# BMCOW:An approach to modify shared page

Binay Kumar
Department Of Computer Science and Engineering
Central University of Rajasthan
Kishangarh, Rajasthan, India
Email: 2014mtcse008@curaj.ac.in

*Abstract*—**Copy on write technique to modify a shared memory page has been a novel idea by keeping the original page unchanged and modifying a duplicate copy of it. The modifiable content in a page can range from one byte to a complete page. The process which owns the page to be modified is kept suspended until the whole page is copied down. Page copying is done without caring the size of modifiable content. Process suspension and page content copying take further time. Also minor page fault is raised as further activity. COW technique , in virtual machines , to save redundant physical memory space is resulting today in memory disclosure attacks. Our work BMCOW as bytes modification & COW modifies shared pages in a way such that COW works on only unreferenced bytes of the pages and thus it avoids extra time of modification process. This avoids any known timing based attacks on virtual machines' memory.**

Keywords—**Memory deduplication, Copy on write, Shared page modification, Memory disclosure attacks.**

## I. Introduction

Computer memory is an array of bytes. These bytes have their own independent addresses[1]. So, we can say that memory content is byte addressable. Under various memory management techniques, paging is an efficient and most widely usable one. This technique divides memory into fixed size of blocks called pages. Memory pages have their own addresses called page numbers. Bytes in pages have their own addresses called page offsets. In combination page number and page offset form unique address of a byte in memory. A byte content is accessed using this address. Although in an another way, in modern computer, memory address is viewed in two forms logical address and physical address. Address generated for an operand by CPU after fetching instructions from programs is called logical address and this address is different than that which can access a memory byte. The address which lets a memory byte accessible is physical address. Logical address is translated into physical address by a device called MMU(Memory Management Unit). A page table is used by the MMU to perform this translation. Contents of memory pages are either program instructions or data. A process has its own logical address space. A process' contents in terms of instructions or data are stored in physical memory in pages but CPU access them by generating logical addresses. Contents of memory pages may or may not be identical. But in practical it has been seen that two or more memory pages may have same content when there execute two or more concurrent processes in a computer. We say that such pages with same content are duplicated ones. Occupied space by duplicated pages have been a major concern for computer science researchers[2]. The same situations have also been found in cloud computing. A hypervisor on a cloud server executes concurrent virtual machines. A virtual machine has its own virtual appliance which consists of operating systems and other applications. Hypervisor manages physical memory and provides virtual memory space to the virtual machines. Again these virtual machines have their own virtual address space. A byte in virtual address space is addressed by virtual address. This virtual address is generated by a virtualized instance of CPU called VCPU. So there is the situation of memory page duplicacy in virtual memory or cloud computing also.

Memory deduplication[3] as a solution to the above problematic situation is to keep only a single copy of duplicated memory pages. But it is cared that the kept single page must be accessed by all owner processes or virtual machines of the duplicated pages. Here kept memory page of the duplicated ones are termed as shared page. Owner processes or owner virtual machines are allowed to read such pages but not to modify. However, Copy on write[2] as a mechanism in operating system and transparent page sharing[4, 5] as memory sharing technique for virtual machines are implemented to modify shared memory pages. As positive side of these mechanisms support cloud computing or operating systems to execute several processes concurrently in minimum amount of memory space but the negative side is its vulnerability for security attacks. Attackers notice time difference between modifying a shared and a non-shared memory page. This time difference is sufficient to notice use of duplicated pages. In cloud computing memory based attacks referring to this time delay have been observed[6]. Such memory disclosure attack causes multiple attacks and it is dangerous even if the network is encrypted by TLS/SSL[6].

We propose our idea to modify shared memory pages in a way that is based on bytes wise modification rather than page wise modification. We analyze that time taken in modifying a local page and a shared page is closer. This can avoid timing attack on virtual machines' memory. BMCOW still requires implementation so it is raw. Section II is the survey of memory deduplicacy, copy on write, transparent page sharing and suggested solutions for memory disclosure attacks. We define observed problems in section III. Approach and processes of BMCOW are given in section IV and V respectively. Later on we analyze our works in section VI whether they solve the defined problems or not. Finally we conclude and give the future scope of our approach in storage deduplicacy in section VII and VIII respectively.

## II. Literature Survey

Memory deduplication[3] is a technology to free allocated copies of memory blocks with identical contents by keeping one of them. The blocks' content range from bytes to pages. Memory blocks are of two types on the basis of sizes: dynamic and static size. Static sized blocks are popularly used as pages having 4KB size. For memory deduplication, every page of each virtual machine is compared with each other. The comparison is done by either byte-by-byte matching or by taking fingerprints of blocks when block size is constant. Once it is discovered that two or more memory pages have same fingerprints,  the whole content is matched for confirmation. One of such page is kept and others are removed to free allocated space by duplicated pages. Page table of participating virtual machines are updated by setting copy on write bit to 1 and replacing page number pointer with the address of kept page. Consequently the kept page is available as read only and shared among those virtual machines which have duplicated copies of it.

For keeping memory deduplication operatable copy on write[4, 1] is probably the first accepted technique to modify shared memory pages in an efficient way but today it causes some memory based attacks. Copy on write creates a duplicate of a shared memory page when kernel discovers that a minor page fault has been raised because of an attempt to modify the shared page. The shared page is replaced with this duplicated page entry in the page table of the page modifier process and then the process resumes modification in this new page not in the shared one. This technique takes further time in order to raising and handling minor page fault , copying entire memory bytes in the new page , suspending the modifier process and resuming it. The extra time taken is noticeable and causes the memory based attacks.

ESXi[5] as a hypervisor performs transparent page sharing to reclaim memory space on virtual machines. ESXi scans memory pages of guest virtual machines by comparing hash values of pages with available hash values in a global hash table. It removes a page if its hash value is matched in the global hash table otherwise the page is kept and new hash value is added . The kept page is shared with processes whose page's hash values are same as hash of the kept page. The hypervisor implements copy on write technique to let users of virtual machines for modification of shared pages by creating their duplicate copies.

Kuniyasu Suzaki et. al.[6] work suggests memory deduplication operatable only for read only pages. As read only pages will never be modified copy on write will never be required and resultantly no attacks on memory will exist. This work does not appear feasible in real life's user activities as users and operating system have many important deduplicable files that can require modification any time.

Jidon Xiao et al.[7], to defend against a specific memory attack called covert channel attack, suggest to not deduplicate all memory pages in one time. They deduplicate and share memory pages randomly. This results in giving less information to covert channel attacker about all shared pages. Complete use of memory deduplication does not appear here as this work does not deduplicate all the identical memory pages. Also the countermeasure is limited to only defend covert channel attack.

## III. Problem Definition

The copy on write technique enables an owner process to modify a shared page. This includes some extra activities (1) Raising a minor page fault. (2) Suspend the process and handling the page fault. (3) Allocating a new page (4) Copying the shared page into a blank page. (5) Updating the process page table. (6) Resume the suspended process[1, 2, 8]. These all steps take plenty of time , experimentally it has been seen that modifying one byte in a shared page takes 5 microseconds on an average[6]. This says that time to modify a shared page and a local page are not closer to each other.

If the page is shared, unexpectedly whole page of the addressed byte is copied down rather than copying only the addressed byte. This avoids redundant copying, if CPU references next bytes of the same page. Also minor page fault and other corresponding activities can be avoided to incur repetitively for the same page. But extra time is consumed in order to copy the whole shared page. Copying time of all pages are constant, it doesn't matter whether the process require partial or full modification in the page. Also, It is common to say that modification in a page may vary from one byte to the whole page in size. Resultantly a process remains suspended for an extra time. Again in other view, 5 microseconds is much sufficient for a user to notice whether the byte of a page which is being modified is shared or local. This makes users aware that the shared page is also available in other virtual machines on the same hypervisor. Consequently memory content of virtual machines is revealed to users[6, 9, 10].

Possible attack to reveal memory content is termed as memory disclosure attack. Other variations of this attack have also been seen.

We observe some concerns with copy on write as following

1. Raising minor page fault.
2. Suspending a process.
3. Copying whole shared page
4. Extra time taken to modify a shared page than the time taken in modifying a local page.
5. Memory based attacks.

## IV. Proposed Work

We propose our approach to modify a shared page which performs modification only on addressed bytes and do copying only of the unaddressed bytes. This is unlike copy on

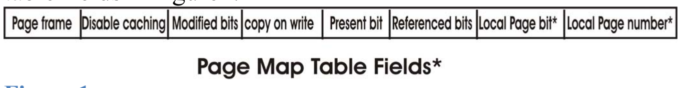
write which copies all the bytes first then modification. Our approach does it by reading bytes to be modified from the shared page , modifying the content , and writing the modified content to a new page, After modification done , in background , a daemon process copies the remaining bytes of the shared page into the newly allocated memory page. This part is made invisible to users, resultantly users can't notice the time taken by the daemon process. Offsets of the bytes in both the pages are kept same. i.e If CPU references i$^{th}$ byte of a p$^{th}$ shared page then the byte must be copied or written at the i$^{th}$ offset in the n$^{th}$ new local page of the modifier process.
We add two extra fields in page table of processes and introduce a new data object as word table which we describe in following subsections. Once it is discovered that users want modifying a shared memory page, our work allocates a new page and assign its number in local page number field* of modifier process. This update is done in the field of shared page entry only. This makes clear to the modifier process that a local page is ready for the shared page. Modified contents can be written into this page. Also a daemon process[8] copies unaddressed bytes into this local page. Here we first introduce our new structures and then give the complete methodology to modify a shared page. The modification operation involves insert, delete, change. The proposed work gets advantage using Page Map Table, Local page bit* , Local page address* , Word Table* , Daemon Process .

### A. *Local page bit**

This is a binary data type field in a page table. Value 1 in this field indicates that a local page has been created for a shared page. The shared page is the one which is referenced to be modified. During modification each time when CPU refers to a byte , this field is checked for getting confirmation whether to allocate a new page or use already allocated new page to write modified content into.

### B. *Local page number**

An unsigned integer type field in a page table, this contains number of a page. It is updated when a new page is allocated in response to modify a shared page. Value of this field is used when it is discovered that Local page bit* is set. i.e no more new page should be created to write modified/copied bytes content. In other words, a modifier process checks Local page bit* for 1 and then writes the modified bytes of a shared page into the page referred by the value of this field. We show page table fields in figure1.



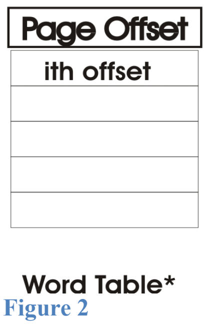
| Page frame | Disable caching | Modified bits | copy on write | Present bit | Referenced bits | Local Page bit* | Local Page number* |
|---|---|---|---|---|---|---|---|

Page Map Table Fields*

Figure 1

### C. *Word table**

Word table*, shown in figure 2, is introduced as a new data structure and it is used by each process separately. The table consists of only one field. The field records page offsets whose content have been modified. This is populated when modifier process modifies shared page bytes and then it is used by a daemon process to know the offsets of modified bytes.



| Page Offset |
|---|
| ith offset |
| |
| |
| |
| |

Word Table*

Figure 2

### D. *Daemon process*

A background process[8] which runs continuously and gets job to copy unreferenced bytes of a shared page to a new page. This process is assigned with its job when it is discovered that all the required bytes of a page have been modified. The fact that no more bytes are required to be modified is identified when CPU refers another page than the shared page for modification.
The daemon process starts from beginning page offset of the shared page. It checks for the unavailability of shared page offsets in the word table* and then copies the content in a new page at the same offsets. It also , in final , removes all the records of word table* and removes the record of shared page entry from page table unlike copy on write which replaces the shared page number with new page number. Daemon process operations are depicted in figure 3.

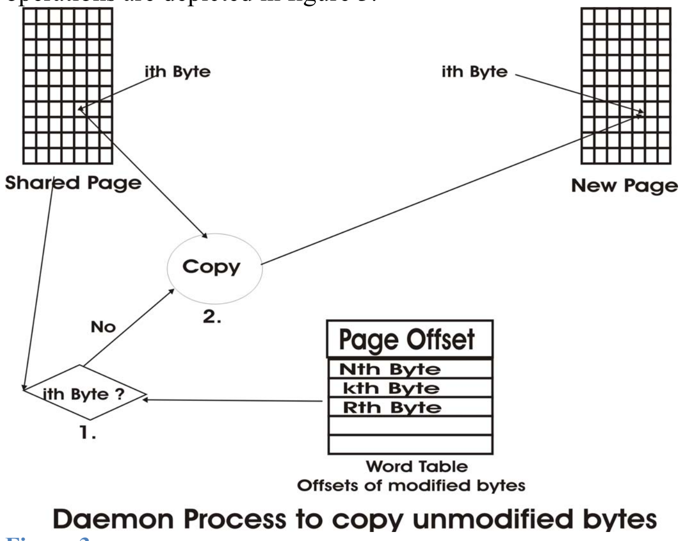


Figure 3

### E. *Methodology*

Once it is confirmed that a page is required to be modified, the copy on write bit in page table of the modifier process in the record of shared page address is checked whether it is set. Again newly introduced binary bit Local page bit* for the same record is checked for 0 to get confirmation whether to

allocate a new page to write into. A new blank page as local page for the modifier process is allocated. Number of this newly allocated page is written in the Local page number* field. If the Local page bit* is 1 then it is interpreted that a new page is already allocated so the Local page number* is read to get already allocated local page number. CPU reads content of a referenced shared page byte, modifies as it is required and writes the modified content in the local page. The page offset of the referenced shared page's byte which has been read and the page offset where modified content is written in the new page are kept same. Word table* of the modifier process is populated with the same page offset. The procedure continues till CPU references the same shared page. To copy remaining unmodified bytes of the shared page into the new page , daemon process starts from the first byte of the shared page. It checks for its offset unavailability in the Word table* and then it writes the exact content in the new page. Again, the page offset of the shared page's byte which has been copied and page offset where copied content is written in the new page are same. This finishes modification process with the entry for shared page is deleted from the page table of the modifier process. Also word table is made empty. Figure 4 shows modification operations.
In all our methodology we describe three jobs to complete the process.
1. Allocating a new page
2. Writing modified bytes in the allocated new page.
3. Copying remaining unmodified bytes in the new page from the shared page.

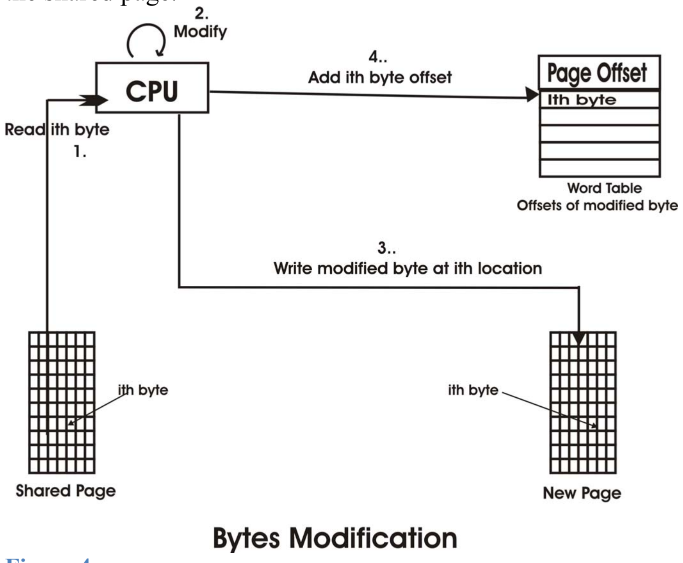


Figure 4

*F. Different operations*

Users perform different operations on shared memory pages Read, Insert, Delete, Change. Read operation does not involve above methodology as it does not modify page. Insert, Delete and Change operations require modification which we describe as below.

*1) Insert*

For a shared page which is partially populated, insert operation is required to store contents at the offsets with null values. In other words, insert operation adds contents in a page when it is partially empty.
For this operation a page table is checked whether the copy on write bit is set. If it is true then a new blank page is created. Inserted contents are written into this page. The page offset is kept same as CPU references it for the shared page Word table* is populated with these offsets. After insert, daemon process copies the remaining content of shared page in this new page at the respective page offsets.

*2) Delete*

Deleting contents in memory refers to erasing data from some page offsets of a page. To delete from a shared memory page, a new memory page is allocated. Word table* is populated with the referenced page offsets. After delete a daemon process copies only those bytes of the shared page into the new blank page whose offsets are not available in the word table*.
Since the new page does not contain erased content so it is considered that data have been deleted.

*3) Change*

Change refers to replace content at some page offsets with other content. A new blank page is allocated. Content from referenced shared page offset is read, modified as it is required then Changed data is written at the same page offsets of new allocated page. Word table* is populated with these page offsets. A daemon process copies only those bytes of shared page whose offsets are not available in the word table*.

## V. Proposed Process

We illustrate our work here in mode of operations and steps. We have considered that in the course of instruction execution, a shared memory page modification is required. We have also considered that Local page bit*, Local page number* as extra fields in the page table and Word table* for a process are available in operating system/hypervisor.

### *A. New Page Allocation*

It is considered that modification instruction has been incurred. Every fields of page table here correspond to record of shared page.

```
1. If copy on write bit ==1 then
       If Local page bit == 0 then
I. Allocate a new page N.
II. Local page number = #N. // # denotes page number
III. Local page bit=1
IV. copy on write bit(N)=0
V. Local page bit(N)=0
VI. Local page address(N)= NULL // NULL denotes no data
2. A Word table as T is allocated for the process
```

### *B. Bytes Modification*

Repeat the procedure till page number is same in each reference // The procedure works till page number doesn't change.

```
1. If copy on write bit ==1 then
      If Local page bit ==1 then
```

```
I. Read Local page number field as #N // N denotes the new allocated page above.
II. Read byte $W_i$ from ith offset of shared page S. // i denotes page offset
III. Modify $W_i$ as $M_i$ .
IV. Write byte $M_i$ at ith offset in N as referred by #N.
V. Write i as en entry in word table T
 else // for second if
VI. call procedure New Page Allocation
continue
else // for first if , starts modification as on a local page
VII. Read byte Wi from ith offset of local page L. // L is the page for which copy on write bit =0 in page table
VIII. Modify $W_i$ as $M_i$.
IX. Write byte $M_i$ at ith offset of local page L.
continue.
```

### *C. Bytes Copying*

Start the procedure as page number differs from the shared page in the referenced address. // This works when Modification procedure stops on a particular page.
Repeat the procedure till the last byte of a page.

```
1. Read byte $W_j$ from the shared page S // j is an offset of a word in S.
2. If j != i in T then// is an entred offset in T
     write $W_j$ in page N at j // N is the same new page whose
                                   page number is #N in the above
                                   subsection.
   else
    continue // for else part
    continue // to repeat the procedure from here
3. Delete all entries from word table T.
4. Delete shared page record from the page table.
```

## VI. Analysis

It has been understood that our computation system is based on byte wise operation[11]. CPU performs operations on a byte at one time rather than on a whole page. Also CPU register is sized for a byte or multibytes[11]. In such a scenario some required application based operations in memory can be experimented as byte wise rather than page wise. That is why, we propose an approach which is fully based on byte wise modification. Our analysis notices whether the proposed algorithms solve the defined problems.

### *A. No minor page fault*

In our approach, minor page fault is avoided by checking Local page bit* in the page map table. Once it is found that instruction exists for modification and copy on write bit is set , Local page bit* is checked for 1 and the address of a new page to write modified content into is read. CPU reads referred byte from the shared page, modify the content and write it in the new page. If Local page bit* is unset a new page is allocated and the address of this page is written in Local page number* field. Here process continues and it does not require to wait for any other type of operating system activity. Resultantly, the process is not suspended.

### *B. Copy of partial page*

Our approach is based on the concept that we want to avoid copying all bytes from shared memory page. The word table* which keeps records of addresses of modified bytes help in happening so. As the modification process continues, addresses of modified bytes are written here. Once modification is done, daemon process starts its job in background to copy the unmodified bytes from the shared page. Daemon process checks Word table* each time to get knowledge of the modified bytes.

### *C. Less time to modify a shared page*

To determine the difference between times of shared page modification by our proposed approach and copy on write approach, first we observe the major operations involved in the page modification.
For copy on write technique
1. Whether write instruction is issued.
2. Whether a page to be modified is shared/Read only.
3. Minor page fault is issued.
4. Current process is suspended.
5. Page fault is handled, a new page is allocated
6. Whole shared page is copied in the new page.
7. Process is resumed.
8. Modification is performed.
Although, time determination is a subject of experiment that we have not done yet. However, comparing the above operations with our approach's algorithm given in section 5, we find difference from operation 3. Raising minor page fault which interrupts CPU instruction cycle take more time than checking a bit(Local page bit*) for 1. Suspending process, handling page fault are incomparable activities with our approach. New page allocation is done in both the approaches. Again copy on write requires a process resume activity but ours don't. Modification is done in both the approaches. Copy is performed on only unmodified bytes in our approach whereas copy on write copies all the bytes. Remarkably we observe copying full page will always take constant time but copying partial bytes depend on number of bytes which is unmodified. After suspension, resuming a process depends on number of processes running already. Handling page fault is a kernel based activity which is an overhead for the system. Our approach uses a word table, populate it with entries of only modified bytes addresses.
In total, summed time by copy on write technique is higher than the time taken by our approach.

### *D. Avoiding memory based attacks*

Our approach modifies local page and shared page in closer time. But the same is 5 microseconds per byte in copy on write mechanism which is remarkably higher. Our analysis says that BMCOW approach can remove this higher time difference. This implies that if BMCOW approach is used for memory page modification then the time difference between local page modification and shared page modification can not be

sufficient for an attacker to notice a shared page available in his/her virtual machine. We estimate the time difference between local page modification and shared page modification as follows.
Total time required to modify a non-shared page according to our proposed work involves following.
1. Whether the instruction issued is write.
2. Whether a page to be modified is non-shared.
3. Writing modified data at some page offsets in the non-shared page.
The extra operations in shared page modification are done with local page bit, local page address, new page allocation. Copying unmodified bytes is again an extra activity which we perform after user is noticed about the page modification. Therefore, our approach does not include it in the total time taken. On the same, copy on write involves raising and handling page fault, suspending process, new page allocation , copying all bytes , resuming the process. These operations comparably require more computation than the ours.
Hence we analyze that the times estimated to modify shared page and non-shared page are closer. It does not ratio as larger as it is found in the COW work. Users will hardly notice the time difference between two types of modification. So the proposed work can avoid the timing attacks on virtual machines.

## VII. Conclusion

Shared memory page modification has been a subject of research among computer science scholars. Copy on write as a technology has solved the problem in fair and acceptable way. But by the time of technology advancements, its implementations in new applications like cloud computing researchers analyze some issues with it. The computation overhead in copy on write has been so alarming as a software is vulnerable for attacks. We have proposed our approach Bytes Modification and Copy on Write (BMCOW) as a solution whose implementation can require less computation in modifying shared pages. We also have compared our approach with per the operations involved in copy on write and conclude that BMCOW suits better.
The markable time difference between local page modification and shared page modification by copy on write has also been observed by some researchers. The resultant attack and risk of these attacks have also been sought. Our approach shows that time difference can be eliminated if we operate on only those bytes of the shared page which are required to be modified. So chances of such attacks can be minimized.
As our approach is only a novel idea, still it requires implementation. So advancement in the work keeping the concept same is feasible on per the implementation. Implementation of this work can clarify all the estimated analyzed results to be true.

## VIII. Future work

This work can be implemented and further extended to modify shared disk space in Data Centre where data of different computers are found same. The shared and stored data files can be modified in less disk operations overheads. As similar storage content revealing attacks have been observed by some researchers, this approach can eliminate reasons of attacks. Also the same idea in, different way, can be implemented in distributed computing to modify remote shared contents.

## References

[1] G. G. Abraham Silberschatz, Peter Baer Galvin, Operating System Concepts, B. L. Golub, Ed. John Wiley & Sons , Inc., 2013.
[2] M. J. Bach, The Design of the Unix Operating Sys, M. J. Bach, Ed. Prentice/Hall international , Inc., 1986.
[3] F. S. et al., “Vmdedup:memory de-duplication in hypervisor,” in International Conference on Cloud Engineering. IEEE, 2014.
[4] K. et al., “Efficient memory sharing in the xen virtual machine monitor,” Department of Computer Science, Aalborg University, Tech. Rep., 2006.
[5] F. Guo, “Understanding memory resource management in Vmware vsphere½o 5.0,” Vmware , Inc., Tech. Rep., 2011.
[6] k. et. al. Suzaki, “Memory deduplication as a threat to the guest os,” in Fourth European Workshop on System Security. ACM, 2011.
[7] J. X. et. al., “Security implications of memory deduplication in a virtualized environment,” in Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on. IEEE, 2013.
[8] A. S. Tanenbaum, Modern Operating Systems, T. Dunkelberger, Ed. Pearson E, 2007.
[9] M. et. al. Mulazzani, “Dark clouds on the horizon: Using cloud storage as attack vector and online slack space,” in USENIX Security Symposium, 2011.
[10] H. et. al., “An analysis of security issues for cloud computing,” Journal of Internet Services and Applications, 2013.
[11] M. M. Mano, Computer System Architecture, M. M. Mano, Ed. Pearson Education, 2007.