A Similarity based Technique for Detecting Malicious Executable files for Computer Forensics

Jun-Hyung Park[1], Minsoo Kim[2], Bong-Nam Noh[3], James B D Joshi[1]
[1]School of Information Science, University of Pittsburgh, USA
{jpark, jjoshi}@mail.sis.pitt.edu
[2]Division of Information Engineering, Mokpo National University, Korea
phoenix@mokpo.ac.kr
[3]Division of Electronics Computer & Information Engineering
Chonnam National University, Korea
bbong@jnu.ac.kr

Abstract

With the rapidly increasing complexity of computer systems and the sophistication of hacking tools and techniques, there is a crucial need for computer forensic analysis techniques. Very few techniques exist to support forensic analysis of unknown executable files. The existing techniques primarily inspect executable files to detect known signatures or are based on metadata information. A key goal of such forensic investigation is to identify malicious executable files that hackers might have installed in a targeted system. Finding such malware in a compromised system is difficult because it is hard to identify the purpose of the fragments of executable files. In this paper, we present a similarity-based technique that analyzes targeted executable files to identify a malware present in a compromised system. The technique involves assigning a similarity value to the fragments of executable files present in a compromised hard disk against a set of source files. We present some results based on the comparison of assembly instruction sequences of well-known hacking tools with those of various executable files, and suggest various ways to reduce the false positives.
Keywords – assembly instruction code, malicious program, computer forensics, similarity

1. Introduction

With the rapidly increasing complexity and interconnectedness of emerging information systems, the number of cyber crimes is increasing sharply. While there are significant advancements in security technologies, there is also a similar proliferation of sophisticated hacking tools and techniques. Therefore, the protection of systems as well as the establishment of evidence-based accountability for malicious actions poses significant challenges [1]. In addition to protective mechanisms, we also need tools and techniques to analyze and identify the cyber crimes and the culprits committing them so that the proper evidence-based actions can be taken against the malefactors. To facilitate such evidence-based actions, computer forensics is emerging as a significant tool. While computer forensics emphasizes techniques to identify and trace malicious activities in a system, the knowledge of the existence of such tools itself can act as a deterrent to potential hackers.

In general, forensic investigators use logs and metadata for reconstruction of cyber crimes [2], for instance, by attempting to recreate the malicious activities by analyzing and constructing a time line of activities. However, as it is still in its infancy, the computer forensics area lacks the adequate tools and techniques to support sophisticated investigations. In general, we are only able to detect the use of well-known malicious hacking tools through a signature-based technique, which also allow us to check whether normal system programs have been modified [3]. In particular, using existing forensic techniques, we cannot determine which tools the malicious users have used without analyzing the file system metadata. Typically, we can not identify a malicious program without executing it even though we may find that suspicious executable files have been installed, particularly, if it is not a known tool with a recognizable signature. Skilled hackers typically take extra measures to hide the evidence of their crimes so that forensics analysts cannot detect them. For instance, there are tools which can read and write parts of malicious executable file to the slack space of a file.

Existing techniques for computer forensic investigation primarily focus on using metadata and text information [4, 5, 6, 7]. For example, forensic investigators would draw a time line showing file usage with an *i*-node table after making a back-up image of the hard drive. The investigator would then typically search for particular words, hidden files, and suspicious file names throughout the entire file system. In general, the investigators also attempt to verify the checksums of a system's instruction files with a CRC algorithm to see if they are exactly same as the original to ensure that a potential hacker has not changed the executable files.

0-7803-9788-6/06/$20.00 ©2006 IEEE.

In this paper, we present a technique for forensic analysis of compromised systems by analyzing the hard drives. Our technique extracts executable content from the hard drive to identify whether or not it is part of a malicious program. We compare assembly instruction sequences of well-known hacking programs with those of fragments of executable files scattered in the disk, and calculate their similarity values. Using executable content to identify malware, to the best of our knowledge, has not yet been attempted. In particular, the contributions of this paper are as follows:

- We suggest a profiling technique using the assembly instruction sequence in the executable files
- We introduce the notion of similarity between two executable files by aggregating the similarities between sequential blocks of instructions separated by instructions for conditional transfer of execution flow.
- We propose various techniques to compute similarity measures to reduce false positives in the malware identification process.

The paper is organized as follows. In Section 2, we present background on computer forensics. In Section 3, we propose a technique for profiling executable files and calculating the similarity between two files. In Section 4, we present techniques to reduce false positives and present experimental results. Section 5 concludes and discusses future work.

2. Background

The techniques for hacking have been changing in response to development of security systems. The CERT gives an overview of recent trends in attack tools after observing intrusion activities since 1988 [2]. We believe that the most dangerous aspect of the field is the ease of propagation of malicious programs. It is critical to address this, as systems are continually facing types of attacks which have not been seen before, especially with the increasing level of automation of attack tools. Further, with the increasing modularization of attack tools, newer, more powerful attack tools can be easily created. Unlike early attack tools that implement one type of attack, such tools now can be changed quickly by upgrading or replacing their components. This causes rapidly evolving attacks and, at the extreme, results in polymorphic tools that self-evolve, changing with each active instance of the attack. There exists another serious problem of anti-forensic techniques. Anti-forensic techniques can hide the nature of the attack tools, making it more difficult and time consuming for security experts to analyze the tools and to understand responses to rapidly developing threats.

When we conduct forensic analysis, we follow general process of computer forensics investigation [8]. Even though the process for forensic analysis consists of many steps and techniques (from recovering deleted files to searching for pattern strings and file fragments), it is clear that investigators will use common techniques to search for some signatures of known malicious programs or strings of suspicious fragments in the hard drives [9]. Such techniques, however, can only be used when we know what we want to find. In general, we cannot find malicious programs when they are new with no known signatures and only some fragments exist [5].

If forensics investigators can not trace the activities of malicious users with common skills, they will have to analyze data more closely and precisely [10]. There are static and dynamic approaches to detect malicious programs. Static analysis involves various forms of examination without executing or running the executable files. By executing an executable file during dynamic analysis, using specialized monitoring utilities such as debuggers, we can trace or alter the program. Static analysis can eventually allow us to "know all" about the tool, whereas dynamic analysis may be limited simply by the virtue of how the programmer allows the user to interact with the application. However, in some cases, a full static analysis can't be accomplished without performing dynamic analysis also.

If the forensic investigators are sophisticated enough and there is not an overwhelming amount of data on the hard drives to investigate using reverse engineering techniques, they might be able to find malicious programs as evidence. However, there is a limit on the human and time resources for forensics investigations.

3. Similarity Based Detection

In this section, we present the propose similarity based technique for comparing executable files to detect a malware present in a compromised hard disk. In particular, the technique focuses on analyzing the instruction sequences that can be found in the hard disk to identify the malware. We compare assembly instruction sequences of well-known hacking programs with those of the targeted executable files, and calculate the similarity between them. We note that the experiments presented in this paper have been carried out for the MIPS assembly language for Intel-based system in a Linux platform.

3. 1 Executable File Profile

A key aspect of a program execution is the sequential execution of assembly instructions and the flow control. The machine executes one assembly instruction at a time sequentially until a transfer of execution flow to another portion of the executable file occurs. The assembly instruction *cmp* is one comparison condition that is associated with such a transfer of execution points. The *cmp* instruction involves checking for jump conditions. Furthermore, a conditional statement also has the important characteristic similar to the block rule of a programming language, i.e., a block has a start and the

end as indicated by the opening and the closing curly brackets in a C program block. Thus, in general, the set of assembly instructions between two *cmp* statements are sequentially executed. Table 1 shows an example of a *cmp* block associated with some high level program statements.

Table 1. An example of *cmp* instructions

High-level language	Assembly language

`IF(AX = 0 and BX = 0)`	`CMP AX, 0`
	`JNZ L1`
	`CMP BX, 0`
	`JNZ L1`

`THEN CX:= 10`	`MOV CX, 10`
	`JMP L2`
	`L1 :`

`ELSE CX := 20`	`MOV CX, 20` `L2 :`

We use this feature associated with the *cmp* commands to identify good instruction patterns. In particular, we use the *cmp* assembly instruction as a basis for dividing the instruction sequence into *cmp* blocks. Each *cmp* block (CB) is a sequence of assembly instructions between a pair of consecutive *cmp* commands. Within each CB we compute the frequencies of each instruction. Each CB can thus be expressed as a frequency vector (*fv*) that contains frequencies of each instruction. We represent by fv_i the frequency vector of cmp_i. Figure 1 shows *n* CBs and their associated *fv*s. For instance, the frequency of the second instruction in block cmp_0 is 3 and that of cmp_2 is 1. The total number of bytes for the instructions in each CB is shown in the last column. A frequency matrix (*fm*) can thus be constructed to capture the instruction frequency for a given set of assembly instructions.

Table 2. Sample *cmp* blocks and frequency vectors

cmp blocks	Frequencies of each assembly instructions (fv_i)									Length (bytes)
cmp_0	0	3	0	1	…	0	0	0	0	9
cmp_1	3	2	1	5	…	1	1	0	6	47
cmp_2	1	1	0	0	…	3	5	2	6	90
					…					…
cmp_{n-2}	2	6	2	1	…	4	0	7	6	192
cmp_{n-1}	0	2	0	3	…	0	2	1	0	27
cmp_n	2	5	0	0	…	5	6	4	0	98

3. 2 Calculating Similarity Between *cmp* Blocks

After profile a sequence of assembly instructions of an executable file, we have to find out whether there exist CBs in the target program which is similar to any CBs of the source program (Figure 1). We use the following *cosine distance* similarity measure to calculate the similarity of two CBs as follows:

$$Similarity\quad(\vec{s},\vec{t}) = \frac{\vec{s} \bullet \vec{t}}{|\vec{s}||\vec{t}|}$$

where $\vec{s}$ and $\vec{t}$ are vectors of CBs of source and target program.

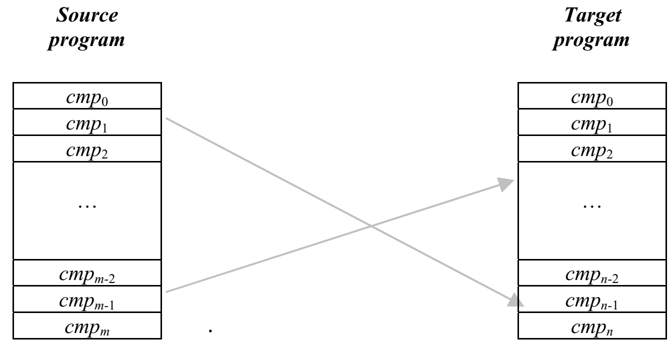

Figure 1. Similarity between profiles of CBs of source and the target executable files

It is important to note, however, that there exist short CBs which consists very few instructions – this may be because the conditional statements are located in the program continually. In such a case, the length of the CB is very short and the frequency of assembly instructions is very low.

3. 3 Detection of Similar Sections

After we have found all similar blocks we create a similarity matrix indicating the similarity values for the CBs in the source and the target programs. Next, we have to judge what parts of the target file are similar with that of the source program. Typically, just some blocks can be expected to appear similar. If blocks from the source are detected to be similar to those in the target, but are not actually part of the target program, they should be flagged as *false positives* and removed.

```
Let n is the number of CBs of source program
Let m is the number of CBs of target program
SET a n×m matrix
FOR row = 1 to n-2
   FOR column = 1 to m-2
    IF ∏_{i=0}^{2} SM[row+i,column+i] > Critical value
    Then
      STORE SM[row, column], SM[row+1,column+1],
            SM[row+ 2,column+ 2]
    ENDIF
  ENDFOR
ENDFOR
```

Figure 2. Algorithm for detecting similar blocks

Note that we detect similar blocks by calculating frequencies of assembly instructions. In addition, more similar blocks should have similar instruction sequences as well. To address this issue, we later investigate combining a number of contiguous blocks during similarity computation to reduce false positives. In particular, we combine three similarity values associated with the continuous CBs in the matrix to check against a *critical value* to compute the similarity measure, in the

experiments. Figure 2 shows the algorithm for detecting series of similar CBs from the similarity matrix.

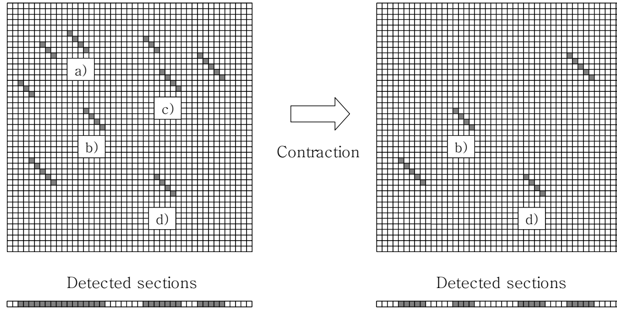

Figure 3. A sample of contraction for detected sections from target program

Figure 3 shows possible distribution of series of similar CBs from the target file - here *X*-axis represents a target program and *Y*-axis represents a source program. We want to detect some sections in the target program that are similar to some parts of source program. Hence, we assume one CB on the *X*-axis could take just one CB on the *Y*-axis. For example, the fourth block of series a) and the first block of series b) are overlapped on the *X*-axis in Figure 3. This means one block of target program is similar with two different blocks of the source program. For this case we used real length of CBs in the profile without considering the number of blocks. If series b) is detected, then the length of series b) is longer than that of series a). By iteration of this kind of *contractions*, we can detect several series of similar blocks that can now be regarded as similar fragments.

3. 4 Distribution of Similar Fragments

There are four types of possible distributions of similar fragments we can inspect in the target executable file with respect to the source program, which are as follows.

a) *Fully Contained*: Source program is contained in the target file. Here, the similar blocks are distributed contiguously diagonally in the similarity matrix.
b) *Partial Existence*: Some parts of the source program exist in the target file. Here, the line of distribution covers *X*-axis fully but not the *Y*-axis.
c) *Disconnected, But Contained* – The source program is contained in the target file but it is modified. If the line of similar fragments in the similarity matrix is not distributed contiguously, we can expect that the source program has been modified for some purpose. Even though similar fragments are not distributed contiguously, the target may contain source program with high possibility.
d) *Irregularly Ordered, but Contained*: The fragments in the target program that are similar to those in the malicious program but the sections in the target program are not ordered as are the corresponding sections in the malicious program that are similar to them. In this case, we cannot be sure if the target program contains some parts of the source.

3. 5 Similarity Algorithm

We make unique profile of each assembly instruction sequence so that they are distinguishable from each other. For this work, the key issue is the exact description of these sequences into the frequency vector. Before calculating the similarity with detected fragments, we assume two conditions for two sequences that are exactly matching: (i) frequencies of elements in two sequences must be same, and (ii) all the elements consisting sequences must be allocated sequentially. Note that these conditions are independent to each other. We can calculate the similarity by considering both these factors to make the similarity measure more effective.

3. 5.1 Similarity from Detected Fragments

In this section, we will explain how to calculate the similarity with detected similar fragments between two executable files. If we detect *n* similar fragments from target program then there exist *n* fragments in source program even though they can be duplicated. We can describe this as below.

$S = \{c_1, c_2, c_3, \dots, c_n\}$, $T = \{c_1', c_2', c_3', \dots, c_n'\}$

Also, $\forall c_i \in S$ there must be a $c_i' \in T$ which is similar to c_i, and each pair of c_i and c_i' has its own similarity. It is to be noted that there is another problem we have to consider for calculating similarity of detected similar fragments. All similarity values for every pair of c_i and c_i' generally do not have same weights because the lengths of detected fragments are totally different. Sometimes the length of one fragment is shorter than 10 bytes but in a long section it could be longer than 100 bytes. These two cases of similarities should not have the same weight for calculating similarity for the total detected fragments. Hence, it is natural to calculate weighs for each fragment depending on its length. Accordingly, we use the following formula for calculating the similarity values:

$$similarity_of_CBs(s,t) = \sum_{i=1}^{n} \left(w_i \times sim\left(c_i, c_i'\right) \right)$$

$$w_i = \frac{\min(size(c_i), size(c_i')) \times sim\left(c_i, c_i'\right)}{\sum_{j=1}^{n} \left(\min(size(c_j), size(c_j')) \times sim\left(c_j, c_j'\right) \right)}$$

$$Where \;\; c_i' = \arg\max\left(sim\left(c_i, c_k'\right)\right), \;\; for\,(k = 1, \dots, n)$$

3. 5.2 Order of Similar Sections

Once similarity has been detected by taking the length of the blocks into consideration, we have to calculate similarity based on the instruction order of the detected similar fragments. We estimate this value based on how similarly the detected sections are ordered. Fortunately, it

is not a new problem. We can use the same mechanism that has been used for comparing two strings [15]. For this we use the *Damerau-Levenshtein* edit distance [15]. *Damerau-Levenshtein* edit distance counts a transposition as a single edit operation. We calculate similarity of their order using the edit distance as follows.

$$sequence_matching(s,t) = 1 - \frac{dist(s,t)}{\max(Length(s), Length(t))}$$

We next calculate two probabilities for the similarities between frequencies of two sets of assembly instructions and their order in the sequence. If there are some sections that satisfy the above two conditions, we can now consider them similar with more certainty. As the two conditions are independent, we can get the final similarity value by multiplying the two results. Thus, the formula for calculating the similarity of between two sequences is as follows.

$$Similarity(s,t) = Similarity_of_CBs(s,t) \times sequence_matching(s,t)$$

4. Experimental Results

In this section, we present some experimental results based on the proposed similarity based detection of malicious code. In particular, we experimented to see whether any CB can be distinguished from the others by using the proposed profiling technique. We tested about 100 executable files to observe the result of applying this technique for distinguishing the CBs.

4.1 Test to Distinguish Conditional Statement

The graph in Figure 4 depicts the distribution of results of calculating similarity with proposed method for 10 sample cases. We can regard blocks with high similarity as noise, which could be caused by the inclusion of short CBs. Such small CBs lack discriminatory power as they can easily result in a similarity measure close o 100% with other short blocks.

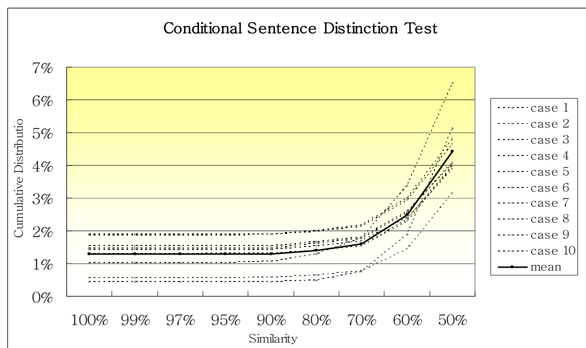

Figure 4. Cumulative distribution of false positives

To discard such false positives resulting from presence of short CBs, we excluded short CBs containing less than five instructions. We could reduce more than 70% of false positives on an average this way. In addition to this, by taking two blocks as similar only if their similarity is higher than 90%, we obtained fewer false positives than 0.5% with the 100% of accuracy.

4.2 Number of Blocks and False Positives

Despite of 0.5% of false positives of the proposed profiling technique, it is still not enough to decide whether two CBs are similar because the length of the executable file could be really long (more than 1 or 2 Giga bytes) and it might consist of millions of CBs. Note that if detected blocks are contiguous, then there is a higher probability that the series of detected blocks are really similar than shorter ones. We conducted an experiment to measure the relation between the reduction in false positives and the number of CBs.

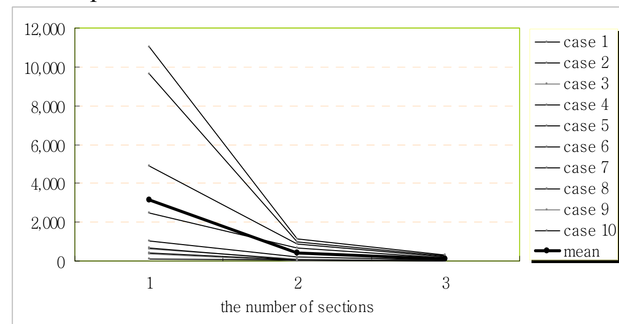

Figure 5. Relation between false positives and the number of contiguous blocks

Figure 5 shows an example of a similarity matrix for comparing two assembly instructions sequences of source program with target program. With a threshold fixed at 80%, we could detect similar CBs as in the figure. Some detected blocks were not similar and they should be indicated as false positives. The graph also depicts the results of our experiment to decide the minimum number of contiguous blocks that would be meaningful. After a large number of comparisons (thousands) between CBs, we could find that the false positives decreased by more than 97%.

Table 3. Test results for detecting a malware

Malicious program	Sniffer	Backdoor	DDOS	LKM
The mean of Similarities	91%	94%	100%	91%

4.3 Test Results with Malicious Program

Next, we conducted an experiment to detect malicious programs installed in the real environment with our detection method. The experiment involved detecting 4 kinds of 15 malicious programs out of 100 normal executable files. We detected almost all malicious executable files except one program among the Linux Kernel Module, and we tried to find the reason why it could not be detected by the proposed method. After

observing the file that could not be detected, we found the file was short and consisted of many conditional statements. This allowed us to investigate increasing contiguous similar blocks to make the matching process more effective. Table 3 shows the results.

4.4 Detection Rate and the Critical value

The graph in Figure 6 shows the Receiver Operating Characteristic (ROC) curve of the final results. To measure the detection rate using the proposed methods, we tried to find 100 similar files out of 312 executable files. Similar files consisted of all possible cases of transform like insertions, deletions, substitutions, and transpositions.

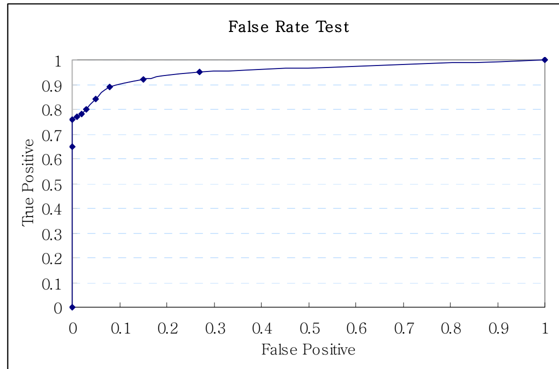

Figure 6. A test result for determining threshold

The graph shows the relation between detection rate and the false positives. A good critical point is the similarity level after which the inclination of the curve is decreased suddenly. This is because such sharp turns indicate that false positives are increasing significantly at those points. In Figure 6, there are three points we can choose as a critical values.

Table 4. Thresholds and detection rates

Threshold	0.8	0.5	0.1
True Positive	76%	80%	91%
False Positive	0%	3%	47%

With this experiment we found that 80% of detection rate and 3% of false positives are possible with 0.5 of similarity. This means we can decide two files are similar when their similarity measure is over 0.5. By controlling the critical values for their environments, the forensics analysts may learn deeper insights into the characteristics of the malware being analyzed.

5. Conclusion

In this paper, we presented a method to find out whether a detected executable file is similar to a malicious executable file by comparing assembly instruction sequences. Using the existing techniques, we would not be able to detect any modified malicious programs without executing the program or reverse engineering. In addition, there exists no appropriate method, to the best of our knowledge, to detect fragments of a malicious executable file which is hidden or deleted. In order to calculate similarity between two executable files, we described the assembly instruction sequences as a unique profile. By using the assembly instruction sequence based technique, we could find fragments similar to malicious program even when they are slightly modified. The method could be used to also identify the nature of executable fragments stored in slack space and free data blocks. We summarized our experiments to support the technique we have proposed and tested.

Acknowledgments

This work was supported by the Ministry of Information & Communication, Korea, under the Information Technology Research Center (ITRC) Support Program and the University of Pittsburgh, USA.

References

[1] H. Chen, W. Chung, J. Jie Xu, Gang Wang, Yi Qin, Michael Chau, “Crime Data Mining: A General Framework and Some Example”. In *Computer*, April, 2004. pp.50-56

[2] B. D. Carrier, E. H. Spafford, “Defining Event Reconstruction of Digital Crime Scenes,” In Cerias Tech Report 2004-37.

[3] A. J. Marcella, R. S. Greenfield, “Cyber Forensics,” Auerbach Publications, 2002.

[4] L. Garber, “EnCase: A Case Study in Computer-Forensic Technology,” IEEE Computer Magazine, Jan., 2001, pp202-205.

[5] Guidance Software, "EnCase Legal Journal," 2nd Edition, Mar., 2003.

[6] D. Farmer, S. John, W. Venema, “The Coroners Toolkit (TCT) v1.12,” http://www.porcupine.org/forensics/tct.html.

[7] B. Carrier, “TCTUTLs v1.01,” May, 2001, http://www.cerias.purdue.edu/homes/carrier/forensics.html,

[8] R. Nagpal, “Recovery of Digital Evidence,” http://www.asianlaws.ort/cyberlaw/li-brary/cc/dig_evi.htm, 2002.

[9]. F. Buchholz, E. Spafford, “On the role of file system metadata in digital forensics,” In Journal of digital investigation, Dec. 2004.

[10] K. J. Jones, R. Bejtlich, C. W. Rose, Real Digital Forensics, Addison-Wesley, 2006.

[11] A. Housebolder, K. Houle, C. Daugberty, “Computer Attack Trends Challenge Internet Security,” In IEEE Security & Privacy, 2002.

[12] Intel Corporation, “The IA-32 Intel® Architecture Software Developer's Manual,” Intel Corporation, 2003.

[13] A. Chuvakin, “Linux Data Hiding and Recovery,” http://www.linuxsecurity.com/feature_stories/data-hiding-forensics.html, October, 2002.

[14] J. R. Vacca, “Computer Forensics : Computer Crime Scene Investigation,” Charles River media, 2002.

[15] E. Mays, F.J. Damerau, R.L. Mercer. Context-based spelling correction. In *information Proceeding and Management*, 27(5):517-522, 1991.