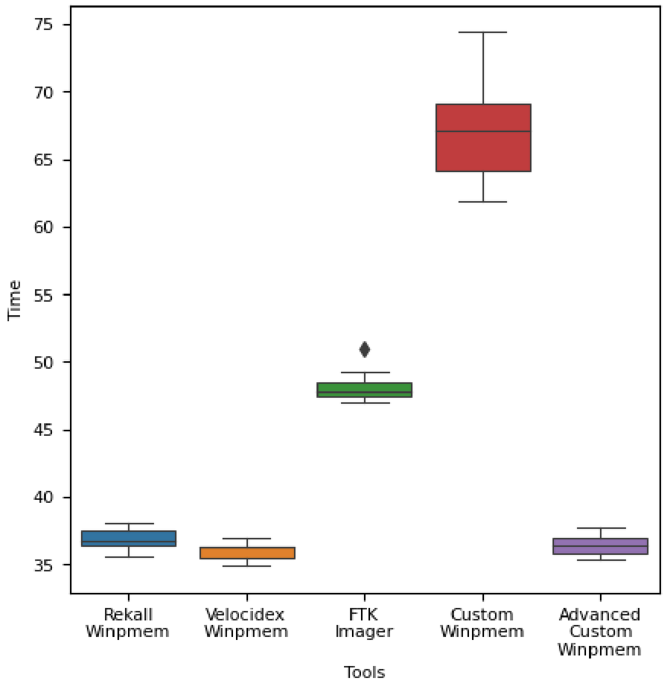

Figure 10. Operation time for each memory acquisition tool in a 16 GB memory environment.

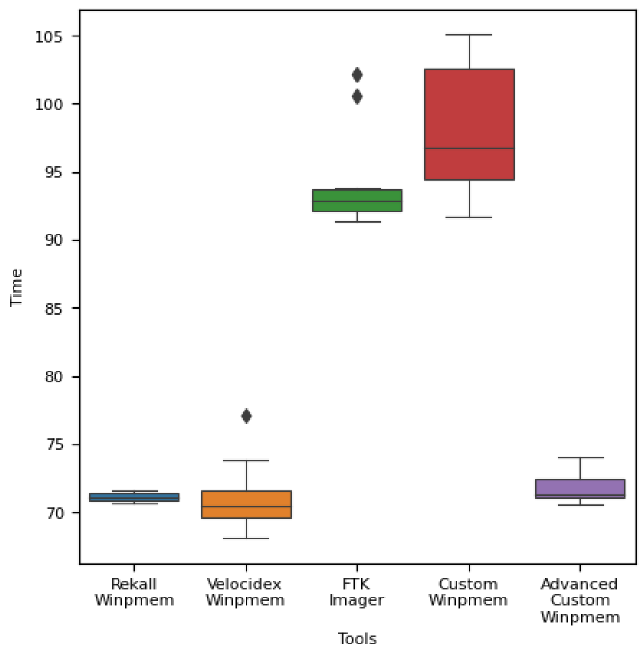

Figure 11. Operation time for each memory acquisition tool in a 32 GB memory environment.

4.3. Discussions and Limitations

The proposed method compares the value of *NumberOfRuns*, which is the basis of the memory layout, and compares the memory layout for each case such that the memory acquisition process can be securely and accurately performed. It was confirmed that Custom Winpmem with the application of the method proposed in here is resistant to anti-memory forensics techniques that interfere with memory acquisition. Moreover, we confirmed that the proposed method can be used not only for Windows 10 64-bit, but also for Windows 10 32-bit and Windows 8 32/64-bit. However, the program operation time increases due to the process of acquiring a PDB file suitable for the target Windows operating system from the MS Symbol Server. The average time required to download and parse a PDB file is 30 s, which contributes the most to the increase in the operation time. Additionally, when the Internet environment is not provided or the speed is slow, obtaining a PDB symbol and extracting data are not effective. Therefore, we configured a custom repository that collected the GUID value of the Windows kernel and *MmPhysicalMemoryBlock* to shorten the operation time. However, even if BasePage, PageCount, and .Translated values as well as *NumberOfRuns* and *NumberOfPages* were modified, there is no problem with the Windows operation, except for special circumstances. Therefore, because they can be targets of an attacker, research is needed to obtain data from areas where data changes cannot easily be attempted and to recover memory layout information based on the findings.

5. Conclusions

In this paper, we proposed a memory layout acquisition and verification method to enable reliable memory acquisition based on a secure memory layout. The proposed method acquires and extracts the memory layout and *NumberOfRuns* using the kernel API functions, the registry, and kernel objects, classifies cases through a comparison of *NumberOfRuns*, and verifies the memory layout to select a secure memory layout. In general, memory forensics tools run on damaged systems, so the results of the analysis may not be reliable. In terms of memory analysis, if the acquired physical memory data themselves are unreliable, the meaning of the analysis fades. In this paper, the proposed method was applied to Winpmem, an open source memory acquisition tool, and through experiments, it was shown that the physical memory data acquired with the tool can be trusted. In future research, we plan to develop memory acquisition technology by studying whether an attacker can acquire a memory layout from an area where data modification cannot easily be attempted. Furthermore, in the future, we plan to develop memory acquisition technology by studying whether investigations can acquire memory layouts from areas where attackers cannot easily modify data. In addition, in terms of memory analysis, we intend to study a systematic memory acquisition and analysis framework by examining countermeasures for anti-memory forensics techniques.

Author Contributions: The contributions of the authors are as follows: Conceptualization, S.J., S.S. and Y.K.; methodology, S.J.; software, S.J.; validation, S.J.; formal analysis, S.J.; investigation, S.J. and S.S.; resources, S.J.; data curation, S.J. and S.S.; writing—original draft preparation, S.J.; writing—review and editing, S.J., S.S. and Y.K.; visualization, S.J. and S.S.; supervision, Y.K. and C.L.; project administration, Y.K. and C.L.; funding acquisition, Y.K. and C.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Defense Acquisition Program Administration (DAPA) and Agency for Defense Development (ADD) (UD200006ED).

Acknowledgments: We thank Byoungjin Seok (Seoul National University of Science and Technology) for the discussions.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Case, A.; Richard, G.G., III. Memory forensics: The path forward. *Digit. Investig.* **2017**, *20*, 23–33. [CrossRef]
2. Halderman, J.A.; Schoen, S.D.; Heninger, N.; Clarkson, W.; Paul, W.; Calandrino, J.A.; Feldman, A.J.; Appelbaum, J.; Felten, E.W. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM* **2009**, *52*, 91–98. [CrossRef]
3. Ligh, M.H.; Case, A.; Levy, J.; Walters, A. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, 1st ed.; John Wiley & Sons: Hoboken, NJ, USA, 2014; pp. 189–217.
4. Kirmani, M.S.; Banday, M.T. Analyzing Detection Avoidance of Malware by Process Hiding. In Proceedings of the 2018 3rd International Conference on Contemporary Computing and Informatics (IC3I), Gurgaon, India, 10–12 October 2018; pp. 293–297.
5. Harris, R. Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. *Digit. Investig.* **2006**, *3*, 44–49. [CrossRef]
6. Gong, S.; Cho, J.; Lee, C. A reliability comparison method for OSINT validity analysis. *IEEE Trans. Ind. Inform.* **2018**, *14*, 5428–5435. [CrossRef]
7. Zhang, N.; Zhang, R.; Sun, K.; Lou, W.; Hou, Y.T.; Jajodia, S. Memory forensic challenges under misused architectural features. *IEEE Trans. Inf. Forensics Secur.* **2018**, *13*, 2345–2358. [CrossRef]
8. Advanced Configuration and Power Interface Specification. Available online: https://uefi.org/sites/default/files/resources/ACPI_6_3_final_Jan30.pdf (accessed on 12 January 2021).
9. Intel 64 and and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide. Available online: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html (accessed on 10 January 2021).
10. Blunden, B. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System: Escape and Evasion in the Dark Corners of the System*, 2nd ed.; Jones & Bartlett Learning: Burlington, VT, USA, 2012; pp. 103–110.
11. Stüttgen, J.; Cohen, M.I. Anti-forensic resilient memory acquisition. *Digit. Investig.* **2013**, *10*, S105–S115. [CrossRef]
12. Haruyama, T.; Suzuki, H. One-byte modification for breaking memory forensic analysis. In Proceedings of the Black Hat Europe, Amsterdam, The Netherlands, 14–16 March 2021.
13. Rekall Forensics. Available online: http://www.rekall-forensic.com/ (accessed on 20 October 2020).
14. Rekall. Available online: https://github.com/google/rekall (accessed on 15 January 2021).
15. Winpmem. Available online: https://github.com/Velocidex/WinPmem (accessed on 15 January 2021).
16. FTK Imager. Available online: https://accessdata.com/product-download/ftk-imager-version-4-5 (accessed on 15 January 2021).
17. Carrier, B.D.; Grand, J. A hardware-based memory acquisition procedure for digital investigations. *Digit. Investig.* **2004**, *1*, 50–60. [CrossRef]
18. Zhang, L.; Wang, L.; Zhang, R.; Zhang, S.; Zhou, Y. Live memory acquisition through firewire. In Proceedings of the International Conference on Forensics in Telecommunications, Informaion, and Multimedia, China, Shanghai, 11 November 2010; Springer: Berlin/Heidelberg, Germany, 2010; pp. 159–167.
19. Dolan-Gavitt, B. Forensic analysis of the Windows registry in memory. *Digit. Investig.* **2008**, *5*, S26–S32. [CrossRef]
20. Registry. Available online: https://docs.microsoft.com/en-us/windows/win32/sysinfo/registry (accessed on 24 December 2020).
21. CM_PARTIAL_RESOURCE_DESCRIPTOR Structure (wdm.h). Available online: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_cm_partial_resource_descriptor (accessed on 24 December 2020).

22. Windows-Driver-Framworks Wudfwdm.h. Available online: https://github.com/microsoft/Windows-Driver-Frameworks/blob/main/src/publicinc/wdf/umdf/2.17/wudfwdm.h (accessed on 24 December 2020).
23. Defeating Windows Memory Forensics. Available online: https://fahrplan.events.ccc.de/congress/2012/Fahrplan/events/5301.en.html (accessed on 14 November 2020).
24. Memoryze: Find Evil in Live Memory. Available online: https://www.fireeye.com/services/freeware/memoryze.html (accessed on 21 December 2020).
25. How to Stop Memory Acquisition by Changing One Byte. Available online: http://blog.rekall-forensic.com/2014/03/how-to-stop-memory-acquisition-by.html (accessed on 21 October 2020).
26. Latzo, T.; Palutke, R.; Freiling, F. A universal taxonomy and survey of forensic memory acquisition techniques. *Digit. Investig.* **2019**, *28*, 56–69. [CrossRef]
27. Understanding Symbol Files and Visual Studio's Symbol Settings. Available online: https://devblogs.microsoft.com/devops/understanding-symbol-files-and-visual-studios-symbol-settings/ (accessed on 10 January 2021).
28. What's Inside a PDB File? Available online: https://devblogs.microsoft.com/cppblog/whats-inside-a-pdb-file/ (accessed on 10 January 2021).
29. Update on Microsoft's Symbol Server. Available online: https://docs.microsoft.com/ko-kr/archive/blogs/windbg/update-on-microsofts-symbol-server (accessed on 10 January 2021).
30. Yosifovich, P.; Russinovich, M.; Solomon, D.; Ionescu, A. *Windows Internals, Part 1: System Architecture, Processes, Threads, Memory Management, and More*, 7th ed.; Microsoft Press: Hoboken, NJ, USA, 2017; pp. 82–83.