國立交通大學

資訊科學系

碩 士 論 文

剖析與加速三種具有字串比對之內容安全應用

Profiling and Accelerating String Matching Algorithms in Three Content Security Applications

研 究 生： 李志祥

指導教授： 林盈達 教授

中 華 民 國 九 十 四 年 六 月

剖析與加速三種具有字串比對之內容安全應用

Profiling and Accelerating String Matching Algorithms in Three Content Security Applications

研 究 生： 李志祥　　　　Student : Zhi-Xiang Li

指導教授： 林盈達　　　　Advisor : Dr. Ying-Dar Lin

國 立 交 通 大 學

資 訊 科 學 系

碩 士 論 文

A Thesis

Submitted to Institute of Computer and Information Science

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

June 2005

HsinChu, Taiwan, Republic of China

中華民國九十四年六月

剖析與加速三種具有字串比對之內容安全應用

學生：李志祥　　　　指導教授：林盈達

國立交通大學資訊科學研究所

摘要

網路內容安全已經成為網際網路中重要的議題。在內容處理中，字串比對演算法效率的必要性也漸漸被證實。字串比對演算法的效能主要受到樣本的個數、最短特徵值的長度與特徵值所組成的字元集而有所影響。這份研究將復審與剖析一些典型的演算法來了解各種演算法適合在什麼情況下使用。Aho-Corasick 演算法適合使用在特徵值最短的長度為 1 的時候，Modified-WM 演算法適合使用在特徵值最短的長度為 2 且樣本個數小於 1,000 的情況下，FNPw2 則適合使用在特徵值最短的長度為 2 且樣本個數大於 1,000 的情況下，特徵值最短的長度為 3 的時候則適合使用 Modified-WM 演算法，特徵值大於等於 4 的時候則適合使用 2-gram BG+演算法。接著，各種有效率的演算法皆實作到開放原始碼套件中，以便觀察在實際應用上效能的差異。如果字串比對處理在總執行時間的比例重的話，效能上則有很大的改善。如在 ClamAV 的實驗中，新的方法在效能上提升了 5 倍以上。另外，將會餵真實的資料與人造的資料到這些套件中，讓它們處理。以便觀察這些套件處理真實資料與人造資料效能上的差異。由於字元集分布的影響，它們處理真實資料所需的時間會比處理人造資料來的長。最後，這份研究亦觀察出在字串比對演算法中的一些實際設計議題。

Keywords: *字串比對，演算法，內容安全*

Profiling and Accelerating String Matching Algorithms in Three Content Security Applications

Student: Zhi-Xiang Li Advisor: Dr. Ying-Dar Lin

Department of Computer and Information Science

National Chiao-Tung University

Abstract

Network content security has become a critical issue of the Internet. It is shown that the efficiency of string matching algorithms is essential to content processing. The performance of a string matching algorithm is sensitive to the number of patterns, the minimum length of the signature and the character set that the signatures are composed of. This work reviews and profiles some typical algorithms to understand which algorithm is suitable in which situation. The AC algorithm is suitable for LSP=1, the Modified-WM algorithm is suitable for LSP =2 when the pattern set size is smaller than 1,000, the FNPw2 algorithm is suitable for LSP=2 when the pattern set size is larger than 1,000, the Modified-WM algorithm is suitable for LSP=3 and the BG+ algorithm is suitable for LSP≥4. Then, these algorithms are implemented on open-source content security applications to observe the performance in practice. The performance improvement is significant if the percentage of string matching processing on total execution time is great. For example, the novel method is five times faster than the original method on the experiment of ClamAV. In addition, these applications are fed with the real and synthetic data. The differences of performance between the real and synthetic data are also compared. The execution time for processing the real data is longer than that

for processing the synthetic data due to the character set distribution. Finally, the practical design issues for string matching are also observed in this work.

Keywords: *string matching, pattern matching, algorithm, content security*

CONTENTS

CHAPTER 1 INTRODUCTION **1**

CHAPTER 2 RELATED WORKS **3**

2.1 TYPICAL ALGORITHMS 3

2.1.1 Automaton-based *4*

2.1.2 Heuristic-based *4*

2.1.3 Hashing-based *6*

2.1.4 Bit-parallelism-based *6*

2.2 SELECTED PACKAGES 7

2.2.1 ClamAV *7*

2.2.2 DansGuardian *8*

2.2.3 Snort *9*

CHAPTER 3 PRACTICAL DESIGN ISSUES **10**

3.1 VERIFICATION APPROACH 10

3.1.1 Propose the CRKBT Algorithm *10*

3.1.2 Experiments *12*

3.1.3 Analysis *14*

3.2 HASH FUNCTION 15

CHAPTER 4 PROFILING ALGORITHMS **17**

4.1 EXTERNAL PROFILING 17

4.1.1 Experiments with Earlier Algorithms *17*

4.1.2 Experiments with Novel Algorithms *18*

4.2 INTERNAL PROFILING 19

4.2.1 Shift Distance *19*

4.2.2 Potential Matching *20*

4.2.3 Memory Accesses *21*

4.3 PROFILING SUMMARY 25

CHAPTER 5 EXPERIMENTS ON REAL APPLICATIONS **28**

5.1 IMPLEMENTATIONS IN THREE PACKAGES 28

5.1.1 ClamAV *28*

5.1.2 DansGuardian *29*

5.1.3 Snort *29*

5.2 BENCHMARKING 29

5.2.1 ClamAV *29*

5.2.1.1 Benchmarking Methodology 29

5.2.1.2 Benchmarking Results 29
5.2.2 DansGuardian *30*
5.2.2.1 Benchmarking Methodology 30
5.2.2.2 Benchmarking Results 30
5.2.3 Snort *31*
5.2.3.1 Benchmarking Methodology 31
5.2.3.2 Benchmarking Results 32
5.3 Real Data vs. Synthetic Data 32
5.3.1 ClamAV *33*
5.3.2 DansGuardian *33*
5.3.3 Snort *33*
CHAPTER 6 CONCLUSIONS **35**

LIST OF FIGURES

FIGURE 1: THE RKBT ALGORITHM VS. THE CLASSIFIED RKBT ALGORITHM 11
FIGURE 2: THE RKBT ALGORITHM VS. THE CRKBT ALGORITHM 12
FIGURE 3: THE 2-GRAM SOG ALGORITHM VS. THE 2-GRAM SOG+ ALGORITHM 13
FIGURE 4: THE 2-GRAM BG ALGORITHM VS. THE 2-GRAM BG+ ALGORITHM 13
FIGURE 5 EARLIER ALGORITHMS BENCHMARKING RESULTS 17
FIGURE 6 NOVEL ALGORITHM BENCHMARKING RESULTS 18
FIGURE 7 THE SHIFT DISTANCE PROFILING 19
FIGURE 8 THE POTENTIAL MATCHING PROFILING 20
FIGURE 9 RKBT VS. CRKBT 21
FIGURE 10 WU-MANBER VS. MODIFIED-WM 22
FIGURE 11 BG+ VS. SOG+ 22
FIGURE 12 THE NUMBER OF MEMORY ACCESSES 23
FIGURE 13 THE NUMBER OF L2 CACHE MISSES 23
FIGURE 14 THE SIZE OF MEMORY USAGE 24
FIGURE 15 THE PROFILING RESULT OF LSP=1 25
FIGURE 16 THE PROFILING RESULT OF LSP=2 26
FIGURE 17 THE PROFILING RESULT OF LSP=3 26
FIGURE 18 THE PROFILING SUMMARY 27
FIGURE 19 THE PROFILING SUMMARY 28
FIGURE 20 THE BENCHMARKING RESULTS FOR CLAMAV PACKAGE 30
FIGURE 21 THE BENCHMARK RESULTS FOR DANSGUARDIAN PACKAGE 31
FIGURE 22 THE BENCHMARKING RESULT FOR SNORT PACKAGE 32

LIST OF TABLES

TABLE 1: CLASSIFICATION OF TYPICAL ALGORITHMS 4
TABLE 2: SELECTED OPEN-SOURCE PACKAGES 7
TABLE 3: THE PROFILING RESULT FOR SOME DIFFERENT HASH FUNCTIONS 16

Chapter 1 Introduction

A growing number of intrusions, worms, viruses and inappropriate Web pages spread all over the Internet. Detecting and filtering them require *content classification* at the application layer, as opposed to traditional *packet classification* at the network and transport layers. *Content classification* requires string matching for designated signatures. Unlike traditional packet classification, which looks for fields of fixed lengths and at fixed positions, the position and the length of the matching signature are unknown before hand. Thus content signature matching is usually more elaborate than packet classification. In addition, string matching is reported a bottleneck for network content applications [1-5]. Consequently, the efficiency of the string matching algorithm is critical to content processing.

No existing string matching algorithms can scan signatures of various characteristics more efficient than all the others. For example, the Wu-Manber algorithm [6] is inefficient for a huge pattern set [7]. Furthermore, content security applications have signatures of different characteristics. For example, the anti-virus applications have a large number of signatures, and the intrusion detection systems have short patterns of one or two characters. This work investigates the types of signatures in these content security applications and the type that each string matching algorithm can scan most efficiently, and hence the most efficient algorithm is derived for each application.

The efficiency of six typical string matching algorithms are profiled for signature sets varying in sizes, the minimum length of the signatures and the character set that the signatures are composed of. Sample sets of both synthetic and real signatures are studied to see if there are deviations in the profiling results for both cases. The edges and limitations of each algorithm are better understood after the profiling. The

impacts on performance of memory and cache accesses are also measured quantitatively.

These algorithms are also implemented on three open source packages of content security: ClamAV [8] for anti-virus, DansGuardian [9] for content filtering and Snort [10] for Network Intrusion Detection System (NIDS). The performance is benchmarked to see if the gain in the actual environment is significant after the implementation.

In addition, this work also proposes a classified RKBT (Rabin-Karp with binary search and two-level hashing) to enhance the performance of the original RKBT algorithm [11, 12] for a huge signature set. The RKBT algorithm can serve as the verification algorithm after a potential match in some string matching algorithms [7] and hence its performance is essential in these algorithms. The contributions of this work are summarized as follows:

- Finding out the most efficient algorithm for each application of content security and telling why.
- Proposing the Classified RKBT algorithm to enhance the performance of the original RKBT algorithm.
- Comparing the performance for synthetic and real data in these algorithms.

The rest of this work is organized as follows. Chapter 2 reviews six typical algorithms and three selected packages. Chapter 3 discusses the practical design issues, such as the verification algorithm and the hash function. Chapter 4 shows the profiling results and identifies the most efficient algorithms for each situation. The performance gain on real packages is demonstrated in Chapter 5. Chapter 6 concludes the study.

Chapter 2 Related Works

2.1 Typical Algorithms

The string matching problem is to find all the occurrences of a string *p*, called the pattern, in the text $T=t_1t_2t_3...t_n$ on the same alphabet, where *n* is the length of the text. Multiple string matching is to find the appearance of a string p^i in a set of strings $P=\{p^1, p^2..., p^r\}$ in the same manner as a single string. This research focuses on exact string matching because the majority of the content security applications must use it to find out the signatures.

A number of string matching algorithms have been proposed for exact string matching. They are usually grouped into three general approaches, prefix searching, suffix searching and factor searching, depending on the way the pattern is searched for in the text [13, 14]. However, this work categorizes the algorithms into four major approaches according to the data structure that drives the matching to emphasize on the evolution. These categories are automaton-based, heuristic-based, hashing-based and bit-parallelism-based. An automaton-based algorithm tracks the partial match of the pattern prefixes in the text. A heuristic-based algorithm relies on one or two heuristic function to finish looking up the shift distance. The shift distance of 0 indicates a possible match, so a verification algorithm follows to verify. A hashing-based algorithm checks a possible appearance of the patterns by hashing a block of characters in the text and compares the hash value with those from hashing the blocks in the patterns. A bit-parallelism-based algorithm takes advantage of the parallelism of the bit operations inside a computer word to simulate the operation of a finite automaton [15]. Table 1 compares some typical algorithms in these four categories.

TABLE 1: Classification of typical algorithms

Algorithms	Approach	Time Complexity	Search Type	Multiple Pattern	Key Ideas
Aho-Corasick	Automaton-based	Linear	Prefix	Yes	Finite automaton
Optimized-AC		Linear	Prefix	Yes	Full matrix or Sparse matrix
Boyer-Moore	Heuristic-based	Sub-linear	Suffix	No	Bad character, Good suffix
Horspool		Sub-linear	Suffix	No	Bad character
Set-wise BMH		Sub-linear	Suffix	Yes	Bad character
Wu-Manber		Sub-linear	Suffix	Yes	Shift and hash Table
Modified-WM		Sub-linear	Suffix	Yes	Change table size and hash function
Rabin-Karp	Hashing-based	Linear	Prefix	No	Hash function
RKBT		Linear	Prefix	Yes	Two-level hash, Binary search
FNP		Sub-linear	Prefix	Yes	Skip distance table
SOG	Bit-parallelism-based	Linear	Prefix	Yes	Bit-parallelism, q-gram
BG		Sub-linear	Factor	Yes	Bit-parallelism, q-gram

2.1.1 Automaton-based

The Aho-Corasick (AC) algorithm [16] was proposed for multi-pattern matching. It uses the data structure of a finite automaton that accepts all strings in the pattern set. The automaton is fed the input characters one by one in the text and tracks partially matched patterns. The time complexity is O(n). However, a large pattern set demands large memory space for the transition table. It can be slower in case of a large pattern set because of the worse cache locality in accessing the transition table.

The derived algorithm, Optimized-AC algorithm [17], was proposed to reduce the memory requirement by compressing the transition table with the structure of a full matrix or a sparse matrix. The running time of the Optimized-AC algorithm is faster than the standard AC algorithm due to its better cache locality.

2.1.2 Heuristic-based

The Boyer-Moore (BM) algorithm [18] is extensively used due to its efficiency in single-pattern matching. It uses two heuristic functions, bad-character function and good-suffix function, to reduce the number of character comparisons by skipping over

characters that cannot be a match. The shift distance of the search window is the maximum of shift distances indexed from the two heuristics. The time complexity is sub-linear of O(n/m) on the average.

Horspool proposed the Boyer-Moore-Horspool (BMH) algorithm [19] that uses only the bad character function and is shown to be more efficient than the BM algorithm in practice, because the BMH algorithm is faster than the BM algorithm in each iteration. Furthermore, this algorithm is also a single-pattern matching algorithm. The Set-Wise BMH algorithm [3] extends the BMH algorithm to handle multiple patterns.

The Wu-Manber (WM) algorithm [6], a variation of the Set-Wise BMH algorithm, was proposed for multi-pattern matching. It is based on the similar heuristic function of the BMH algorithm to build the shift table and hashes the block of B characters in the suffix of the search window for the shift distance, where B is the size of the hash block, so its time complexity can be sub-linear on the average. However, the performance of the WM algorithm depends considerably on the length of the shortest pattern (denoted as LSP), because the maximum shift distance equals $m–B+1$, where m is the length of the shortest pattern and B is the size of the hash block.

The derived algorithm, Modified-WM algorithm [17], was proposed by the Snort team leader, Marc Norton. It is based on the implementation of the Agrep package [20], which is written by the author of the WM algorithm. It changes the size of hash table from 8,192 to 65,535 and accomplishes the purpose grouping all patterns with the same hash value, which is not implemented in the Agrep package. The performance of the Modified-WM algorithm is more efficient than that of the WM algorithm [17]. It is also proved that the performance of all string matching algorithms is sensitive to the tuning of the hash table in practice.

2.1.3 Hashing-based

The Rabin-Karp (RK) algorithm [11] is designed to handle single-pattern matching with less memory. Hash values for the patterns are calculated and saved during the preprocessing stage. Then, matching can be done by calculating the hash value for each m-character string of the text and comparing it with the hash value, where m is the length of the pattern. If the hash values are the same, the pattern is compared with the specific position of the text. In order to cope with large pattern sets, the RK algorithm with binary search was proposed. In addition, Muth and Manber use two-level hashing to enhance the performance of the RK method (denoted as RKBT) [12].

The FNP algorithm [21] also uses hashing method to accelerate the pattern matching. It is designed particularly for the pattern set in which the length of the shortest pattern is extremely short, say 2 or 3. It keeps the average shift distance as close to B as possible by considering the characters within the block of B characters, namely prefix sliding window, to determine the best shift distance. The shift distances are recorded in a skip distance table that is similar to the shift table in the WM algorithm for indexing during the search. Like the WM algorithm, the verification for a true match follows if a partial match within the search window is found.

2.1.4 Bit-parallelism-based

The Shift-Or (SOR) algorithm [22] and Backward Nondeterministic Dawg Matching (BNDM) algorithm [11] were proposed for the single-pattern matching with bit-parallelism. The SOG and BG algorithms [7] extend the SOR and BNDM algorithms for the multi-pattern matching. Multiple patterns are thought of as a single pattern, and hence the same position of characters in each pattern is grouped into the same class. A class is processed instead of a character, so the bit-parallelism for matching a single pattern can be applied to multiple patterns. They use q-grams to

reduce the possibility that the *q*-gram characters in the text appear in one of the patterns in the pattern set so that fewer verifications are required. These algorithms use the RKBT algorithm to accelerate the verification of whether a true match occurs.

2.2 Selected Packages

We select open source packages for observation and experiments in the profiling because the source code is available. Table 2 lists the number of patterns, the maximal pattern length, the minimal pattern length and all supported algorithms in three open-source packages for each network content security application. It is also recognized that whether all patterns are classified into different group and the single group is searched for during the searching stage. In addition, the character set distribution for all patterns is also observed. As to detailed description about the searching stage for all packages, we explain as follows.

TABLE 2: Selected open-source packages

Application	Packages	Version	Algorithms	Number of Patterns	Classified	Pattern Length	Char. Set Distribution
Anti-Virus	ClamAV	0.85	**Aho-Corasick** **Wu-Manber**	26467	No	10~210	Type 1
Content-Filter	DansGuardian	2.8.0.4	**Horspool** **DFA**	5867	No	2~64	Type 2
IDS/IPS	Snort	2.3.3	AC-std AC-Full AC-Sparse AC-Banded AC-SB **Modified-WM** LowMemTrie	Patterns for all groups: 14295 Total rules: 2246	Yes 173 groups Max group size: 1174 Min group size: 12	1~107	Type 1

Type1 : close to uniform distribution

Type2 : biased to English character set

2.2.1 ClamAV

ClamAV is an open-source anti-virus package. It contains two types of virus patterns: a basic pattern is a simple sequence of characters that identify a virus, and a multi-part pattern is composed of more than one basic sub-pattern. First, basic

patterns are scanned by a multi-pattern matching algorithm, the WM algorithm. This type of virus patterns occupies 93% of total patterns. If no virus is found in this stage, it then handles the multi-part patterns. The type of the multi-part patterns occupies 5% of total patterns. All sub-patterns of a multi-part pattern must match so as to find a virus. This part is scanned by the extended AC algorithm, a trie with two levels [23], because the AC algorithm could handle regular expressions. During preprocessing, all sub-patterns beginning with the same prefix are stored in a linked list under the appropriate trie leaf node. In the searching stage, it is fed with input characters one by one in the text in order to transform the state. If a leaf node is encountered, all patterns inside the linked list are checked using sequential string comparisons. This process keeps until the last input character is read, or a match is found. If a match is found, it will check all parts constructed by a candidate multi-part pattern to verify whether a true match occurs or not. If a match is not found, it will use the message digest to find out whether the text is malicious tool or not. This is the whole searching method for the ClamAV package.

2.2.2 DansGuardian

DansGuardian is an open-source content filtering package. It uses content keywords classified in the configuration files to find out whether the content is inappropriate content or not. All content keywords are searched once during the scanning period, because the content can not be classified into the correct group before the matching. Further, the BMH algorithm and a deterministic finite automata (DFA) algorithm are implemented for the searching. They are implemented as follows. During preprocessing, it builds a big array to simulate a matrix in order to set up a whole DFA, called the graph data by its author. All content keywords go through the processing of the graph data to avoid repetition. Then it searches the graph data to find out the nodes with fewer than 12 branches and deletes them from the graph data.

These content keywords are classified into another group because they have the same prefix and have no more than 12 keywords. They are searched with the BMH algorithm one by one in the searching stage in order to avoid coping with the nodes with fewer branches. Then it continues to search all content keywords with the graph data. Finally, it checks all searched keywords to decide whether the content is inappropriate or not. This is the filtering approach for the DansGuardian package. In addition, its configuration also supports the force quick search method. If the *forcequicksearch* flag is enabled, all content keywords are searched with the BMH algorithm one by one.

2.2.3 Snort

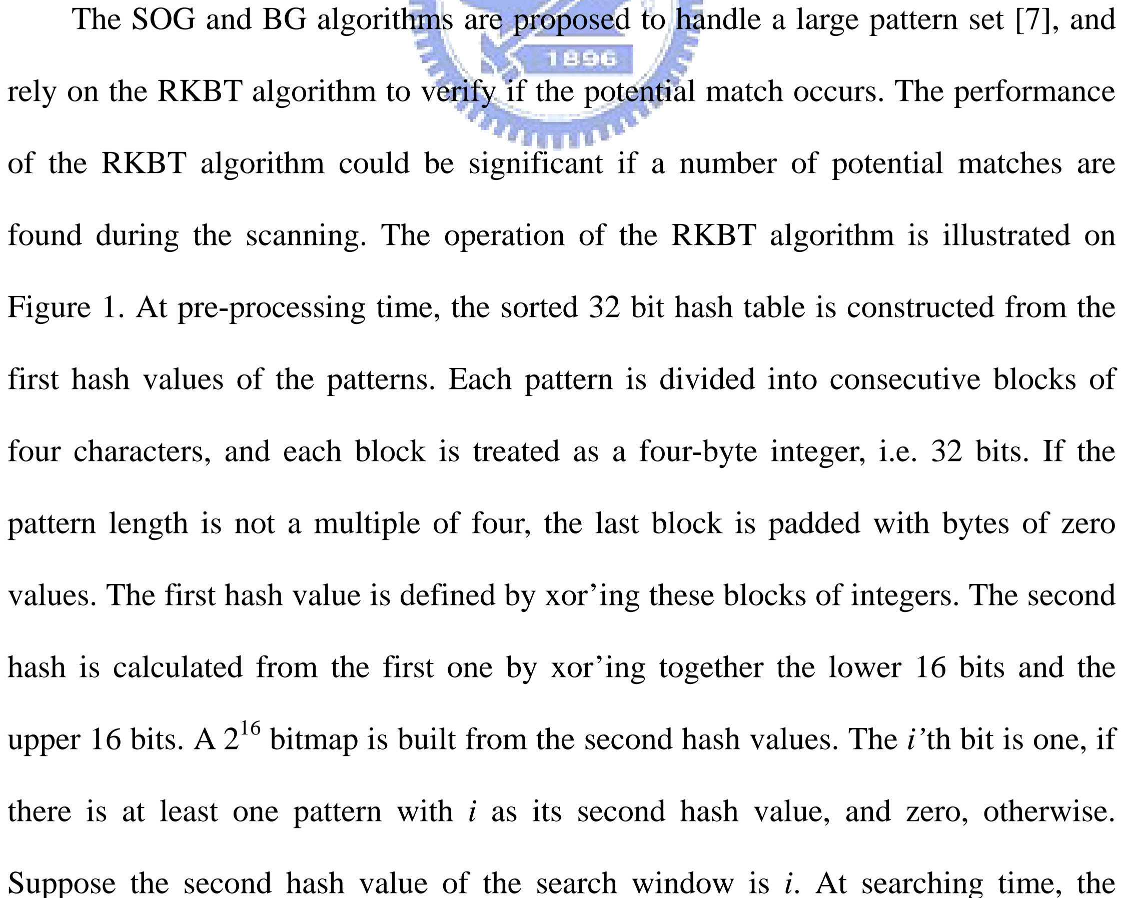
Snort is a popular open-source package for network intrusion detection. As opposed to other packages, it uses the packet header to classify all rules. During run-time it uses a two-stage architecture to inspect data and find the matching rules. The first stage quickly identifies potential match rules based on the packet header. The next stage uses the Modified-WM algorithm by default. If a potential match is found, Snort queues the match and inspects until all packet matches are located. When the inspection is finished and needs to validate the rest of the rule, it runs into the second stage. The inspection of the second stage performs a complete rule inspection to test the rest of the rule using the standard parameterized rule processing. If a complete rule match has been found, it inserts the rule into the event queue. Finally, it processes the event queue and selects a single event for logging. This is a complete inspection manner. Moreover, the hybrid method is also proposed based on the size of the set in order to enhance the performance of content inspection [3]. The hybrid method is the BMH algorithm when coping with the sets of size 1, the Set-wise BMH when coping with the sets of sizes between 2 and 100, and the AC algorithm when coping with the sizes larger than 100.

Chapter 3 Practical Design Issues

Besides the algorithm itself, the implementation can impact the performance significantly. This chapter focuses on the practical design issues in the implementation. Some existing algorithms, such as the SOG and BG algorithms, have to use the verification algorithm to verify the potential match. The RKBT algorithm can serve the purpose. The RKBT algorithm could be inefficient for a huge pattern set. A Classified RKBT (CRKBT) is proposed to boost its performance. In addition, a number of the existing algorithms rely on hashing during the searching process. We will discuss various ways proposed by some existing algorithms to understand how to choose the most efficient one.

3.1 Verification Approach

3.1.1 Propose the CRKBT Algorithm



The SOG and BG algorithms are proposed to handle a large pattern set [7], and rely on the RKBT algorithm to verify if the potential match occurs. The performance of the RKBT algorithm could be significant if a number of potential matches are found during the scanning. The operation of the RKBT algorithm is illustrated on Figure 1. At pre-processing time, the sorted 32 bit hash table is constructed from the first hash values of the patterns. Each pattern is divided into consecutive blocks of four characters, and each block is treated as a four-byte integer, i.e. 32 bits. If the pattern length is not a multiple of four, the last block is padded with bytes of zero values. The first hash value is defined by xor'ing these blocks of integers. The second hash is calculated from the first one by xor'ing together the lower 16 bits and the upper 16 bits. A 2^{16} bitmap is built from the second hash values. The i'th bit is one, if there is at least one pattern with i as its second hash value, and zero, otherwise. Suppose the second hash value of the search window is i. At searching time, the

bitmap is checked. If the *i*'th bit in the bitmap is zero, no true match can occur and the verification fails. When the corresponding bit of the second hash value is one, say the 2345'th bit on Figure 1, the 32 bit hash table is searched with binary search. If a first hash value is found, the characters of the pattern with that value is compared with those in the search windows one by one to check if a true match occurs. Otherwise, the verification fails.

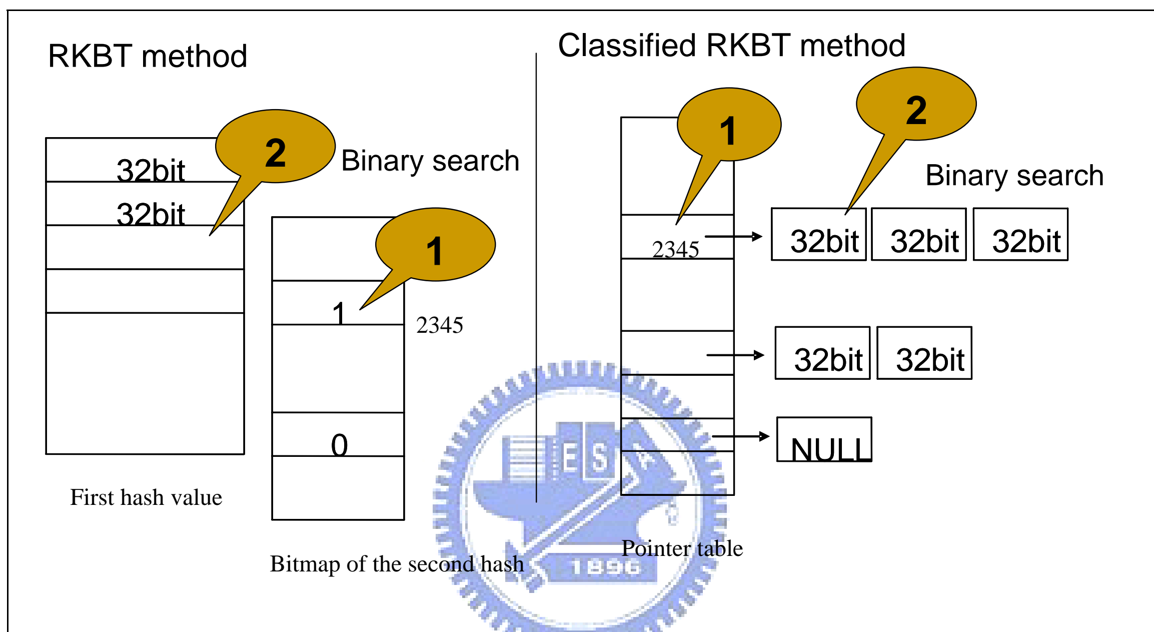


FIGURE 1: the RKBT algorithm vs. the Classified RKBT algorithm

As the number of pattern grows huge, say 100,000 patterns, the probability that a bit is set to 1 on the bitmap will be closer to 1, and consequently searching the first hash table is almost un-avoided. The size of the first hash table is equal to the pattern set size, and so the search is slow. A classified approach, namely the CRKBT algorithm, is proposed to improve the original RKBT algorithm. The CRKBT algorithm also used two-level hashing and binary search. The CRKBT algorithm uses the pointer table instead of the bitmap. The *i*'th pointer points to a sorted array, which is constructed at least one pattern with *i* as the second hash value, and point to NULL, if no pattern has *i* as the second hash value. At searching time, the pointer table is checked. If the corresponding pointer of the second hash value is not NULL, the

sorted array that the pointer points to is searched with binary search. The search scope is reduced to a small subset of the patterns that have the same second hash value, and so the binary search can be much faster. The improvement and analysis are presented below.

3.1.2 Experiments

The RKBT and CRKBT algorithms are benchmarked as follows. The text of 32 MB is randomly generated from the alphabet of 256 characters. The patterns are generated from the same alphabet and the length of the shortest pattern is 8. Both the text and the patterns reside in the main memory in the beginning. The tests run on a computer with a 2.8 GHz Pentium 4 processor, 1 GB of memory and 512 kB cache. The algorithms are written in C, compiled with the gcc compiler, and running on Linux 2.6.5.

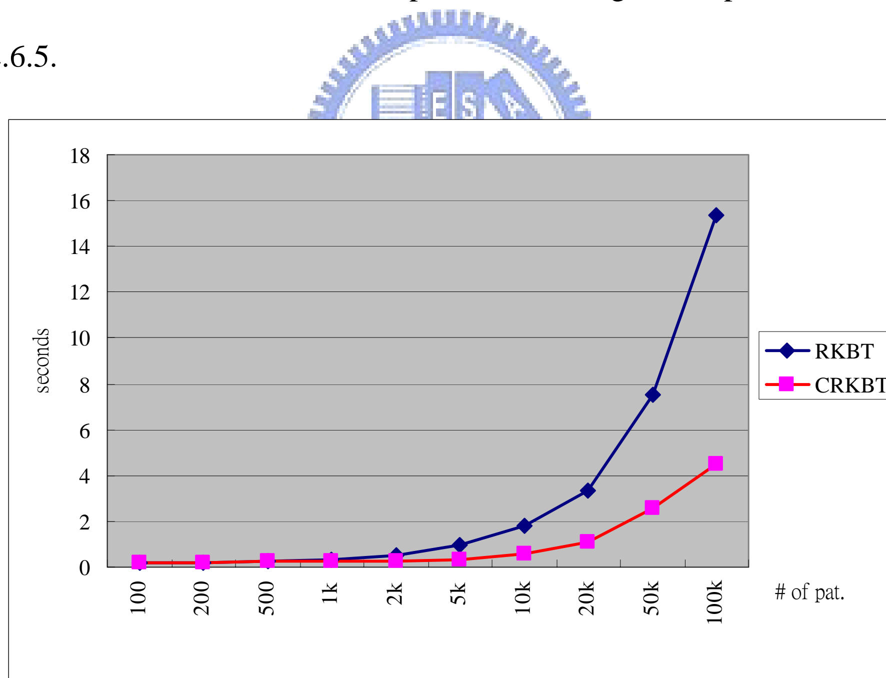


FIGURE 2: the RKBT algorithm vs. the CRKBT algorithm

Figure 2 shows the experimental results from benchmarking both algorithms. When the number of patterns is small, the execution time of both algorithms is very close, because possible matches are unlikely to happen and few chances of binary search in the first hash table are needed. As the number of patterns increases gradually,

the chances of binary search also increase. Because the search scope of binary search in the CRKBT algorithm are smaller than that of the RKBT algorithm, the benefit of the CRKBT algorithm becomes significant and so the CRKBT algorithm is faster than the RKBT algorithm. The CRKBT algorithm can be four times faster than the RKBT algorithm when the number of pattern grows to 100,000 patterns. Therefore, the CRKBT algorithm is suitable for huge pattern sets.

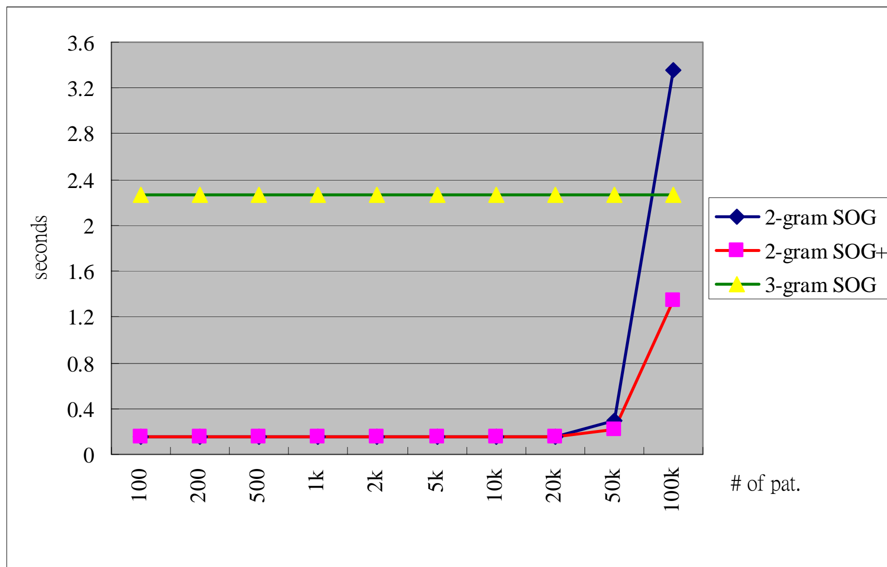


FIGURE 3: the 2-gram SOG algorithm vs. the 2-gram SOG+ algorithm

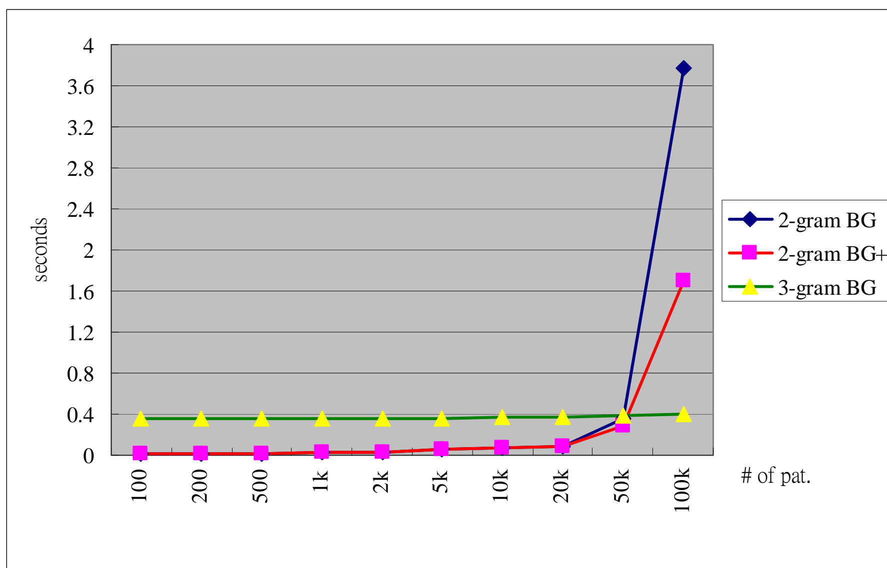


FIGURE 4: the 2-gram BG algorithm vs. the 2-gram BG+ algorithm

We implement the CRKBT algorithm instead of the RKBT algorithm into the SOG and BG algorithms, denoted as the SOG+ and BG+ algorithms. Figure 3 and 4 show the performance of the SOG and BG algorithms are also improved. For instance, the BG+ algorithm is twice faster than the BG algorithm when the number of pattern grows to 100,000 patterns. The efficiency of both algorithms will be significantly improved as the number of potential matches increases.

3.1.3 Analysis

The difference between the RKBT and CRKBT algorithms is the binary search in the first hash table. The number of memory accesses is essential in this stage. Suppose the number of pattern is *r*. The number of memory accesses in each algorithm is estimated in Formala (1) and (2) below respectively.

The RKBT algorithm

$$\begin{cases} p \times \log_2 r + (1-p) \times 1, & r \le 65536 \\ \log_2 r, & r > 65536 \end{cases} \quad \text{..... (1)}$$

The CRKBT algorithm

$$\begin{cases} p \times \log_2 (1 + \frac{r}{65536}) + (1-p) \times 1, & r \le 65536 \\ \log_2 (1 + \frac{r}{65536}), & r > 65536 \end{cases} \quad \text{..... (2)}$$

When the RKBT algorithm is running into searching stage, it checks the second hash table first. Suppose the probability that the corresponding bit of the second hash value is set to one is *p*. If the bit is set to one, the binary search on the first hash table is followed. The expected number of memory accesses is $p \times \log_2 r$. Otherwise, that is $(1-p) \times 1$. The expected number of memory accesses in both conditions is $p \times \log_2 r + (1-p) \times 1$. When the number of pattern is larger than 65,536 patterns, the number of memory accesses is $\log_2 r$. So Formula (1) is derived. The difference

between the RKBT and CRKBT algorithms is search scope, and consequently the expected number of memory accesses on the CRKBT algorithm is $\log_2\left(1+\frac{r}{65536}\right)$. The expected number of memory accesses for the CRKBT algorithm is written in Formula (2). However, $\log_2\left(1+\frac{r}{65536}\right)$ is much smaller than $\log_2 r$ in the RKBT algorithm. Therefore, the CRKBT algorithm is more efficient than the RKBT algorithm.

3.2 Hash Function

Many algorithms need using hash function to enhance the performance. For example, the RKBT algorithm uses the hash value to find out the potential matching and the Wu-Manber algorithm uses the hash table to index the shift distance. It is thus clear that the hash method is a critical component for string matching algorithm. This section will discuss the selection of the hash function.

There are four implementations of the same hash functions, but they can be translated into different machine codes. They are enumerated as follows:

char str=”abcdefgh”;

(1) *(unsigned short *)str

(2) (unsigned short)(*(unsigned int *)str)

(3) ((*(str+1))<<8) | *str

(4) ((*(str+1))<<8) + *str

We want to know what hash function is best choice for the multi-pattern matching algorithm. Thus, we profile the performance with these hash functions under a situation, calculating the hash value plus using the hash value to look up the hash table.

TABLE 3 lists the profiling result which is running 5,000 times to parse the 32 MB text and getting from the total execution time. The execution time listed in

TABLE 3 is not the same with each other. The type of the first and second hash function has the similar execution time and is faster than that of the other ones, because the first two methods have the fewer memory accesses than the others. According to this observation, we can figure out the multi-pattern matching algorithm using the first and second hash function would get good performance than that using the other ones. But the SPARC architecture can not use the first two methods to calculate the hash value, because the address of an integer should be a multiple of the integer size. The third and fourth hash functions are the only choices in this architecture.

TABLE 3: the profiling result for some different hash functions

Index	Hash function	Time (sec)
(1)	*(unsigned short *)str	1516.829 sec
(2)	(unsigned short)(*(unsigned int *)str)	1516.903 sec
(3)	((*(str+1))<<8) | *str	1585.203 sec
(4)	((*(str+1))<<8) + *str	1584.982 sec

Chapter 4 Profiling Algorithms

This chapter compares the typical algorithms surveyed in section 2.1 in accordance with external profiling and internal profiling. According to the profiling results, we can conclude which algorithm is suitable on what situation.

4.1 External Profiling

The benchmarking environment is the same as that in section 3.1.2. We first implement some earlier algorithms, the AC and WM algorithm, and some novel algorithms, the BG and SOG algorithms, and test them to get average time over 1,000 runs using the same text and patterns. Moreover, the implementation of the WM algorithm refers to that of the Agrep package and the WM algorithm discussed below generally points at the implementation of the Agrep package.

4.1.1 Experiments with Earlier Algorithms

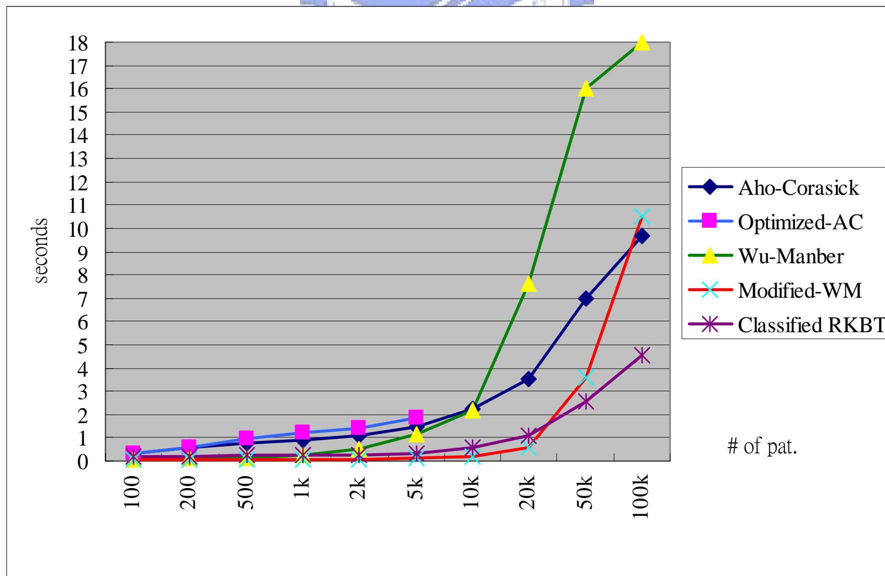


FIGURE 5 Earlier Algorithms Benchmarking Results

A number of multi-pattern matching algorithms were proposed after 1975. The AC and WM algorithms were famous ones for this string matching research area. Also the WM algorithm had been proved that it has good performance under small pattern

sets [7, 13]. After, the Snort research team had proposed the Optimized-AC and Modified-WM algorithms, which are different from the original algorithms due to the variable tuning, to enhance the performance [17]. So we compare them in this section to understand which algorithm is more efficient than the others

Figure 5 shows the benchmarking results which are tested with LSP=8. We also compare with the CRKBT algorithm proposed in section 3.1.1. This experiment demonstrates that the Modified-WM algorithm is more efficient than the others when the pattern set size is smaller than 20,000. However, when the pattern set size is greater than 20,000, the CRKBT algorithm is the most efficient. The Modified-WM algorithm and CRKBT algorithm are the fastest ones, so we select these two algorithms as traditional algorithms and compare them with two novel algorithms, the BG+ and SOG+ algorithms.

4.1.2 Experiments with Novel Algorithms

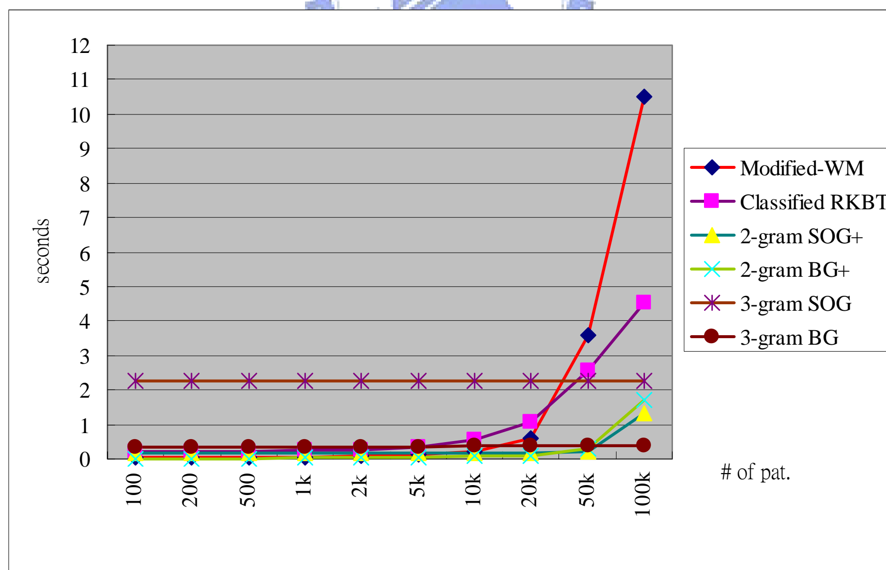


FIGURE 6 Novel Algorithm Benchmarking Results

Figure 6 shows the benchmarking results which are also tested with LSP=8. We can find out the traditional algorithms are less efficient than the others. The 2-gram BG+ algorithm is the fastest one than the others when the pattern set size is smaller

than 50,000. As the pattern set size is greater than 50,000, the 3-gram BG+ algorithm is the fastest one.

In the experiments of earlier and novel algorithms, we can conclude the BG+ algorithm is the more efficient algorithm than the others for LSP=8. As to verify the benchmarking results, we will do internal profiling later.

4.2 Internal Profiling

After the external profiling, we can know what algorithm is the most efficient. But some results need to verify. For example, why does the BG+ algorithm has good efficiency and the Modified-WM algorithm is more efficient than the WM algorithm? We will go through the internal profiling so as to answer the questions. The shift distance, the potential matching and the memory accesses of each algorithm are profiling as follows.

4.2.1 Shift Distance

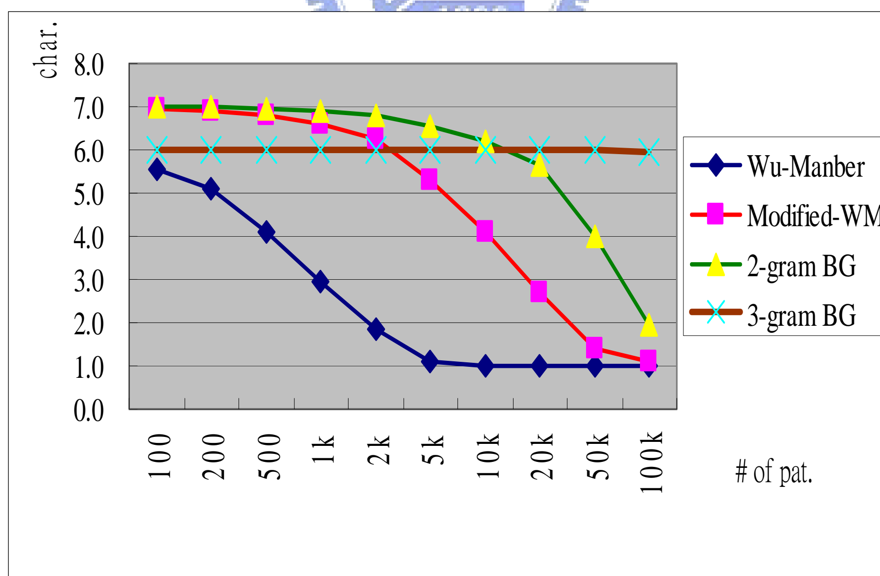


FIGURE 7 the Shift Distance Profiling

Both the WM and BG+ algorithms are the sub-linear ones. The WM algorithm uses the shift table to record the shift value. The BG+ algorithm also uses the B table plus the bit-parallelism method to calculate the shift value, where the B table keeps

whether each character of all patterns occurs or not. So we will profile the shift distance in order to justify the prior results.

Figure 7 shows the profiling results of the average shift distance. According to the results of the average shift distance, we can find out the average shift distance of the WM algorithm is close to one character when the pattern set size between 5,000 and 100,000. So the WM algorithm is not suitable for huge pattern sets. The average shift distance of the Modified-WM algorithm is greater than that of the WM algorithm. This result easily proves the Modified-WM algorithm is more efficient than the WM algorithm. In addition, it is clearly proved that the 2-gram BG+ algorithm is more efficiency when the pattern set size is smaller than 20,000 and the 3-gram BG+ algorithm has larger shift distance than 2-gram BG+ algorithm while the pattern set size between 20,000 and 100,000.

4.2.2 Potential Matching

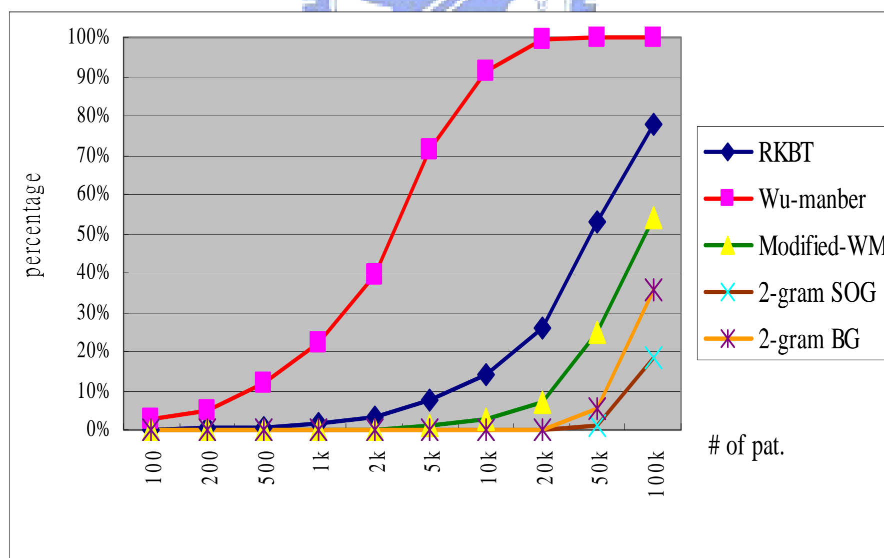


FIGURE 8 the Potential Matching Profiling

Some algorithms are filtering ones that need the verification algorithm to check whether the potential match is a true match or not. As the number of potential matches increase, the string matching performance will decrease and the verification become a bottleneck. The number of the potential matches will be profiled in each filtering

algorithm in this section.

Figure 8 shows the percentage of the potential matching for all filtering algorithms. The result shows the potential matching of the Modified-WM algorithm is less than that of the WM algorithm. This result proves the Modified-WM algorithm is more efficient than the WM algorithm, too. In addition, the thing that the potential matching of the WM algorithm increases fast also proves the WM algorithm is less efficiency while the pattern set size is more than 10,000. Finally, it is also proved that the BG+ algorithm is more efficient than the Modified-WM algorithm.

4.2.3 Memory Accesses

It is insufficient to explain the results in accordance with the results of the shift distance and the potential matching. For example, why does the CRKBT algorithm is the fastest one than the others as the pattern set size is more than 50,000 in figure 5? This section will profile the number of memory accesses to prove it.

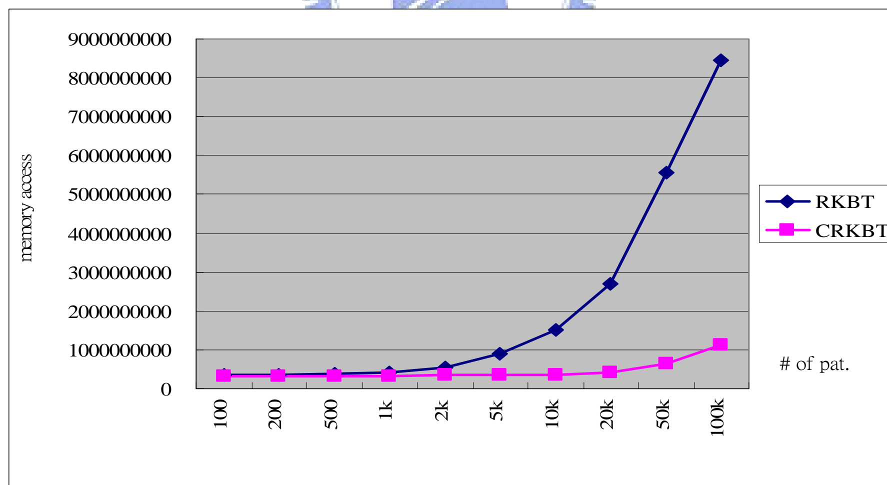


FIGURE 9 RKBT vs. CRKBT

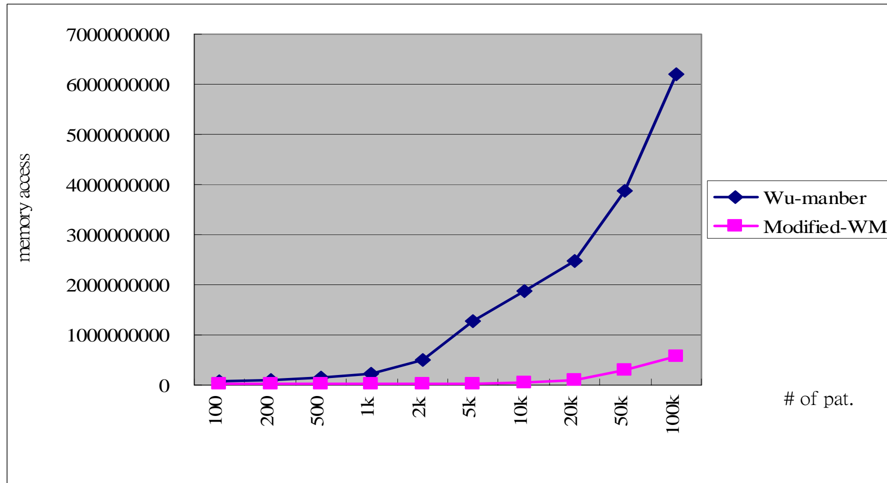


FIGURE 10 Wu-Manber vs. Modified-WM

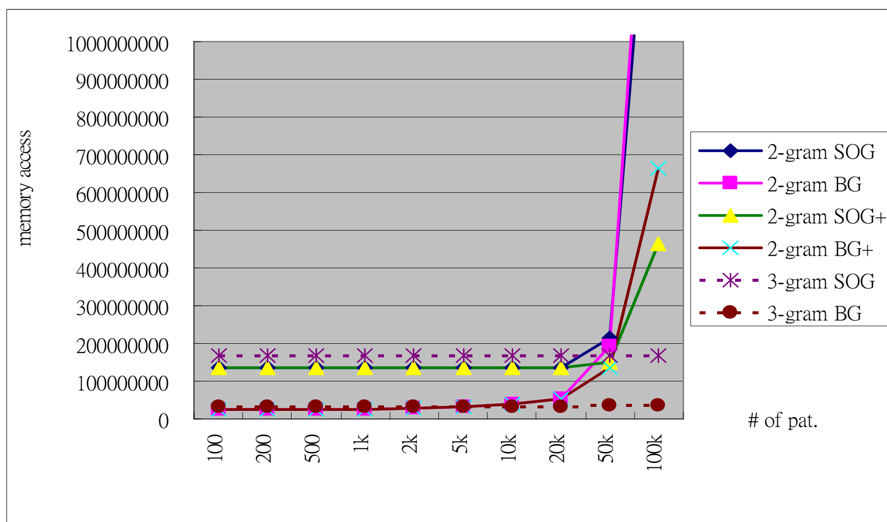


FIGURE 11 BG+ vs. SOG+

Figure 9, 10 and 11 show the results of the number of total memory accesses from program level profiled from Valgrind [24]. The memory accesses of these three figures are the same as well as the results of the external profiling, because the properties of these three types of algorithms are the same. For example, the RKBT and CRKBT algorithms have the same hash function, hash table size and cache miss rate. In addition, it is also proved here again that the CRKBT algorithm is more efficient than the RKBT algorithm.

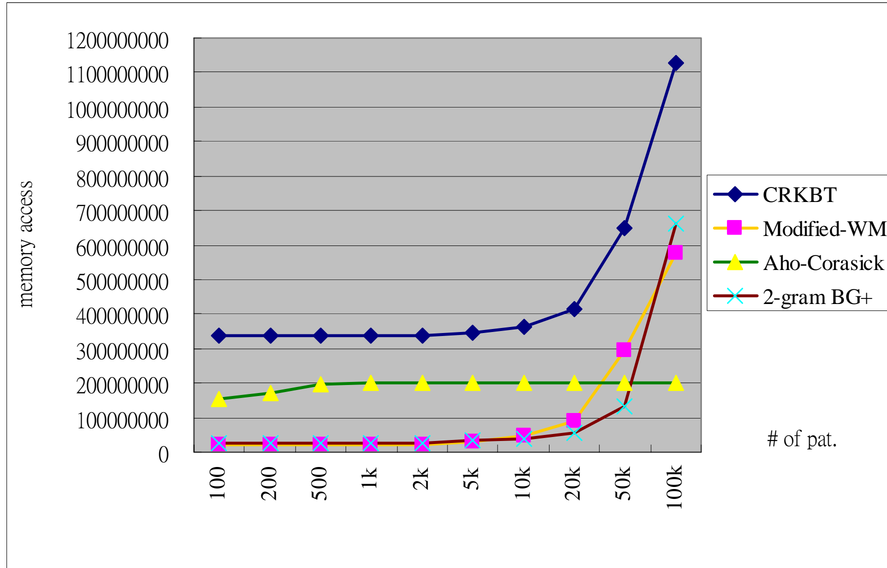


FIGURE 12 the number of memory accesses

When the algorithms of different properties are compared with each other, the results are not the same as above under the huge pattern sets. Because huge pattern sets can bring about many verifications and the cache miss rate are not similar to each other. This result can be observed on figure 12.

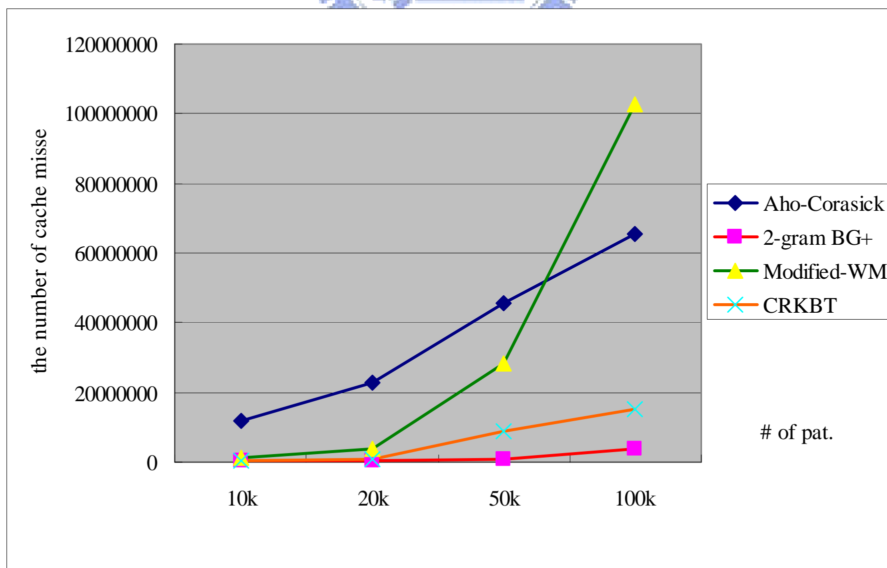


FIGURE 13 the number of L2 cache misses

The number of memory accesses from the program level is insufficient to justify the prior results. Because the penalty of L2 cache misses dominate the total

performance. For this reason, we profile the number of L2 cache misses to verify the exceptional results.

Figure 13 shows the number of L2 cache misses for the CRKBT algorithm is less than that for the Modified-WM algorithm and the number of L2 cache misses for the 2-gram BG+ algorithm is the least. According with these results, we can easily prove the prior results, include that the CRKBT algorithm is more efficient than the Modified-MW algorithm as the pattern set size is larger than 50,000. In addition, it is also proved again that the 2-gram BG+ algorithm has best efficiency under huge pattern sets.

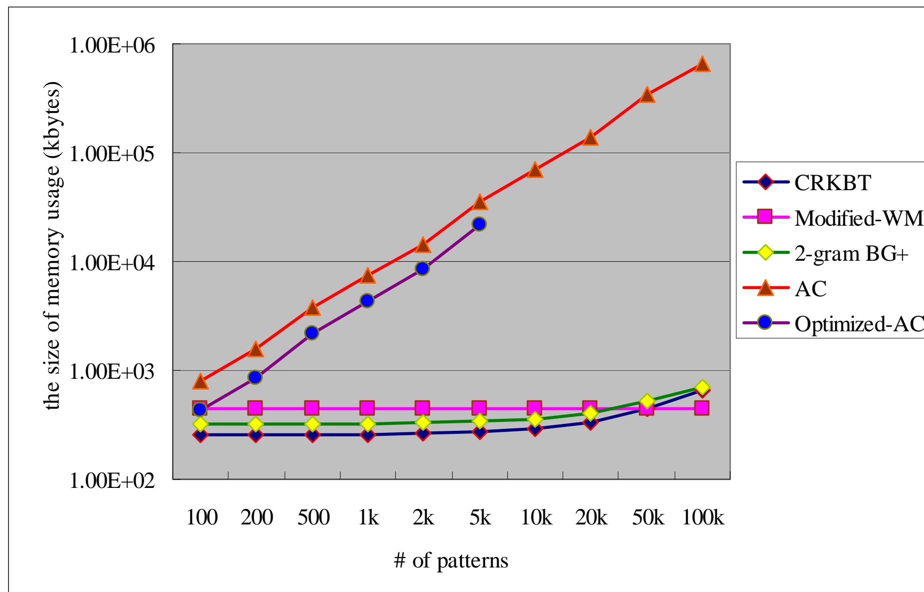


FIGURE 14 the size of memory usage

In addition to profiling the number of total memory accesses, figure 14 shows the size of memory usage for all algorithms. Under the small pattern sets, the CRKBT algorithm uses less memory for building the preprocessing table. The Modified-WM algorithm uses fixed size of memory for building shift table and hash table. The type of the AC algorithms uses large memory space as the pattern sets increasing. However, the Optimized-AC algorithm is more complexity for building deterministic finite automata when the pattern set is large than 10,000. So this area can not provide the

profiling results.

4.3 Profiling Summary

The external and internal profiling demonstrates that the 2-gram BG+ algorithm is the most efficient for LSP=8 with the pattern set size smaller than 50,000 and the 3-gram BG+ algorithm is the more efficient for LSP=8 with the pattern set size between 50,000 and 100,000. In addition, we also profile the length of the shortest pattern between 1 and 7. The rank of efficiency for LSP between 4 and 7 are the same as that for LSP=8. But the rank of efficiency for LSP between 1 and 3 are not the same as well as before. Figure 15, 16 and 17 show the Aho-Corasick, FNPw2 and Modified-WM algorithms are the fastest algorithm for LSP=1, 2 and 3, respectively. The profiling results are summarized in figure 18.

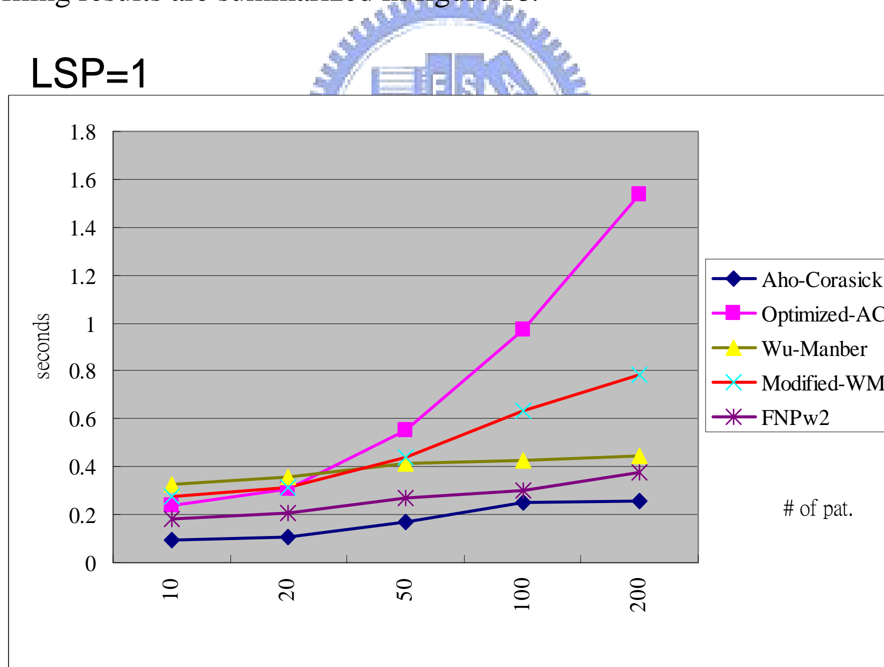


FIGURE 15 the profiling result of LSP=1

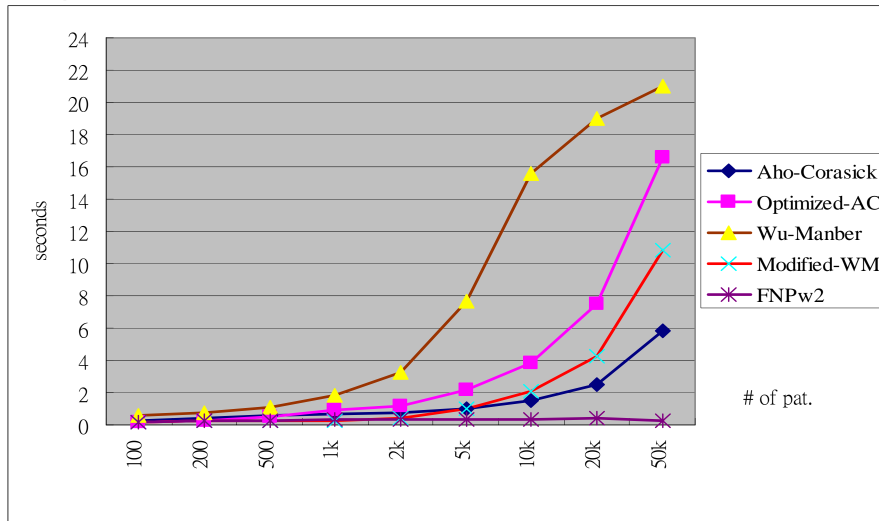


FIGURE 16 the profiling result of LSP=2

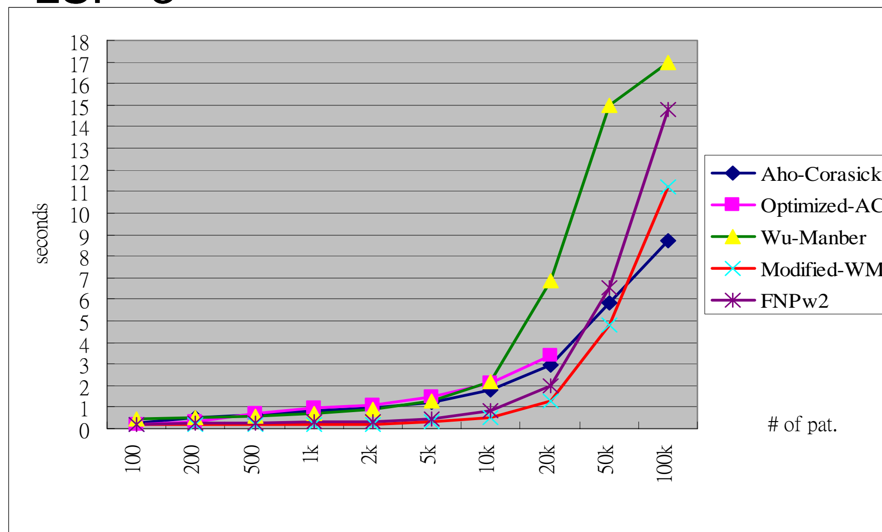


FIGURE 17 the profiling result of LSP=3

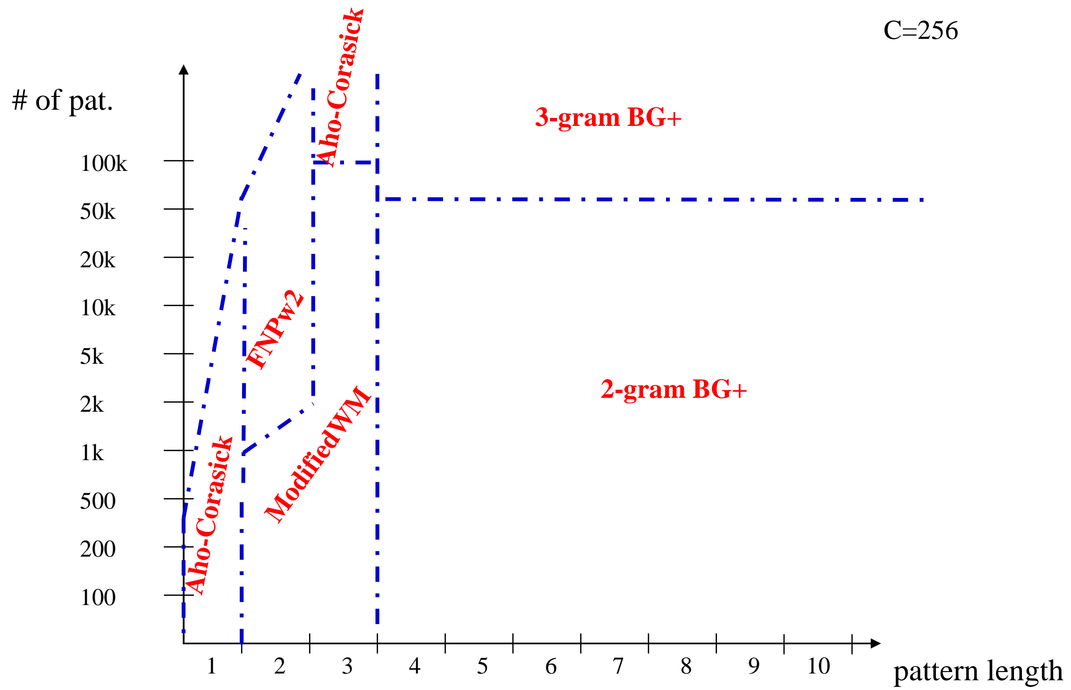


FIGURE 18 the profiling summary

Chapter 5 Experiments on Real Applications

5.1 Implementations in three packages

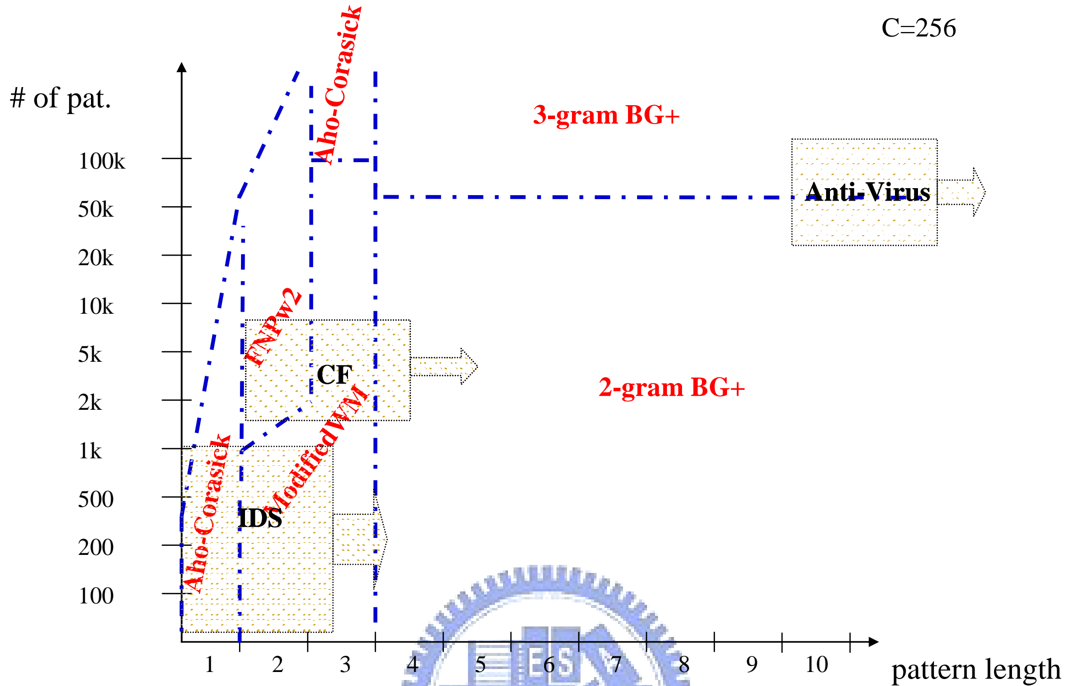


FIGURE 19 the profiling summary

Figure 19 concludes which package is located on which position and suits for which algorithm by means of the profiling results in chapter 4 and survey in chapter 2. The detailed description about the implementation of the real package is explained as follows.

5.1.1 ClamAV

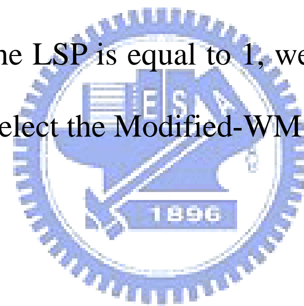
The LSP of exact matching patterns in ClamAV is 10 and the pattern set size is more than 30,000. We implement the 2-gram BG+ algorithm instead of the WM algorithm to handle exact matching, i.e. basic patterns described in chapter 2. If the pattern set size is more than 65,535, we can change from the 2-gram BG+ algorithm to the 3-gram BG+ algorithm to enhance the efficiency. In addition, we keep to using the AC algorithm to handle regular expression, i.e. multi-part patterns mentioned in chapter 2.

5.1.2 DansGuardian

According to our observation, there are 25 content keywords scanned with the BMH algorithm one by one. If the *forcequicksearch* flag is enabled, all patterns are processed with the BMH algorithm, which requires to scanning the text as many times as the number of patterns. We implement the multi-pattern matching algorithm, the Modified-WM algorithm, to handle the short patterns, the LSP between 2 and 3, and also use the 2-gram BG+ algorithm to handle the long patterns, equal to or longer than 4 characters.

5.1.3 Snort

The Snort package uses the packet header to group all patterns. The LSP of every group is not the same. We use the hybrid method instead of the default method, the Modified-WM algorithm. If the LSP is equal to 1, we select the AC algorithm. If the LSP is larger than 1, we also select the Modified-WM algorithm.



5.2 Benchmarking

5.2.1 ClamAV

5.2.1.1 Benchmarking Methodology

We select 10 files for file size between 32 KB and 16 MB from Windows execution files. These files are scanned by original virus scan engine and modified virus scan engine. Then we can measure the execution time to understand whether the novel method is more efficient than the old one or not.

5.2.1.2 Benchmarking Results

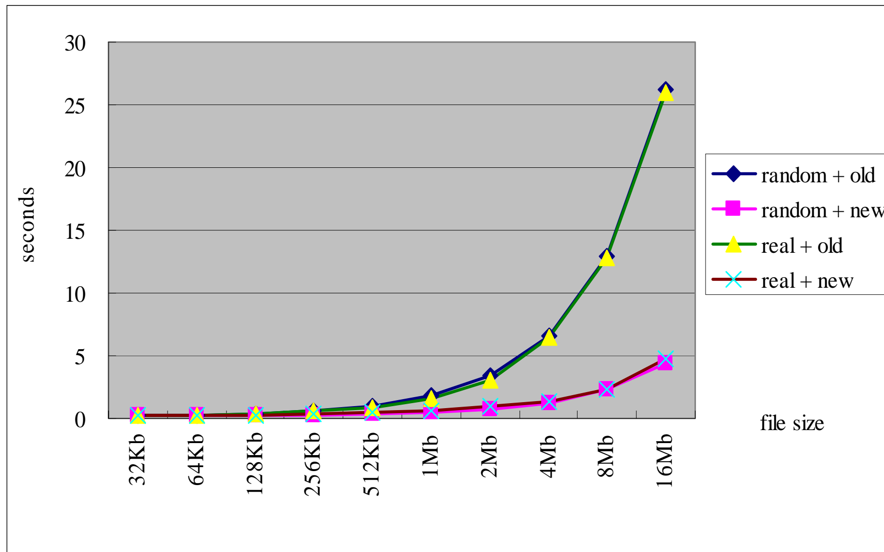


FIGURE 20 the benchmarking results for ClamAV package

Figure 20 shows the benchmarking results. As the file size is increasing, the difference in scanning time between the novel and old methods will become greater. For example, the novel method is five times faster than the old one while the file size is 16 Mbytes. The reason comes from that the percentage of string matching in the entire processing is increasing while the file size is increasing. According to these results, we can justify that the novel method is more efficient than the old method.

5.2.2 DansGuardian

5.2.2.1 Benchmarking Methodology

We only modify the section of the content filtering in DansGuardian package. Because we know the processing time of content filtering occupies 90% of the total execution time after our internal profiling. As to benchmarking content filtering, we use the wget tool [25] to mirror a Web site, the RFC Web site [26]. The RFC Web site contains more than 8,000 files, including HTML files and TXT files. The formats of these two types of files are scanned with DansGuardian's content filtering. This benchmarking method can see the difference between the novel and old methods.

5.2.2.2 Benchmarking Results

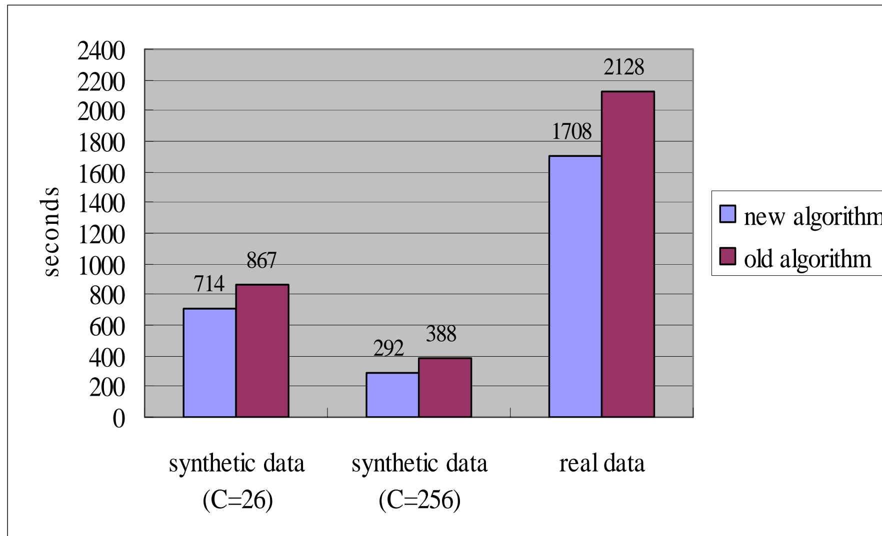


FIGURE 21 the benchmark results for DansGuardian package

Figure 21 shows the old method needs 2128 seconds to mirror the entire site and the new method just needs 1708 seconds to finish it. The efficiency of content filtering in DansGuardian package enhances 20% of total execution time. It does not arrive at significant improvement, because the property of content filtering needs to find all content keywords out and the verification algorithm can not use binary search to find a single keyword out. We use linear search instead of binary search to search all potential matches which have the same hash value. But the performance of content filtering also improves after implementing the new method into the DansGuardian package.

5.2.3 Snort

5.2.3.1 Benchmarking Methodology

The proportion of HTTP traffic accounts for great quantity under general network. More peer-to-peer protocols also use HTTP traffic as control messages. We select HTTP traffic for the experiments in our verification process. The benchmarking method is similar to that of the DansGuardian package. We use the wget client to mirror all RFC files from Web server across the inspection of the Snort package. In addition, the Snort package runs under the inline mode by means of the capability of

the Iptables [27] so as to understand the capability of the Snort package on intrusion prevention manner.

5.2.3.2 Benchmarking Results

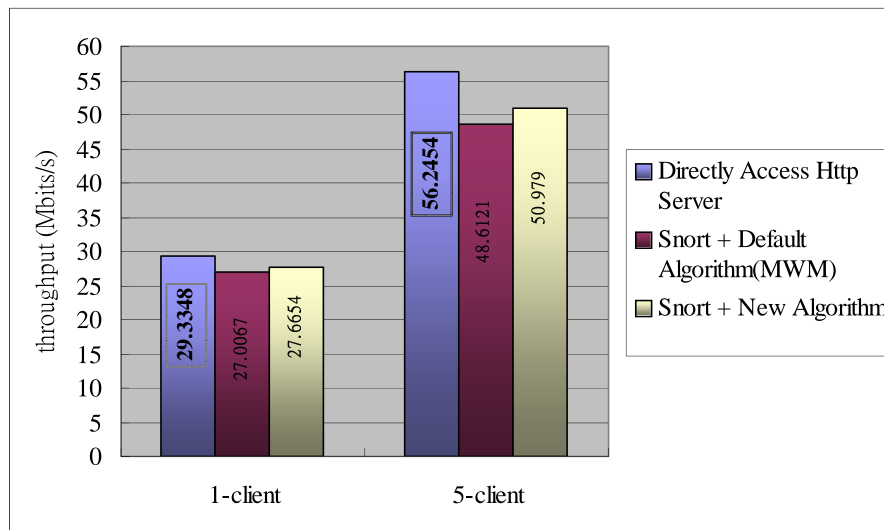


FIGURE 22 the benchmarking result for Snort package

Figure 22 shows the benchmarking result. First, we use a single client to mirror the entire RFC files. The performance of the Snort inspection can not come to significant improvement. After these benchmarking, we use five clients instead of a single client to mirror it. The performance of the Snort package is improved and more efficient than before with the last method. But the performance of the Snort package can not reach significant improvement, because the new version of the Snort package only inspects the HTTP header instead of HTTP header plus HTTP body [28].

5.3 Real Data vs. Synthetic Data

In profiling stage, all experimental data are synthetic data. In past experiment, the algorithm benchmarks also use synthetic data for the experiments to find out whether the algorithm is more efficient than the others or not. But the property of processing data in real applications is not the same as well as synthetic data. In this section, we will observe the difference between real data and synthetic data from three

content applications.

5.3.1 ClamAV

In benchmarking stage, we also generate synthetic data of the same size as the real files to observe the difference between real data and synthetic data. Figure 20 shows the processing time of the ClamAV package in real data and synthetic data are similar to each other. Because the ClamAV package scans the entire text to find the virus out and the character set distribution generated in all virus signatures is close to uniform distribution.

5.3.2 DansGuardian

We generate synthetic Web pages for external benchmarking. Except the content of the pages, the file size and the file names of every file are the same as all RFC files. First, we generate data with character set of 256 characters and observe the difference between real data and synthetic data. The processing time for synthetic data is faster than that for real data, because the character set distribution of synthetic data is close to uniform distribution and that of real data biases to English character set.

We also generate data with character set of 26 characters and observe the results again. The probability of potential matching is increasing and the processing time of content inspection becomes 3 times slower than the prior experiment. But the total potential matching for synthetic data is no more than real data; the processing time has some difference between the synthetic data generated with character set of 26 characters and real data. It is also a reason that the probability of common keywords that appear in real data is higher than they appear in synthetic data. For example, the common keywords are 'reference', 'issue' and 'chapter'. In addition, the property of English word also affects the results.

5.3.3 Snort

The observation between real data and synthetic data can not carry out. Because

the inspection content in Snort package is restricted within the HTTP header. If we generate synthetic data instead of the HTTP header, the communications between HTTP client and HTTP server will disappear. So we give up the observation in this package.

Chapter 6 Conclusions

In this research, some typical algorithms are reviewed and profiled to observe the performance of the string matching algorithms under various conditions and provide an insight of choosing the most efficient algorithm for designing content security applications. The AC algorithm is suitable for LSP=1, the Modified-WM algorithm is suitable for LSP =2 when the pattern set size is smaller than 1,000, the FNPw2 algorithm is suitable for LSP=2 when the pattern set size is larger than 1,000, the Modified-WM algorithm is suitable for LSP=3 and the BG+ algorithm is suitable for LSP$\geq$4. In addition, these results are also justified by means of the experiments of the real applications. Some applications have dramatically improved if the percentage of string matching processing on total execution time is great. These results also help to select an efficient algorithm to design a novel application in the future.

Meanwhile, the CRKBT algorithm is proposed and it is justified that the CRKBT algorithm is more efficient than the RKBT algorithm. The efficiency of the CRKBT algorithm is four times faster than the RKBT algorithm for huge pattern sets. Moreover, the BG+ and SOG+ algorithms that use it as the verification algorithm are also twice faster than the original algorithm.

This work also observes that the difference of performance between the real and synthetic data by means of the experiments of the real applications. The ClamAV package is not sensitive to the synthetic data or the real data, because the character set distribution is close to uniform distribution. But otherwise the application of content filtering is sensitive to the real data or synthetic data, because all patterns in the DansGuardian package is biased to English word. Moreover, it is observed that the bottleneck in content filtering application is to verify all potential matches in order to find out all matched content keywords, because the plenty of content keywords have

the same hash value.

Finally, the contributions of this work are summarized as follows: finding out the most efficient algorithm for each application of content security and telling why, proposing the CRKBT algorithm to enhance the performance of the original RKBT algorithm and comparing the difference of performance between the real and synthetic data in practice.

References

[1] S. Antonatos, K. G. Anagnostakis and E. P. Markatos, Generating Realistic Workloads for Network Intrusion, *WOSP 2004 ACM*, January 2004.

[2] S. Antonatos, K. G. Anagnostakis and E. P. Markatos, Performance Analysis of Content Matching Intrusion Detection Systems, *Institute of Computer Science*, January 2004.

[3] M. Fisk and G. Varghese, Fast Content-Based Packet Handling for Intrusion Detection, *UCSD Technical Report CS2001-0670*, 2001.

[4] C. W. Jan, Y. D. Lin and Y. C. Lai, An integrated proxy architecture for anti-virus, anti-spam, intrusion detection and content filter, Computer and Information Science, National Chiao-Tung University, June 2004.

[5] F. H. Huang, Y. D. Lin and Y. C. Lai, A fast accurate proxy for multi-language text webpage classification, Computer and Information Science, National Chiao-Tung University, June 2004.

[6] S. Wu and U. Manber, A Fast Algorithm for Multi-pattern Searchin, *Report TR-94-17, Department of Computer Science, University of Arizona*, 1994.

[7] J. Kytojoki, L. Salmela and J. Tarhio, Tuning String Matching for Huge Pattern Sets, *CPM 2003,* LNCS 2676, pp. 211-224, 2003.

[8] ClamAV, http://www.clamav.net/ .

[9] DansGuardian, http://dansguardian.org/.

[10] Snort, http://www.snort.org/.

[11] R. Karp and M. Rabin, Efficient Randomized Pattern-Matching Algorithms, *IBM Journal of Research and Development 31,* pp. 249-260, 1987.

[12] R. Muth and U. Manber, Approximate Multiple String Search, *CPM 96, Combinatorial Pattern Matching, Lecture Notes in Computer Science 1075,*

pp.75-86, *1996*.

[13] G. Navarro and M. Raffinot, Flexible Pattern Matching in Strings, *Cambridge University Press*, 2002.

[14] L. Cleophas, B. W. Watson and G. Zwaan, A New Taxonomy of Sublinear Keyword Pattern Matching Algorithms, *Department of Mathematics and Computer Science*, Technische Universiteit Eindhoven, April 22, 2004.

[15] G. Navarro and M. Raffinot, Fast and Flexible String Matching by Combining Bit-Parallelism and Suffix Automata, *ACM Journal of Experimental Algorithms 5,* pp. 1-36*,* 2000.

[16] A. Aho and M. Corasick, Efficient String Matching: An Aid to Bibliographic Search, *Communications of the ACM 18,* pp. 333-340, 1975.

[17] M. Norton, Optimizing Pattern Matching for Intrusion Detection, http://www.snort.org/.

[18] R. Boyer and S. Moore, A Fast String Searching Algorithm, *Communications of the ACM 20,* pp. 762-772, 1977.

[19] N. Horspool, Practical Fast Searching in Strings, *Software – Practice and Experience* 10 (1980), 501-506.

[20] S.Wu and U. Manber, Agrep – A Fast Approximate Pattern-Matching Tool. *Proc. Usenix Winter 1992 Technical Conference*, 1992, 153-162.

[21] R. T. LIU, N. F. HUANG, C. H. CHEN and C. N. KAO, A Fast String-Matching Algorithm for Network Processor-Based Intrusion Detection System, *Transactions on Embedded Computing Systems of the ACM,* Vol. 3, No. 3, pp. 614-633, August 2004.

[22] R. Baeza-Yates and G. Gonnet, A New Approach to Text Searching, *Communications of ACM 35,* pp.74-82, 1992.

[23] Y. Miretskiy, A.Das, Charles P. Wright and E. Zadok, Avfs: An On-Access

Anti-Virus File System, *USENIX Security Symposium*, 2004.

[24] Valgrind, http://valgrind.org/.

[25] Wget, http://www.gnu.org/software/wget/wget.html.

[26] RFC, http://asg.web.cmu.edu/rfc/rfc-index.html.

[27] Netfilter, http://www.netfilter.org/.

[28] M. Norton and D. Roelker, Snort 2.0 Protocol Flow Analyzer, http://www.snort.org/docs/.