Identifying Volatile Data from Multiple Memory Dumps in Live Forensics

Frank Law, Patrick Chan, Siu-Ming Yiu, Benjamin Tang, Pierre Lai, Kam-Pui Chow, Ricci Ieong, Michael Kwan, Wing-Kai Hon, Lucas Hui

Chapter 13

IDENTIFYING VOLATILE DATA FROM MULTIPLE MEMORY DUMPS IN LIVE FORENSICS

Frank Law, Patrick Chan, Siu-Ming Yiu, Benjamin Tang, Pierre Lai, Kam-Pui Chow, Ricci Ieong, Michael Kwan, Wing-Kai Hon and Lucas Hui

Abstract One of the core components of live forensics is to collect and analyze volatile memory data. Since the dynamic analysis of memory is not possible, most live forensic approaches focus on analyzing a single snapshot of a memory dump. Analyzing a single memory dump raises questions about evidence reliability; consequently, a natural extension is to study data from multiple memory dumps. Also important is the need to differentiate static data from dynamic data in the memory dumps; this enables investigators to link evidence based on memory structures and to determine if the evidence is found in a consistent area or a dynamic memory buffer, providing greater confidence in the reliability of the evidence. This paper proposes an indexing data structure for analyzing pages from multiple memory dumps in order to identify static and dynamic pages.

Keywords: Live forensics, volatile data, memory analysis

1. Introduction

In recent years, there has been a growing need for live forensic techniques and tools [10]. Best practices have been specified to ensure that acquisition methods minimize the impact on volatile system memory and that relevant evidentiary data can be extracted from a memory dump [3, 12, 16]. However, most approaches focus only on a single snapshot of system memory, which has several drawbacks.

One of the most significant drawbacks is that dynamic activities cannot be detected and analyzed using a single snapshot of memory. Exam-

ples of dynamic activities include P2P file sharing and botnet communications. Investigators can uncover valuable evidence by analyzing the memory of processes corresponding to these activities. However, one of the major challenges in undertaking such an analysis is that memory allocation to processes is highly dependent on the system. Different programs often use different memory addressing schemes, causing great discrepancies in memory data structures. Consequently, data recovered from different portions of memory may require different interpretations. By classifying memory into static and dynamic regions, and mapping these regions to logical processes or files, investigators may be able to link evidence found in different portions of different dumps.

Using a single memory snapshot can bring into question the reliability of the extracted evidence and the veracity of the corresponding analysis. Multiple consecutive snapshots of memory can help address this issue. If the memory regions can be classified as static and dynamic, then evidence found in the static area would exist in several consecutive dumps and the integrity of the static evidence can be verified. Conversely, evidence found in a dynamic area can be correlated with other evidence by linking it to the corresponding logical process or file.

The complete analysis of multiple consecutive memory dumps is a challenging, multifaceted problem. This paper addresses one component of the larger problem: Given multiple consecutive snapshots of memory from a Windows system, how can static regions be efficiently differentiated from dynamic regions?

Solving this problem can help understand the memory structure of processes, but the solution is not as simple as it might appear. The memory dumps to be analyzed are huge (2 GB or more). Reading a single 2 GB dump takes more than 30 minutes; processing multiple dumps can take days. Additionally, the definition of the term "static" varies according to the memory processes analyzed and the time interval between consecutive dumps. Thus, the method should be flexible enough to answer which pages are identical in X consecutive dumps and which pages vary in all X consecutive dumps for different values of X. A brute force approach to scanning all the memory dumps for multiple values X would take far too much time.

In this paper, we assume that application programs employ a static memory allocation algorithm for their stack frames. Since memory is divided into pages (4 KB/page on a Windows platform), we adopt a hashing algorithm to assist in the identification of changes between memory pages. Then, we create an indexing data structure to store hash values so that by scanning the dumps just once, it is possible to efficiently

answer which pages are static (identical) or dynamic (different) in X consecutive dumps for different values of X.

2. Related Work

Given the limitations of conventional digital forensic techniques for dealing with volatile memory, the acquisition and analysis of machine memory are currently hot topics of research [5, 9]. Substantial research has focused on tools that can acquire memory images without altering memory content [7, 10, 11, 15]. However, the dynamic nature of memory means that obtaining a complete and consistent perspective of memory is impossible without taking multiple memory snapshots. In addition to problems posed by memory fragmentation, the analysis of data is complicated by the fact that memory structures vary considerably for different systems [16].

Microsoft Windows is the most common operating system encountered by digital forensic examiners. Much research has been directed at extracting relevant data from live Windows systems [4]. However, the closed source nature of Windows makes it difficult to verify the results, potentially increasing the likelihood of challenges when the evidence is presented in court.

Instead of focusing on data acquisition [4, 6] and memory object reconstruction [3, 12, 14], Arasteh and Debbabi [1] investigated the process memory stack. By analyzing the stacking mechanism in Windows memory, they were able to discover the partial execution history of a program in the memory process stack, which can be of value in forensic investigations. Chow, *et al.* [8] pointed out the possibility of differentiating static data corresponding to a UNIX memory process in order to identify useful data from inconsistent data in a memory dump. Balakrishnan and Reps [2, 13] analyzed memory accesses by x86 executables and proved the viability of distinguishing various regions of memory data created by executables. The heap and global data regions are areas where persistent data can be found. Balakrishnan and Reps also demonstrated the importance of understanding memory structures and of classifying static and dynamic data in memory dumps. In the following, we examine this issue in more detail in the context of forensic investigations.

3. Methodology

The problem studied in this paper can be stated as follows. Given K consecutive memory dumps, each containing N pages, and a sequence of m queries related to which pages have identical contents (static pages) in X consecutive dumps or which pages have different contents (dynamic

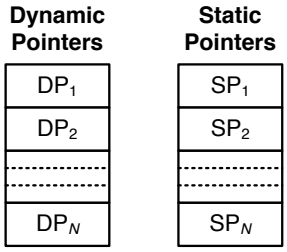

Figure 1. Data structure for the indexing approach.

pages) in all X consecutive dumps, then identify the page numbers and their corresponding dumps. Note that the value of X can be different in different queries in problem specification.

In the following, we assume that a hash value (e.g., MD5 or SHA-1) is computed and stored for each page. Comparing the hash values for a given page in different dumps identifies if their contents are identical or different.

3.1 Brute Force Approach

First, we describe a brute force approach in which the memory dumps taken at different times are read once for each query. We show how this approach is used to identify dynamic pages; static pages are identified in a similar manner.

The first page of each of the K dumps is read, and their hash values are computed and stored in an array. The array is scanned once: if X or more consecutive entries in the array are different, then the page is a dynamic page. This procedure is repeated for the other pages.

The brute force approach involves significant overhead for each query. In particular, considerable I/O time is expended to read all the memory dumps repeatedly. This problem is addressed in our indexing approach.

3.2 Indexing Approach

The indexing approach involves reading all the memory dumps only once and building a data structure so that queries can be answered efficiently. The data structure is essentially an array of linked lists as shown in Figure 1. Two linked lists are maintained for each of the N pages in the memory, the dynamic list and the static list. For Page i, the dynamic pointer DP_i and the static pointer SP_i point to the dynamic list and static list, respectively. The dynamic list pointed to by DP_i helps locate groups of dynamic dumps corresponding to Page i. Similarly, the static list pointed to by SP_i helps locate groups of static dumps for Page

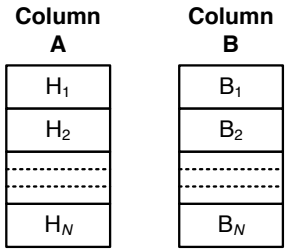

Figure 2. Working array for building the data structure.

i. Therefore, each node in a list – whether static or dynamic – denotes a group of dumps. In addition to pointers to the next node and the previous node, a count and starting position are stored for each node. The count gives the number of dumps in the group and the starting position gives the position of the first dump in the group. All the lists are sorted in descending order based on the count.

We now show how this data structure can handle a query for dynamic pages; the query for static pages is handled in a similar manner. The first node pointed to by DP_i is scanned once for each i from 1 to N. For each node with count C and starting position S, if C is greater than or equal to X, then Page i in Dump S with length C is a dynamic page. This procedure is repeated for the next node pointed to by DP_i until a node with count less than X is reached. At this stage, the procedure continues with the next dynamic pointer DP_{i+1} until the last dynamic pointer DP_N is processed.

A 2-D temporary working array is required to create the data structure (Figure 2). As Dump 1 is read, the hash value of each Page i in the dump is stored as H_i. When Dump 2 is read, the hash value of each Page i in this dump is compared with H_i. If the hash values are identical, then B_i is set to False, a new node is created with count equal to 2 and starting position equal to 1, and the static pointer SP_i is set to point to the new node. Otherwise, B_i is set to True, a new node is created with count equal to 2 and starting position equal to 1, and the dynamic pointer DP_i is set to point to the new node.

For Dump 3 onwards, the following steps are performed. Read Dump j. For each Page i in Dump j with $H_i^{'}$, compare $H_i^{'}$ with H_i. There are four cases.

- **Case 1:** If $H_i^{'}$ equals H_i and B_i is True, then create a new node with count equal to 2 and starting position equal to $j-1$, and insert the new node at the head of the list pointed to by SP_i. Set B_i to False.

- **Case 2:** If $H_i^{'}$ equals H_i and B_i is False, then increase the count of the head node of the list pointed to by SP_i by 1.

- **Case 3:** If $H_i^{'}$ is not equal to H_i and B_i is False, then create a new node with count equal to 2 and starting position equal to $j-1$, and insert the new node at the head of the list pointed to by DP_i. Set H_i to $H_i{'}$ and B_i to True.

- **Case 4:** If $H_i{'}$ is not equal to H_i and B_i is True, then increase the count of the head node of the list pointed to by DP_i by 1. Set H_i to $H_i^{'}$.

Finally, sort all the lists in descending order based on the count.

3.3 Time and Space Complexity

This section conducts an analysis of the time and space complexity of the two approaches. Let K be the number of memory dumps, N be the number of pages in each dump and m be the number of queries to be answered.

In the case of the brute force approach, an array of just K entries is needed; therefore, the space complexity is $O(K)$. For each query, all the memory dumps have to be scanned once; therefore, the time complexity for each query is $O(N \times K)$. Since m is the number of queries to be answered, the overall time complexity is $O(mNK)$. Note that all these operations involve I/O as the memory dump has to be read each time a query is answered. Consequently, the brute force approach requires $O(mNK/B)$ I/O operations where B is the I/O block size.

In the case of the indexing approach, for K memory dumps with N pages per dump, a 2-D working array with 2 columns and N rows is needed; therefore, the space complexity for the working array is $O(N)$. The data structure requires two lists per page, a total of $2N$ lists. In the worst-case situation, where the dumps are the same for every two pages, there are about $K/2$ nodes in each list. Thus, there are $N \times K$ nodes for the entire data structure and the space complexity is, therefore, equal to $O(N \times K)$.

Next, we consider the time complexity of the indexing approach. Building the data structure requires each memory dump to be scanned once, and this is the only step involving I/O operations. The time complexity for this step is $O(NK/B)$ I/O operations where B is the I/O block size. Next, all the linked lists have to be sorted; since there are at most $K/2$ nodes in each list, the time complexity for this step is $N \times 2 \times (K/2)\log(K/2)$. Thus, the overall time complexity is $O(N \times K \log K)$.

To answer a query, it is necessary to scan the first node in all the dynamic lists or in all the static lists. If L is the number of nodes that have to be scanned, then the time complexity is $O(L)$ for each query (these operations are much faster because no I/O operations are involved). The total time complexity for handling m queries is $O(NK \log K + mL)$. Note that in real-world scenarios, where L is much less than $N \times K$, the indexing approach is much faster than the brute force approach. In summary, the overall time complexity consists of two parts: $O(NK/B)$ I/O operations to read the memory dumps and $O(NK \log K + mL)$ RAM operations to handle m queries.

4. Experimental Results

This section compares the performance of the two approaches in terms of running time and running space. The two approaches were implemented in C++ code and executed on a Core2Duo P8400 2.26GHz computer with 4 GB RAM. The memory dumps used in the experiments were obtained from a video playback program, which was playing a video when the memory dumps were acquired. Obviously, memory dumps taken under such conditions are very dynamic.

The experiments involved three rounds. In Round 1, ten memory dumps were taken of the video playback program; each dump was about 59,660 KB. The two approaches were executed to identify pages with K consecutive dynamic dumps ($K = 5..10$). Therefore, a total of six queries were issued.

Table 1. Running times (Round 1).

	K = 5	**K = 6**	**K = 7**	**K = 8**	**K = 9**	**K = 10**
Brute Force	6.27 min	3.87 min	3.84 min	3.88 min	5.64 min	3.77 min
Indexing	2.68 min	1.13 min	1.03 min	1.06 min	1.07 min	1.04 min

Table 1 presents the running times obtained for the two approaches for various values of K. Note that the running times were measured using the C++ internal time library. The running space, which was obtained using the process manager, was 496 KB for the brute force approach compared with 504 KB for the indexing approach.

In Round 2, ten additional dumps were taken. As in Round 1, each dump was about 59,660 KB. Table 2 presents the running times for six values of K ($K = 15..20$). The running space was 584 KB for the brute force approach compared with 500 KB for the indexing approach.

Table 2. Running times (Round 2).

	K = 15	**K = 16**	**K = 17**	**K = 18**	**K = 19**	**K = 20**
Brute Force	10.03 min	12.94 min	13.82 min	15.74 min	17.70 min	19.94 min
Indexing	20.86 min	41.89 sec	40.89 sec	40.77 sec	40.83 sec	40.82 sec

Table 3. Running times (Round 3).

	K = 5	**K = 6**	**K = 7**	**K = 8**	**K = 9**	**K = 10**
Brute Force	154 min	160 min	166 min	170 min	162 min	180 min
Indexing	260 min	15 min	15 min	15 min	15 min	15 min

Round 3 involved ten larger memory dumps (about 2 GB each) to simulate the analysis of the entire memory dump of a computer. The two approaches were then used to identify pages with K consecutive dynamic dumps ($K = 5..10$). Table 3 presents the running times. The running space was 668 KB for the brute force approach compared with 516 KB for the indexing approach.

The experimental results show that the indexing approach requires significantly less time than the brute force approach except for $K = 15$ in Round 2 and $K = 5$ in Round 3. This is because, in order to answer the first query, the indexing approach has to build the data structure, which takes some time. However, the indexing approach uses the same data structure for subsequent queries. Consequently, in the later runs, the indexing approach is much faster than the brute force approach. Finally, as far as the running space is concerned, the two approaches require approximately the same amount of memory.

5. Conclusions

The indexing approach is designed to identify static and dynamic pages in multiple consecutive memory dumps. It is much faster than the brute force approach and uses about the same amount of memory

It is important to note that our approach is just the first step in the analysis of volatile data using multiple memory snapshots. Nevertheless, the approach could be refined and augmented to help identify static and dynamic data corresponding to a process in memory, track the address of the stack area, identify data that is consistently retained within memory, and link evidence related to the dynamic activities of users. These are all challenging research problems in their own right and are deserving of further investigation.

References

[1] A. Arasteh and M. Debbabi, Forensic memory analysis: From stack and code to execution history, *Digital Investigation*, vol. 4(S), pp. S114–S125, 2007.

[2] G. Balakrishnan and T. Reps, Analyzing memory accesses in x86 executables, *Proceedings of the Thirteenth International Conference on Compiler Construction*, pp. 5–23, 2004.

[3] Bugcheck, GREPEXEC: Grepping executive objects from pool memory (www.uninformed.org/?v=4&a=2&t=pdf), 2006.

[4] M. Burdach, An introduction to Windows memory forensics (forensic.seccure.net/pdf/introduction_to_windows_memory_forensic.pdf), 2005.

[5] M. Burdach, Digital forensics of the physical memory (forensic.seccure.net/pdf/mburdach_digital_forensics_of_physical_memory.pdf), 2005.

[6] M. Burdach, Windows Memory Forensic Toolkit (forensic.seccure.net), 2007.

[7] B. Carrier and J. Grand, A hardware-based memory acquisition procedure for digital investigations, *Digital Investigation*, vol. 1(1), pp. 50–60, 2004.

[8] K. Chow, F. Law, M. Kwan and P. Lai, Consistency issues in live systems forensics, *Proceedings of the International Workshop on Forensics for Future Generation Communication Environments*, pp. 136–140, 2007.

[9] D. Farmer and W. Venema, *Forensic Discovery*, Addison-Wesley, New York, 2005.

[10] G. Garcia, Forensic physical memory analysis: Overview of tools and techniques, Telecommunications Software and Multimedia Laboratory, Helsinki University of Technology, Helsinki, Finland (www.tml.tkk.fi/Publications/C/25/papers/Limongarcia_final.pdf), 2007.

[11] E. Huebner, D. Bem, F. Henskens and M. Wallis, Persistent systems techniques in forensic acquisition of memory, *Digital Investigation*, vol. 4(3-4), pp. 129–137, 2007.

[12] N. Petroni, A. Walters, T. Fraser and W. Arbaugh, FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory, *Digital Investigation*, vol. 3(4), pp. 197–210, 2006.

[13] T. Reps and G. Balakrishnan, Improved memory-access analysis for x86 executables, *Proceedings of the Seventeenth International Conference on Compiler Construction*, pp. 16–35, 2008.

[14] A. Schuster, Searching for processes and threads in Microsoft Windows memory dumps, *Digital Investigation*, vol. 3(S1), pp. S10–S16, 2006.

[15] I. Sutherland, J. Evans, T. Tryfonas and A. Blyth, Acquiring volatile operating system data: Tools and techniques, *ACM SIGOPS Operating Systems Review*, vol. 42(3), pp. 65–73, 2008.

[16] R. van Baar, W. Alink and A. van Ballegooij, Forensic memory analysis: Files mapped in memory, *Digital Investigation*, vol. 5(S), pp. S52–S57, 2008.