JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

Hash-AV: Fast Virus Signature Scanning by Cache-Resident Filters

Omkar Vikas Bakare

Abstract: *As today's world is moving from offline to online all the process now are been done with the help of computer's, any data we require is present on the internet so it's a challenging task to develop a system that will provide search Fast Hash AV algorithm to Scan virus signatures online . Hence we present a system that will provide the Virus scanning technique that will scan the system for the virus very quickly by using the concept of bloom filter and many other different databases, where accuracy and efficiency are most important terms required. This paper proposes Hash-AV, a virus scanning "booster" technique that aims to take advantage of improvements in CPU performance. Various analyses shows user is not restricted to formulate any kind of query so this system provides result to users any type of query he fires to the system accurately and efficiently, even if any user make signature mistake the system will automatically identify the virus in the system . The key to Hash-AV's success lies in a set of "bad but cheap" hash functions that are used as initial hashes.*

***Index Terms* - Hash AV, bloom filter.**

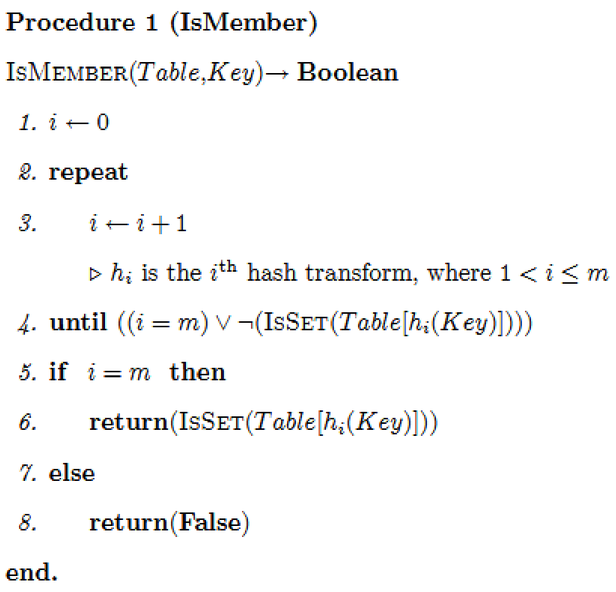
I. INTRODUCTION

In the age of Internet and the Web, viruses proliferate and spread easily. As a result, anti-virus technologies are a must in today's wired world. An effective defense needs virus-scanning performed at every major network traffic stop and at the end-host computers. Today, anti-virus software applications scan traffic at e-mail gateways and corporate gateway proxies1, and they run on end-hosts such as file servers, desktops and laptops. Unfortunately, while the speed of network-based intrusion detection has improved over the years to over 1Gb/s today, the speed of virus scanning has not kept pace. Virtually all virus-scanning programs spend the bulk of their time matching data streams with a set of known virus signatures, and they all utilize some form of multi-pattern string matching algorithm. The number of virus signatures today is over 100,000 and is growing constantly. Unlike intrusion detection signatures, virus signatures cannot be neatly separated into rule sets consisting of a small number of strings. Traditional matching algorithms require at least one random memory access per scanned byte. The performance of random memory accesses, however, does not improve nearly as much as the CPU speed or even sequential memory access throughput. For example, in the past decade, CPU processing speed has been doubling every 18 months, yet memory speed only improved at a rate of less than 10% per year.

Bloom Filter Technique

Using bloom filters to speed up signature matching is not a new idea. For example, research groups have proposed specialized hardware solutions that use parallel bloom filters to scan packets at very high speed. If the hash functions are not chosen well, computing the hashes can easily take enough CPU cycles to obliterate the advantage of cache-resident filters. Hash-AV addresses the problem by using "bad but cheap" hash functions as initial hashes, and relying on serial hash lookups. The "good but expensive" hash functions are only calculated when the cheap initial hashes indicate a match, effectively reducing the CPU computation of the bloom filter probe. We have applied Hash-AV to Clam-AV , the most popular open source anti-virus software. Hash-AV improves Clam-AV's scanning throughput to 29.4 MB/s for executables, 16.6 MB/s for web pages, and 29.5 MB/s for random data, on an Athlon XP 2000+. This represents a speed-up factor of 1.7 to 4.4. The speed improvement doubles as the size of the signature database increases from 20K to 120K. Furthermore, if polymorphic viruses are handled separately using emulation,.

For the purposes of the analysis, we are presenting only the essentials of Bloom filters — the algorithms are for single bit elements. The analysis of filters with more complicated cells is essentially the same. IsMember is presented immediately below



```
Procedure 1 (IsMember)
IsMember($Table$,$Key$)$\rightarrow$ Boolean
$i \leftarrow 0$
repeat
    $i \leftarrow i + 1$
    $\triangleright$ $h_i$ is the $i^{\text{th}}$ hash transform, where $1 < i \leq m$
until $((i = m) \vee \neg(\text{IsSet}(Table[h_i(Key)])))$
if $i = m$ then
    return(IsSet($Table[h_i(Key)]$))
else
    return(False)
end.
```

Algorithm Search signature

II. LITERATURE STUDY

Multi-string pattern matching algorithms is a well-studied topic with applications in many domains [14]. In the network-ing area, the two prominent applications are IDS (Intrusion Detection Systems) and virus scanners. Recently, several in-novations have been proposed for pattern matching in IDS, for example, hardware-based parallel bloom filters [19], and novel compression techniques to reduce memory requirements of IDS and improve hardware implementation performance[16]. However, these studies have not looked into virus scanning applications, which are quite different from IDS systems[6]. Our focus on virus scanning applications and software implementation distinguishes our study from the above efforts. Virus scanning applications are commonly host-based, as opposed to IDS systems which are commonly network-based. As a result, software implementations running on generic pro-cessors are more appropriate for virus scanners than hardware implementations. Software implementation is different from hardware implementation due to serial applications of hash functions, stringent requirements on the CPU cost of a hash function, and the performance impact of good cache locality. As a result, design choices for software implementation are quite different from those of hardware implementations. Recently, there have been renewed focus on improving the scalability of Clam-AV [5], [24]. In addition, the Avfs paper [24] provides an excellent study of the issues involved in integrating virus scanners in file system implementation. The techniques described in these studies are complementary to Hash-AV, and the techniques should be combined together to further improve Clam-AV performance. Because of their importance, there have been constant improvements on multi-string matching algorithms and their variations. Hash-AV is a "booster" technique that is indepen-dent of the underlying string matching algorithm, and can be combined with any improved matching algorithm. The benefit of Hash-AV is in quickly determining no-match cases in a CPU cache-friendly manner, and Hash-AV is beneficial to any systems where the no-match cases are the vast majority.

III. 3. PROPOSED WORK

System Description

There are three main techniques used in virus-scanning: 1. signature-matching: check if a file contains a known virus by searching for a fixed string of bytes (the "signature" of the virus) in the data. 2. emulation: check if a file contains a polymorphic virus (those that change from occurrence to occurrence) by executing the instructions of an executable in an emulated environment, and then looking for a fixed string (the "signature") in the memory region of the process [22]. 3. behavior-checking: check if a file contains an unknown virus by running the file in an emulated environment and observing its behavior.

Behavior-checking is typically run on a specific file to deter-mine if the file contains a new virus. Since it is not routinely run, its performance is usually not a concern. Emulation is typically used only on executables matching certain criteria. Signature-matching is routinely run on all files. Clam-AV: Clam-AV is the most widely used open-source anti-virus scanner [12]. It is used by many organizations in their mail servers, and has been incorporated into com-mercial anti-virus gateway products [10]. As of July 2005,it has a database of over 30,000 viruses, and consists of a core scanner library and various command-line programs. The database includes over 28,000 plain-text strings and over 1,300 strings with wild-card characters embedded. The plain-text strings are for non-polymorphic viruses, and the strings with wild-card characters are for polymorphic viruses. The current version of Clam-AV uses an optimized version of the Boyer-Moore (BM) algorithm [2] for non-polymorphic signatures, and uses the Aho-Corasick (AC) algorithm [1] for polymorphic ones. The Boyer-Moore implementation in Clam-AV uses a "shift-table" to reduce the number of times the Boyer-Moore routine is called. At start up, Clam-AV walks over every signature, byte by byte, and hashes the three-byte chunk to initialize a global shift table. Then, at any point in the input stream, Clam-AV can determine if it can skip up to three bytes by performing a quick hash on them. Clam-AV also creates a hash table based on the first three bytes of the signature, and uses this table at run-time when the shift table returns a match. Since this algorithm uses hash functions on all bytes of a signature, it is only applicable for non-polymorphic signatures. The Aho-Corasick implementation uses a trie to store the automaton generated from the polymorphic signatures. To quickly perform a lookup in this trie, Clam-AV uses a 256 element array for each node. It also modifies Aho-Corasick such that the trie has a height of two, and the leaf nodes contain a linked list of possible patterns. Clam-AV fixes its trie depth to two because its database of polymorphic viruses have signatures with prefixes as short as two bytes.

Hash-AV utilizes the fact that, while virus scanning must be done on network traffic, the vast majority of the data do not contain viruses. Therefore, it aims to determine the no-match cases with high accuracy, minimal main-memory access and a small number of CPU instructions. It achieves the goals by using a filter that fits in CPU caches and acts as a first-pass scan to determine if the data need to go through an exact-match algorithm. Specifically, Hash-AV moves a sliding window of β bytes down the input stream. For each β bytes under the window, k hash functions are applied to calculate their hashes. The hash results are then used to probe into a bit array of N bits, which is a bloom filter [3] pre-constructed from the virus signatures.

Bloom filter : Bloom filters use hash transforms to compute a vector (the filter) that is rep-preventative of the data set. membership is tested by comparing the results of hashing on the potential members to the vector. In its simplest form the vector is composed of N elements, each a bit. An element is set if and only if some hash transform hashes to that location for some key. Figure 2 shows such a filter with m = 4 hash transforms and N = 8 bits.

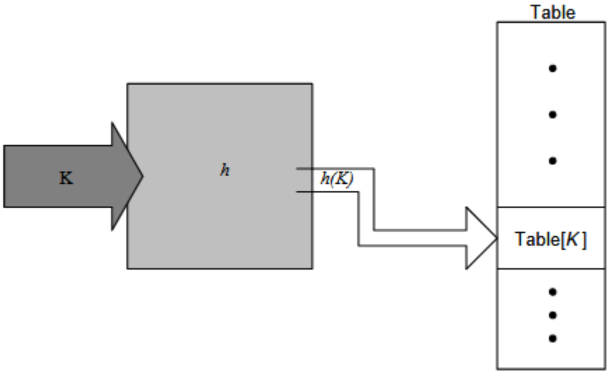


Figure 1: A typical hash transform in action

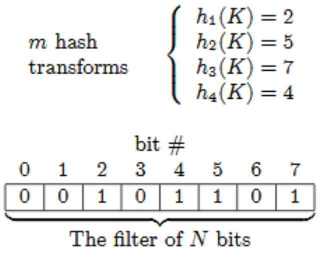


Figure 2: A simple Bloom filter

Bloom filters can be combined with other methods, such as signatures [1, 2]. Figure 3 depicts a case in which the filter contains references to information related to records rather than only the records. In that case the hash transforms will hash to N/(b + 1) cells, where b is the size of the signature and the extra bit is used to flag cells containing signatures [1].

Hashing: Hashing transforms are typically pseudo-random mathematical transforms used to compute addresses for lookup [18, 19]. Figure 1 shows the use of the hash transform h, to find an item with a key K, stored at address h(K). The time complexity of searches by hashing can be as low as O(1) or as high as O(N), for a hash table with N elements. The worst-case behavior occurs when two or more distinct keys Ki 6 = Kj collide, i.e., h(Ki) = h(Kj), and the entire table must be searched to find the correct entry [18, pp. 507 – 508]. Bloom filters are a fast method in which the hash transforms always have constant time complexity — there is no attempt at collision resolution. Knuth [18] described Bloom filters as a type of superimposed coding because all of the hash transforms map to the same table.

Hash-AV utilizes the fact that, while virus scanning must be done on network traffic, the vast majority of the data do not contain viruses. Therefore, it aims to determine the no-match cases with high accuracy, minimal main-memory access and a small number of CPU instructions. It achieves the goals by using a filter that fits in CPU caches and acts as a first-pass scan to determine if the data need to go through an exact match algorithm. Specifically, Hash-AV moves a sliding window of B bytes down the input stream. For each B bytes under the window, k-hash functions are applied to calculate their hashes. The hash results are then used to probe into a bit array of N bits, which

is a bloom filter pre-constructed from the virus signatures. A. Basic Mechanisms Hash-AV constructs a bloom filter from the set of plaintext signatures. The bloom filter is a vector of N bits,

initially all set to 0. For each plain-text signature, k hash functions are applied to its first B bytes a, with results h1(a); h2(a); :::; hk(a), all in the range of 1; :::;N. The bits at positions h1(a); h2(a); :::; hk(a) are then set to 1. At scanning time, Hash-AV moves over the input data stream one byte at a time. For each B byte block b, the scanning algorithm applies the first hash function, h1(b), and checks the corresponding bit in the bloom filter. If the bit is 1, it computes the next hash function h2(b); if not, it immediately goes over to the next byte, and starts applying hash functions over the next B-byte block.

In the case where all k functions have positive bloom filter matches, Hash-AV needs to check for exact match. There are two alternatives here. One is to use Boyer-Moore. Another is to pre-construct a "secondary hash table" using the last hash function hk, with each entry holding a linked list of signatures which are checked linearly. Hash-AV adopts the latter approach, since the number of signatures in each entry is low. Several aspects of Hash-AV differentiate it from other approaches.

Most commercial scanners use hash-tables to speed up string matching , similar to Hash-AV use of the secondary hash table. However, the data structure involved usually does not fit in cache, and the false positive ratio from a single hash function is higher than the bloom filter. Clam-AV uses a cache-resident shift table to reduce the number of times the Boyer-Moore algorithm is called. Unfortunately, since the shift table has to fit in cache, only 3 bytes are used and the resulting false positive ratio is high. In essence, compared to these schemes, bloom filters are much more compact, and the use of multiple hash functions results in much lower false positives. Other researchers have proposed using hardware bloom filters to perform high-speed network intrusion detection. However, in those designs, all hash functions are calculated at the same time using parallel ASIC hardware. Hash-AV applies hash functions serially, in an effort to reduce the number of CPU instructions consumed. Based on our prior experience in using bloom filters, k = 4 works well. Therefore, there are three choices left in setting up Hash-AV:

_ Choosing four hash functions;
_ Choosing the size of the bloom filter;
_ Choosing B;

Below, we use a simple model to briefly analyze the impact of each choice. B. A Simple Performance Model Assume that the four hash functions are h1, h2, h3, and h4, applied in that order. Furthermore, assume that the function hi can be calculated at ci MB/s. Let the total number of signatures be M, and the size of the bloom filter be M _ K bits. The function h1 then has a false positive probability of p1 in the bloom filter. The probability p1 is determined by both the hash function and the bloom filter's expansion factor K. Similarly, h2 has a false positive probability of p2;1 in h1's false positive cases. In other words, p2;1 is the conditional probability of false positive under h2 given that h1 has a false positive. The ratios p3;2;1 and p4;3;2;1 are defined similarly.

The performance of the scanning algorithm can be modeled using the above parameters. Note that h2 is called when h1 hits in the bloom filter (i.e. h1's bit is 1), h3 is called when both h1 and h2 hit in the bloom filter, and h4 is called when all three previous hash functions hit in the bloom filter. Thus, the throughput of the scanning algorithm is:

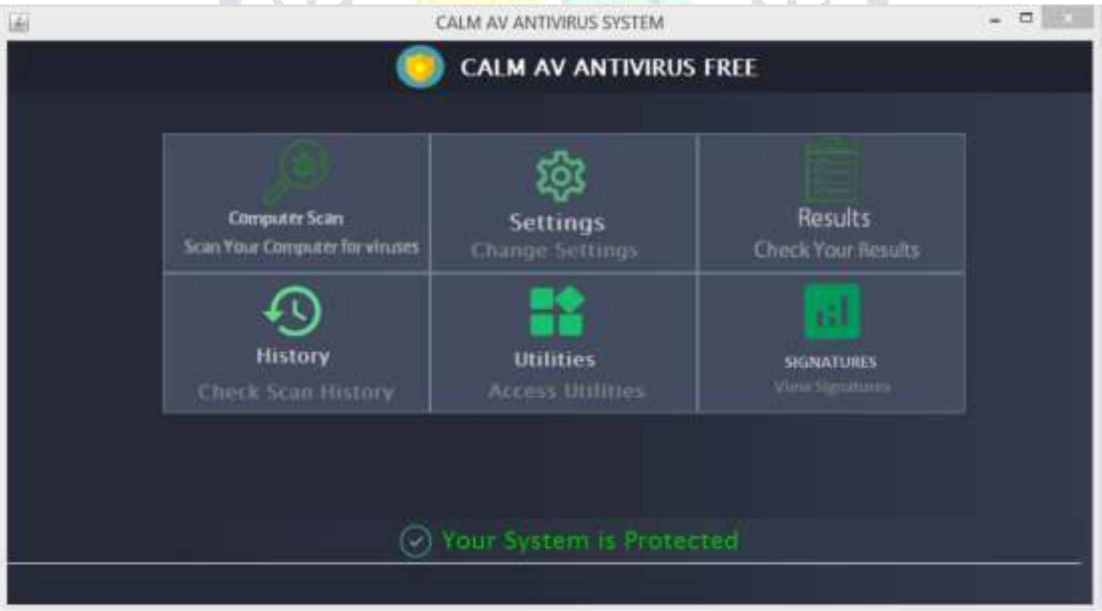
$$c1 + p1 * c2 + p1 * p2;1 * c3 + p1 * p2;1 * p3;2;1 * c4 + p1 * p2;1 * p3;2;1 * p4;3;2;1 * C$$

where C is the cost of the exact string matching algorithm. Clearly, since all the probabilities are between 0 and 1, the hash functions should be ordered from the cheapest (computationally) to the most expensive. The above formula leads to a number of insights. First, it pays to use very fast, but mediocre hash functions for h1 and h2. A hashing function which has 15% error rate but takes five CPU cycles to calculate is a poor choice in other circumstances, but serves very well to our purpose. In fact, these cheap functions help us make the theoretical argument that the Hash-AV scanning algorithm can potentially perform at near memory system throughput. Second, it's important to choose hash functions that are independent. Completely independent hash functions would have the conditional false positive probabilities the same as the unconditional false positive probabilities. On the other hand, non-independent hash functions tend to have the conditional probabilities close to 1, defeating the purpose of multiple hash functions. Third, the probabilities are affected by the bloom filter's expansion factor K. Since the cost of the exact string matching algorithm, C, might be one or two orders of magnitude higher than the cost of the hash functions, it's important that the bloom filter do not contribute significantly to the false positive ratios. In the sections below, we use experiments to determine the appropriate K.

Finally, there is a lower bound on the probabilities p1* p2;1* p3;2;1 * p4;3;2;1, which is determined by the parameter B. In other words, there is a probability that strings that match the first B characters do not match the full signature. In general, longer _s are better. However, a longer B also means that shorter signatures (those of length < B + 3) must be handled by a different mechanism. Hence, the choice of B also affects the performance. C. Evaluation methodology We evaluate the benefits of Hash-AV for both the current Clam-AV database (about 30,000 signatures) and a database containing 120,000 signatures. The signature database for Clam-AV is growing very rapidly. Hence, it is essential that Hash-AV scales for large signature databases. To generate more signatures, we wrote a synthetic virus generator that examines the properties of the current Clam- AV database, and tries to generate realistic virus signatures. The generator works as follows. At startup, it reads in the non-polymorphic and polymorphic signatures in Clam-AV's database into different arrays in memory. Then it extracts two pieces of information: the distribution of virus signature lengths, and the percentage of polymorphic patterns in the database. Based on these pieces of data, for each "new" virus, the generator first chooses its length and its type (i.e. non polymorphic or polymorphic). For byte i in the new virus, the generator randomly picks an existing signature, and copies its byte i. For each byte index, this algorithm statistically favors the most common byte for that index. Since Clam-AV's polymorphic signatures only use wild-card ASCII characters * and ?? in between bytes (with * matching any string and ?? matching any single character), this approach generates viruses that are as polymorphic as the ones in the database. For most experiments, the sample file is an 120MB file created by concatenating together widely used Windows executables
that are over 3 MB in size, including MS Office executables, messenger programs, third-party software used for scientific and entertainment purposes. We focus on windows executable files since the majority of viruses spread through executables and commercial scanners focus heavily on

executable files. In our selection, we pay attention to including only the executable binaries, and avoid setup programs since Hash-AV runs much faster on them. Our experiments are run on an Athlon 64 3200+ PC, with 2.0 Ghz CPU, 128KB L1, and 512KB L2 cache. We havealso repeated the experiments on an Athlon XP 2000+ and a Pentium-4 2.6Ghz PC, and found matching results.

IV. RESULTS



Your System is Protected

figure 3. System Designed UI1

In the above figure we shown about the proposed systemfor various features that can be implemented using Calm AV. future we will try to implement the above discussed goals. In our system, Implementation is for scanning viruses by using specified virus signatures deployed on online signatures' database.

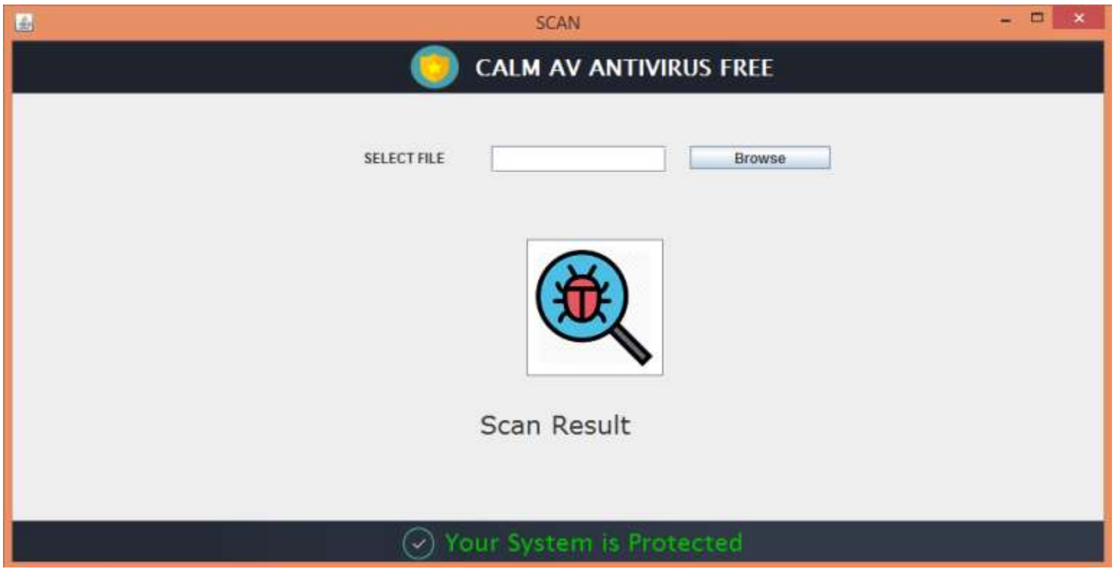


figure 4. Search For Virus UI2

Above figure shows an interface for searching a file for any virus infections. It search whole file for any virus infection using system virus database. Proposed system scans files with very high speed using Calm-AV techniques with above 95% of accuracy.

V. CONCLUSION

In this paper we proposed the Antivirus system by using Calm-AV algorithm for faster virus detection with very low lower false positives rate. Proposed system uses a trained database of various virus signatures' and can easily identify an availability of any infections to given file or files within very limited time with 100% true negative rate.

VI. REFERENCES

[1] Woods, W.A. et al, "**The Lunar Sciences Natural Language Information System**", BB&N Report 2378, June**1972.**

[2]Hendrix, G.G., Sacerdoti, E.D., Sagalowicz, D., Slocum, J. "**Developing a natural language interface to complex data**", in ACM Transactions on database systems, 3(2), pp. 105-147, **1978**.

[3]Tomek S., Fang L., Jose Perez-Carballo and Jin W. "**Building Effective Queries in Natural Language Information Retrieva***l*" GE Corporate Research &Development Research Circle, Niskayuna, NY 12309

[4] JOSEPH, S.W, ALELIUNAS, R. "**A Knowledge-Based Sub System For A Natural Language Interface To A Database That Predicts And Explains Query Failures**", IN IEEE CN, PP. 80-87, **1991**.

[5]Qing C., Mu L., Ming Z."**Improving Query Spelling Correction Using Web Search Results**",Natural Language Processing and Computational Natural Language Learning, pp. 181–189, Prague, June **2007**.

[6]Huangi, Guiang Z., Phillip C-Y S. "**A Natural Language database Interface based on probabilistic context free grammar**", IEEE International workshop on Semantic Computing and Systems **2008**.

[7] Mrs. Gauri R."**Natural Language Query Processing Using Semantic Grammar**". / (IJCSE) International Journal on Computer Science and Engineering Vol. 02, No. 02, **2010**, 219-223

[8]Aminul Islam and Diana"**Correcting Different Types of Errors in Texts**",Inkpen University of Ottawa, Ottawa, Canada diana@site.uottawa.ca, **2011.**

[9]Ziqi W., GU X., Hang L., and Ming Z."**A Probabilistic Approach to String Transformation**" IEEE transactions on knowledge and data engineering VOL:PP NO:99 YEAR **2013**.

[10] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. Communications of the ACM, 18(6):333–340,1975.

[11] R. S. Boyer and J. S. Moore. A fast string searching algorithm. Communications of the ACM, 20(10), 1977.

[12] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In Allerton 2002, page http://www.eecs.harvard.edu/m˜ ichaelm/NEWWORK/papers.html,2002.

[13] Cisco. Network-based application recognition and distributed network-based application recognition. In http://www.cisco.com/univercd/cc/td/doc/product/software/ ios122/122newft/122t/122t8/dtnbarad.htm, 2004.

[14] M. Dounin. Clamav developer forum. In http://sourceforge.net/mailarchive/forum.php?forum=clamav-devel, June 2004.

[17] O. Erdogan and P. Cao. Hash-av: Fast virus signature scanning by cache-resident filters. In http://crypto.stanford.edu/˜cao/hash-av/, 2005.

[18] J. S. Gardner. Pc motherboard technology. In http://www.extremetech.com/article2/0,1558,1148755,00.asp, June 2001.

[19] H. D. GmbH. Dazuko. In http://www.dazuko.org, 2004.