BITS FILTER: A HIGH-PERFORMANCE MULTIPLE STRING PATTERN MATCHING ALGORITHM FOR MALWARE DETECTION

by

Dan Lin

B.Sc., Beihang University, 2006

A Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
Spring 2010

APPROVAL

Name: Dan Lin

Degree: Master of Science

Title of Thesis: Bits Filter: A High-Performance Multiple String Pattern Matching Algorithm for Malware Detection

Examining Committee: Dr. Tamara Smyth
Chair

Dr. Rob Cameron, Senior Supervisor

Dr. Mohamed Hefeeda, Supervisor

Dr. Kay Wiese, SFU Examiner

Date Approved: Dec. 20th, 2007

Declaration of
Partial Copyright Licence

Abstract

Multiple string pattern matching is the kernel and key technique of many security applications such as anti-virus scanning and intrusion detection. The growing size of on-line content and increasing network and CPU speed push the need for a fast multi-string search algorithm. This thesis investigates single-instruction multiple-data (SIMD) and parallel bit stream technologies in high performance text processing. A three-pass filtering algorithm called Bits Filter based on these technologies was developed not only to provide fast calculation but also to minimize the data cache misses. This algorithm is then studied by balancing the tradeoffs and considering various parameters such as text size, pattern set size, filter size and segment size. Comparisons are made with implementations of the Aho-Corasick algorithm extracted from the open-source security applications Snort and ClamAV. Whereas the Aho-Corasick algorithm typically requires 50-300 cycles per input byte of the text in these implementations, the Bits Filter algorithm requires only about 2-7 cycles per byte.

Acknowledgments

I am profoundly grateful to my senior supervisor, Dr. Rob Cameron, for his continuous support, encouragement and guidance through my research over the last three years. He provided valuable insights, a lot of help during my study, taught me how to do research. He also came on weekends and made numerous editing and proof-reading of the material presented in this thesis. This thesis would not have been possible without him.

I would like to thank my supervisor Dr. Mohamed Hefeeda and my thesis examiner Dr. Kay Wiese for being on my committee and reviewing this thesis. I would like to thank Dr. Tamara Smyth for taking the time to chair my thesis defense.

I would also like to thank my uncle Dr. Dekang Lin for doing proof-reading of this thesis and giving me a lot of valuable suggestions.

Last but not least, I would like to thank my family for their unquestioning love, care and support. Their expectations have always propelled me to explore my full potential. This thesis is dedicated to them.

Contents

Chapter 1

Introduction

Multiple string pattern matching is a process of finding a substring of a long string known as the text such that the substring matches any one pattern from a set of multiple patterns. It has many applications. It is needed in security applications to detect certain suspicious keywords or patterns. For example, anti-virus scanning and intrusion detection both involve searching through large collections of known virus signatures or malicious commands. It is also used in content filtering, data mining, DNA searching, and many other applications.

Network applications that are capable of providing high-speed processing are always in demand. Thus, multiple string pattern matching, one of the bottlenecks of these applications, becomes an issue as the size of on-line content grows, network speeds increase and known threats multiply. For example, the number of intrusion detection rules for the community-based bleedingthreats.net grew from 1000 to 8000 within three years [28]. The virus signatures in ClamAV database tripled in number in less than two years. The database is updated multiple times a day and currently contains more than 500,000 signatures [1].

One of the common solutions for multiple string matching problem is Aho-Corasick (AC) [4], a linear time algorithm based on an automata approach. A linear time algorithm is optimal in the worst case, but as demonstrated in Boyer-Moore (BM) [7] algorithm, skipping a large portion of the text is often much faster, especially when the focus is mainly on searching the occurrences of rare patterns or unwanted patterns that should not occur at all, which are the common cases in many of the multi-string matching applications. Commentz-Walter [13] presented an algorithm that combines the BM algorithm with AC algorithm. Later, Baeza-Yates [26] gave another algorithm using an improved BM. Coit [12] also implemented a combined algorithm based on the work described by Gusfield's book [17]

and applied it to Snort [27], a light weight Network Intrusion Detection System (NIDS). In practice, some applications use both of the two algorithms and apply them in different cases. One example is ClamAV, an anti-virus scanning software using BM for regular pattern matching and AC for wildcard pattern matching. However, BM is designed for single string matching. When dealing with a large number of strings, it has to scan the text once for each of the patterns.

This thesis proposes a new algorithm, Bits Filter (BF), based on the parallel bit stream technique, which speeds up traditional byte-at-a-time processing by using SIMD (single-instruction multiple-data) registers to extract bit streams and enable a multi-byte at-a-time processing. For example, the 128-bit SIMD registers of the SSE or Altivec and the 64-bit SIMD registers of MMX can be used to process 16 or 8 bytes at-a-time. Transformation from byte stream to eight parallel bit streams provides the possibility to process 128 positions at-a-time in some high performance SIMD text processing applications [11, 9].

BF is a three-pass filtering algorithm. Filtering processes include both text filtering and pattern filtering. In the first filter, the text is partitioned into equal-size segments with a single signature calculated per segment. The signature is then used as the index of a bitmap to rule out the impossible matches of text segments. In the second filter, another bitmap is applied, but only to those candidate segments remaining from the first filter. In the final filtering process, new signatures for the rest of the segments are calculated and looked up in a much larger signature array. This signature array maps each signature to a small set of patterns and positions for full match evaluation (exact matching).

The intuition of the three-pass filter is the following. Currently, one cache miss may cost hundreds of CPU cycles and the processor-memory performance gap keeps increasing [22]. During the filtering process, the CPU has to access lookup table frequently. A small filter that fits within the cache is ideal to avoid data cache misses. At the same time, in order to support hundreds of thousands of patterns, the lookup table needs to have as many entries as possible so that the false positive ratio could be low. However, if each entry contains pattern and position information, the filter quickly grows too large for the cache. By using one bit per entry, the filter size in the first two passes is minimized. Therefore, a three-filter structure is applied and a lightweight lookup table, Bit Map, is used in the first and second filters to substantially reduce the work load of the third filter where a much larger lookup table, Signature Table, is needed.

To evaluate the performance of this algorithm, extensive tests are run and comparisons

are made with the implementations of AC algorithm that are extracted from Snort and ClamAV. We apply both random pattern sets, which ensure the generality and rareness of matches and the virus databases, which show the applicability of BF algorithm. Those tests showed a substantial improvement of the performance.

The rest of this thesis is organized as follows. Chapter 2 introduces the background of multiple string pattern matching and its application in malware detection. Chapter 3 presents the architecture and implementation of BF algorithm. Chapter 4 analyzes the performance with various parameters such as text size, pattern set size, filter size and segment size. Chapter 5 shows performance comparisons with the AC algorithm extracted from Snort and ClamAV as well as some other multiple string pattern matching algorithms. The thesis concludes with Chapter 6, which discusses the results and future work.

Chapter 2

Background

2.1 Basics of String matching

The problem of a string matching is that given a string P called the pattern and a longer string T called the text, find an occurrence, if any, of pattern P in text T.

BM is the best-known algorithm for single string matching. It compares the search pattern with the input text starting from the rightmost character of the search pattern. The idea is to skip the text as much as possible. If the corresponding text character does not appear in the pattern at all, we can skip the text for the length of the search pattern. If it appears in the pattern, we can also skip a certain length of the text depending on the position of the matching character.

Horspool [18] gave a simplified Boyer-Moore algorithm, which trades space for time. It does some preprocessing of the pattern, as a means of determining how far to skip ahead in the text if an initial matching attempt fails. The results are stored in a vector that is used during the searching process.

Baeza and Conned [6] developed a bitap algorithm (also known as shift-or method). This algorithm uses a set of bitmasks containing one bit representing each element of the pattern. Then it is able to perform bitwise operations, which are extremely fast and solve the exact matching problem very efficiently for small patterns.

Although some improvements have been done to BM, to extend this single string matching algorithm to a multiple string matching algorithm, we need to apply the single pattern algorithm to the text for each pattern. Obviously this does not scale well to larger sets of patterns to be matched.

2.2 Multiple String Pattern Matching

The problem of a multiple string matching is that given multiple strings called the pattern set P and a longer string called the text T, find an occurrence, if any, of one or more patterns from P in text T.

Fist and Varghese [23] extended the BM algorithm to support a set-wise string matching. They build a trie by using the end-point of the shortest pattern as the initial position and storing the reversed patterns. Then the comparison can be done by looking from right to left until a character is found in the text that does not match the next character in the trie. This method avoids the problem of applying the BM algorithm separately for each pattern. However, it only performs better than AC with small pattern sets, say less than 100 patterns. This cannot satisfy the current needs for most of the security applications.

One of the widely used multiple string matching algorithms is AC. It works by first constructing a finite state machine from the search strings and then running the text as input through the automaton. Unlike the skip based method, this state machine based algorithm is not affected by the size of the smallest or largest pattern in a pattern set and its worst-case and average-case performance are the same. Although it can achieve a better performance for larger set of patterns, the size of the pattern set still has a great effect of the performance for AC, because smaller pattern sets can fit into the cache and benefit from memory caching.

Another multiple string matching algorithm was proposed by Wu and Manber [33]. It is based on the bad character heuristic similar to BM but using a shift table constructed by preprocessing all the patterns. When scanning text, the hash value of current text position is computed and used as an index into the shift table. A multiple byte shift may be applied or a match evaluation may be required at the current position before advancing. However, this algorithm supporting tens of thousands patterns still suffers from the memory problem.

Several optimized versions of the AC algorithm are developed by Norton for Snort [25] to cut down the memory consumption, but they still generate a lot of data cache misses as the number of patterns increases in order to meet the actual needs. The optimized AC required 28 MB memory for the Snort rule set with about 1,500 rules [25], while ClamAV required 10M for a database of 20,000 signatures [16].

2.3 Bloom Filter

Bloom filter is a space efficient probabilistic data structure for performing set membership queries. It can result in false positives, but never gives false negatives. An initial filter is a bitmap of m bits, all set to 0. k hash functions are defined and each of them maps a key value to one of the bitmap positions. A bit in the filter is set to one if and only if one or more keys hash to that location. To query for an element, calculate the keys by hashing the element with the k functions. If any of the positions are 0, the element is not in the set. If they are all set to 1, then it is possible that this element is in the set.

Assume all the hash functions select each position with the same probability. The probability of getting a false positive with n inserted elements is following [8]

$$(1 - (1 - \frac{1}{m})^{kn}))^k \approx (1 - e^{-kn/m})^k$$

Obviously, for a given n (the number of inserted elements) and k, the probability of false positives decreases as m (the number of bits in the array) increases.

Although Bloom filter may result in false positives, it has a significant space advantage over other data structures for representing sets, such as binary searching trees, tries, hash tables and so on. Because most of them require storing at least the data items themselves while Bloom filter represents the items with one single bit.

Bloom filter is now used in many applications. Some of the algorithms that applies Bloom filter will be discussed in the next section.

2.4 Multiple String Matching in Security Applications

2.4.1 Deep Packet Inspection

Deep Packet Inspection(DPI) is used to prevent attacks such as buffer overflow, Denial of Service, sophisticated intrusions, and virus or worms inside a packet. DPI devices have the ability to look at headers and data protocol structures as well as the actual payload of the messages. An important building block of these processors, a real-time string matching has to keep pace with the high speed network and performs at a wire speed.

Many hardware-based techniques are developed to provide high performance string matching. Programmable Gate Arrays (FPGAs), a reconfigurable hardware technology

has added new dimension to DPI.

A commonly used algorithm for FPGA-based string matching is based on automata.

Sidhu [29] proposed a fast construction of Nondeterministic Finite Automation(NFA) for regular expression matching. Once constructed, NFA as well as Deterministic Finite Automata(DFA) can process each text character in a constant amount of time for a regular expression. This method was implemented on both FPGA and Self-Reconfigurable Gate Array(SRGA) and compared with grep on a 800 MHz Pentium III machine. However, the main concern of this paper is a real-time state machine construction, therefore NFA is chosen instead of DFA.

Later, Moscola [24] provided a content-scanning module for an Internet firewall using DFA. Since all the signatures are known and can be preprocessed, DFA is preferable for multiple string matching because DFA can only have one activate state and thus are generally more compact.

A more recent work done by Kumar [19] introduced a new representation for regular expressions, called the Delayed Input DFA (D_2FA). D_2FA is constructed by replacing several transitions of the automation with a single default transition, which reduces space requirements.

However, all the state machine based algorithms suffer the scalability problem. The states can consume many hardware resources and lower the efficiency. Therefore, most of the hardware-based state machine method only focuses on a small set of patterns. Moscola tested their content-scanning module with only 21 regular expressions. Sailesh used datasets with less than a thousand expressions.

Another approach that has been widely used is Bloom filter, which works with a larger set of patterns.

Suresh [31] built a 256 bytes Bloom filter on FPGA (XC2V8000), which can process 256 bits per cycle. However, this method only deals with patterns of length eight.

Dharmapurikar [15] described a technique using parallel Bloom filters. It grouped the predefined signatures by their lengths and stored in a set of parallel Bloom filters in hardware so that each Bloom filter contains the signatures of particular length. Each string can be tested for its membership in the Bloom filter. Positive results from the Bloom filter are then candidates to be supplied to an analyzer module to decide whether they are full matches or false positives. A prototype was implemented on Field Programmable Port Extender(FPX) [20], which is a platform that performs data processing in FPGA hardware. Although this

method was designed for fast Bloom filter processing of up to 10,000 patterns, the final analyzer in Dharmapurikar's system operated quite slowly at 20 times the clock period of the system.

Song and Lockwood [30] improved Dharmapurikar's work and develop a new algorithm named Extended Bloom Filter (EBF). EBF eliminates the need of another process for match verification. Instead, the pattern informations for exact matching can be gained from the Bloom filter to accelerate the lookups.

However, this doesn't completely solve the problem of a limited resource for hardware-based methods. They cannot offer searching solutions for the increasing pattern sets size and the number of Bloom filters they could use also set the limit of the number of distinct pattern lengths. Although EBF segments the longer patterns into a set of sub-patterns, adding extra process and two more tables can decrease the throughput. Moreover, the clock frequency of those hardwares are measured by MHz and they are much slower than the commodity processors that are generally several GHz.

Some software-based algorithms are also developed for commodity processors. Previously mentioned algorithm proposed by Fisk and Varghese and the BM-AC-combined algorithm developed by Coit also focus on the fast string matching solution for intrusion detection.

Lu [21] also designed a memory efficient string matching algorithm called transition-distributed parallel DFAs (TDP-DFA). It employed efficient representations for the transition rules in each DFA based on the overlapping relationships among real-life NIDS signatures. It also used a scheme to share storage of transition rules among multiple DFAs, which decreases the total storage cost.

Tuck [32] presented two string matching algorithms based on AC that can reduce the memory usage to as low as 2% of that required by previous algorithm while maintaining bounded worst case performance. The first one is called Bitmap Compression, which uses a single pointer to point to the first valid next state and maintains a 256 bit bitmap instead of having 256 next state pointers. The second one is called Path Compression, which handles nodes lower in the tree by squeezing as many of these sequential nodes as possible into a single node. Both of these algorithms have a similar performance with the optimized AC and Wu-Manber algorithm on average case, but can achieve a much better performance on worst cases, which is just slightly higher than the original AC.

However, none of the existing software-based implementations are able to match the

speed of fast links.

2.4.2 Snort

Snort is an open source Network Intrusion Detection System (NIDS). NIDS is software and/or hardware designed to detect malicious attacks through a network.

Since most of the known attacks can be represented by strings or multiple substrings, Snort uses a set of rules to define suspicious activities that includes at least a type of packet to search, a string to match, a location where that string is to be searched for and the corresponding action.

Unlike the traditional network based security devices such as firewalls, NIDS scans the network traffic packets by searching those strings in not only packet headers but also payloads. As a result of the complexity of string search algorithms, this matching process in Snort costs 40-70% of the total processing time and takes 60-85% of the processor instructions [5].

In earlier days, Snort used brute force pattern matching, which is very slow. Later, a partial BM algorithm was implemented to improve the performance. After that, AC algorithm is applied and used as the next generation [14].

Snort is currently using an optimized AC algorithm to perform the searching process. Norton [25] presents some basic sparse matrix and vector storage formats and applies one to the AC state table to implement the optimized AC algorithm. The selected sparse storage method provides solutions to minimize memory and offers better performance than the original AC. This algorithm was also compared with Wu-Manber, which is generally a very fast algorithm on average, but does not have a good worst-case guarantee.

2.4.3 ClamAV and Related Work

ClamAV is an open source anti-virus software providing a command line scanner, a multi-threaded daemon and a tool for automatic database updates. The database consists of a large set of signatures. A signature is a unique pattern that is specific to a piece of malicious code. There are more than 500,000 signatures including both regular signatures and signatures with wild-card characters that are used for detecting polymorphic viruses.

Viruses are polymorphic if their body is self-changing during replication to avoid the presence of constant search strings. Such viruses cannot be detected using a fixed signature.

As a result, ClamAV represents polymorphic viruses using multi-part signatures with wildcards in order to map down multiple versions of a virus originating from the same source. This method can handle many but not all polymorphic virus, other systems may apply more complex method such as known plaintext cryptanalysis, statistical analysis, emulation and so on.

The current ClamAV database has 500,000 signatures where about 1% are polymorphic signatures represented by the following format.

- ?? Match any byte;
- a? Match high nibble (high four bits);
- ?a Match low nibble (low four bits);
- * Match any number of bytes;
- `{n}` Match n bytes;
- `{-n}` Match n or less bytes;
- `{n-}` Match n or more bytes;
- `(a|b)` Match a or b (more alternate characters is allowed).

One of the approach that improves the performance of ClamAV is called HashAV [16]. It is a virus scanning technique that adds a Bloom filter before ClamAV's scanning process to determine if the data needs to go through an exact-match process. Specifically, the algorithm moves a sliding window down the input stream. For the bytes under the window, several hash functions are applied iteratively to calculate their hashes. Each hash is looked up in the Bloom filter. If there is a hit, it moves on to the next hash function. Otherwise, it immediately slides the window and process the next window. If a hit occurs with all of the hash functions, the exact-match algorithm is them applied to see if there is a match with the virus.

However, the scanning algorithm used in HashAV is not able to process signatures with wildcards. To address this problem, they provide a "two-scan" approach where all the signatures with wildcards are taken care of by the original ClamAV algorithm in the second scanning process.

Another recent work called Multi-block Recursive Shift Indexing(MRSI) [34] is also designed for anti-virus scanning and applied to ClamAV. MRSI is based on RSI, which is a multiple string matching algorithm designed for IDS application. The core idea of RSI is to use two levels of block heuristic to produce shifts instead of using one character as BM did. But the probability of getting two zeros from the two shifts tables increases quickly as the number of pattern grows. Therefore, multiple blocks are used to decrease this ratio. However, this BM-based algorithm still suffers the low probability of non-zero shift with larger pattern sets. The performance of this algorithm shows a substantial decrease as the number of patterns grows and the comparisons are only done with an extended version of BM. Furthermore, this algorithm cannot deal with wildcard patterns.

Chapter 3

A Fast Three-Pass Multi-string Matching Algorithm

3.1 Problem Statement

Let P = $\{P_1, P_2, ..., P_k\}$, where $k > 1$, be a set of patterns. Let pattern $P_i = p_{i,1}, p_{i,2}, ..., p_{i,n_i}$ be a character string. Let T = $t_1, t_2, ..., t_j$ be a large text. The problem is to find any match with pattern P_j at position m where there is a $p_{j,x} = t_{m+x-1}$ for all x in $1, 2, ..., j$.

We note that, in real life applications, tens or even hundreds of thousands of patterns need to be supported and most of them are unlikely to match the text.

3.2 Algorithm Overview

As shown in Figure 3.1, this algorithm has two stages. The first stage is the preprocessing of the set of patterns. The second (scanning) stage, includes three filtering processes and a full match evaluation process. In the preprocessing stage, two Bit Maps and a Signature Table are constructed. In the scanning stage, we first use one of the Bit Maps to filter out those segments that cannot match any pattern. We stored the positions of "possible matching" segments and supply it to the next filter. The candidates are then checked and ruled out by the other Bit Map in the second filter and sent to the final filter where the Signature Table is used to further narrow down the text segments and greatly decrease the number of pattern candidates for each text segment. Information from the Signature Table is also used in the Full Match Evaluation Module for identification of pattern candidates.

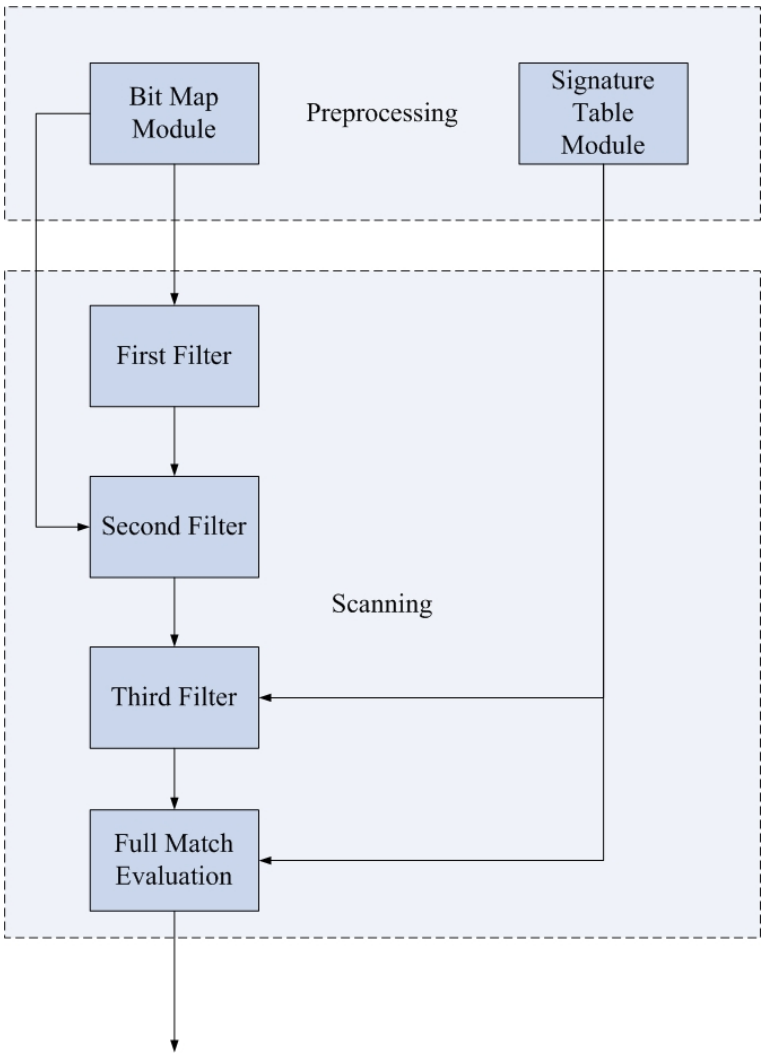

Figure 3.1: Bits Filter Architecture

The filters can determine the majority of the non-matching cases quickly and improve the performance since most of the patterns are unlikely to match the text. The three-pass structure helps to achieve the greatest possible filtering effect by devoting as much memory to first two filtering processes as possible. That is, just one bit per signature is used in the first two filters for a yes or no decision for potential matches of a segment. Additional information of signature is stored in the Signature Table used in the third filter to identify the position and possible patterns that match candidates.

Section 3.3 introduces the construction of the Bit Map and how the first two filters works with the Bit Map.

Section 3.4 presents the architecture of Signature Table and how it is used in the third filter and full match evaluation process.

Section 3.5 considers the problem of searching patterns with wildcards. It also extends our algorithm so that it has no limitation on patterns length.

3.3 Bit Map

We partition the text into segments of size s. We define a text segment as a subsequence of text, $t_j, t_{j+1}, ..., t_{j+s-1}$ where $j \bmod s = 1$.

We define a pattern segment as a subsequence of pattern $p_{i,k}, p_{i,k+1}, ..., p_{i,k+s-1}$ where i is in range of 1 to s.

We define a partial bits value of size n (PBV_n) a binary value containing n bits extracted from the segment.

We calculate s PBV_ns and s PBV'_ns for s segments of each pattern by shifting the start position and use these PBV_ns and PBV'_ns to construct the first and the second Bit Map separately. We also calculate one PBV_n for each segment of the searching text in the first filter and one PBV'_n in the second filter. The strategy to extract n bits from the segments can be varied but once chosen must be consistently applied to both the pattern set and text segments. Moreover, the bits used to form PBV_n and PBV'_n must be different. Normally, we extract consecutive bits from the segments in order to simplify the calculation of PBV_n by using SIMD operations and we choose lower bits because lower bits tend to contain more important information in some cases such as alphabetical and numeric patterns in which higher bits are the same.

For example, we can take the last bit, bit7, of each character in both the text segments

seg1 seg2 ..... seg8

Pattern	A	B	c	d	E	F	g	h	L	M	n	o	P	Q	r
bit4	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0
bit5	0	0	0	1	1	1	1	0	1	1	1	1	0	0	0
bit6	0	1	1	0	0	1	1	0	0	0	1	1	0	0	1
bit7	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0

	text1																text2															
	Segment1								Segment2								Segment3								Segment4							
text	x	x	x	A	B	c	d	E	F	g	h	L	M	n	o	P	Q	r	A	B	c	d	E	F	a	b	c	d	n	o	P	Q
bit4	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
bit5	0	0	0	0	0	0	1	1	1	1	0	1	1	1	1	0	0	0	0	0	0	1	1	1	0	0	0	1	1	1	0	0
bit6	0	0	0	0	1	1	0	0	1	1	0	0	0	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0	1	1	0	0
bit7	0	0	0	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0	1	0	0	1	0	1

Figure 3.2: Example

and pattern segments to form the PBV_n and bit6 to form PBV'_n. As shown in Figure 3.2, s = 8, the eight PVB_8 of the eight consecutive pattern segments, ABcdEFgh, BcdEFghL, cdEFghLM, dEFghLMn, EFghLMno, FghLMnoP, ghLMnoPQ and hLMnoPQr, are 0xAA, 0x55, 0XA9, 0x52, 0XA5, 0x4A, 0x95 and 0x2A. The eight corresponding PBV'_8 are 0x66, 0xCC, 0x98, 0x31, 0x63, 0xC6, 0x8C and 0x19. The four PBV_8 of the four text segments, 000ABcdE, FghLMnoP, QrABcdEF, abcdnoPQ are 0x15, 0x4A, 0xAA and 0xA5. The four corresponding PBV'_8 are 0x0C, 0xC6, 0x59 and 0x6C.

In order to generate s PBV_ns, there is a limitation on the pattern length. Later we will discuss the wildcard BF algorithm, which will treat short pattern as wildcard patterns and therefore has no limitation on the pattern length. Here, the length of a pattern, L, has to be at least $2 \times s - 1$, because we need s different pattern segments of length s. The segments don't have to be consecutive, but their start positions i mod s have to different. Since there are no wildcards involved, which will change the false positive ratio and affect the efficiency, we can simply choose the s consecutive segment from position 0 to $s - 1$.

Bit Map is a series of 2^n bits, where each bit individually corresponds to the existence of a PBV_n of a pattern. For example, given a Bit Map x1xxx1...x, it means that there exist at least one pattern containing a PBV_n that equals to 1 and one containing a PBV_n that

equals to 5.

A Bit Map is initialized with all zeros and is set to one at the position where the PBV_n it represents exists in one of the patterns. In the algorithm, we use an integer array as Bit Map. The high bits of the PBV_n are used as the index to locate the position in the array and the low 5 bits (if integer is 2^5 bits) are used to calculate the values. For example, as shown in Figure 3.2, the PBV_n of the first segment, ABcdEFgh, is 0xAA. The high 3 bits of 0xAA is 5 and the low 5 bits is 10. Therefore, the 11th bits from the end of `bit_map[5]` is set to one.

```
bit_map [5] = xxxxxxxxxx1x...xx
```

We define a bit stream as a sequence of bit values in one-to-one correspondence with the character in the searching text. To calculate the PBV, we first use a SIMD operation, `movemask`, to get the corresponding bit stream.

```
bit7 = simd_movemask(simd_slli_8(text,7));
```

The movemask operation takes the first bit from each byte within the 128-bit register to give a 32-bit integer value with 16 leading zeros followed by the 16 selected bits. By shifting the selected bits to the first position, we can get all the bit streams we need. It is much more efficient than extracting necessary bits from every byte and combining them together.

In the example shown in Figure 3.2, to calculate the PBV_8s of the four text segments, we only need bits7, the last bit stream from the text. The code below shows the calculation of these four PBV_8s. We can then apply this method to calculate PBV_8' by extracting the second last bit stream. PBV_8 and PBV_8' of patterns segments can also be calculated in the same way. But since this calculation is done in preprocessing stage and won't affect the scanning speed, we can also use byte at time method.

```
bits7 = simd_movemask(simd_slli_8(text1,7)) = 0x154A
pbv1 = bits7 >> 8; ( xxxABcdE: 0x15)
pbv2 = bits7 & 0xFF; ( FghLMnoP: 0x4A)
bits7 = simd_movemask(simd_slli_8(text2,7)) = 0xAAA5
pbv3 = bits7 >> 8; ( QrABcdEF: 0xAA)
pbv4 = bits7 & 0xFF; ( abcdnoPQ: 0xA5)
```

From the result of this code, we can see that the PBV_8 of both text segment two and four match one of the PBV_8 of pattern ABcdEFghLMnoPQr. Therefore, we store the start

position 9 and 25 of these two text segment as a candidate position and pass it to the second filter and kick out the first and the third text segment.

We now show that all the text segments whose PBVs don't exists in Bit Map can be ruled out. Suppose PBVs of text segment $t_j, t_{j+1}...t_{j+s-1}$ is not set in Bit Map. This text segment won't match any subsequence of the first $2 \times s - 1$ character of pattern in the set, because all the PBVs of these pattern segments are set in Bit Map. One possibility is that this text segment matches $p_{i,s+n}, p_{i,s+n+1}, ..., p_{i,2s+n-1}$ where $n > 0$. Let $x = (n + s) \bmod s$, $p_{i,x}, p_{i,x+1}, ..p_{i,s+x-1}$ must match with $t_{j-(s+n-x)}, t_{j-(s+n-x)+1}...t_{j-(n-x)}$ as long as $j - (s + n - x) > 1$. Since $j \bmod s = 1$, $n \bmod s = x$, as a result, $j - (s + n - x) \bmod s = 1$, which means $t_{j-(s+n-x)}, t_{j-(s+n-x)+1}, ..., t_{j-(n-x)}$ must be a text segment whose PBV will be checked in the filters. Therefore, all the patterns that text segment, $t_j, t_{j+1}...t_{j+s-1}$ could match would be covered by other text segments. Another possibility is that this text segment doesn't match any subsequence of the pattern. In both of cases, text segment $t_j, t_{j+1}...t_{j+s-1}$ could be considered as non-matching cases. This method improves the performance by processing s bytes at a time instead of one byte at a time.

In the second filter, PBV'_ns are calculate for each candidate segments. In the example of Figure 3.2, PBV'_8 of text segment at position 9 and 25 is 0xC6 and 0x6C. 0xC6 matches the sixth PBV'_8 of pattern ABcdEFghLMnoPQr while 0x6C doesn't match any PBV'_8s. So we pass position 9 to the third filter for further processing.

3.4 Signature Table

Signature Table is an array of pattern lists where each pattern in the same list contains the same signature value. To calculate the signature, we can apply a similar method used for calculating PBV and use this value as the index of Signature Table. Since we are not worried about the cache behavior in this filter, we usually extract a large number of bits from each segment so that during the third filtering process we could get rid of more text segments and feed less possible matches to the Full Match Evaluation Module. Let 2^m be the size of the Signature Table. Therefore, we need to extract m bits from each segments. It is meaningless to choose the same bits that are used to calculate PBV_n since they are already matched in the first filter. Ideally, we would like to use completely different bits for PBV'_n. The mutual-exclusiveness of the two PBVs can lower the false positive ratio. However, when dealing with small segment and large filter size, there might not be enough

distinct bits we can select. In this case, overlaps are allowed.

The Signature Table is responsible for filtering not only the text segments but also the pattern segments. Different from the Bit Map, which only has only one bit for each PBV to tell whether a text segment is possible to match a pattern or not, Signature Table entries provide a list of patterns that could each possibly match this segment. Since the number of the Signature Table entries is much larger than the number of signatures, there is a good chance that we only need to compare with one pattern or a very small set of pattern in Full Match Evaluation Module.

Therefore, certain information is needed in the Signature Table including the pattern, pattern length and the start position to calculate the corresponding signature. As shown in Figure 3.2, bit4 and bit5, are extracted to represent a 16-bit signature. For example, the first segment ABcdEFgh in pattern ABcdEFghLMnoPQr generates signature 0x011E and the next segment BcdEFghL generates signature 0x033D and so on. The last segment of this pattern hLMnoPQr has a signature of 0xF878 at position 8. We can then append pattern information including string "ABcdEFghLMnoPQr", length 15 and start position 1 to pattern list `sig_tab[0x011E]` and string "ABcdEFghLMnoPQr", length 15 and start position 2 to `sig_tab[0x033D]` and go on until the last piece of information is appended to `sig_tab[0xF878]`. Since each pattern has s signatures base on s start positions from 0 to s-1, one pattern is stored in at most s times in Signature Table.

In the last filter, we first calculate the signature of each possible matching position gathered from the previous filter. As shown in Figure 3.2, the signature of candidate position 9 in the text is 0x3EDE. We can then look up Signature Table and get pattern list `sig_tab[0x3EDE]`. If it is not empty, we will go through each pattern in the list. In this list, we should find pattern ABcdEFghLMnoPQr with start position 5.

Then we can send pattern ABcdEFghLMnoPQr and its matching position 9-5=4 (9 is the candidate position of the text and 5 is the start position of signature 0xC64A) to the Full Match Evaluation Module. In the Full Match Evaluation Module, we compare the text at the fourth position with the pattern ABcdEFghLMnoPQr and find out whether it is an exact match or not.

3.5 Wildcards

Bits Filter also supports patterns with wildcards. The basic idea is to extend PBV and signature with all the possibilities. For example, suppose there is a pattern ABcdE?ghLMnoPQr ("?" matches any single characters). Let s = 8 and PBV is formed by the last bit of each character. Since the last bit of the missing character could be either 0 or 1, we should set the Bit Map to contain the two PBVs that result. For example, the first 8 bytes ABcdE?gh has two possible PBVs, 10101010 and 10101110; these are both placed in the Bit Map. Therefore, any text position that matches one of the values is a candidate for further checking.

With the wildcard extension, restriction on minimum pattern length can be relaxed, because all the shorter patterns can be treated as pattern of $s \times 2 - 1$ characters with question marks at the end. For example, pattern AbcdEFghLMno can be treated as AbcdEFghLMno??.

For wildcards representing random characters of arbitrary length, such as "*" or "`{n}`", we can split the pattern by those wildcards so that each resulting subsequence contains only fixed length wildcards. Each subsequence can then be processed by the method shown above. However, we only choose one subsequence for filtering and don't consider other subsequences until the full match evaluation process. Generally, the target subsequence is the longest one to minimize the number of PBVs needed. One reason is that the length of selected subsequence could be less than $s \times 2 - 1$. In this case, the fewer wildcards added at the end, the higher probability of the non-matching cases and the better performance we will get. Another reason is that longer patterns with wildcards can be optimized with the following method to reduce the number of PBVs.

The basic idea of this method is that instead of using pattern segment at position n to calculate PBV, we are allowed to use any segment at position $n + m \times s$ where n is in $1, 2, ..., s$ and m could be any integer as long as $n + m \times s + s - 1$ is less then the total length of the pattern. We can then select one of the pattern segments that doesn't have a wildcard or has the minimum number of wildcards. For example, we have a pattern ABcdE??hLMnoPQrABcdEF. Instead of calculating the 8 byte ??hLMnoP at position 6 we can choose the 8 bytes QrABcdEF at position 6+8. Clearly, ??hLMnoP has 4 possible PBVs 00001010, 10001010, 01001010 and 11001010, but QrABcdEF only has one, which is 10101010. Therefore, we reduced the number of elements inserted to the lookup tables. To see why this won't affect the correctness of the original algorithm, suppose there is a

match between pattern segment $p_{i,n}, p_{i,n+1}, ..p_{i,s+n-1}$ and text segment $t_j, t_{j+1}, ..., t_{j+s-1}$. Although the match won't exist after we replace PBV of $p_{i,n}, p_{i,n+1}, ..., p_{i,s+n-1}$ by PBV of $p_{i,n+m\times s}, p_{i,n+m\times s+1}, ..., p_{i,n+m\times s+s-1}$, we know that there must be a match between the new pattern segment $p_{i,n+m\times s}, p_{i,n+m\times s+1}, ..., p_{i,n+m\times s+s-1}$ and subsequence of text $t_{j+m\times s}, t_{j+m\times s+1}, ..., t_{j+m\times s+s-1}$. Since $j \bmod s = 1$, $j + m \times s \bmod s = 1$ and text subsequence $t_{j+m\times s}, t_{j+m\times s+1}, ..., t_{j+m\times s+s-1}$ must be a text segment. Therefore, position $j + m \times s$ will be sent to the next stage as a candidate position replacing j.

Although Bits Filter is able to deal with pattern of arbitrary length and with all wildcards, both single-position and variable-length, the added signatures for wildcards could increase the probability of matches and adds work load to all of the filters. Let W denote the number of wildcard bits that we used to calculate PBVs or signatures. Since all of bits can be either 0 or 1, the number of combination is 2^W, which means for a pattern segment, instead of inserting one signature to the lookup tables, we inserted $2^W - 1$ more. Therefore, there are practical limits on using Bit Filter on pattern with too many unknown characters or with very short length.

3.6 Implementation

The algorithm was implemented in C and compiled with GCC 4.1.2 with optimization level 3 on Ubuntu Linux 8.04. The non-wildcard algorithm running with random database requires 1000 lines of C code. There are also two python scripts involved. One was used to compile the algorithm with different parameters, such as filter order, segment size and several evaluation parameters as well. Another one was an execution script that can run programs generated by the compilation script on different pattern sets and text files. Choosing different parameters can make a substantial difference to the performance. Analysis will be discussed in the next Chapter.

Chapter 4

Analysis

All experiments in this thesis were conducted on a 2.1 GHz Intel Core2 Duo processor desktop machine with 2GB of memory and 2MB cache. Performance Application Programming Interface (PAPI) Version 3.5.0 [2] toolkit was installed on the test system to facilitate the collection of hardware performance monitoring statistics.

4.1 Preprocessing

Most of the security applications applies the model of a continuously-running process, in which the patterns that need to be detected are known and the preprocessing cost is paid only when the process needs to be started up again or the pattern set needs to be updated. Therefore, the time required for constructing the lookup tables is not a big concern.

Nonetheless, the preprocessing can be done very quickly. Although the calculation seems quite complicated, with SIMD operations, only 0.54 seconds is needed for 100,000 patterns.

4.2 Scanning Model

Assume that B_n is the processing time per lookup for the nth filter. This value includes the time consumed by basic operations required for each lookup and the cost of data cache misses caused by assessing the lookup table.

According to the algorithm, if we know the text size is T bytes and segment size is S bytes, the number of lookups for the first filter L_1 is T/S. Therefore, the processing time of the first filter is $B_1 \times T/S$.

With false positive ratio of the first filter $R_1(R_1 < 1)$, we can calculate the number of lookups for the second filter $L_2 = R_1 \times T/S$. Similarly, the number of lookups for the third filter $L_3 = R_1 \times R_2 \times T/S$.

Thus, the processing time of the algorithm is

$$B_1 \cdot \frac{T}{S} + B_2 \cdot \frac{T}{S} \cdot R_1 + B_3 \cdot \frac{T}{S} \cdot R_1 \cdot R_2 = \frac{(B_1 + B_2 \cdot R_1 + B_3 \cdot R_1 \cdot R_2) \cdot T}{S} \tag{4.1}$$

This processing time is measured as CPU cycles per byte. Thus, we would like this value to be as small as possible.

4.3 False Positive Ratio

The probability of getting a false positive for the Bloom filter is $(1 - e^{-kn/m})^k$. Since we only use one hash function, therefore $k = 1$. The number of inserted elements $n = P \times S$, because each pattern will be inserted S times for S different positions. The number of the bits in the Bit Map is 2^F, which is the filter size.

The false positive ratio R_n can be calculated by

$$R_n = 1 - e^{-P*S/2^{F_n}} \tag{4.2}$$

According to Eq.4.1, to bring down the processing time, this ratio has to be small. Since B_3 is much larger than B_1 and B_2, the effect of this ratio becomes more important.

4.3.1 Segment Size (S)

From Eq.4.2 we can see that the larger the segment size the higher the false positive ratio. But segment size is also one of the important factors to decide the processing time shown in Eq.4.1. In Eq.4.1, segment size has a direct effect on the processing time while in 4.2, we can achieve a desirable ratio by adjusting the filter order F. Therefore, in general cases, we would like to use a larger segment size for longer patterns.

4.3.2 Filter Order (F)

Filter order is one of the key factors for R. Ideally, We would like F to be as large as possible so that we could have a smaller R. However, given a limited cache size, increasing

Pattern Set Size	Filter Order								
	21	22	23	24	25	26	27	28	29
10k	0.142	0.073	0.037	0.019	0.009	0.005	0.002	0.001	0.0006
20k	0.263	0.142	0.073	0.037	0.019	0.009	0.005	0.002	0.001
30k	0.367	0.204	0.108	0.056	0.028	0.014	0.007	0.004	0.002
40k	0.457	0.263	0.142	0.073	0.037	0.019	0.009	0.005	0.002
50k	0.534	0.317	0.174	0.091	0.047	0.024	0.012	0.006	0.003
60k	0.600	0.367	0.204	0.108	0.056	0.028	0.014	0.007	0.004
70k	0.656	0.414	0.234	0.125	0.065	0.033	0.017	0.008	0.004
80k	0.795	0.457	0.263	0.142	0.073	0.037	0.019	0.009	0.005
90k	0.747	0.497	0.291	0.158	0.082	0.042	0.021	0.010	0.005
100k	0.783	0.534	0.317	0.174	0.091	0.047	0.024	0.012	0.006

Table 4.1: False Positive Ratio

the filter order will cause larger data cache misses which means we will end up with a large B_n. The tradeoff between these two will be discussed in the next section.

4.3.3 Number of Patterns (P)

Larger pattern sets will cause larger false positive ratios. However we usually cannot decide the number of patterns. It is a requirement for the algorithm to work fast with a certain amount of patterns. But it is important to know the relationship between P and R so that we can choose the right F to get the best performance.

Table 4.1 shows the false positive ratio of different pattern set size with different filter order. The segment size used to calculate this table is 32.

As shown in this table, even for a large pattern set, we can easily pick two filter orders and get a less than one percent ratio for third filter.

4.4 Data Cache Misses

As previously defined, B_n is equal to the time consumption of basic operations plus the data caches misses per lookup. The number of CPU cycles needed by the basic operations is a fixed number calculated from the algorithm. In order to lower cost of each lookup, we have to decrease the data cache misses. There are several ways to solve this problem.

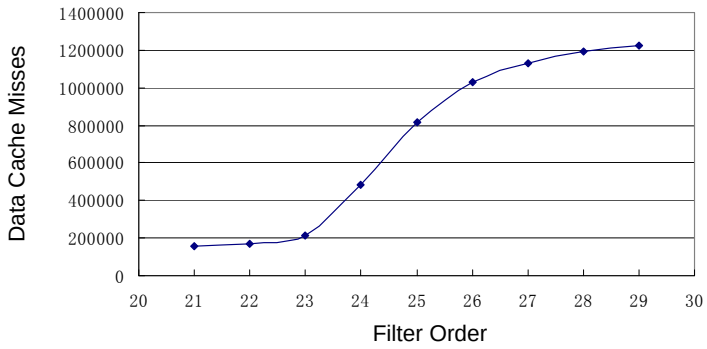

Figure 4.1: Data Cache Misses of First Filter (P=100k,T=10M,S=16)

4.4.1 Filter Size (F)

The most significant optimization to the cache behavior is achieved by decreasing the filter size. A small filter that fits within the cache is ideal to avoid data cache misses.

Fig 4.1 shows the data cache misses of the first filter when we scan 100,000 patterns on a 10M file using segment size 16.

The reason we discuss the cache behavior by observing the first filter is because the numbers of lookup for this filter are the same when we test on the same size of text file. The numbers of lookup for the second and third filter are different as the filter order changes. However, by considering the false positive ratio, we can find a similar cache behavior in the second filter. Here we only use the first filter as an example.

From Fig 4.1, we can see a substantial increase when the filter size exceeds 2^{23} bits. It means that when the filter can no longer fit into the cache, the number of data cache misses will go up quickly.

The rate of increase in data cache misses abates after filter size exceeds 2^{27} bits, because basically for each memory access to the lookup table, in this case, the Bit Map, there is a great chance of generating a data cache miss.

Fig 4.2 shows the data cache misses of the first filter with same number of patterns on a same text file except the segment size we are using is 32 bytes. Although the number of

Figure 4.2: Data Cache Misses of First Filter (P=100k,T=10M,S=32)

Figure 4.3: Data Cache Misses of First Filter (P=10k,T=10M,S=32)

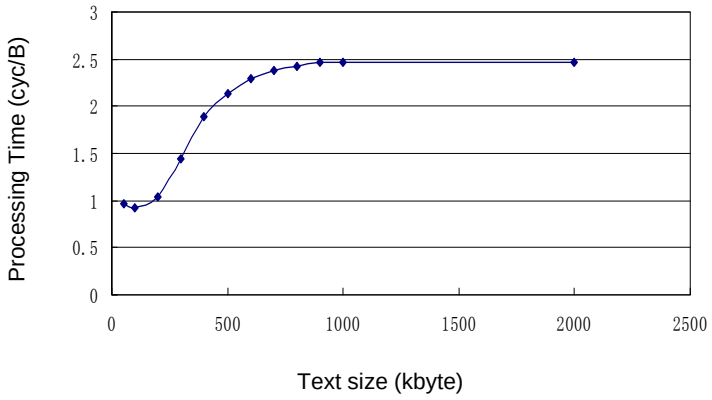

Figure 4.4: Processing Time for Different Text Size (F=29, S=32, P=100k)

lookups ($L_1 = T/S$) is cut into half, the data cache misses of filter order from 21 to 23 is about the same with the number shown in Fig 4.1 because when the filter can fit the cache, it won't create much more data cache misses by adding the number of lookups. However, the data cache misses of filter order 29 is half of what is shown in Fig 4.1 because every lookups in a Bit Map of size 2^{29} bits tends to generate one data cache misses. So when the number of lookups is doubled, the data cache misses is doubled as well.

This Comparison further proves that we can achieve better data cache behavior if the filter size is smaller than cache size.

Increasing the segment size does reduce the number of lookups and give us different experimental result to observe the data cache behavior, but it doesn't have a direct effect on it. In fact, it is the filter size that changes the cache behavior.

Fig 4.3 is similar to Fig 4.2. The only difference is that this measurement is tested with 10,000 patterns. With a smaller set of patterns, the data cache misses remains the same, which means that the size of pattern set is also irrelevant to data cache misses.

4.4.2 Text Size (T)

Fig 4.4 shows the processing time of our search algorithm running on files of different sizes. When the text file is small and can be fit into the cache, the performance is much better.

Otherwise, each filter is possible to have one more data cache misses cause by the memory access to the text file.

In Intrusion Detection Systems, the size of a search text is usually less than a few thousands bytes. We can easily fit the text file into the cache without affecting the lookup table. However, in virus detection, we usually have to scan a larger file or large number of files. Therefore, keeping the text in the cache is no longer feasible. In fact, we would like to keep the text out of cache to make sure that the Bit Map can stay within the cache.

Another useful information we can get from Fig 4.4 is that the performance becomes stable when the text size is larger than 1M. Therefore, it is reasonable for us to use a 10M text file for testing.

4.4.3 Other Factors

We did some other experiments to improve the cache behavior. For example, to keep the text out of cache, we apply a CLFLUSH instruction[3] after reading the text segments from memory and calculate their $PBVs$. This instruction invalidates the cache line associated with the linear address that contain the byte address of the memory location in all levels of the processor cache hierarchy. Results show that it does cut down the data cache misses but did not improve the performance, because this instruction itself is a big cost for checking the consistency with memory before invalidation.

Another experiment we did is to bypass the cache and write some of the data directly into memory. One of its applications is the text segment positions that have to be passed from the first filter to the second filter. By using this direct storing instruction, the performance is improved whenever the filter size exceeds the data cache. However, when the cache size is enough for the filter, there is no need to keep this position array from the cache.

4.5 Tradeoffs

There are two tradeoff factors, the first is the segment size. As we discussed above, increasing the segment size will increase the false positive ratio, but by giving an observation of Eq.4.1, we can see that segment size plays a more important role to decrease the total processing time.

Fig 4.5 and Fig 4.6 shows the performance of the scanning process with three different segment sizes 8, 16 and 32. Filter order of the first and the second filter are the same in

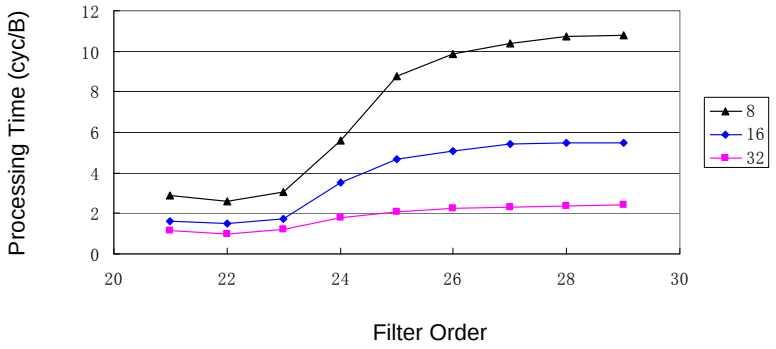

Figure 4.5: Processing Time for Different Segment Size (P=10k,T=10M)

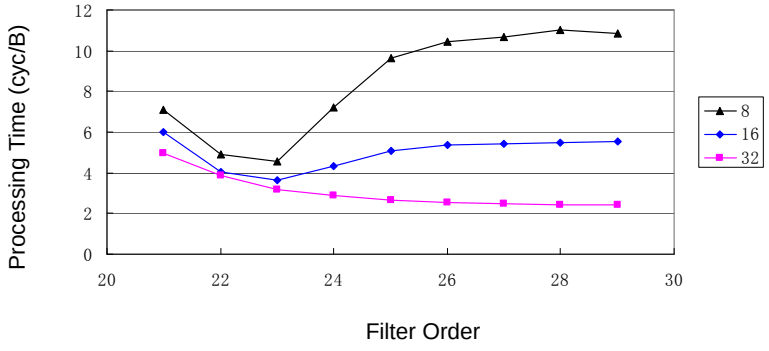

Figure 4.6: Processing Time for Different Segment Size (P=100k,T=10M)

this study. Different filter orders will be applied and their corresponding performance will be discussed later.

In Fig 4.5, we processed 10,000 patterns while in Fig 4.6, we processed 100,000 patterns. In each case, the performance is better with larger segment size.

In Fig 4.5, the best performance is achieved when the filter size is smaller than the cache. However in Fig 4.6, for segment size 32, the processing time keeps dropping down when the filter order increase. When the segment size is 8 or even 16, the ratio of false positive for 100,000 patterns is pretty small. We can get reasonable number when the filter size is smaller than the cache size. But when we use segment size 32 to process 100,000 patterns, as we can see from Table 4.1, its effect to the false positive ratio is more significant.

Even though segment size 32 requires large filter size to balance the false positive ratio, which would cause an increasing of data cache misses, it still gives a better performance in general.

However, we should also remember that the length of pattern has to be at least $2 \times S - 1$ for the non-wildcard algorithm which sets the limit to increase the segment size. Although we could deal with random length patterns with the wildcard algorithm, using large segment size will considerably increasing the ratio of false positive by adding the number of inserted elements to the lookup tables when most of them do not come with the desirable length.

The second tradeoff is the filter size. From the previous section we know that larger filter size can give us lower false positive ratio R but as a result the data cache misses will increase and generate a larger B. So it becomes tricky to pick the right filter size for different situations.

Table 4.2 shows the analysis of each filter under different filter sizes. The number of pattern we are dealing with is 100k and the segment size we are using is 32. Again, we are using the same order for the first two filters.

The number of lookups for the first filter is $T/S = 10M/32 = 312500$. With the same number of lookups, the performance of the first filter is depending on B_1, which follows the trend of data cache misses. Therefore, the processing time is growing when the filter order is increased.

For the second filter, although the number of lookups is calculated by the program, it can actually approximated by the formula $L_1 \times R_1$. As Table 4.2 shows, the number of lookups drops down when the filter order is increased, because R_1 is decreasing. Thus, although B_2 becomes larger and the cost of each lookup increases at the same time, the

total processing time doesn't have to increase. As a matter of fact, the cost of the second filter starts dropping down from the point that the filter order is 23 in this table. In Table 4.3 where 10k patterns are scanned under the same condition, the processing time of the second filter is completely going down.

In the third filter, it is not necessary to consider reducing its data cache misses. Because each entry of the Signature Table contains pattern and position information, which causes the third filter to quickly grow too large for the cache. Therefore, we could use a filter order that is relatively large, but we have to make sure that there is enough memory to allocate for this table.

Besides the caching problem, the process of the third filter is more complicated than the previous two, which also makes each lookup in the third filter more expensive. Therefore, the best solution is to keep the number of lookup in the Signature Table as low as possible, which means the filter order of the first and second filter cannot be too small, but at the same time we have to watch their data cache misses as well.

Ideally, we would like the filter stay within the cache, that is at most 2^{23} for a 2M cache. As we can see from table 4.3, the best performance goes to filter order 22.

However, for virus scanning we're usually dealing with a large set of patterns. From Table 4.1, we can see that the ratio of false positives for 100k patterns is 0.317 when filter order is 23. This ratio is not good enough to break down the number of text segment candidates that have been passed to the third filter. As shown in table 4.3, there are still 31455 candidates left for the third filter to process, when $F_1 = F_2 = 23$, which is the lowest false positive ratio we can get when the first two filters can stay within the cache. Therefore. we should consider using a larger filter order for these two filters in this case although it might generate more data cache misses.

However, the filter order of the first two filters could be different. We conducted some experiments with different filter order on 10M text files scanning 100k patterns using segment size 32 bytes.

Table 4.4 shows some performance results with different F_2 when $F_1 = 22$. The false positive ratio of the first filter in the given condition is 0.534, which is relatively high. Thus, the second filter needs to be larger to reduce the number of lookups in the third filter. We are expecting an improvement when F_2 become larger as it is shown in Table 4.4. When the false positive ratio is reduced to certain level and the cost of the third filter is minor, then there is not much improvement that can be done by increasing F_2. In fact, at this point,

F_1	21	22	23	24	25	26	27	28	29
L_1	312500	312500	312500	312500	312500	312500	312500	312500	312500
TOT_1	0.982	1.208	1.533	1.932	2.226	2.298	2.387	2.407	2.468
F_2	21	22	23	24	25	26	27	28	29
L_2	244297	166895	99281	54462	28344	14536	7450	3619	1788
TOT_2	0.868	1.115	0.904	0.657	0.389	0.212	0.114	0.063	0.040
F_1	27	27	27	27	27	27	27	27	27
L_3	190857	88809	31455	9386	2620	707	195	37	11
TOT_3	3.195	1.722	0.763	0.264	0.087	0.025	0.007	0.002	0.001
TOT	5.045	4.045	3.220	2.854	2.702	2.535	2.508	2.472	2.509

Table 4.2: Analysis on Each filter (P=100k T=10M S=32)

F_1	21	22	23	24	25	26	27	28	29
L_1	312500	312500	312500	312500	312500	312500	312500	312500	312500
TOT_1	0.752	0.762	1.154	1.796	2.083	2.277	2.354	2.433	2.431
F_2	21	22	23	24	25	26	27	28	29
L_2	44342	23055	11865	5911	3096	1584	786	363	178
TOT_2	0.409	0.264	0.154	0.088	0.048	0.027	0.016	0.011	0.007
F_3	27	27	27	27	27	27	27	27	27
L_3	6300	1673	436	102	21	7	2	0	0
TOT_3	0.178	0.054	0.016	0.004	0.001	0.0009	0.0008	0.0006	0.0006
TOT	1.339	1.080	1.324	1.861	2.132	2.305	2.371	2.445	2.439

Table 4.3: Analysis on Each filter (P=10k T=10M S=32)

F_2	22	23	24	25	26	27	28	29	30
TOT(cyc/B)	3.941	3.660	3.405	3.135	3.030	2.920	2.921	2.925	2.931

Table 4.4: $F_1 = 22$

F_2	21	22	23	24	25	26	27	28	29
TOT(cyc/B)	2.515	2.496	2.473	2.464	2.450	2.438	2.446	2.447	2.452

Table 4.5: $F_1 = 28$

increasing F_2 can only cause more data cache misses in the second filter. This is why the total cost starts going up slightly when F_2 is larger than 27.

The other case $F_1 = 28$ is shown in Table 4.5. Changing the filter order of the second filter doesn't make much difference to the final ratio in this case because it is already 0.012 for the first filter. Thus, a smaller filter order that reduces the data cache misses can improve the performance.

In fact, among all the reasonable combinations of filter size, the best performance for 100k patterns goes to $F_1 = 28$ and $F_2 = 26$, which is 2.44 cyc/byte.

Measurements of other pattern set sizes are displayed in Table 4.6 with their best performance that we can achieve for segment size 32 and the corresponding filter orders are listed as well.

As we can see from Table 4.6, a general rule to select the filter order is to keep the filter size within the cache if the false positive ratio is not too high. However, as the number of patterns gets bigger, we can start adjusting the filter order of the second pass to satisfy the need for a low false positive ratio. Since the number of lookups in the second pass is smaller than the first pass, the additional data cache misses will be fewer. When the size of the pattern set is larger than 100,000, changing the filter order of the second pass is not enough, we also need to change F_1. Even though F_2 can ensure a low false positive ratio for the third pass, a false positive ratio 0.32 for 100k and 0.53 for 200k is too high and cause a large amount of lookups in the second pass. Thus, it better to change F_1 to achieve the balance.

Pattern Set Size	10k	20k	30k	40k	50k	60k	90k	100k	200k
TOT(cyc/B)	1.06	1.35	1.57	1.73	1.83	1.98	2.27	2.44	2.50
F_1	22	22	22	23	23	23	23	28	29
F_2	23	23	27	28	29	29	29	26	26

Table 4.6: Best Performance

4.6 Wildcards

The solution for the wildcard algorithm is a little different, but the basic ideas are the same. We still need to consider the two tradeoff factors. However when we consider the number of patterns, we also need to calculate the signatures generated by the wildcards when we are choosing the parameters.

In practical pattern sets, e.g, virus databases, there might exist some very short patterns or patterns with a lot of wildcards. Therefore, we should consider using smaller segment size to avoid processing those wildcards and inserting more signatures into the lookup tables.

We also need to set up a threshold to filter out those patterns that might dramatically increase the false positive ratio. Table 4.7 shows one of the data files main.db we get from ClamAV database. This data file includes 30406 patterns in total where 28802 of them can be processed by the non-wildcard algorithm using segment size 8, which requires pattern length longer than 15 bytes and without any wildcards. There are 395 patterns that have single character wildcards '?' and 552 short patterns that are needed to be treated as patterns with question marks at the end.

The wildcard algorithm used to measure the performance shown in Fig 4.7 only deals with single character patterns. The 1209 patterns with other wildcards are ignored.

Fig 4.7 compares the non-wildcard algorithm or non-wildcard algorithm with two wildcard algorithms of different thresholds on the same text file. The non-wildcard algorithm scans through 29197 random patterns, the same number of patterns as the wildcard algorithm does.

The first wildcard algorithm sets a higher threshold. Therefore, 35 out of 947 wildcards or short patterns are not processed. The second wildcard algorithm used a lower threshold so that it only left 3 patterns unprocessed, but it gets slower than the first wildcard algorithm.

In fact, the first wildcard algorithm generates 275,816 additional partial bits value while

Total Patterns	Pattern Type			
	'?' only	short	wildcards	no wildcards
30406	395	552	1209	28802

Table 4.7: Virus Data

Filter Order	21	22	23	24	25	26	27	28	29
TOT-wildcards1(cyc/B)	5.02	3.69	3.89	6.47	9.77	10.64	10.85	11.01	11.09
TOT-64k(cyc/B)	5.36	3.98	3.74	6.28	9.55	10.38	10.71	10.85	11.04

Table 4.8: wildcards versus non-wildcards

the second wildcard algorithm generates 884,031 additional partial bits value by processing 32 more patterns.

Since each non-wildcard pattern generated 8 partial bits value, 275,816 additional partial bits value is equivalent to adding 34,477 patterns. So its performance should be similar to non-wildcard algorithm processing 64,000 patterns. Table 4.8 gives the performance comparison between the first wildcard algorithm and the non-wildcard algorithm with 64,000 patterns. We can see from the table that its performance is pretty close.

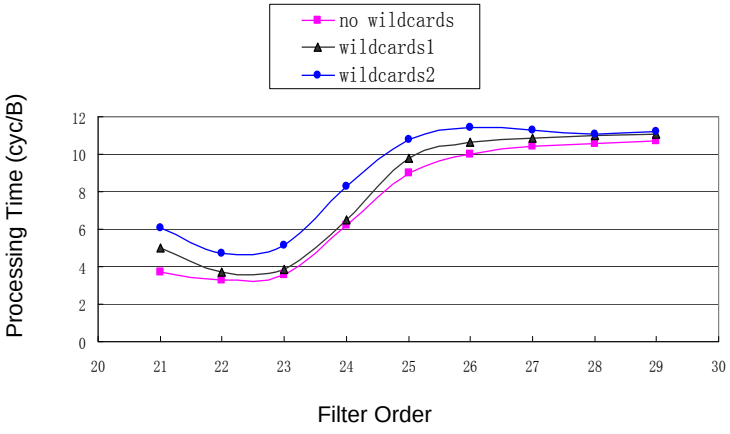

Figure 4.7: Wildcards

buffer size	1k	2k	8k	32k	128k
TOT(cyc/B)	3.50	2.96	2.50	2.31	2.26
latency(ms)	0.00167	0.00282	0.00952	0.0352	0.137

Table 4.9: latency

4.7 Latency

Latency is a measure of time delay experienced in a system. In BF algorithm, the latency is considered as the time delay between receiving an input byte and writing that byte to output. Because BF algorithm reads and process one buffer at a time, the latency is calculated as the time to process one buffer.

As shown in Table 4.9, when the buffer size grows, the processing time for each byte decreases as a result of the cache behavior. Therefore, when dealing with offline content virus scanning, where the latency is not the big concern, we could use larger buffer size to achieve better performance. However, in online applications such as NIDS, the search text maybe processed as packets less than a few thousand bytes each. In this case, the delay of BF algorithm is only a few microseconds.

Chapter 5

Performance Comparison

In this chapter, performance comparisons were made between the non-wildcards algorithm and the AC algorithm extracted from Snort since no wildcards are used in Snort rules. The wildcards algorithm was also measured and compared with the AC algorithm extracted from CalmAV using the virus database.

5.1 Comparison with Snort

Snort implemented an optimized AC algorithm, which supports an optimized full matrix state table, and a memory efficient state table using a sparse matrix based storage method.[25]

Considering the size of Snort rules, random pattern sets from range 1k to 10k are generated and scanned on a 100M random file.

The extracted AC algorithm and BF algorithm with segment size 8 are tested on the same pattern sets where the length of each pattern are from 15 to 30 bytes, because most of the patterns in Intrusion Detection are generally short.

However, to implement the non-wildcards BF algorithm with segment size 32, pattern length has to be at least 63. Therefore, it is tested on some other pattern sets with the same number of patterns but using patterns length from 80 to 100 bytes.

As shown in Table 5.1, by using segment size 32, we can achieve more than 100 fold speed-up. Even by using segment size 8, BF algorithm is more than 30 times faster than AC from Snort, especially with large pattern sets.

Pattern Set Size	1k	2k	4k	6k	8k	10k
AC(cyc/B)	68.5	85.85	99.64	109.19	114.85	116.02
BF-SEG8(cyc/B)	2.13	2.24	2.32	2.49	2.53	2.71
BF-SEG32(cyc/B)	0.66	0.73	0.86	0.94	0.99	1.06

Table 5.1: Comparison with Snort AC

5.2 Comparison with ClamAV

ClamAV has a more complicated data structure. We used a simplified data structure only to load the database and run its AC algorithm.

To compare with the AC algorithm extracted from ClamAV, we need to use larger sets of patterns. But there are not that many wildcards patterns in the actual virus database. Most of the wildcards patterns are included in main.db, which has about 30,000 patterns. We randomly picked 10,000, 20,000 and 30,000 patterns from this file to run the test.

The performance is shown in Fig 5.1. Our algorithm is about 18 times faster than AC for 10k patterns, 20 times faster for 20k patterns and 25 time faster for 30k patterns. The difference is going to be more significant as the size of the pattern set grows, because the number of data cache misses for the AC algorithm is growing more quickly than BF as we can see from Fig 5.2

BF is using the same size of bitmaps in the first two filters for these three pattern sets The increasing number of patterns only increase the size of the signature table where the number of lookups is small due to the low false positive ratio of the previous two filters.

5.3 Comparison with Other Proposed Techniques

5.3.1 Intrusion Detection

Table 5.2 gives the performance of some other techniques for Intrusion Detection. Most of them are FPGA-based methods mentioned in Chapter 2. Although the listed FPGA-based methods show high performance, they are all analytical results based on models of desired hardware, rather than experimental results.

Kumar's method needs at least 8 memory modules and 32 engines that process 32 packets concurrently to achieve 10Gb/s performance. Suresh's method is extremely fast

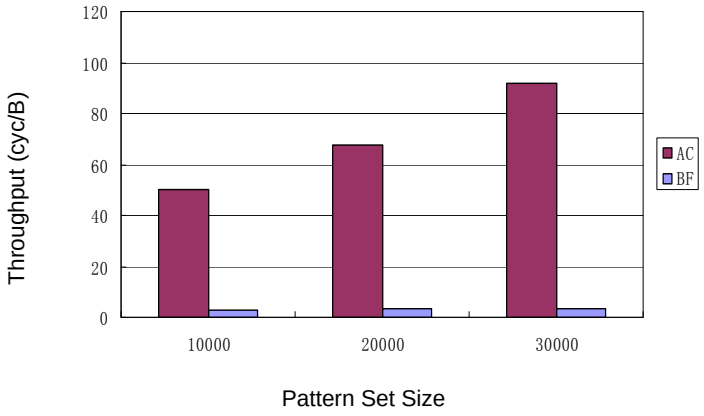

Figure 5.1: Performance Comparison with ClamAV AC (P=100k,T=10M)

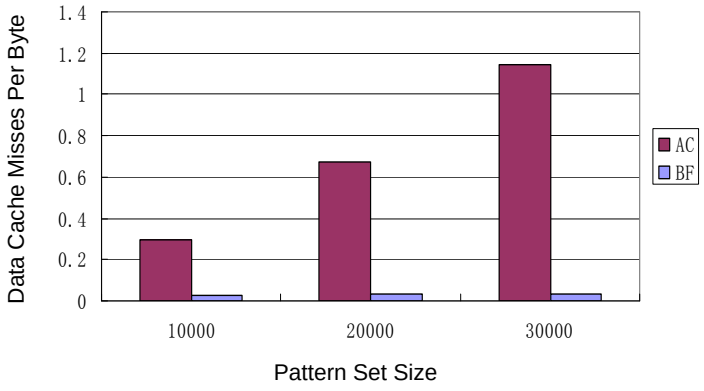

Figure 5.2: Data Cache Misses Comparison with ClamAV AC (P=100k,T=10M)

Algorithm	CPU	Patterns	Throughput(Gb/s)
Daniele	Cell/BE (8 SPEs)	8,400	2 (experimental)
Kumar	FPGAs 300MHz (8 memory modules)	1,000	10(analytical)
Dharmapurikar	FPGAs 81MHz (4 engines)	10,000	2.4 (analytical)
Tuck	ASIC	1,500	8 (analytical)
Suresh	FPGAs 73MHz	100	18 (analytical)
BF	Intel Core2 2.1GHz	10,000	16 (experimental)

Table 5.2: Comparison with Techniques used for Intrusion Detection

but only process the strings in Snort rules with length eight. Tuck's method is based on a application-specific integrated circuit (ASIC) design.

However, our software-based BF can achieve 16Gb/s with 10,000 random patterns with segment size 32. For shorter patterns, we can get 6Gb/s performance by using segment size 8 for 10,000 patterns.

5.3.2 ClamAV

Algorithm	CPU	Patterns	Pattern Type	Throughput
HashAV	Pentium4 2.6GHz	20,000	non-wildcards	170 MB/s
BF	Intel Core2 2.1GHz	20,000	non-wildcards	1550 MB/s
HashAV	Pentium4 2.6GHz	20,000	including wildcards	54 MB/s
BF	Intel Core2 2.1GHz	20,000	including wildcards	660 MB/s
MRSI	Pentium4	30,000	non-wildcards	43 MB/s
BF	Intel Core2 2.1GHz	30,000	non-wildcards	1330 MB/s
MRSI	Pentium4	46,000	non-wildcards	10 MB/s
BF	Intel Core2 2.1GHz	46,000	non-wildcards	1160 MB/s

Table 5.3: Comparison with Techniques used on ClamAV

Table 5.3 compares the two software-based methods that are applied to ClamAV with BF algorithm under different circumstances. All of the tests are scanned through random text files.

The non-wildcard BF algorithm we used to compare with HashAV did not use patterns

from virus database since it is using segment size 32. The corresponding HashAV method process roughly 20,000 signatures, because the wildcards signatures are ruled out.

Although HashAV itself doesn't process wildcards patterns. It uses a two-scan approach that process the wildcards signatures during the second scanning process by using the original AC algorithm from ClamAV. We compared this approach with our wildcards algorithm.

MRSI is a BM-based algorithm. As shown in the table, the performance decreases very quickly as the number of patterns grows. However, our algorithm is able to achieve a much better and stable performance.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we discussed the importance of multi-string search especially in security applications such as intrusion detection and virus scanning.

We examined the previous works on string search that has been constantly explored by many researchers due to the increasing pressure caused by the growing size of on-line content and increasing network speeds.

We showed that most of the BM-based algorithms are not able to deal with large number of patterns with high efficiency. DFA-based methods preferred by most of the multiple pattern matching applications suffer from the memory problem caused by building the state machine. Some hardware-based solutions are also proposed for Deep Packet Inspection with ideal architectures. But there are practical problems such as limited resources and low clock frequency of those chips.

A new algorithm called Bits Filter has been proposed that applies Bloom filter and parallel bit stream technique. This three-pass filtering algorithm including both text filtering and pattern filtering partitions the text into segments and use two light weight lookup tables to rule out the impossible matchings. This light-weight lookup table called Bit Map greatly reduce the data cache misses that could be generated by directly accessing the signature table.

Bits Filter was evaluated and compared with results for the AC algorithm implementations extracted from Snort and ClamAV as well as some other works that have been done to improve ClamAV or NIDS.

We demonstrated that the non-wildcard algorithm can achieve more than 40 times speed-ups over the Snort AC implementation based on the current size of the rule set. In comparison with the ClamAV implementation, the wildcard algorithm is about 30 times faster for a significant subset of the virus database with an increasing advantage as the number of patterns grows. Our target virus signatures do not include MD5 signatures, which is a large portion of the virus database and continuously growing quickly. However, the matching of MD5 signature to a target file's MD5 checksum or the checksum of a specific section can be processed in a much shorter time.

Compared with those analytical results from hardware-based technique, BF, a software-based algorithm running on commodity processor can achieve an equivalent or even better performance.

If larger data cache is provided, the performance could be improved by further reducing the data cache misses or generating a smaller false positive ratio. As SIMD register widths and processing technologies improve, the calculation during each filtering process could also be accelerated.

Moreover, with advances in processor SIMD architecture, such as Inductive Doubling Instruction Set Architecture [10], more opportunities to use SIMD operations in the Bits Filter algorithm are expected. This may speed up the basic operations or enhance the performance by allowing algorithm variations using more complex hash functions, for example.

6.2 Future Work

Bits Filter algorithm can be easily adapted to multi-core processors. A simple case is that enough memory space is provided to all the processors, so that each of them can perform as a separate search engine processing one piece of the text data.

Otherwise, if processors can only access limit memory space efficiently, the Bit Map can be split up into several parallel Bit Maps by using a hash function. This hash function could be as simple as a bit extraction from the patterns. The number of extracted bits used as the index of the Bloom filters is depending on the number of available processors. For example, if four processors are provided, patterns can be grouped by their first two bits. Then the processors, each accessing one Bit Map, can process the corresponding text segments concurrently. A pass zero might be applied before the first filter just to quickly sort the text segments and assign them into different processors.

The future work also includes embedding Bits Filter into Snort and ClamAV and exploring new method to deal with shorter patterns that exist in ClamAV database and Snort rule set.

Another possibility is to investigate and develop a new strategy to generate virus patterns based on the characteristics of BF algorithm. Since shorter patterns are generally recognized as a small part of the virus, it should be possible to replace them with longer patterns. Then, the algorithm can be further improved by using larger segment size.

Bibliography

[1] ClamAV: Setting the Standard for Open Source Antivirus and Antimalware Sortware. http://www.sourcefire.com/products/clamav.

[2] Performance Application Programming Interface. http://icl.cs.utk.edu/papi/.

[3] Intel 64 and IA-32 Architectures Optimization Reference Manual. 2009.

[4] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[5] Spyros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. Generating realistic workloads for network intrusion detection systems. *SIGSOFT Softw. Eng. Notes*, 29(1):207–215, 2004.

[6] Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[7] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[8] Andrei Broder, Michael Mitzenmacher, and Andrei Broder I Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[9] Robert D. Cameron, Kenneth S. Herdy, and Dan Lin. High performance xml parsing using parallel bit stream technology. In *CASCON '08: Proceedings of the 2008 conference of the center for advanced studies on collaborative research*, pages 222–235, New York, NY, USA, 2008. ACM.

[10] Robert D. Cameron and Dan Lin. Architectural support for SWAR text processing with parallel bit streams: the inductive doubling principle. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 337–348, New York, NY, USA, 2009. ACM.

[11] Cameron, Robert D. A Case Study in SIMD Text Processing with Parallel Bit Streams. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Salt Lake City, Utah, 2008.

[12] C. Jason Coit, Stuart Staniford, and Joseph McAlerney. Towards faster string matching for intrusion detection or exceeding the speed of snort. *DARPA Information Survivability Conference and Exposition,*, 1:0367, 2001.

[13] Beate Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*, pages 118–132, London, UK, 1979. Springer-Verlag.

[14] N. Desai. Increasing Performance in High Speed NIDS. www.linuxsecurity.com, March 2002.

[15] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep packet inspection using parallel bloom filters. In *IEEE Micro*, pages 44–51. IEEE Computer Society Press, 2003.

[16] Ozgun Erdogan and Pei Cao. Hash-AV: Fast virus signature scanning by cache-resident filters. *Int. J. Secur. Netw.*, 2(1/2):50–59, 2007.

[17] Dan Gusfield. In *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, CA, 1997. University of California Press.

[18] R. Nigel Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10:501–506, 1980.

[19] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 339–350, New York, NY, USA, 2006. ACM.

[20] John W. Lockwood, Naji Naufel, Jon S. Turner, and David E. Taylor. Reprogrammable network packet processing on the field programmable port extender (fpx). In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 87–93, New York, NY, USA, 2001. ACM.

[21] Hongbin Lu, Kai Zheng, Bin Liu, Xin Zhang, and Yunhao Liu. A memory-efficient parallel string matching architecture for high-speed intrusion detection. *Selected Areas in Communications, IEEE Journal on*, 24(10):1793–1804, Oct. 2006.

[22] Nihar R. Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck: problems and solutions. *Crossroads*, 1999.

[23] Mike Fisk and George Varghese. Applying fast string matching to intrusion detection. Technical report, Los Alamos National Lab, Los Alamos, New Mexico, 2002.

[24] James Moscola, John Lockwood, Ronald P. Loui, and Michael Pachos. Implementation of a content-scanning module for an internet firewall. In *in FCCM*, pages 31–38, 2003.

[25] Marc Norton. Optimizing pattern matching for intrusion detection. Technical report, Sourcefire, July 2004.

[26] Baeza-Yates R.A. Improved string searching. In *Software–Practice and Experience*, pages 257–271, 1989.

[27] Martin Roesch. Snort - lightweight intrusion detection for networks. In *LISA '99: Proceedings of the 13th USENIX conference on System administration*, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[28] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Exact multi-pattern string matching on the Cell/B.E. processor. In *CF '08: Proceedings of the 5th conference on Computing frontiers*, pages 33–42, New York, NY, USA, 2008. ACM.

[29] Reetinder Sidhu and Viktor K. Prasanna. Fast regular expression matching using fpgas. In *in IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[30] Haoyu Song and J.W. Lockwood. Multi-pattern signature matching for hardware network intrusion detection systems. In *Global Telecommunications Conference, 2005. GLOBECOM '05. IEEE*, volume 3, pages 5 pp.–, Nov.-2 Dec. 2005.

[31] Dinesh C. Suresh, Zhi Guo, Betul Buyukkurt, and Walid A. Najjar. Automatic compilation framework for Bloom filter based intrusion detection. In *Reconfigurable Computing: Architectures and Applications*, pages 413–418. Springer Berlin / Heidelberg, 2006.

[32] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *In IEEE Infocom, Hong Kong*, pages 333–340, 2004.

[33] Sun Wu and Udi Manber. A fast algorithm for multi-pattern searching. Technical Report TR 94-17, Department of Computer Science, University of Arizona, 1994.

[34] Xin Zhou, Bo Xu, Yaxuan Qi, and Jun Li. MRSI: A fast pattern matching algorithm for anti-virus applications. In *ICN '08: Proceedings of the Seventh International Conference on Networking*, pages 256–261, Washington, DC, USA, 2008. IEEE Computer Society.