Department of Electronic Engineering

FINAL YEAR PROJECT REPORT

BEngIE-2009/10-DP-02-BIEBIM

High speed string matching for virus scanning - Quick Sampling and Verification

Student Name: Wang Xiaoran
Student ID:
Supervisor: Dr. Pao, Derek C. W.
Assessor: Dr. Cheung, Ray C. C.

Bachelor of Engineering (Honours) in Information Engineering
Student Final Year Project Declaration

I have read the student handbook and I understand the meaning of academic dishonesty, in particular plagiarism and collusion. I declare that the work submitted for the final year project does not involve academic dishonesty. I give permission for my final year project work to be electronically scanned and if found to involve academic dishonesty, I am aware of the consequences as stated in the Student Handbook.

**Project Title:** High speed string matching for virus scanning – Quick sampling and verification

<table>
<thead>
<tr>
<th>Student Name : Wang Xiaoran</th>
<th>Student ID:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>Date : 11/04/2010</td>
</tr>
</tbody>
</table>
Table of Contents

List of Figures .................................................................................................................. ii
List of Tables .................................................................................................................... iii
Abstract .............................................................................................................................. iv

Chapter 1 Introduction ...................................................................................................... 1
  1.1 Antivirus status quo ..................................................................................................... 1
  1.2 Statement of the problem ............................................................................................. 1
  1.3 Objectives ................................................................................................................... 3
  1.4 Report outline ............................................................................................................... 3

Chapter 2 Background Study ............................................................................................ 4
  2.1 String matching .......................................................................................................... 4
    2.1.1 Automata theory .................................................................................................... 4
    2.1.2 Aho-Corasick pattern recognition algorithm ......................................................... 6
  2.2 Previous works on software and hardware string matching ....................................... 8

Chapter 3 Overview of QSV ............................................................................................ 10
  3.1 ClamAV virus database statistics .............................................................................. 10
  3.2 QSV and Aggregation Unit Architecture .................................................................... 11
    3.2.1 Simple string searching algorithm ........................................................................ 11
    3.2.2 QSV Architecture ................................................................................................ 12
    3.2.3 Tables lookup operation ....................................................................................... 15
    3.2.4 Calculation of the checksums .............................................................................. 17
  3.3 Calculation of the hash values ................................................................................... 19
  3.4 Aggregation Unit Architecture ................................................................................. 19

Chapter 4 Implementation detail ....................................................................................... 23
  4.1 QSV module construction work flow ......................................................................... 23
  4.2 Data flow ................................................................................................................... 24
  4.3 Lookup tables construction ....................................................................................... 25
    4.3.1 Rearrange the dataset ......................................................................................... 25
    4.3.2 Perform Bit-selection ........................................................................................... 28
    4.3.3 Construction of the tables T_1, T_2, T_3 ................................................................ 30
    4.3.4 Construction of tables A_0, A_1 .......................................................................... 37
    4.3.5 Output of tables T_1, T_2, T_3, A_0 and A_1 ......................................................... 45

Chapter 5 Performance Evaluation .................................................................................... 48

Conclusion ......................................................................................................................... 51

Glossary .............................................................................................................................. 53
List of Figures

Figure 1 Simple Transition Graph .................................................................5
Figure 2 Formal definition of non-deterministic finite automaton ......................... 5
Figure 3 Sample AC dictionary ..................................................................... 7
Figure 4 Sample AC simulation on an input text ............................................. 7
Figure 5 Simple string matching algorithms and their associated time complexities ..... 8
Figure 6 Pattern length distribution in the ClamAV pattern set ............................ 10
Figure 7 String searching engine diagram ..................................................... 12
Figure 8 Block diagram of the QSV component ............................................. 14
Figure 9 Differentiating patterns using bit-selection ....................................... 16
Figure 10 Brief program flow chart ............................................................... 23
Figure 11 Brief program flow chart explanation ............................................ 24
Figure 12 Sample outlook of sim_data.txt .................................................... 26
Figure 13 Visual “pushdown linked list” data structure .................................... 27
Figure 14 Snapshot of “data_rearrangedByH1.txt” ....................................... 28
Figure 15 Organization of T2 ..................................................................... 31
Figure 16 Sample bit selection tree ............................................................... 34
Figure 17 Snapshot of “ChainCode.txt” ......................................................... 38
Figure 18 Sample Patterns segmentation and AU state transition graph ............. 38
Figure 19 Overlapping segment ID reassignment .......................................... 43
Figure 20 Sample contents of “QSV-T1.txt” .................................................. 45
Figure 21 Sample contents of “QSV-T2.txt” .................................................. 46
Figure 22 Sample contents of “QSV-T3.txt” .................................................. 46
Figure 23 Sample contents of “QSV-A0.txt” .................................................. 47
Figure 24 Sample contents of “QSV-A1.txt” .................................................. 47
List of Tables

Table 1 ClamAV static strings (patterns) statistics ........................................................................................................ 10
Table 2 Comparison of the lookup tables T_1 and T_2 .................................................................................................. 14
Table 3 Sample lookup tables for 8 patterns .................................................................................................................. 15
Table 4 Probabilities of false positive against different pattern lengths ..................................................................... 18
Table 5 Region boundaries for the bit-selection algorithm ......................................................................................... 19
Table 6 Input files to the program .................................................................................................................................. 24
Table 7 Output files to the program .................................................................................................................................. 25
Table 8 Sim_data format .................................................................................................................................................. 26
Table 9 LNO – one bit differences .................................................................................................................................... 33
Table 10 Sample Segment ID in decimal and binary .................................................................................................. 41
Table 11 Sample reassigned Segment ID in decimal and binary .................................................................................. 42
Table 12 Sample pattern ID reassignment .................................................................................................................. 45
Table 13 Memory cost for lookup tables of the QSV module ....................................................................................... 48
Table 14 Memory allocation statistics for table T_2 ...................................................................................................... 49
Table 15 Sizes of 8 parallel memory modules in table T_2 .......................................................................................... 49
Table 16 Simulation results of the string searching engine .......................................................................................... 49
Abstract

This report introduces the lookup table constructions for the QSV and AU modules of a string searching engine of virus detection purpose. The QSV (quick sampling and verification) method is based on quick sampling of the input data with fixed-length prefixes, and on-demand verification of variable-length suffixes. Patterns are required to possess distinct 16-byte prefix, which is ensured by the segmentation process. Some short and exceptional cases patterns are transferred to pipelined Aho-Corasick (P-AC) module.[1] The AU (aggregation unit) is responsible for concatenating partial segment matches to produce the final pattern match. Only 3 lookup table entries in QSV are occupied for a pattern with unique 16-byte prefix and the overall memory cost is 1.4Mbyte. The lookup tables for QSV are T₁, T₂, and T₃, while the lookup tables for AU are A₀ and A₁. Several methods and algorithms are introduced to improve the memory efficiencies of the lookup tables by reducing the total memory cost.
Chapter 1 Introduction

1.1 Antivirus status quo
With the rapid transformation of technology, the internet is embracing larger and larger volume of data. According to a research conducted by the University of Minnesota [2], the internet traffic doubles every 100 days. However, the security threats always grow with the same speed. In this trend, the burden casted on the anti-virus software, firewall and intrusion detection system is huge. In fact, as the internet becomes mature, people tend to concern more about their privacy and security. Conventionally, every computer should be equipped with at least one set of anti-virus software. However, argument arises when comes to the trade-off between the security and speed, simply because those software consume a large amount of computer resources when scanning is in progress. Moreover, if the anti-virus software is passed on to be installed on the application gateway, demanded throughput in a network is difficult to reach for traffic of Terabyte per second. According to the Microsoft, the speed of the computer would be five times faster if there were no anti-virus software running in the background.

1.2 Statement of the problem
Because of the incapacity of the current software based anti-virus and IDS, it is necessary to develop hardware assisted approaches to mitigate the pressure on the software end.

The way that anti-virus and IDS software to detect virus or malicious patterns is rule based, which can be classified into two groups, namely static string and regular expression. ClamAV [3] virus database was used to explore the distribution of virus signatures. In total, there are
82,888 static strings and 7017 regular expressions. Hence, the majority of the signatures to handle are static strings and the QSV is designed only to handle all of the static strings.

There are abound active researches on the hardware-assisted pattern recognition engines based on the most popular Aho-Corasick (AC) algorithm [4], hashing, Bloom Filters [5] [6] and so forth. Most of their work is based on Snort IDS [7] virus signature database, containing around six thousand patterns, which is more than 10 times smaller than the ClamAV database used in our design.

The first challenging problem is the memory cost. Since the QSV is built on a single FPGA chip, the maximum memory allowed is only up to 2Mbyte. However, if all virus signatures were to be stored in the on-chip embedded memory, it would require at least 20Mbyte-200Mbyte memory space. In our approach of performing string matching over a very large virus signature set (82888 static strings), only 1.4Mbyte on chip memory is required, a.k.a about 1.4 bits per character of the signature set. In this way, a significant amount of memory cost is reduced.

The second challenging problem is the update to the FPGA chip. The ClamAV virus database updates itself every two hours. In addition, to reconfigure a FPGA chip needs from several hours to a day, rendering the intermediate configuration time without protection. However, our approach is memory based, meaning that the QSV system can be updated by only updating the lookup tables, without reconfiguring the circuit.
1.3 Objectives

The objectives of the project are as follows.

- Pre-process the virus signature set
- Construct the hierarchical Prefix Sampling (PS) lookup tables T1 T2 T3
- Construct the aggregation unit and produce two lookup tables A0 and A1
- Reassign patterns and segments IDs to enhance the memory efficiency.
- Evaluate the system as a whole and perform several memory allocation methods to further improve the memory usage.

1.4 Report outline

In chapter 2, background information such as automata theory, Aho-Corasick algorithm, software and hardware string matching related work will be introduced. Later in chapter 3, an overview of the QSV module including its workflow, architecture as well as the ClamAV virus signature database statistics will be discussed. In chapter 4, the implementation detail of QSV and AU with the construction of their several lookup tables will be explained. Then in chapter 5, performances of the memory and the whole module will be evaluated. Last comes the conclusion.
Chapter 2 Background Study

2.1 String matching

2.1.1 Automata theory
The engine that tries to match the virus patterns is essentially a finite state machine. The finite state machine (FSM) consumes one input character at once and checks against its state graph to see whether there could be a valid transition from the current state to a next state. An automaton is a mathematical model for a finite state machine (FSM) and is represented formally by the 5-tuple \( (Q, \Sigma, \delta, q_0, F) \) [9], where:

- \( Q \) is a finite set of states.
- \( \Sigma \) is a finite set of symbols.
- \( \delta \) is the transition function, that is \( \delta: Q \times \Sigma \rightarrow Q \).
- \( q_0 \) is the start state, that is, the state in which the automaton is when no input has been processed yet, where \( q_0 \in Q \).
- \( F \) is a set of states of \( Q \) called accept states.

For example, a very simple example of the automaton \( M_0 \) is illustrated below.

- \( Q = \{q_1, q_2, q_3\} \)
- \( \Sigma = \{0, 1\} \) (binary)
- \( \delta: \)
  - \( q_1 \times 0 = q_1 \)
  - \( q_1 \times 1 = q_2 \)
  - \( q_2 \times 0 = q_3 \)
  - \( q_2 \times 1 = q_2 \)
  - \( q_3 \times 0 = q_2 \)
  - \( q_3 \times 1 = q_2 \).
- \( q_0 = q_1 \)
- \( F = \{q_2\} \)
We say that the finite automaton accepts input string \( \omega \) if \( \omega \) can transit from the start state to the end state. For example, in our case, \( M_0 \) accepts 1, 01, 11, 100, 101, 1101 and so forth.

A deterministic finite automaton (DFA) is a finite automaton with only deterministic transition rules, meaning that the transition function of one state and one input symbol can produce exactly only one next state.

A nondeterministic finite automaton (NFA), in contrast, can have several next states from one state one input symbol. In addition, every NFA can be converted to an equivalent DFA.

A formal definition of NFA is shown below in Figure 2. [9]

A **nondeterministic finite automaton** is a 5-tuple \( (Q, \Sigma, \delta, q_0, F) \), where

1. \( Q \) is a finite set of states,
2. \( \Sigma \) is a finite alphabet,
3. \( \delta: Q \times \Sigma \rightarrow \mathcal{P}(Q) \) is the transition function,
4. \( q_0 \in Q \) is the start state, and
5. \( F \subseteq Q \) is the set of accept states.

**Figure 2 Formal definition of non-deterministic finite automaton**

In the QSV system, since some of the virus signatures are segmented, the system will report a positive match if all of the segments are found consecutively in the correct order. This job is done by the aggregation unit, which essentially is a multi-active-states NFA and will be
introduced in Chapter 3 later. Notice that the input symbols to the NFA are the segment IDs instead of simple alphabets.

2.1.2 Aho-Corasick pattern recognition algorithm
Aho-Corasick algorithm is a dictionary matching algorithm that searches for elements of a finite set of strings in the input text, developed by Alfred V. Aho and Margaret J. Corasick. Since it locates all patterns in one time, the time complexity of the algorithm is proportional to sum of the length of the patterns, length of the input text and the number of matches.

In this algorithm, a trie with suffix tree-like set of links from is established from each node representing a string to the node corresponding to the longest proper suffix. Since it also consists of links from each node to the longest suffix node that connect to a match string, all of the matches can be traversed by going along the resulting linked list. The trie is utilized at runtime to keep track of the longest match and the suffix links are used to make sure the computation is proportional to the length of the input. For every link along the dictionary suffix linked list and every node in the dictionary located, a match is found.

Since most of the time, the pattern database is known ahead, program can be created to build the trie, compile it and save it for later use. In this case, the computational complexity in the runtime is proportional to the sum of the length of the inputs and the number of matched entries.

Figure 3 [10] shows an example of AC data structure made up from a couple of strings. Each row represents a node in the trie while each column indicates the distinct order of characters from root to the node.
In every step, the current node will try to find its child recursively if the suffix child does not exist until it reach the root node. Steps taken when scanning “abccab” are shown below in Figure 4. [10]

Since there may be two or more dictionary entries at a character location in the input text, more than one dictionary suffix link may need to be followed.
2.2 Previous works on software and hardware string matching

There are myriad of software that can perform string matching and most of them are used for anti-virus purpose. Examples of string matching software other than anti-virus applications are “grep” in UNIX, regular expression matching in Perl, “preg_match” in PHP and so forth. As for the algorithms to perform string searching, several popular ones are shown below in Figure 5. [10]

Let \( m \) be the length of the pattern and let \( n \) be the length of the searchable text.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Preprocessing time</th>
<th>Matching time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive string search algorithm</td>
<td>( \Theta(0) )</td>
<td>( \Theta((n-m+1) \cdot m) )</td>
</tr>
<tr>
<td>Rabin-Karp string search algorithm</td>
<td>( \Theta(m) )</td>
<td>average ( \Theta(n+m) ), worst ( \Theta((n-m+1) \cdot m) )</td>
</tr>
<tr>
<td>Finite state automaton based search</td>
<td>( \Theta(m \cdot</td>
<td>\Sigma</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt algorithm</td>
<td>( \Theta(m) )</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Boyer-Moore string search algorithm</td>
<td>( \Theta(m +</td>
<td>\Sigma</td>
</tr>
<tr>
<td>Bitap algorithm (shift-or, shift-and, Baeza-Yates-Gonnet)</td>
<td>( \Theta(m +</td>
<td>\Sigma</td>
</tr>
</tbody>
</table>

**Figure 5 Simple string matching algorithms and their associated time complexities**

As for the hardware string matching, it’s a relatively new territory and does not have as many examples as does the software. However, some work done by the previous researchers is still worth mentioning. Dharmapurikar and Krishnamurthy used Bloom filter to perform string searching. However, the system can only determine if the input key is a member of the pattern set or not. Ho and Lemieux [11] later used Bloomier filters in their PERG architecture, extending the capacity of Bloom filter by selecting the hash function carefully to match the pattern in the input data. However, only trial-and-error is available to create the hash function and it’s infeasible to the pattern updates. Once update causes hash collision, a set of new hash functions must be recreated. Cho [12] designed a pattern detection module (PDM), which uses selected bytes from the input data to calculate a hash index. Those
patterns saved in the hash index will be retrieved and compared with the input. Since the limitation of PDM, only up to 8 byte of pattern is supported. Hence long patterns are segmented and will later be aggregated using dedicated long pattern state machines (LPSM). Song [13] presented a method to add one additional cached state to the DFA to eliminate those backward transition edges pointing back to nodes that are two hops away. Hua [14] also proposed a novel idea to reduce the memory cost and enhance the speed performance of AC algorithm by transforming the input data and string patterns to another alphabet set using a “content-invariant” variable-stride segmentation method.
Chapter 3 Overview of QSV

3.1 ClamAV virus database statistics

The virus database used in this project is from ClamAV. The latest version of the database is v51 released on 14 May 2009. Since only static strings are taken into account, all the other MD5 checksums and regular expressions are excluded. In total, there are 82,888 static strings found in two files, main.db and main.ndb. Following is a description table of their statistical information [15].

<table>
<thead>
<tr>
<th>Statistics</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total No. of static strings</td>
<td>82,888</td>
</tr>
<tr>
<td>Minimum pattern length</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Maximum pattern length</td>
<td>392 bytes</td>
</tr>
<tr>
<td>Average pattern length</td>
<td>102 bytes</td>
</tr>
<tr>
<td>Patterns having length less than 16 bytes – short patterns</td>
<td>1258 (1.5%)</td>
</tr>
<tr>
<td>Patterns having length more than 16 bytes – long patterns</td>
<td>81630 (98.5%)</td>
</tr>
<tr>
<td>Patterns having length more than 180 bytes</td>
<td>172 (0.2%)</td>
</tr>
</tbody>
</table>

94% of the long patterns have distinct 16-byte prefix

Table 1 ClamAV static strings (patterns) statistics

The following figure shows the histogram of the pattern length distribution.

![Histogram of pattern length distribution](image)

Since zero value cannot be shown using logarithmic scale, the number of patterns for each length in the plot is set to 1 plus the actual value.
3.2 QSV and Aggregation Unit Architecture

3.2.1 Simple string searching algorithm
A simple approach can be used to perform the string searching. Let \( \Gamma \) be the pattern set, and \( L \) be the set of discrete pattern lengths. Let \( C_j \) be the set of checksum for patterns in \( \Gamma \) with length equal to \( j \). The checksums can be generated using a predefined CRC polynomial. The following pseudo code shows the simple algorithm.

```c
//Pseudo code of the string searching method
for (i = 0; i < in_stream.length; i++)
{
    for each j in L
    {
        if(i+j < in_stream.length)
        {
            c_in=checksum of in_stream[i..(i+j-1)];
            if(c_in==pattern_k.checksum in C_j)
            {
                report a possible match (i, pattern_k);
            }
        }
    }
}
```

As discussed above, there are more than 82,000 static strings in the ClamAV dataset, resulting in the inefficiency of running such program on a common sequential computer. However, if hardware is used to assist the above computation, high efficiency can be achieved, in one condition that all of the patterns in the dataset should have distinct 16 bytes prefix. In fact, this condition can be fulfilled by pre-processing the patterns and segment the long ones so that every single one of them has a unique prefix.

In the input stream, the system will do a quick sampling by extracting a couple of predefined selected bits and try to find a match with the pattern prefixes lookup table. If no one is found, no further process is needed on this byte location. Otherwise, the system will proceed to calculate the CRC checksum for the corresponding pattern length. If the checksum is equal to the value listed in the lookup table, a possible match is found.
3.2.2 QSV Architecture
In the proposed string searching engine, short patterns and a small amount of patterns are processed by the pipelined Aho-Corasick (P-AC) match engine [1]. In contrast, the majority of the patterns will be handled by the quick sampling plus verification (QSV) engine. As for the byte count, QSV will intake 99.3% of the whole pattern set.

The following diagram (Figure 7) illustrates the basic architecture of the string searching engine. The whole engine basically consists of three components, the P-AC, QSV and the AU.

![String searching engine diagram](image)

**Figure 7 String searching engine diagram**

The P-AC architecture utilizes pipelined processing to eliminate all the failure and backward transition edges in the state transition graph of the AC algorithm such that the overall size of the lookup table can be reduced very significantly.

The QSV component comprises of two parts, the prefix sampling unit (PS) and CRC checksum verification unit (CRC). Strings are required to have distinct 16-byte prefix in order to be processed by the QSV component. However, if some strings share the same 16-byte prefix, they will be cut into several segments, each of which has length of 16 bytes or
longer. Moreover, segments of length 1 to 3 bytes will be ignored by the QSV and will be verified by software.

The AU is a finite state machine which keeps more than one active states and is used to concatenate each of the partial match to produce the final pattern match result.

Let $\Gamma_{PAC}$ denote the set of strings processed by P-AC and $\Gamma_{QSV}$ denote the set of strings processed by QSV. Every checksum for the prefix of patterns in $\Gamma_{QSV}$ is computed in advance and stored in the embedded memory. The hardware will utilize a shift register to buffer 16 bytes of data (Figure 8). In addition, a circuit is used to calculate the 16 bit CRC checksum of the 16-byte data in the buffer. If the calculated checksum is the same as the one stored in the memory, further verification will be conducted. Otherwise, execution of the current byte location stops here.

The checksum of the 16-byte data in the buffer is required to be compared with around 82K possible values. This problem can be resolved by Direct Index plus Bit Selection (DIBS) algorithm using a two-level hash table architecture. The unique 16-byte patterns prefix are used to be the set of keys $K$. Then, bit selection will be performed to select 30 bits out of the 128-bit keys to construct a 30-bit hash index. The bit selection algorithm is designed to limit the maximum number of elements in one bucket to be 8. Supposing that a bucket contains $e$ entries ($e > 8$), then $(e - 8)$ patterns in this bucket will be moved to $\Gamma_{PAC}$. After enforcing this restriction, only up to 8 prefix checksums will be compared in parallel.

There are three advantages of using the bit selection algorithm.

i. The size of the bucket is limited and in control

ii. It’s simple to generate the hash index
iii. No hardware reconfiguration is needed since only reprogramming of the hash function is enough if modification and update to the pattern set is demanded.

Figure 8 Block diagram of the QSV component

When the 30-bit hash value is adopted, the size of the conceptual hash table can be quite large, e.g. 2^30. However, only around 82K entries are utilized. Hence, we construct a hierarchical structure of the lookup hash table containing tables T1 and T2 illustrated in Figure 8. In addition, up to 8 parallel memory modules are used to construct the lookup table T2. In terms of the sizes, table T1 is 32K and T2 is 93K (sum of all of the parallel memory modules). When accessing these two tables, direct index is available for T1 while direct indexing plus bit-selection (DIBS) [1] is used for T2. Following Table 2 summarizes the characteristics of the lookup tables T1 and T2.

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of memory modules</td>
<td>Single</td>
<td>Multiple (up to 8 parallel memory modules)</td>
</tr>
<tr>
<td>Size</td>
<td>32K</td>
<td>93K</td>
</tr>
<tr>
<td>Accessing method</td>
<td>Direct index</td>
<td>Direct indexing plus bit-selection (DIBS)</td>
</tr>
</tbody>
</table>

Table 2 Comparison of the lookup tables T1 and T2
### 3.2.3 Tables lookup operation

We use an example to show the detailed table lookup operation of tables T₁ and T₂. For simplicity, the number of parallel memory modules is limited to one. Table 3 shows the sample patterns and lookup tables.

<table>
<thead>
<tr>
<th>Sample patterns</th>
<th>length (byte)</th>
<th>selected bits</th>
<th>prefix checksum</th>
<th>pattern checksum</th>
<th>partial checksum_1</th>
<th>partial checksum_2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>h₁</td>
<td>h₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₁</td>
<td>16</td>
<td>0002</td>
<td>605c</td>
<td>b169</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>P₂</td>
<td>26</td>
<td>0003</td>
<td>5104</td>
<td>841e</td>
<td>7ac0</td>
<td>b2bc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7ac0</td>
</tr>
<tr>
<td>P₃</td>
<td>17</td>
<td>0003</td>
<td>04f3</td>
<td>a72b</td>
<td>b5a9</td>
<td>b5a9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b5a9</td>
</tr>
<tr>
<td>P₄</td>
<td>37</td>
<td>0004</td>
<td>15e8</td>
<td>b81e</td>
<td>e535</td>
<td>98f7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f676</td>
</tr>
<tr>
<td>P₅</td>
<td>64</td>
<td>0005</td>
<td>0369</td>
<td>967e</td>
<td>2e14</td>
<td>e6fc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>369d</td>
</tr>
<tr>
<td>P₆</td>
<td>112</td>
<td>0005</td>
<td>75a2</td>
<td>89a6</td>
<td>48fe</td>
<td>2e94</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f709</td>
</tr>
<tr>
<td>P₇</td>
<td>40</td>
<td>0005</td>
<td>2391</td>
<td>74b9</td>
<td>e1b8</td>
<td>e1c6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6551</td>
</tr>
<tr>
<td>P₈</td>
<td>45</td>
<td>0005</td>
<td>3240</td>
<td>3e7d</td>
<td>a22e</td>
<td>c271</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1885</td>
</tr>
</tbody>
</table>

**Table 3 Sample lookup tables for 8 patterns**

The 30 selected bits are divided into two parts, h₁ and h₂, each of which contains 15 bits. Hexadecimal values of the h₁, h₂, the bit mask, checksums are provided in the tables above. h₁ is used as the address to lookup table T₁. To be specific in this example, patterns P₂ and P₃ are in the bucket 0003 in T₁, while patterns P₅ to P₈ sit in the bucket 0005. In addition, T₁ stores the 15-bit mask vector and a 16-bit base address used for accessing T₂ with DIBS approach. Since table T₂ is constructed by using several parallel memory modules, although the size of it is about 93K, the address range is no more than 64K. If there are more than one patterns residing in a bucket in T₁, the bit mask will be utilized to create an offset value for accessing T₂. Finally, if a logical bucket is empty, the base address value of the corresponding bucket is set to all ‘f’.
Take an example of the group of patterns $P_5$ to $P_8$ that sitting in the bucket 0005 in $T_1$. The 4 patterns can be differentiated on the $0^{th}$ and the $4^{th}$ bits of their $h_2$ hash values together. Figure 9 illustrates it.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>value of $h_2$ in binary</th>
<th>address offset by taking the value of bits 0 and 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_5$</td>
<td>0000 0011 0110 1001</td>
<td>01</td>
</tr>
<tr>
<td>$P_6$</td>
<td>0111 0101 1010 0010</td>
<td>00</td>
</tr>
<tr>
<td>$P_7$</td>
<td>0010 0011 1001 0001</td>
<td>11</td>
</tr>
<tr>
<td>$P_8$</td>
<td>0011 0010 1101 0000</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 9 Differentiating patterns using bit-selection

Therefore, the bit mask stored in the bucket 0005 of $T_1$ is equal to 0011 (hexadecimal). The offset is created by a bit selection circuit taking the value of $h_2$ and the bit mask as inputs.

Assume $g$ bits of the bit mask, $b_{s1}, b_{s2}, \ldots, b_{sg}$, are equal to 1, where $s1 < s2 \ldots < sg \leq 15$. Let the value of $h_2$ be $i_{s1} \ldots i_{sh}$. The offset value produced by the bit-extraction circuit is equal to $0..0i_{sg} \ldots i_{s2}i_{s1}$.

In this case, a block of size $2^g$ entries in $T_2$ is allocated for the corresponding $T_1$ bucket.

Moreover, we can artificially allocate each of the blocks to start from an integral multiple of $2^g$, then the last $g$ bits of the base address must be ‘0’. In this way, a simple bitwise logical-OR operation is needed to perform the addition of the base address and the bit mask values.

Supposing that the values of $h_1$, $h_2$, and the checksum of the buffered data are “0005”, “1234”, and “abcd”, respectively. The system will access the address 0005 of $T_1$ first and retrieve the base address (0004) and the bit mask (0011) from it. Then since bits 0 and 4 of “1234” are equal to “10” in binary, the offset value is calculated to be 2. Hence, the address used to access $T_2$ is equal to 0004+2 in the next cycle. Later on, the system lookup the checksum from $T_2$ (3e7d) and compares with the calculated one (abcd). Since this is not a match, no further processing at this byte location is needed.
Take another scenario as an example where the values of $h_1$, $h_2$, and checksum of the buffered data are “0005”, “2391” and “74b9” respectively. The system will access address 0005 of T1 and then 0004+3 of T2. Since the calculated and retrieved checksums match with each other, a verification command for the current pattern will be got from T3 with the address equal to pattern ID. The verification command consists of four components. (Byte count, pattern checksum, partial checksum_1, partial checksum_2) If the length of the pattern is equal to 16, the byte count retrieved from T3 is zero. Then this particular pattern will be reported to the AU without further verification. On the other hand, if the pattern length is greater than 16, then the system will wait for the subsequent bytes and check whether they match the pattern suffix. Hence, verification is necessary if the length of the pattern is greater than 16. There is no need to verify the patterns from the first byte since the previous 16 byte prefix is assumed to have a match. If the checksum is created by a 16-bit CRC polynomial and the CRC verification unit consumes 1 byte of data per cycle, then 3 bytes of data is the needed a least to be processed. Let the bytes of a patterns are numbered from 0 to L-1, where L is the pattern length. Only bytes from 14 to L-1 are computed to get the pattern checksum. Therefore, L-14 is the byte count value in the verification command, if L is larger than 16.

### 3.2.4 Calculation of the checksums

Let $i$ denotes the current location index and $t$ be the current number of cycle. In cycle $t$, the system lookup table T1 and then T2 in $t+1$. If there is a match for the calculated and the stored checksums, a verification command is fetched from T3 in cycle $t+2$ and sent to a free CRC unit. Then the CRC unit will begin to calculate the pattern checksum in cycle $t+3$, by which time, byte 14 of the corresponding input data would have moved forward to the 11th location of the input buffer. Therefore, byte 11 of the shift register is fed to the CRC units. In addition, every CRC unit is equipped with a count-down counter of value equal to the byte
count received in the verification command. At the end of each cycle, the counter is decremented by 1. When the value of the counter is zero, the checksum is ready to be compared with that in the verification command. If a match is found, then it’s sent to AU for later pattern concatenation.

In order to improve the efficiency of the QSV system when dealing with long patterns, e.g. up to 392 bytes, two partial checksums are used after cycles 7 and 17 in the verification process. Partial checksum_1 and partial checksum_2 are the checksums computed from byte 14 to 20 or to 30 respectively for the given patterns. In addition, 2 partial checksums are equal to the pattern checksum if the pattern length is between 17 to 21 bytes, while the partial checksum_2 is the same as the pattern checksum if the pattern length is between 22 and 31 bytes. Moreover, the CRC unit will initiate two checking procedure in the 7th and 17th byte location. If the current checksum is not equal to the partial checksum at those given location, the verification process will be stopped. In this way, better efficiency is achieved for the long patterns since there is a chance that the verification process will be aborted in the middle.

Table 4 below summarizes the false positive of patterns with different lengths.

<table>
<thead>
<tr>
<th>Pattern length (bytes)</th>
<th>Necessary verification checksums</th>
<th>Probability of false positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>= 16</td>
<td>Prefix checksum</td>
<td>$4.7 \times 10^{-10}$</td>
</tr>
<tr>
<td>&gt; 16 and &lt; 22</td>
<td>Prefix checksum and pattern checksum</td>
<td>$7.1 \times 10^{-15}$</td>
</tr>
<tr>
<td>&gt;= 22</td>
<td>Prefix checksum, pattern checksum and one or two partial checksum(s)</td>
<td>$1.1 \times 10^{-19}$</td>
</tr>
</tbody>
</table>

**Table 4 Probabilities of false positive against different pattern lengths**
3.3 Calculation of the hash values

There are 15 regions for the total 128-bit input stream (Table 5), each of which will provide 2 bit to form h1 and h2. In addition, some regions like R8 to R14 and R0 to R7 have some overlapping areas. This arrangement actually prevents the selected bits from concentrating on a few bytes of the 16-byte prefix as well as providing much more combinations to try when performing the bit-selection algorithm. The hash value h1 is formed by extracting one bit from each region, while h2 collects the remaining bits. As for the bit-selection circuit, only two 16-to-1 (or 17-to-10 multiplexors are two 4-bit (or 5-bit) register that saves the offset of the selected bits are required. The generation process of the two hash values will need another 15 bits of storage and 15 copies of 2-bit cross-bar switch. It’s simple to reprogram the hash function by just modifying the values stored in those registers.

<table>
<thead>
<tr>
<th>region</th>
<th>lower bound</th>
<th>upper bound</th>
<th>region</th>
<th>lower bound</th>
<th>upper bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_0</td>
<td>0</td>
<td>15</td>
<td>R_8</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>R_1</td>
<td>16</td>
<td>31</td>
<td>R_9</td>
<td>19</td>
<td>36</td>
</tr>
<tr>
<td>R_2</td>
<td>32</td>
<td>47</td>
<td>R_{10}</td>
<td>37</td>
<td>54</td>
</tr>
<tr>
<td>R_3</td>
<td>48</td>
<td>63</td>
<td>R_{11}</td>
<td>55</td>
<td>72</td>
</tr>
<tr>
<td>R_4</td>
<td>64</td>
<td>79</td>
<td>R_{12}</td>
<td>73</td>
<td>90</td>
</tr>
<tr>
<td>R_5</td>
<td>80</td>
<td>95</td>
<td>R_{13}</td>
<td>91</td>
<td>108</td>
</tr>
<tr>
<td>R_6</td>
<td>96</td>
<td>111</td>
<td>R_{14}</td>
<td>109</td>
<td>126</td>
</tr>
<tr>
<td>R_7</td>
<td>112</td>
<td>127</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Table 5 Region boundaries for the bit-selection algorithm

3.4 Aggregation Unit Architecture

In order to fulfill the required distinctiveness of 16-byte prefix, a small amount of patterns are divided into segments. In the system, patterns are assigned IDs from 1 to N, while segments possess IDs from M to a certain value, where M > N. Therefore, if the potential match result has an ID that is smaller than M, it will be sent to the output interface directly. Otherwise, a partial match is found. The aggregation unit (AU)’s job is to keep track of the partial matches to produce the final result.
There are four parts in the partial match results received from either P-AC or QSV module, \textit{pid}, \textit{patLoc}, \textit{refLoc}, \textit{verified}, which are the pattern ID, location of the last byte of the pattern found in the input stream, location of the last byte of the 16-byte prefix, flag indicating whether the pattern has been verified by a CRC unit respectively. The last two tuples are used to handle buffer overflow exception of the CRC pool. The traditional way of constructing the aggregation unit is to establish a mathematical model for the AU using deterministic finite automaton (DFA). However, this method is not available since every segment has different length, resulting in the loss of mutual exclusiveness of the input symbols (i.e. segment IDs) to the FA. Taking an example of the initial state $q_0$ of the transition graph of the aggregation unit, it has around 1800 transition edges. To be specific, for instance, say there are two transition edges $e_1 = (q_0, \text{pid}_1, \text{ns}_1)$ and $e_2 = (q_0, \text{pid}_2, \text{ns}_2)$, in which the three symbols denote the current state, input symbol and next state respectively. In this case, the segments $\text{pid}_1$ and $\text{pid}_2$ may have different lengths, as well as a chance of containing each other as a substring. Supposing that the aggregation unit is in state $q_0$ and receives an input segment $\text{pid}_1$, it then transits to state $\text{ns}_1$. While the FA is waiting for the next segment from state $\text{ns}_1$, another valid segment $\text{pid}_2$ could arrive. To solve this problem, the AU is modelled as a non-deterministic finite automaton (NFA) that allows multiple active states.

The transition edge in the NFA consists of 7 components (current state, segment ID, segment length, next state, bit-mask, TTL, state-type), where the TTL (time-to-live counter) is equal to the maximum segment length of the corresponding next state’s forwarding edges. In addition, the state-type indicates whether the next state is a terminal state, an output state, a combination of the two states or internal state.
Two transition rules lookup tables are constructed, A0 and A1. The former stores all of the transition rules originating from the initial state while the latter stores the others. If the transition rules for the edges originating from the initial state are numbered from $M$ to $M+\delta$, where $\delta$ is the number of edges going out from the initial state, then the address used to index the table $A_0$ is equal to the segment ID $- M$.

The way to access Table $A_1$ is Direct Indexing plus bit-selection (DIBS), same method as $T_2$. The address used to lookup $A_1$ is the base address plus the offset generated by the segment ID and the bit mask. Moreover, segment IDs can be reassigned to optimize the performance of bit selection, resulting in better memory efficiency. In this case, segmented patterns and patterns processed by the P-AC module are numbered in the range from 1 to 16K-1. In addition, by reassigning the patterns and segments IDs, the state ID of the AU transition graph can represent the pattern ID of the corresponding output pattern.

As mentioned before, the AU is essentially a non-deterministic finite automaton (NFA). Hence it keeps a list of active states (AS_list), each of which comprises of a bit-mask, a reference location and an expiry location. The bit mask is used to generate the offset to access the table $A_1$, while the reference and expiry location are the last byte of the segment (patLoc) that triggers the state transition, and the reference location plus the TTL counter respectively. The list of active states (AS_list) can be updated based on two methods. The first one is by referring to the expiry location. For example, if the expiry location is smaller than the location of the last byte of the segment (patLoc), then the corresponding active state will be deleted from the AS_list. The other approach is to utilize the compatibility property of the segments. Supposing that pattern $P_i$ is cut into segments $s_1, s_2$ and so forth, and then segment $s_1$ is considered to be the first segment. If $s_1$ is not a substring (midfix) of any other patterns,
then it’s said to be incompatible. In this case, when AU receives an incompatible segment, all of the other active states can be removed since the expected segments of the current state cannot be found within the required location range. Moreover, the compatibility is decided by the \( h_1 \) hash value, the prefix and the pattern checksums rather than the actual segment values. In the investigated pattern set, 84% of the first segments are found to be incompatible.

The pseudo code for the aggregation process is provided below. [15]

```plaintext
//pseudo code for the aggregation process
//PM = partial match result; AS = active state
for each entry in the AS_list before the arrival of the partialMatch {
    if((PM.verified==1 && PM.patLoc > AS.expiryLoc) || (PM.verified==0 && PM.refLoc > AS.expiryLoc))
        remove the active state from the list;
    else
    { retrieve the transition rule from table A1;
        if(rule.segment_ID==PM.segment_ID &&
            AS.location + rule.segment_len == PM.patLoc)
            if (rule.next_state is an output state)
                report a possible match(rule.next_state,
                PM.patLoc);
            if (rule.next_state is not a terminal state)
                addToList(rule.next_state,
                PM.location,
                PM.patLoc + rule.TTL,
                bit_mask);
    }
}
if (PM.segment_ID is within the range M to M+delta) {
    retrieve the transition rule from table A0;
    if (PM.verified == 1 && rule.compatible == 0)
        remove all current entries in AS_list;
    addToList(rule.next_state,
    PM.location,
    PM.patLoc + rule.TTL,
    bit_mask);
    if (rule.next_state is an output state)
        report a possible match (rule.next_state, PM.patLoc);
}
```
Chapter 4 Implementation detail

4.1 QSV module construction work flow

The QSV system construction has two main parts.

The first part is to collect the statistical information about the virus signatures, segment the long patterns into trunks with appropriate sizes, and perform bit selection & calculate the checksums of each signature. This part was done by Dr. Derek Pao and produced several necessary output files.

The second part takes the output files from the first part as the input files and construct system lookup tables as well as the aggregation unit. The term “program” used later in this report refers to the coding of this part. A brief program flow is as follow.

Figure 10 Brief program flow chart
4.2 Data flow

The program takes 6 files as input and produces 12 output files. A simple data flow chart is as follow.

The followings are explanations to those input and output files.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim_data.txt</td>
<td>sim_data produced from PreProcess-1</td>
</tr>
<tr>
<td>ChainCode.txt</td>
<td>Segmented Patterns with a sequence of their segments</td>
</tr>
<tr>
<td>PAC.txt</td>
<td>Patterns and Segments to be processed by P-AC system</td>
</tr>
<tr>
<td>QSV.txt</td>
<td>Patterns and Segments to be processed by QSV system</td>
</tr>
<tr>
<td>segment_U4.txt</td>
<td>Small segments that are less than four bytes</td>
</tr>
<tr>
<td>Segment_compatibility.txt</td>
<td>Denotes the compatibility of the “1st segments”, used in the aggregation unit.</td>
</tr>
</tbody>
</table>

Table 6 Input files to the program
<table>
<thead>
<tr>
<th>Category</th>
<th>File Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup tables</td>
<td>QSV-T1.txt</td>
<td>QSV Lookup table T1</td>
</tr>
<tr>
<td></td>
<td>QSV-T2.txt</td>
<td>QSV Lookup table T2</td>
</tr>
<tr>
<td></td>
<td>QSV-T3.txt</td>
<td>QSV Lookup table T3</td>
</tr>
<tr>
<td></td>
<td>QSV-A0.txt</td>
<td>Aggregation Unit transition rule table 0</td>
</tr>
<tr>
<td></td>
<td>QSV-A1.txt</td>
<td>Aggregation Unit transition rule table 1</td>
</tr>
<tr>
<td></td>
<td>OldNewIDs.txt</td>
<td>The matching pair of the old and new segment IDs</td>
</tr>
<tr>
<td></td>
<td>sim_data_final.txt</td>
<td>Revised sim_data with new segment IDs and pattern IDs</td>
</tr>
<tr>
<td>Updated datasets</td>
<td>segment_U4_final.txt</td>
<td>Revised short segments with new segment IDs and pattern IDs</td>
</tr>
<tr>
<td></td>
<td>QSV_final.txt</td>
<td>Revised QSV database with new segment IDs and pattern IDs</td>
</tr>
<tr>
<td></td>
<td>ChainCode_final.txt</td>
<td>Revised chaincode with new segment IDs and pattern IDs</td>
</tr>
<tr>
<td></td>
<td>PAC_final.txt</td>
<td>Revised PAC database with new segment IDs and pattern IDs</td>
</tr>
<tr>
<td></td>
<td>Segment_compatibility_final.txt</td>
<td>Revised segment_compatibility with updated Segment ID</td>
</tr>
</tbody>
</table>

Table 7 Output files to the program

4.3 Lookup tables construction

4.3.1 Rearrange the dataset
The input file to the program is “sim_data.txt” in this part, each observation of which contains the following tuples.

<table>
<thead>
<tr>
<th>Tuple Name</th>
<th>Description</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern/segment flag</td>
<td>Flag to indicate whether the record is of a pattern or a segment</td>
<td>Pat</td>
</tr>
<tr>
<td>Pattern/segment id</td>
<td>ID of the pattern/segment</td>
<td>79885</td>
</tr>
<tr>
<td>Pattern/segment length</td>
<td>Length of the pattern/segment</td>
<td>50</td>
</tr>
<tr>
<td>h₁</td>
<td>Value of the h₁ hash</td>
<td>3092</td>
</tr>
<tr>
<td>h₁,h₂</td>
<td>Decimal value of the selected 30 bits</td>
<td>150225451</td>
</tr>
<tr>
<td>Row ID</td>
<td>Number of the current row (or observation)</td>
<td>20</td>
</tr>
<tr>
<td>Prefix Checksum</td>
<td>Checksum of the prefix</td>
<td>b128</td>
</tr>
</tbody>
</table>
## Pattern checksum

<table>
<thead>
<tr>
<th>Pattern checksum</th>
<th>Checksum of the whole pattern</th>
<th>2e9c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partial checksum_1</td>
<td>Checkpoint checksum 1</td>
<td>8998</td>
</tr>
<tr>
<td>Partial checksum_2</td>
<td>Checkpoint checksum 2</td>
<td>4cea</td>
</tr>
<tr>
<td>Pattern value</td>
<td>Hexadecimal display of the pattern</td>
<td>00</td>
</tr>
</tbody>
</table>

**Table 8 Sim_data format**

Figure 12 shows a sample outlook of the sim_data.txt.

![Sample outlook of sim_data.txt](image1)

In this step, the task is to divide all of the patterns into groups of the same $h_1$ value. In this way, bit-selection can be performed on their $h_2$ values in the later step. In order to divide the patterns into groups, the following “pushdown linked list” data structure is created to record the pattern’s information.

The “pushdown linked list” has a horizontal linked list (main list) as well as a number of vertical lists originated from each node of the main list. The main list records different $h_1$ values and the vertical list stores patterns that possess the same $h_1$ value.

Figure 13 depicts the “pushdown linked list” data structure, followed by a C code segment of its data structure declaration.
The operation of this step is straight forward. Whenever the program scans a new pattern, it will check whether the new h1 value has already been encountered before. If so, the program will append the current pattern to the corresponding vertical list. Otherwise, it will create a new node on the horizontal list and a corresponding vertical list along with it, then append the
current pattern to the newly created vertical list. The pseudo code of the operation is provided below.

```plaintext
//pseudo code of the dataset rearrangement
If (!"EOF")
{
    Scan a new pattern and record its file location;
    Scan the h1 value;
    For (every node on the horizontal list && !found)
    {
        If (h1 = current_node.h1)
        {
            found = true;
            append the current pattern to the vertical list;
        }
    }
    If (!found)
    {
        Create a new horizontal node and insert it into the main list;
        Create a vertical list;
        Append the current pattern to the vertical list;
    }
}
```

After this step, one intermediate data file is created called “data_rearrangedByH1.txt”. This file is similar to “sim_data.txt” except that patterns are grouped by their h1 values. Figure 14 is a snapshot of “data_rearrangedByH1.txt”.

![Figure 14 Snapshot of “data_rearrangedByH1.txt”](image)

### 4.3.2 Perform Bit-selection

In order to construct the lookup table T2 using DIBS, we need to do the bit selection on the h2 value for each group of patterns sharing the same h1 value. The bit selection algorithm tries to restrict the number of elements one bucket to be at most 8. For a group of x 15-bit keys
\( K = \{ k_1, k_2, \ldots, k_x \} \), the bit selection algorithm will first compute the bit-count of each bit location. The bit-count is the minimum of two competing values, the number of bits that equals to 1 and number of bits that equals to zero. In every round of bit selection, one bit with the greatest bit count will be selected and each group will be divided into two halves. The bit selection algorithm will terminate if no subgroup has more than 8 patterns. If a group has \( n \) patterns that are more than 8, the \( n-8 \) patterns will be moved from \( \Gamma_{QS V} \) to \( \Gamma_{PAC} \). The following pseudo code illustrates the operation of the bit selection algorithm on the \( h_2 \) value.

```plaintext
//pseudo code of the bit selection algorithm on the h_2 value
While (there is still some group has size larger than 8 && all of the h_2 values in a group are not the same)
{
    Max_bitcount=0;
    //calculate the maximum bitcount from the unselected bits
    For (every bit location)
    {
        If (the current bit is not selected before)
        {
            //calculate the bit count
            Current_bitcount=0;
            For (every pattern in the group)
            {
                //ignore the subgroup that has size smaller than 8
                If (!small_subgroup && bit==1)
                    Current_bitcount++;
            }
            Current_bitcount = minimum of
            (current_bitcount, group_size - current_bitcount);

            If (current_bitcount > max_bitcount)
            {
                Max_count = current_bitcount;
                Bit_loc = current_bitloc;
            }
        }
    }
    //rearrange the subgroups based on the newly selected bit
    Rearrange_groups();
}
Move the excessive patterns to the \( \Gamma_{PAC} \);

After this step, every \( h_1 \) value has its associated bit mask, which can be used later to generate the offset.
```
4.3.3 Construction of the tables $T_1$, $T_2$, $T_3$

Construction of tables $T_1$ and $T_3$ are straightforward, since they are indexed by $h_1$ and $pid$, correspondingly. Certain data structure is used to hold the data temporarily before printing out to the final lookup table files.

```c
//data structure to record the signature information
typedef struct {
    int h2[MASK_LENGTH];        //h2 value
    long h12;                   //t2 value shown in the sim data
    long file_loc;              //information offset
    int bits[MASK_LENGTH];      //1 denotes selected bits
    int weight;                 //weight denotes the number of signatures sharing the
                                 //same h1&h2
    int PAC;                    //flag to indicate whether the signature will be transferred
                                 //to PAC
    int len;                    //pattern length
    int id;                     //pattern id
    int h2_temp;                //incomplete selected bits decimal value
} sig_info;

//data structure to record pattern information
typedef struct {
    int id;                     //pattern id
    int len;                    //pattern length
    char pre_cs[4];             //prefix checksum
    char pat_cs[4];             //pattern checksum
    char par_cs1[4];            //partial checksum1
    char par_cs2[4];            //partial checksum2
} pattern_info;

//data structure to record the table $T_1$ elements
typedef struct {
    char address[MAX_ADDR_DIGIT];  //current memory address
    char mask[HEX_MASK_LENGTH];    //bit mask of this entry
    char base[MAX_ADDR_DIGIT];     //base address of $T_2$ in this $T_1$ entry
    int occupied;                 //flag indicates whether the current address is occupied
} t1_element;

//data structure to record the $T_3$ elements
typedef struct {
    char address[MAX_ADDR_DIGIT];  //current address
    int bytecount;                //pattern length
    char pat_cs[4];               //checksums
    char par_cs1[4];              //checksums
    char par_cs2[4];              //checksums
    int occupied;                 //whether the segment is in QSV now
    int newID;                    //new id assignment for id mapping
    int machine;                  //in QSV or PAC initially before transferring some
                                 //pattern in QSV to PAC
    int aucount;                  //number of appearance in the AU
                                 //used for efficient memory allocation
} t3_element;

//data structure to record the $T_2$ elements
typedef struct
```

30 | Page
Table T₂, on the other hand, is designed using up to 8 hierarchical parallel memory modules as shown in Figure 15, in order to ensure that the lookup operation can be done in 1 clock cycle. Several methods are introduced to improve the memory efficiency of the T₂ table.

Method 1 Least Overhead Allocation (LOA)

Least overhead allocation (LOA) is to try every possible number of parallel memory modules to see which one possess the least overhead. The overhead of an allocation trial is the total memory allocated minus the space occupied in the allocated memory (e.g. group size). Hence, the higher the memory efficiency is, the less the overhead is. The number of parallel memory modules that a group tries should be larger than or equal to the weight of the group.

The definition of weight is the maximum number of patterns that share the same h₂ value in a group. For example, there are 7 patterns in group Gₖ and 2 of them share the same h₂ value. Then the weight of this group will be 2 and the program will try to allocate this group to 3, 4, 8 parallel memory modules and select the one with the least memory overhead, given that there are 1, 3, 4, 8 parallel memory modules available at the same time as shown in Figure 15.

Figure 15 Organization of T₂
The pseudo code of the LOA is provided below.

```plaintext
//pseudo code of the Least Overhead Allocation (LOA) algorithm
Memory_module = 1;
While (memory_module <= 8) {
    //if the weight of this group is smaller than
    //the current number of parallel memory modules
    //then proceed
    If (group.weight < memory_module) {
        //perform bit selection for the current number of
        //parallel memory modules
        //return the number of bits selected
        Bitcount = Bit_selection(memory_module);
        //calculate the memory allocated
        Base = 1;
        For (i=0; i<bitcoutn; i++)
            Base *= 2;
        Allocated_memory = base * memory_module;
        //calculate the overhead
        Overhead = allocated_memory - group_size;
        //compare the overhead
        If (overhead < min_overhead) {
            Optimal_memory = memory_module;
            Min_overhead = overhead;
        }
    }
    //upgrade the memory module
    //1->3, 3->4, 4->8, 8->UNKNOWN (represented by a large integer)
    Memory_module = upgrade_memory(memory_module);
}

LOA is essentially a heuristic approach to explore the optimal memory allocation method.

**Method 2 Leaf Node Omission (LNO)**

LNO is an assisting method for the bit selection algorithm. In some cases, differences in one bit can double the memory cost. Supposing that there are 12 patterns and their $h_2$ values are listed below.

<table>
<thead>
<tr>
<th>Patterns</th>
<th>$h_2$ value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>000100000000000</td>
</tr>
<tr>
<td>$P_2$</td>
<td>001000000000000</td>
</tr>
<tr>
<td>$P_3$</td>
<td>010000000000000</td>
</tr>
</tbody>
</table>
Since the weight of this group is 8, this group of patterns can only be allocated in the 8 parallel memory modules. Hence, if we were to allocate them in an ordinary way, the resultant overhead would be quite large. For example, the first four bits must be selected to differentiate the patterns. In this case, \(2^4 \times 8 = 128\) byte memory is allocated and the overhead is \(128 - 12 = 116\) (91%), which is very large. If we observe the \(h_2\) values of the patterns carefully, we will find out the reason for this large overhead is Patterns 1 to 4. In \(P_1\) to \(P_4\), each of their \(h_2\) value only differs in one distinct bit location so the bit-count for selecting these bits is only one (only one pattern can be differentiated after the bit selection on a bit). This results in a very inefficient memory usage. Hence, the Leaf Node Omission (LNO) algorithm is proposed to deal with this kind of situation.

We can view the bit selection as a tree structure. The LNO algorithm will transfer certain subgroup of patterns of size equal to Gain from \(\Gamma_{QSV}\) to \(\Gamma_{PAC}\) if the Gain is less than some value (e.g. 2). The Gain maximum number of patterns that can be differentiated after selecting one bit. Supposing that there are 30 patterns in the group as shown in the Figure 16 and the weight of this group is 8 (e.g. it’s only able to be allocated in the 8 parallel memory modules), the number in the node of the tree indicates the size of the group or subgroup after each bit selection round. For example, for this group with size 30, after the first bit is selected, it is divided into two subgroups with size 16 and 14 respectively and the Gain of this bit selection is 14. The optimal result of the bit selection is that the gain is the half of the initial group size (even division). In the case of this example, \(\text{Gain}_{\text{optimal}} = 15\). The bit selection performance seems good at the first few bits but encounters obstacles when comes
to the fourth level of the tree. Only one pattern can be differentiated after selecting one bit and the same morass happens three times from the fourth level through the sixth level. The overhead of this bit selection would be $2^6 \times 8 - 30 = 482$ (94%) if no assisting algorithm is adopted. However, if LNO is used instead, pattern a, b, and c in the figure will be transferred to $\Gamma_{PAC}$ and the new overhead will be $2^3 \times 8 - 30 = 34$ (51%). In this way, $(2^6 - 2^3) \times 8 = 448$ bytes of memory are saved.

Figure 16 Sample bit selection tree

Moreover, certain restrictions can be enforced on this algorithm to optimize the trade-off between the memory efficiency and number of pattern transferred to $\Gamma_{PAC}$. For example, we can enlarge the minimum threshold value of Gain so that more patterns will be transferred to $\Gamma_{PAC}$ and higher memory efficiency can be achieved. On the other hand, we also can reduce the threshold to limit the number of patterns transferred to $\Gamma_{PAC}$ and sacrifice some memory efficiency.
The pseudo code of the LNO algorithm is shown below in yellow highlight.

```plaintext
//set the minimum Gain value to be 2 in order to transfer 
//corresponding patterns to PAC using LNO algorithm
Min_gain = 2;
//pseudo code of the bit selection algorithm on the h^2 value
While (there is still some group has size larger than 8
     && all of the h^2 values in a group are not the same)
{
    Max_bitcount=0;
    //calculate the maximum bitcount from the unselected bits
    For (every bit location)
    {
        If (the current bit is not selected before)
        {
            //calculate the bit count
            Current_bitcount=0;
            For (every pattern in the group 
                && !current_patterns.transferred)
            {
                //ignore the subgroup that has size smaller than 8
                If (!small_subgroup && bit==1)
                    Current_bitcout++; 
            }
            Current_bitcount = minimum of
                          (current_bitcount, group_size - current_bitcount);
                    
            If (current_bitcount > max_bitcount)
            {
                Max_count = current_bitcount;
                Bit_loc = current_bitloc;
            }
        }
    }

    //LNO algorithm
    If (max_bitcount < min_gain)
    {
        For (every pattern in the group)
        {
            //assume that the patterns with selected bit equal
            //to one belong to the small group
            If (current_pattern.h2[bit_loc] == 1)
                Current_pattern.transferred = true;
        }
    }

    //rearrange the subgroups based on the newly selected bit
    Rearrange_groups();
}

Move LNO algorithm transferred patterns to H_PAC ;
Move the excessive patterns to the H_PAC ; //patterns with bucket size larger than 8
```

**Method 3 Reduce Bucket Size (RBS)**

RBS is also an assisting algorithm designed to optimize the bit selection performance in order to enhance the memory efficiency of lookup table T_2. The intention behind RBS is to reduce
the bucket size of the current group to release the pressure of high level memory modules. The word Level used here refers to the number of parallel memory modules in the T_2 table. If the number of parallel memory modules is large, the memory level is high. For example, in the current design of the T_2 table, we use 1, 3, 4, 8 piece of parallel memory modules shown in Figure 15. In this case, their memory levels are 1, 3, 4 and 8 respectively. A group of patterns of weight \( w \) must be allocated to a parallel memory module whose level is larger than \( w \). As we will see in the performance evaluation, the higher the memory level is, the less efficient the memory allocation will be. Hence, we can take out certain patterns to reduce the weight of the group, and then this group will have additional choices in memory allocation.

Consider the same example of Table 9 shown along with the LNO algorithm. The weight of this group is 8 and the allocated memory would be \( 2^4 \times 8 = 128 \) byte large if no assisting algorithm is enforced. However, if we transfer patterns \( P_9 \) through \( P_{12} \) to P-AC, then this group will have a weight of 4. This time, the allocated memory would be \( 2^4 \times 4 = 64 \) bytes, which is half of the previous allocation. Moreover, we also can try to further reduce the bucket size to 3 by transferring an additional pattern to \( \Gamma_{PAC} \), say \( P_8 \). In this way, the current weight of this group is 3 and the memory requirement becomes \( 2^4 \times 3 = 48 \) bytes.

There is also a need to keep the balance between the memory efficiency and the number of patterns moved to \( \Gamma_{PAC} \). Since P-AC is concerned with the pattern length, longer patterns need higher computational powers. Hence, when patterns are being transferred to \( \Gamma_{PAC} \), short patterns will be considered first. For example, if patterns \( P_3 \) to \( P_{12} \) have lengths 5, 6, 7… 12 respectively, the probability of them being transferred to \( \Gamma_{PAC} \) is in descending order.
Following is the pseudo code of the RBS algorithm. This code segment tries to mitigate the memory allocation pressure on level 8 parallel memory modules only. However, it also can be revised to optimize other levels of parallel memory modules.

```java
//pseudo code segment for the RBS algorithm
//this code segment tries to mitigate the memory allocation
//pressure on level 8 parallel memory modules only.
If (group.memory_level == 8)
{
    //sort the patterns based on their h2 value
    Sort_patterns(group);
    For (every patterns in the group)
    {
        If (current_pattern.weight == 8)
        {
            //for the current pattern and the subsequent 7 patterns
            //pick up four patterns with minimum lengths
            For (I = 0; I < 8; I++)
            {
                Record the minimum four patterns;
            }
            Mark those patterns as transferred to $\Gamma_{PAC}$.
        }
    }
}
```

### 4.3.4 Construction of tables $A_0$, $A_1$

#### 4.3.4.1 Build the state transition graph

In this step, we shall introduce the way of constructing aggregation unit lookup tables $A_0$ and $A_1$.

First of all, the input file to this part is “ChainCode.txt”, and it contains the segmented pattern ID with a chain of its segments and their associated IDs and lengths. The format of this file is like this.

```
Pattern ID No. of segments: SegID ID len length, SegID ID len length, ...
```

A snapshot of the “ChainCode.txt” is shown below in Figure 17.
As we mentioned in the Chapter 3, the AU acts as a non-deterministic finite automaton (NFA). A sample transition graph is shown below in Figure 18.

**Figure 17 Snapshot of “ChainCode.txt”**

There is one exceptional case that needs to be shown here. Since segments of length less than 4 are ignored by the QSV module, those segments will also be ignored when constructing the AU transition graph. However, the problem emerges when two or more segments are ignored from the same state. For example, consider the following example.

```
pattern 7183 2 : segID 89094 len 21, segID 91142 len 2,
pattern 7184 2 : segID 89094 len 21, segID 94253 len 3,
```

From the initial state $q_0$ after receiving segment 89094, the AU will transit to $q_I$ and wait for the subsequent segments. Since segment 91142 and 94253 are both short segments that are going to be ignored by the QSV, $q_I$ will become an output state. In this case, patterns 7183
and 7184 are possible both match and AU needs to report both patterns to the software for verification. In order to get around this problem, these kind of patterns are artificially assigned consecutive IDs and the AU will just report the lowest IDs of those patterns. The software, in turn, will check multiple IDs starting from the received lowest one. In our study, this exceptional case, though exist, is very rare. Among 2000 segmented patterns, we found out only 4 patterns behaved like this.

Data structures defined for the transition state and edge are shown below.

```c
//data structure for the transition rule
typedef struct
{
    int currState;   //the originating state
    int segID;       //-1 stands for no segment located in this position
    int segLength;   //length of the transition segment
    int pid;         //pattern for transition rule
    int nextState;   //id of the next state
    int bitmask;     //bit mask
    int ttl;         //time to live counter
    int type;        //obsolete
    int output;      //whether output state
    int terminal;    //whether terminal state
    int compatibility;  //compatibility of the first segment
}transition_rule;

//data structure for the transition graph
typedef struct edge
{
    int currStatid;
    int segid;
    int segLength;
    int pid_count;   //a maximum of 10 patterns is assumed in
                     //one output state
    int pid[MAXPID]; //pattern id for output state
    int offset;      //new id offsetted from the base
                     //id indicated in its belonging state
                     //including both state and segment ids
    int compatibility;
    struct edge *nextEdge;  //next edge in the edge list
    struct edge *prevedge;   //previous edge in the edge list
    struct state *nextstate; //the next state associated with this edge
}EDGE;

typedef EDGE *edgeptr;

typedef struct
{
    EDGE head;
    EDGE tail;
}edgelist;

typedef struct state
{
int currStatid;
int edge_count;    //number of edges of a state
int bit_mask;      //number of least significant bits
int base_id;       //base new id of the whole group's segments
int range;         //range = 2*bit_mask
int ttl;           //size of the longest segment
int output;        //whether this state is an output state
int terminal;      //whether it's a terminal state
edgelist edgelist;
}STATE;

typedef STATE *stateptr;

The pseudo code of building the state transition graph of AU is shown below.

//pseudo code of building the state transition graph
//read in the pattern id and the number of segments associated with it
Fin = fopen("ChainCode.txt", "r");
Pattern_id = read_id(fin);
Seg_count = read_seg(fin);

//read in all of the segments
Curr_state = &root;
For (i=0; i<seg_count; i++)
{
    Seg_id = read_seg(fin);
    //check whether the transition state already exist in the graph
    Exist = have_edge(seg_id, curr_state);
    If (!exist)
    {
        //if it’s a new state, allocate the new state
        Edge = malloc(sizeof (EDGE));
        State = malloc(sizeof(STATE));
        Initiate the new edge and the new state;
        Curr_state = state;
    }
    //update the current state's status
    If (there is at least one segment that has length over 4 bytes left)
        Curr_state->terminal = false;
    //update the new state’s status
    Change_state(seg_id, curr_state);
    If (newstate)
    {
        If (there is at least one segment that has length over 4 bytes left)
        {
            Curr_state->terminal = false;
            Curr_state->output = false;
        }
        Else
        {
            Curr_state->terminal = true;
            Curr_state->output = true;
            insert_id(curr_state, patter_id);
        }
    }
    Else
    {
...
If (there is at least one segment that has length over 4 bytes left)
    Curr_state->terminal = false;
Else
    {
        Curr_state->output = true;
        insert_id(curr_state, patter_id);
    }

//assign compatibility bit to the first segments
Read compatibility list from “segment_compatibility.txt”;
Assign compatibility bit if the transition edge is originated from the root state;

4.3.4.2 Reassign segment IDs and generate the bit-mask
As we introduced in Chapter 3, table A₁ is implemented using DIBS. Hence, bit selection is used to generate the bit mask. The bit selection used here is the same as that used in constructing table T₂, except that unlike table T₂, we’re able to manipulate the segment IDs to optimize the memory efficiency. Therefore, we manually reassigned segment IDs to ensure the minimum memory cost. For example, consider the following segments and their corresponding IDs in decimal and binary.

<table>
<thead>
<tr>
<th>Segment</th>
<th>ID (decimal)</th>
<th>ID (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₁</td>
<td>1</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>S₂</td>
<td>2</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>S₄</td>
<td>4</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>S₈</td>
<td>8</td>
<td>1 0 0 0</td>
</tr>
</tbody>
</table>

Table 10 Sample Segment ID in decimal and binary

In the above example, four bits are needed to be selected just for the four segments. In this case, the overhead will be \(2^4 - 4 = 12\) (75%), which is on the high side. However, with the ability to revise the segment IDs, we can optimize the memory efficiency by assigning consecutive segment IDs to those segments originating from one state. For example, IDs of S₁, S₂, S₄ and S₈ can be revised to the followings shown in Table 11.
Table 11 Sample reassigned Segment ID in decimal and binary

<table>
<thead>
<tr>
<th>Segment</th>
<th>ID (decimal)</th>
<th>ID (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₁</td>
<td>4</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>S₂</td>
<td>5</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>S₄</td>
<td>6</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>S₈</td>
<td>7</td>
<td>0 1 1 1</td>
</tr>
</tbody>
</table>

After reassigning the segment IDs, only the least two bits are needed to differentiate the four segments, reducing the memory cost by $2^2 = 4$ times.

The algorithm of reassigning segment IDs is as follows. The whole available segment IDs are considered as a linear pool and every state will try to allocate a block of available segment IDs for its segment from the pool. Since the IDs of the segments will be reassigned consecutively, only several least significant bits will be enough to distinguish the segments.

Since one segment can appear on more than one outgoing edges from different states, it’s possible that during the segment ID reassignment of one state’s outgoing edges, some segments’ IDs have already been reassigned. In this case, the program will consider these segments first by extracting their several least significant bits as needed and marking the corresponding location of the allocated block as unavailable. For example, supposing that there is a segment that has been reassigned ID of 90001 (binary: 10101111110010001) and this very same segment appears in another group of 12 segments of the outgoing edges from a different state. The program will allocate a block of $2^4=16$ available segment IDs starting from M (e.g. M=96000) for the new group. Since segment 90001’s ID has already been reassigned, the program will extract the 4 least significant bits and calculate the offset, which is 1. Then offset 1 will be marked unavailable and will not be reassigned to the other
segments. The intention to keep the offsets unique with each segment in a group is because the sum of offset and the base address together will be used to access table $A_1$ to retrieve transition rules. In addition, since the segment IDs are reassigned in group manner, a group of size less than 3 is no need to undergo bit selection. Note that bit selection is used to select several least significant bits to differentiate a group of segments. If there is only one segment, there is nothing to differentiate with. If there are two segments, due to the distinctiveness of the segment IDs, there is at least one bit that the two segments differ. Hence, there is also no need to use the least significant bits to differentiate the two segments.

Since lookup table $T_3$ is indexed by the pattern and segment IDs, the more concentrated the ID assigned, the less memory it requires. In order to improve the memory efficiency of $T_3$, we can allocate available segment ID block inside the vacant area of other group’s block that has already been occupied. This situation is illustrated in the Figure 19 below.

![Figure 19 Overlapping segment ID reassignment](image)

Supposing that Group 1, 2 and 3 have been allocated their available segment ID blocks in the range of AB, CD and EF respectively. In addition, the shadowing area indicates the occupancy of the block. Before allocating Group 4, we can see clearly that there is still vacant space in Group 1’s block. Hence, instead of allocating blocks after F, the program offers Group 4 a segment ID block inside Group 1 from X to Y. In this way, better memory efficiency is achieved in the table $T_3$. 
The pseudo code of this algorithm is given below.

```c
//pseudo code of the segment ID reassignment and bit selection algorithm
Initiate the bitmap of segment ID pool;
For (every state starting from root)
{
    If (state->edge_count == 1 && ID has not be reassigned)
    {
        Start = Find_space(bitmap, 1);
        Edge->ID = start;
    }
    Else if (state->edge_count == 2)
    {
        If (both of the segments have been reassigned ID)
        {
            Find out the bit difference and build the bit mask;
        }
        Else
        {
            Start = find_space(bitmap, number of segments that have not
            Been reassigned ID);
            Assign ID to the segments from the “Start”; 
            Build the bit mask;
        }
    }
    Else
    {
        Start = find_space(bitmap, state->edge_count);
        //consider the segments that have been reassigned ID first
        For (every segment originating from the current state)
        {
            If (segment ID has already been assigned)
            {
                Extract the least significant bits;
                Mark the corresponding offset on the bitmap as unavailable;
            }
        }
        //reassign IDs to those that have not been reassigned ID before
        For (every segment originating from the current state)
        {
            If (segment ID has NOT already been assigned)
            {
                Start to reassign segment ID from “Start”; 
            }
        }
    }
}
```

4.3.4.3 Reassign pattern IDs

In the last step, we reassigned IDs to the segments in order to maximize the performance of $T_3$. In this part, we also reassign IDs to patterns aiming to further reduce the use of memory for the AU lookup table $A_0$ and $A_1$. In table $A_1$ and $A_0$, every entry stores 9 tuples (address,
segment ID, segment Length, next State, bitmask, TTL, output flag, terminal flag, pattern ID) and the “pattern ID” is the ID of the pattern that was segmented. If the next state is an output state, the “pattern ID” will be the ID of the pattern that AU is going to report a possible match. Otherwise, it will be zero. An interesting idea is that if we set the value of pattern ID as “next state”, the tuple “pattern ID” can be omitted since it’s the same as “next state”. For example, consider the example in Table 12, the next state value is 01c08 (hexadecimal) and the output pattern ID is 60340. After pattern ID reassignment, the new pattern ID is 7176, which is the decimal value of 01c08 and the “pattern ID” column is omitted. In this case, each entry in A0 and A1 saves five bytes.

<table>
<thead>
<tr>
<th>Address</th>
<th>segID</th>
<th>segLength</th>
<th>nextState</th>
<th>Bitmask</th>
<th>TTL</th>
<th>Output</th>
<th>Terminal</th>
<th>patternID</th>
</tr>
</thead>
<tbody>
<tr>
<td>00006</td>
<td>94251</td>
<td>43</td>
<td>01c08</td>
<td>00050</td>
<td>20</td>
<td>1</td>
<td>1</td>
<td>60340</td>
</tr>
<tr>
<td>00006</td>
<td>94251</td>
<td>43</td>
<td>01c08</td>
<td>00050</td>
<td>20</td>
<td>1</td>
<td>1</td>
<td>7176(01c08)</td>
</tr>
</tbody>
</table>

Table 12 Sample pattern ID reassignment

4.3.5 Output of tables T1, T2, T3, A0 and A1
After introducing the steps and algorithms of constructing the lookup tables, following Figure 20-24 display the sample contents of those five lookup tables.

Figure 20 Sample contents of “QSV-T1.txt”
Figure 21 Sample contents of “QSV-T2.txt”

<table>
<thead>
<tr>
<th>address</th>
<th>bytecount</th>
<th>pattern_cs</th>
<th>partial_cs1</th>
<th>partial_cs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>02f63</td>
<td>38</td>
<td>9Gb0</td>
<td>7d65</td>
<td>49a</td>
</tr>
<tr>
<td>02f64</td>
<td>39</td>
<td>bc93</td>
<td>7ac6</td>
<td>3648</td>
</tr>
<tr>
<td>02f65</td>
<td>39</td>
<td>e47d</td>
<td>6ea5</td>
<td>d5e3</td>
</tr>
<tr>
<td>02f66</td>
<td>37</td>
<td>4972</td>
<td>f319</td>
<td>00a3</td>
</tr>
<tr>
<td>02f67</td>
<td>39</td>
<td>a820</td>
<td>b2c8</td>
<td>e9e6</td>
</tr>
<tr>
<td>02f68</td>
<td>37</td>
<td>3ac6</td>
<td>7de9</td>
<td>e6f6</td>
</tr>
<tr>
<td>02f69</td>
<td>39</td>
<td>e061</td>
<td>e915</td>
<td>49d3</td>
</tr>
<tr>
<td>02f6a</td>
<td>38</td>
<td>8e6a</td>
<td>3273</td>
<td>267a</td>
</tr>
<tr>
<td>02f6b</td>
<td>39</td>
<td>8308</td>
<td>a95</td>
<td>1fb8</td>
</tr>
<tr>
<td>02f6c</td>
<td>37</td>
<td>7cc6</td>
<td>9f47</td>
<td>925e</td>
</tr>
<tr>
<td>02f6d</td>
<td>38</td>
<td>50f1</td>
<td>39c5</td>
<td>b7be</td>
</tr>
<tr>
<td>02f6e</td>
<td>38</td>
<td>c7a0</td>
<td>c73</td>
<td>e074</td>
</tr>
<tr>
<td>02f6f</td>
<td>38</td>
<td>4f0a</td>
<td>4a23</td>
<td>C009</td>
</tr>
<tr>
<td>02f70</td>
<td>37</td>
<td>6e1b</td>
<td>2c18</td>
<td>4c13</td>
</tr>
<tr>
<td>02f71</td>
<td>38</td>
<td>37f6</td>
<td>a4bc</td>
<td>74f0</td>
</tr>
<tr>
<td>02f72</td>
<td>39</td>
<td>8571</td>
<td>2ba5</td>
<td>95f1</td>
</tr>
<tr>
<td>02f73</td>
<td>37</td>
<td>d5f2</td>
<td>d3be</td>
<td>ff4a</td>
</tr>
<tr>
<td>02f74</td>
<td>38</td>
<td>5c05</td>
<td>42ec</td>
<td>4d</td>
</tr>
<tr>
<td>02f75</td>
<td>39</td>
<td>6c26</td>
<td>a942</td>
<td>77db</td>
</tr>
<tr>
<td>02f76</td>
<td>38</td>
<td>bd12</td>
<td>641f</td>
<td>4c96</td>
</tr>
<tr>
<td>02f77</td>
<td>37</td>
<td>382</td>
<td>c936</td>
<td>d11c</td>
</tr>
<tr>
<td>02f78</td>
<td>37</td>
<td>b026</td>
<td>b096</td>
<td>1447</td>
</tr>
<tr>
<td>02f79</td>
<td>37</td>
<td>b235</td>
<td>b5a9</td>
<td>f43c</td>
</tr>
<tr>
<td>02f7a</td>
<td>39</td>
<td>1fad</td>
<td>e9e</td>
<td>647c</td>
</tr>
<tr>
<td>02f7b</td>
<td>37</td>
<td>2a38</td>
<td>3b3e</td>
<td>c68f</td>
</tr>
</tbody>
</table>

Figure 22 Sample contents of “QSV-T3.txt”
<table>
<thead>
<tr>
<th>address</th>
<th>segLength</th>
<th>nextState</th>
<th>bitmask</th>
<th>TTL</th>
<th>output</th>
<th>terminal</th>
<th>compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001</td>
<td>24</td>
<td>00002</td>
<td>0001</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00002</td>
<td>18</td>
<td>00004</td>
<td>0001</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00003</td>
<td>21</td>
<td>00006</td>
<td>0001</td>
<td>43</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00004</td>
<td>26</td>
<td>0000c</td>
<td>0001</td>
<td>55</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00005</td>
<td>92</td>
<td>00010</td>
<td>0003</td>
<td>57</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00006</td>
<td>21</td>
<td>0000e</td>
<td>0000</td>
<td>20</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00007</td>
<td>16</td>
<td>00014</td>
<td>0001</td>
<td>29</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00008</td>
<td>18</td>
<td>00016</td>
<td>0001</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00009</td>
<td>33</td>
<td>00018</td>
<td>0001</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000a</td>
<td>24</td>
<td>0000f</td>
<td>0000</td>
<td>41</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000b</td>
<td>23</td>
<td>0001a</td>
<td>0000</td>
<td>40</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000c</td>
<td>16</td>
<td>0001c</td>
<td>0001</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000d</td>
<td>16</td>
<td>0001e</td>
<td>0001</td>
<td>29</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000e</td>
<td>19</td>
<td>00020</td>
<td>0001</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0000f</td>
<td>36</td>
<td>00022</td>
<td>0001</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00010</td>
<td>21</td>
<td>0001b</td>
<td>0000</td>
<td>27</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00011</td>
<td>17</td>
<td>00024</td>
<td>0001</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00012</td>
<td>21</td>
<td>00028</td>
<td>0003</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00013</td>
<td>41</td>
<td>00026</td>
<td>0000</td>
<td>29</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00014</td>
<td>29</td>
<td>0002c</td>
<td>0001</td>
<td>80</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00015</td>
<td>22</td>
<td>00027</td>
<td>0000</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00016</td>
<td>19</td>
<td>0002e</td>
<td>0001</td>
<td>38</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 23 Sample contents of “QSV-A0.txt”

<table>
<thead>
<tr>
<th>address</th>
<th>segID</th>
<th>segLength</th>
<th>nextState</th>
<th>bitmask</th>
<th>TTL</th>
<th>output</th>
<th>terminal</th>
<th>compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>91142</td>
<td>70</td>
<td>01c0a</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00002</td>
<td>94213</td>
<td>15</td>
<td>01c01</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00003</td>
<td>94214</td>
<td>15</td>
<td>01c02</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00004</td>
<td>94345</td>
<td>15</td>
<td>01c03</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00005</td>
<td>94346</td>
<td>28</td>
<td>01c04</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00006</td>
<td>94251</td>
<td>43</td>
<td>01c08</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00007</td>
<td>94252</td>
<td>43</td>
<td>01c09</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00008</td>
<td>92574</td>
<td>6</td>
<td>01c05</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00009</td>
<td>92575</td>
<td>6</td>
<td>01c06</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0000a</td>
<td>92576</td>
<td>6</td>
<td>01c07</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0000b</td>
<td>94329</td>
<td>19</td>
<td>00001</td>
<td>00000</td>
<td>70</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0000c</td>
<td>94330</td>
<td>65</td>
<td>01c0b</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0000d</td>
<td>91142</td>
<td>20</td>
<td>01c0f</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0000e</td>
<td>91142</td>
<td>41</td>
<td>01c16</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>92833</td>
<td>67</td>
<td>01c0c</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>92834</td>
<td>55</td>
<td>01c0d</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00012</td>
<td>92835</td>
<td>66</td>
<td>01c0e</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00013</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>00014</td>
<td>94253</td>
<td>29</td>
<td>01c10</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00015</td>
<td>94254</td>
<td>6</td>
<td>01c11</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00016</td>
<td>94353</td>
<td>27</td>
<td>01c12</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00017</td>
<td>94354</td>
<td>27</td>
<td>01c13</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00018</td>
<td>95719</td>
<td>4</td>
<td>01c14</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>00019</td>
<td>95720</td>
<td>4</td>
<td>01c15</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0002a</td>
<td>91142</td>
<td>40</td>
<td>01c17</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0001b</td>
<td>91142</td>
<td>27</td>
<td>01c20</td>
<td>00000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Figure 24 Sample contents of “QSV-A1.txt”
Chapter 5 Performance Evaluation

The latest ClamAV virus database version 51 is used and evaluated. There are in total 82,888 static strings and 94% of them possess unique prefix. For those patterns share the same prefix, they are divided into 2.44 segments on average. The bit-selection algorithm performs quite well, with over 97% of the 16-byte prefixes are hashed to unique buckets. There are only three buckets have more than 8 entries, resulting in the transfer of 4 patterns to the P-AC unit of length 95 bytes in total. In all, 99.3% of the patterns are handled by the QSV module.

As for the memory costs of lookup tables, several methods and algorithms are introduced to improve the memory efficiency. As mentioned in the previous chapter, methods such as reassigning IDs to patterns and segments, algorithms like LNO, LOA and RBS are successfully implemented and reduce the memory cost around 20% percent. Finally, the memory cost of five lookup tables are shown in Table 13 below.

<table>
<thead>
<tr>
<th>Lookup Table</th>
<th>No. of entries</th>
<th>Memory Cost</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1</td>
<td>30K</td>
<td>32K</td>
<td>93.75%</td>
</tr>
<tr>
<td>T_2</td>
<td>80K</td>
<td>93K</td>
<td>86.19%</td>
</tr>
<tr>
<td>T_3</td>
<td>80K</td>
<td>82K</td>
<td>97.76%</td>
</tr>
<tr>
<td>A_0</td>
<td>1841</td>
<td>2K</td>
<td>89.9%</td>
</tr>
<tr>
<td>A_1</td>
<td>5229</td>
<td>7K</td>
<td>72.9%</td>
</tr>
</tbody>
</table>

Table 13 Memory cost for lookup tables of the QSV module

Since table T_2 is implemented using different levels of parallel memory modules, different levels of memories have different utilizations. Table 14 below lists the memory allocation statistics for table T_2 in detail.
As we can see above, the memory utilization goes up with the decrement of memory level. In fact, level 8 memory should be further optimized to reduce the total memory costs. In addition, we also can view the 8 parallel memory modules RAM0 – 7 separately and Table 15 below shows their sizes.

| RAM 0 | 54K |
| RAM 1 | 16K |
| RAM 2 | 16K |
| RAM 3 | 3K |
| RAM 4 | 1K |
| RAM 5 | 1K |
| RAM 6 | 1K |
| RAM 7 | 1K |

**Table 15 Sizes of 8 parallel memory modules in table T₂**

Different types of data files have been used to simulate and test the performance of our string searching engine. Table 16 below is the summary of the results. [15]

<table>
<thead>
<tr>
<th>data file</th>
<th>size (Mbyte)</th>
<th>P-AC module segment match</th>
<th>P-AC module pattern match</th>
<th>QSV module segment match</th>
<th>QSV module pattern match</th>
<th>no. of CRC units required</th>
<th>max. length of the AS_list</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG picture</td>
<td>2.96</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MP3 music</td>
<td>7.5</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MP4 video</td>
<td>53</td>
<td>0</td>
<td>0</td>
<td>41</td>
<td>24</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>PDF document</td>
<td>1.05</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Word document</td>
<td>3.1</td>
<td>275</td>
<td>0</td>
<td>271</td>
<td>61</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jdk-6u14-nb-6</td>
<td>149</td>
<td>122</td>
<td>0</td>
<td>91</td>
<td>10</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>ubuntu-7.10</td>
<td>695</td>
<td>1998</td>
<td>0</td>
<td>2538</td>
<td>436</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 16 Simulation results of the string searching engine**
When the engine is running on the word documents, the QSV unit locates a prefix match for every 3.3Kbyte of data on average, and a segment match for every 8.6Kbyte of data. On the other hand, when the engine is running on the other type of files, the QSV unit locates a prefix match for every 350Kbyte of data and a segment match for every 2Mbyte of data on average. There are at most 4 CRC units busy at the same time when scanning the Ubuntu-7.10 ISO image file. In addition, there is no more than 1 active state in the AS_list during the simulation time and the aggregation unit is under little pressure.
Conclusion

In this final year project report, a memory efficient string searching engine that deals with static strings is introduced. The QSV method consists of quick sampling of fixed-length strings and on-demand verification of the variable-length suffix segment. The AU is responsible for concatenating partial segment matches to produce final pattern match. There are three lookup tables $T_1$, $T_2$, $T_3$ associated with the prefix sampling (PS) module and two lookup tables $A_0$, $A_1$ linked to AU.

16-byte prefix of patterns is required to be unique in the QSV module. If there are patterns share the same prefix, they will be segmented to meet the requirement. The lookup tables are constructed using the latest ClamAV database main.cvd version 51. For 82888 static strings in the ClamAV dataset, only 1.4Mbyte embedded memory is required.

Several methods are introduced to improve the memory efficiency of lookup tables, such as reassigning IDs to segments for better bit-selection performance, reassigning IDs to patterns to eliminate the entry size of $A_1$ and so forth. Moreover, algorithms are designed to reduce the memory cost, like Least Overhead Allocation (LOA), Leaf Node Omission (LNO), Reduce Buck Size (RBS), etc.

Our future work will focus on two parts. First of all, there are still some memories that are not utilized very well (e.g. level 8 memories). Hence, additional algorithms will be worked out to further optimize the memory cost. In addition, since the current string searching engine supports only static strings, the second job will be investigating ways to accommodate regular expressions at the same time keeping a balance between memory efficiency and search speed.
References


## Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QSV</td>
<td>Quick sampling and verification</td>
</tr>
<tr>
<td>P-AC</td>
<td>Pipelined Aho-Corasick</td>
</tr>
<tr>
<td>DIBS</td>
<td>Direct indexing plus bit selection</td>
</tr>
<tr>
<td>LOA</td>
<td>Least Overhead Allocation</td>
</tr>
<tr>
<td>LNO</td>
<td>Leaf Node Omission</td>
</tr>
<tr>
<td>RBS</td>
<td>Reduce Bucket Size</td>
</tr>
<tr>
<td>AC</td>
<td>Aho-Corasick</td>
</tr>
<tr>
<td>PS</td>
<td>Prefix sampling</td>
</tr>
<tr>
<td>IDS</td>
<td>Intrusion detection system</td>
</tr>
</tbody>
</table>