2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications

Assessing Overhead Cost Associated with Encrypting Swap File

B. AlBelooshi K. Salah T. Marin A. Bentiba
Department of Electrical and Computer Engineering
Khalifah University of Science, Technology and Research
Sharjah,UAE
Email: bushra.albelooshi@kustar.ac.ae

Abstract— **Privacy and security of information are two important concerns for most computer users. Passwords, keys, and encrypted information can be found unencrypted in the swap file which is used by the operating systems to support the implementation of virtual memory. Therefore, encrypting the swap file is essential to provide more security of users' private and confidential information. However, encrypting the swap file comes with an extra overhead cost. In this paper, we measure the overhead cost associated with encrypting the swap file. To effectively measure this cost, we developed our own benchmarks that will enforce heavy swapping with disk write and read operations. We measured the overhead cost for Windows 7 operating system. In our measurements, we considered a number of popular encryption algorithms which include AES, Blowfish, Twofish, and GOST. Our experimental measurements show that Windows 7 incurs considerable overhead penalties when encrypting the swap file.**

Keywords: Swap file, Encryption, Operating system security, virtual memory

I. INTRODUCTION

Nowadays, most operating systems support two spaces to run processes. The first is the physical main memory which is the computer RAM. The second is the virtual memory that is used to manage the processes along with the physical memory. Virtual memory is "a storage allocation scheme in which the secondary memory can be addressed as though it were part of the main memory" [1]. When the required amount of memory space needed to execute a process exceeds the available physical memory then the swap space or swap file is used. The operating system creates a large space in the hard disk, known as the paging file (or swap space) which holds memory pages. The name of this file in the Windows operating system is pagefile.sys. The main memory moves some of the inactive process pages to the paging file in order to free up memory. This mechanism is called swapping, which is an implementation of the concept of virtual memory [2].

The swap file is a reflection of the running state of the machine, with unencrypted swapped-out main memory pages. When forensically analyzed, the swap space pages can reveal critical keys and passwords. Therefore, it is important to secure the swap file in order to protect this sensitive information. This can be achieved by encrypting the swap file. As shown in Figure 1, swapped out pages first get encrypted using "crypto swap" algorithm before being written to the disk. When reading encrypted swapped pages, by performing memory read operations, the encrypted swapped pages have to be first decrypted before loading them into main memory. This process will definitely secure swapped out pages, but there is a performance penalty associated with encrypting and decrypting the swapped pages. The main contribution of this research is to assess the cost overhead associated with swap file encryption. This overhead cost can significantly slowdown running applications and services. In this paper, we attempt to quantify such a cost. The cost is measured for different encryption algorithms. Such results will be of a great benefit in helping the end user choose how to protect their data while making the minimal impact on performance.

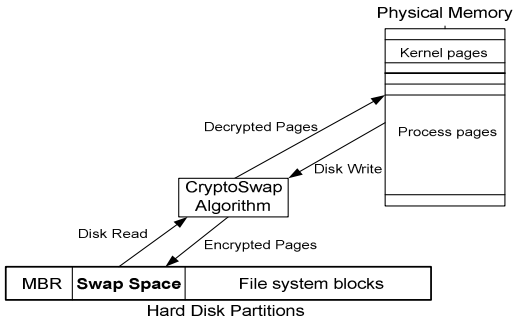

Figure 1. Encrypting swapped pages

The rest of the paper is organized as follows. Section II describes related work. Section III explains how we implemented the experiment to measure the cost overhead generated from swap encryption. Section IV describes the testing and validation process. In section V, we represent the results generated from our experiments. Finally, we conclude in section VI.

II. RELATED WORK

There has been little research conducted in the area of measuring the overhead cost associated with encrypting the swap file. The two main bodies of research which are relevant to this topic are "Encrypting Virtual Memory" [3] and "Efficient Security-Aware Virtual Memory Management" [4]. Both papers had limitations that will be addressed by this research.

Encrypting Virtual Memory was proposed by Niels Provos from University of Michigan. In 2000, Provos suggested encrypting the swap file when pages are being swapped out from memory and decrypting the pages when they are brought back to the physical memory. He suggested dividing the backing store into 512 Kbyte sections. Each section has its own key which consists of 128-bit encryption key, reference counter and the expiration time. The encryption key is generated

978-0-7695-4745-9/12 $26.00 © 2012 IEEE
DOI 10.1109/TrustCom.2012.96

randomly at the first time, while the counter is set to zero and incremented every time the page is encrypted with its encryption key. The reference counter is decremented every time the page is freed from the backing store. Once the counter reaches to zero, the key is deleted. Therefore, all the data encrypted with that key will not be decrypted and will be erased. Once the expiration time is reached, all pages with that reference should be re-encrypted by the new key. The key is stored in the unmanaged part of the kernel so that it will not be swapped out.

The encryption effect on the system performance was then evaluated using a test program that allocated 200 Megabyte of memory and filled the memory sequentially with zeros. After memory allocation, the program read the whole memory sequentially. Provos noticed that there was an increase in the run time when encryption was enabled [3].

One of the main limitations of this research is that it was implemented using one encryption algorithm, namely AES. The implementation was also done on obsolete hardware with 128 MB of main memory and a 333 MHz Celeron processor.

The second relevant work was the Efficient Security Aware Virtual Memory Management that suggested encrypting confidential pages and segments of the process being swapped out. Rather than encrypting the whole swap file, this paper proposed encrypting only the confidential parts. This would reduce the overhead cost associated with encryption by minimizing the amount of cryptographic operations. The idea of this approach was based on adding a security label to each process. The page fault handler then uses this label to decide whether to apply the encryption on the process pages.

The researchers then analyzed the performance overhead generated from their approach. They developed a C++ simulator which showed that the performance of their solution is acceptable and can be used as a solution [4].

One of the main limitations of this paper is that it will add an overhead on the process owner side. The process owner in this case has to decide which portions of the process should be encrypted and which should not. Moreover, an overhead will be added because of the page fault handler which checks the security label for each process and decides whether to apply the encryption or not.

III. IMPLEMENTATION

This section describes the implementation method that we adopted to measure the overhead cost associated with encrypting the swap file. The section explains the different encryption techniques used for the purpose of this research. It will then list the platform specifications of the PC on which the test was performed. It will also present the developed benchmark for assessing and measuring the overhead cost associated with encrypting the swap file.

A. Techniques of Encrypting the Swap File

There are different techniques by which the swap file can be encrypted. This includes the operating system standard utilities or using the available encryption tools. In Windows, the page file can be encrypted using the Encryption File System (EFS) technique. There are many other tools that are available that feature a swap file encryption utility as well. The most popular tools are Crypto-Swap, True-Crypt and Best-Crypt. For the purpose of this research, Crypto-Swap with the different encryption mechanisms available was used. The main encryption algorithms supported by Crypto Swap are AES (Rijndael) 256-bit key, Blowfish 448-bit key, GOST 256-bit key and Twofish 256-bit key [7].

B. Platform Specifications

The testing was implemented on a 32-bit operating system with 4 GB RAM. The system had four Intel Xeon processors running at 2.8 GHz each. The system was running the Windows 7 operating system.

C. Developed Benchmark

Due to the lack of benchmarking software, a program was written in the C programming language. The objective of the program was to stress the memory and therefore force page file swapping. The program allocates 2 GB of memory and fills it up sequentially with any letter, in our case we were filling it up with 'z'. It will then iterate writing to memory five times over the allocated buffer to overwrite the contents of the swap file and force swapping. Another program was also developed that aims to read sequentially from memory.

As shown in Algorithm 1, Lines 1-4 define the input parameters. The *MAX* constant represents the maximum buffer size which is 2 GB in this benchmark. The 2GB of memory is divided into 1 MB blocks that is (1024x1024) bytes. Therefore the total block size dividing the 2GB by 1 Mb is equal to 2000 blocks. As shown in Line 7, the buffer gets initialized with 'z' letter in a while loop. After that in Lines 14-20, the memory pages are writing with different values to force writing operations of process memory pages to the hard disk. The time incurred is recorded with TotalT, and the average is calculated for each run. In our experiments, we record the average of 5 runs. Similarly, a disk read benchmark is constructed but with read operations.

Algorithm 1: Disk Write Benchmark

```
Input:
1  MAX ⟵ 2000
2  b[MAX] ⟵ Memory buffer with the maximum size of 2GB
3  T0 ⟵ Initial time
4  TotalT ⟵ Total time

6  Begin:
7    while ((b[mb]=malloc(1024*1024)) != NULL && mb < MAX) {
9                  t0 = current time
10                 b[mb] = 'z'
11                 TotalT = current time – t0 + TotalT
12                 mb++
13   }
14   for (j=0; j<5; j++) {        // Force swapping by re-writing pages
16       for ( i=0; i<MAX; i++) {
18                 t0= current time
19                 b[i] = 'b'+i+j
20                 TotalT = current time – t0 + TotalT
21       }
22   }
23 End
```

IV. Testing and Validation

The first step was to verify that the developed code was working properly and inserting specific values into the page file. That was done by updating the writing to memory benchmark to write a specific character into the memory, in this case letter 'b'. After that, two instances of the program ran concurrently for several iterations. The swap file was then captured using the FTK imager tool. A hex editor tool was used to read and analyze the page file. It was shown that the letter 'b' occured 3,056,195,533 times in the file. That represented 81.71% of the total file bytes. The second most frequent letter was '0h' which represented only 10.10% of the total bytes in the page file. This proved that the program was inserting letter 'b' in the memory and was being swapped out after running the concurrent benchmarks several times.

It was also verified that the data in the swap file is being encrypted when NTFS or crypto swap encryption was enabled. That was done by enabling the swap file encryption utility and running the writing to memory benchmark several times. It was verified that the data in the swap file was not legible. Moreover, all the bytes had close to the same frequency. This proved that the swap file encryption was being applied.

V. Results and Discussion

To evaluate the performance overhead, two instances from the benchmark were running at the same. The observation was then repeated ten times and the average overhead was calculated for each encryption algorithm. This section describes the generated results in more detail.

A. Writing to Memory Results

The writing to memory benchmark was run ten times and the results were measured. Figure 2 shows the average time required to write to memory while applying the different encryption mechanisms on the swap file. The program took more time to be executed when the encryption was enabled. It took the longest time when the encryption used was AES. Moreover, it took the least time with Blowfish encryption. In ranking terms, the AES encryption was taking the highest execution time, then GOST, NTFS, Twofish and the least execution time was with Blowfish encryption.

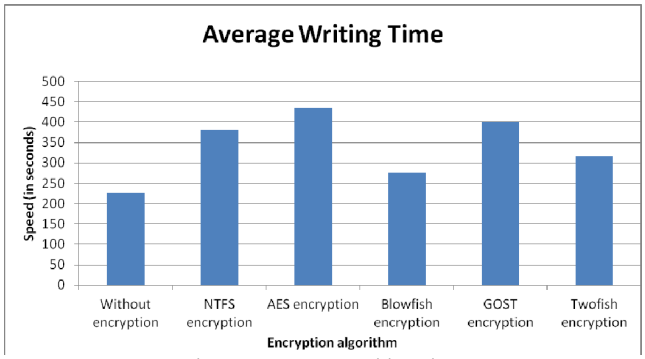

Figure 2. Average Writing Time

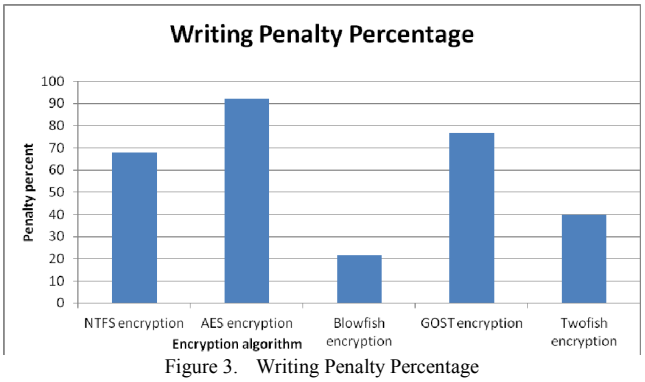

Figure 3. Writing Penalty Percentage

The results generated agreed with the expectation for some of the encryption algorithms. As was stated in Jetico website [5], Blowfish is faster than GOST and the results we got is showing that the program takes less time to execute with Blowfish than with GOST encryption. Furthermore, in the paper "Performance Comparison of the Five AES Finalist", it was shown that AES-256 bit key is slower than Twofish-256 bit key [6]. This explains the result we got when running the program when Twofish encryption was used.

Figure 3 depicts the penalty associated with writing with encryption. The formula to compute this penalty is expressed as follows:

$$Penalty\ (\%) = \frac{AvgWritingWithEncryption - AvgWritingWithoutEncryption}{AvgWritingWithoutEncryption} \times 100$$

This measures the penalty or overhead generated after applying the encryption algorithm compared to running the benchmark without encryption. As shown in Figure 3, the writing time was increased by about 91.9% when AES encryption was applied. The least penalty generated was when Blowfish was used causing 21.5% increase. The overhead generated from GOST and NTFS encryptions is considered high which was above 65%. Furthermore, Twofish encryption overhead was 40% more than running the benchmark without encryption.

B. Reading from Memory Results

The reading from memory benchmark was also executed ten times to measure the penalty issued from each encryption mechanism while reading from the memory. Figure 4 shows the average time needed to read from memory while applying different encryption algorithms. It is shown that there is little difference while applying the different encryptions compared to the differences generated from writing to memory benchmark. AES took the most time to read from memory which was about 53 seconds. Blowfish was again taking the least time to read from memory which was about 49.3 seconds. The ranking of the average time needed to read from the memory was the same as with writing to memory except for the NTFS encryption which was higher than GOST encryption. Thus AES took the most time then NTFS, GOST, Twofish and finally Blowfish.

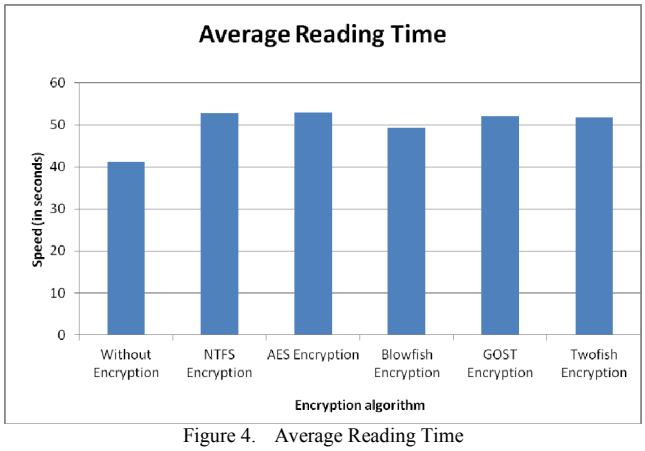

Figure 4. Average Reading Time

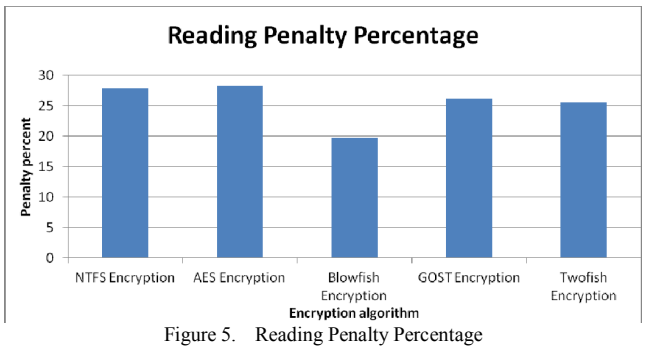

Figure 5. Reading Penalty Percentage

As shown in Figure 5, the highest overhead was associated with AES encryption that was 28.3%. After that was the overhead associated with applying NTFS at 27.8%.Then the GOST encryption penalty at 26.16%. The least overhead was associated with Blowfish and Twofish encryptions at 19.7% and 25.5% respectively.

In general, the time to read from memory was less than the time to write to memory. That was due to the fact that writing to the memory benchmark was updating the pages that were being swapped out from the memory. This will affect the modify bit attached to the page. The modify bit is used to indicate whether a page in memory has been modified since it was last swapped out of memory. If that bit is not set (page is not modified), then the page is just discarded without re-writing it and it is replaced with the new page. If this bit is set which means that the page has been modified, then it must be written back into the swap space [2]. In writing to memory, this bit will be set. Therefore if the page needs to be replaced then it should be written back into the swap space before replacing it. In reading from memory, the benchmark was just reading the data from the memory without updating the values of the pages. Thus the modify bit was not set and the page could be discarded without re-writing it in the swap space. This explains why reading from memory was taking less time than writing to memory.

C. *Performance Monitoring Results*

Two main applications were used to monitor the performance while executing the benchmark in Windows 7. The objective of these tools was to monitor the memory performance and to make sure that it was updating normally as predicted. The first application used was Windows Task Manager. The performance was monitored before and during execution of the benchmark. The below figures were captured using the Task Manager tool.

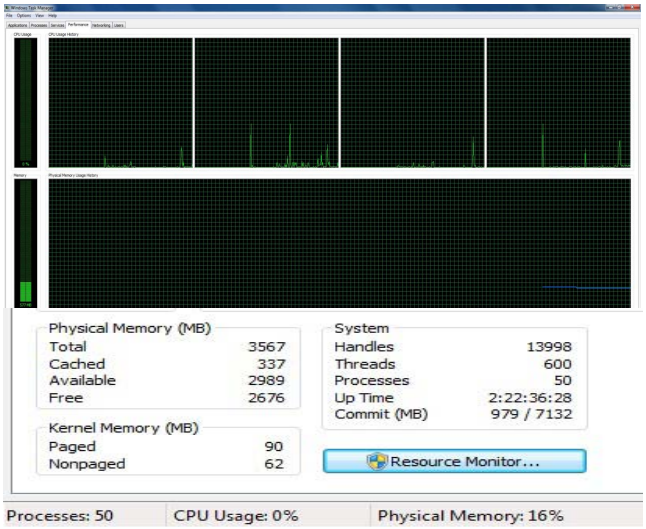

Figure 6. Memory performance prior to running the benchmark (captured from Task Manager)

As shown in Figure 6, there were fifty processes running prior running the benchmark. These were consuming 16% of the total physical memory. The CPU usage was almost 0%. While the benchmark was running, as shown in Figure 7, the physical memory usage increased to 97% with 53 processes. Those three processes were related to the benchmark execution. The CPU usage also increased to 16% during execution. The commit memory prior benchmark execution was 979 MB compared to 4774 MB while running the benchmark. The commit value represents the size of virtual memory that is in use by all processes [8].

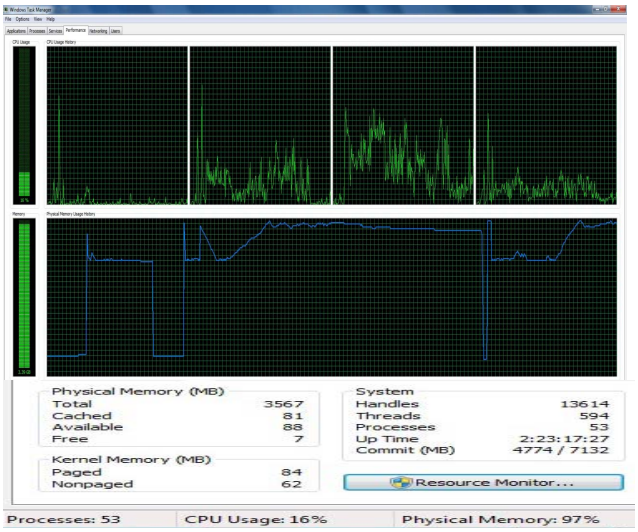

Figure 7. Memory performance while executing the benchmark (shown by Task Manager)

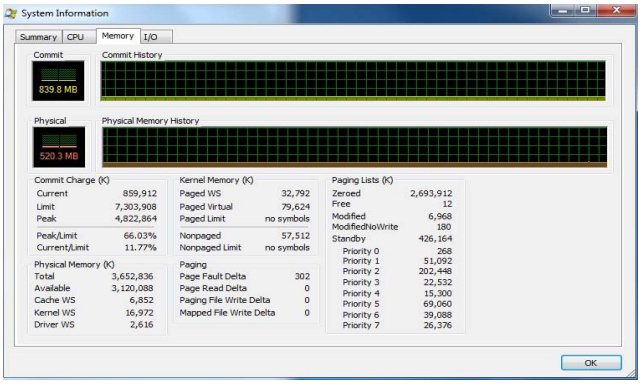

Figure 8. Memory performance prior to running the benchmark (shown by Process Explorer)

The second application used to monitor the memory performance in the Windows operating system was the Process Explorer. Below are the snapshots captured from Process Explorer before and while running the benchmark with and without enabling the swap file encryption.

Figure 8 shows that the current available memory was 3,120,088 kilo bytes and the total available memory was 3,652,836 kilo bytes prior benchmark execution. The total used memory represents 14.58% of the total physical memory. Page fault delta, that represents the difference between the page faults, was 302. The commit memory at that time was 859,912 kilo bytes. On the other hand, after executing the benchmark the available memory was reduced to 18,120 kilo bytes that represent 99.51% of the total memory as shown in Figure 9. The commit memory at that time was 4,792,872 kilo bytes. Therefore the commit memory increased from 11.77% to 65.62% after executing the benchmark.

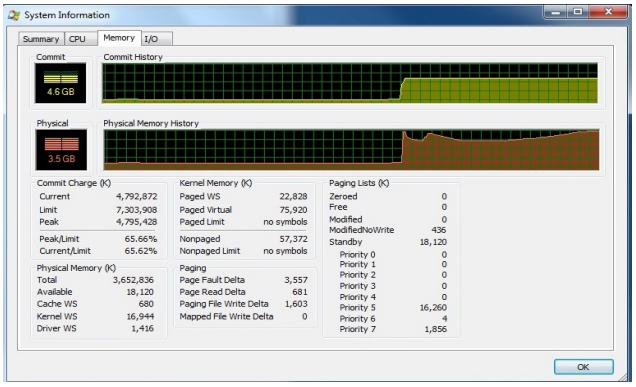

Figure 9. Memory performance while running the benchmark (shown by Process Explorer)

VI. CONCLUSION

In this paper, we have measured the overhead cost associated with encrypting swapped-out pages. We have considered a number of popular encryption algorithms which include NTFS encryption, AES, Blowfish, TwoFish, and GOST. Our results show that a considerable overhead penalty can be incurred when encrypting the swap space. Different encryption algorithms have generated various overhead costs. For both write and read operations, AES encryption produced the highest overhead cost; whereas Blowfish produced the least overhead cost. Our measurements in the paper were conducted for Windows 7 platforms. As a future work, we are in the process of assessing the overhead cost associated with encrypting the swap space under Linux 2.6.32 platforms.

REFERENCES

[1] Stallings, W. (2009). Virtual Memory. *Operating systems: internals and design principles* (6th ed., internat. ed., p. 346). Upper Saddle River, NJ: Pearson/Prentice Hall.

[2] Silberschatz, A., Galvin, P. B., & Gagne, G. (2009). Virtual Memory Management. *Operating system concepts* (8th ed., pp. 357-359). Hoboken, NJ: J. Wiley & Sons.

[3] N. Provos, "Encrypting Virtual Memory," In Proceedings of the Ninth USENIX Security Symposium, pages 35-44, August 2000.

[4] R. Amirsoufi, M. Taghiloo and A. Ahmadi, "Efficient Security-Aware Virtual Memory Management," In Proceedings of 2009 International Conference of Soft Computing and Pattern Recognition, pages 208-2011, December 2009.

[5] "Encryption Algorithms." Jetico. Web. 10 Nov. 2011. <http://www.jetico.com/bc8_web_help/html>

[6] Schneier, Bruce and Doug Whiting, "A performance comparison of the five AES finalists," April-2000.

[7] "Swap File Encryption Utility (CryptoSwap)," Jetico - Military-Standard Data Protection Software - Wiping, Encryption, Firewall. Retrieved March 14, 2011, from http://www.jetico.com/bc8_web_help

[8] "Effective use of Task Manager," (n.d.). Express Computer. Retrieved October 4, 2011, from http://www.expresscomputeronline.com/