A Quick String Matching Employing Mixing Up

Tianlong Yang and Hongli Zhang

School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China
coolskydragon@163.com

Abstract

Most of the current string matching algorithms behave slowly when the amount of patterns increases. In this paper a fast matching algorithm named SSEMatch was designed. PHADDW instruction from SSE (Streamed SIMD Extension) set was used in SSEMatch to produce data confusion, by which the patterns can be distributed into pseudo hash address such that there will be less patterns left for verification matching. With the help of PHADDW, the whole matching time was reduced. Our SSEMatch holds a O(n/m) *complexity. Experiment shows that similarly to WM algorithm, SSEMatch performs better when the length of the shortest pattern increases. Also when the amount of patterns increases SSEMatch performs better than WM.*

Keywords: *String matching; SSE; Fast matching*

1. Introduction

String matching can be understood as the problem of finding a pattern with a property within a given sequence of symbols [1]. Its application can be used in many fields, such as bioinformatics and computer science. This paper focuses on a string matching technique for computer security, especially exact multi-pattern matching for intrusion-detection systems (IDS). Although most commonly used algorithms, such as AC [2], AC BM [3] or WM [4] (in this paper we call these algorithms matching automaton), are thought of as a good choice for a network environment application, and these algorithms mature and perform well, their memory usage may become problematic when they are applied to instances of super large patterns set. SSE (Streamed SIMD Extension) is a kind of efficient instructions set for streamed media. Its application has been extended to scientific computation and medical science except processing for video image and audio. However it hasn't been studied widely for string matching. In this paper we will discuss a fast string matching algorithm employing SSE instruction.

The remaining part of this paper is organized as follows: In Section 1, we talk about the characters of SSE instructions set. In Section 2, we talk about the data mixing up method using SSE instruction and its usage in string matching. In Section 3, we give the experiment design and show the experimental results of the proposed algorithm. Section 4 is the conclusion of this paper.

2. Related Work

2.1. The Characters of SSE Instructions Set

The instructions inside SSE instructions set [5] can be roughly classified as data moving instructions, data conversion instructions, logical instructions, arithmetical

instructions, special usage instructions, and string-dealing instructions. After reviewing the Intel's instructions manual we can see that the most frequently updated sub-instructions set is arithmetical instruction. By modification of data type, such as signed, unsigned, single and double, and the modification of combination of different operand position inside the SSE register, SSE instructions set scale has been enlarged from the first version to the latest version. The most interesting instructions in SSE might be horizontal instruction in SSE3 and SSSE3. The original SSE instruction operation such as adding, subtracting and so on, are done between two registers vertically. For example, if two 128 bits SSE registers xmm1 and xmm2 are separated into 4 parts respectively. After a vertical addition, the first 32 bits of the xmm1 will be the result of the first part from xmm1 and xmm2. While in a horizontal addition, the first 32 bits of xmm1 will be the result of the first and the second part of xmm1. These two kinds of additions can be described as following:

Vertical Operation:

xmm1[0-31] = xmm1[0-31]+xmm2[0-31],
xmm1[32-63] = xmm1[32-63]+xmm2[32-63],
xmm1[64-95] = xmm1[64-95]+xmm2[64-95],
xmm1[96-127] = xmm1[96-127]+xmm2[96-127],

Horizontal Operation:

xmm1[0-31] = xmm1[0-31]+xmm1[32-63],
xmm1[32-63] = xmm1[64-95]+xmm1[96-127],
xmm1[64-95] = xmm2[0-31]+xmm2[32-63],
xmm1[96-127] = xmm2[64-95]+xmm2[96-127]

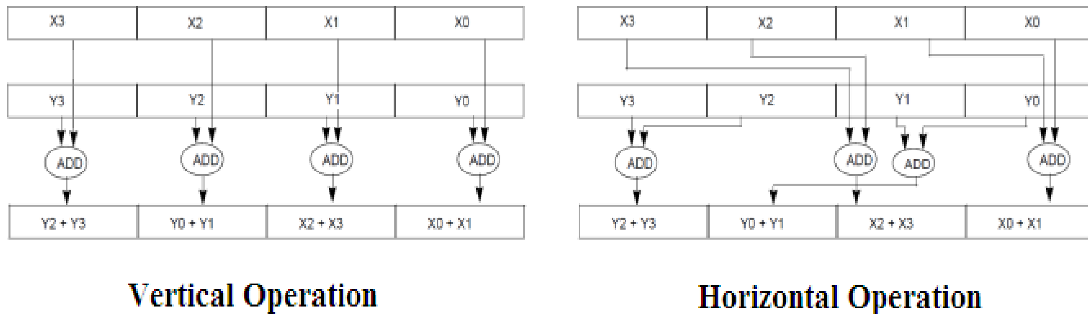


Figure 1. Example of Vertical Operation and Horizontal Operation

From the computation we can see that during a horizontal operation, a 128 bits number can be packed into a 64 bits number. While during a vertical operation the 4 segments of the 128 bits results come directly from relevant segments of xmm1 and xmm2. If xmm2 is set 0, the procedure of horizontal operation can be regarded as a procedure of data compression from xmm1. This special property is utilized to implement a WM-like string matching algorithm in this paper.

SSE instructions set does not only improve the performance of streamed media, but also begins to play more and more important roles in Fourier transform [6-11], generating random number [15], solving system of linear equations [16], parallel FDTD simulation [17] and medical science [18-19].

2.2. The Characters of AC and WM

Exact string matching algorithms can be classified into prefix matching, suffix matching and substring matching, AC (belonging to prefix matching) and WM (belonging to suffix matching) are two kinds of algorithms that have been given special attention. With the help of trie tee and fail transition link AC algorithm can match a text in linear time. However the way of implementing goto transition will influence the matching time. For example, if all possible transition for each symbol in the alphabet are stored into the goto information for a state, it takes only O(1) to retrieve the destination state after a symbol has been read, and the whole matching time will be O(n), where n is the length of the text; if only transition for accepted symbols are stored and binary searching tree is used for storing these goto information, it will take $O(n\log|\sum|)$ [1] to match the whole text. It is obvious that if the matching algorithm demands little memory to store the transition information the matching algorithm matches slowly.

WM utilize a SHIFT table to make the text pointer move as fast as possible so that the text can be matched in high speed. In order to get shift value of the SHIFT table, the minimum length (min_len) of all patterns and all possible continuous characters block BL of length B in the prefix with length min_len of all patterns are concerned [4]. The construction for SHIFT table can be detailed in the following paragraph. And to make the description easily we assume that all patterns are of the same length, *i.e.* min_len is equal to each pattern's length.

If the block BL appears in none of the patterns, the pointer's movement for BL in SHIFT table can be set to $SHIFT(h_1(BL))$=min_len-B+1. If BL appears in some of the p^is in P, the most right position, *j*, which BL appears in these patterns, should be recorded. Then $SHIFT(h_1(BL))$ will be set to min_len-j.

Provided that the shift value $SHIFT(h_1(BL))$ is larger than 0, the pointer can be moved forward without losing any possible matching. Once $SHIFT(h_1(BL))$ is 0, the text should be compared with all possible patterns by another hash function h_2 and table HASH. In order to make the comparison run fast, all the patterns appearing in HASH can be sorted alphabetically. More about the hash table, HASH, can be found in the paper [4].

During the matching procedure, the text pointer's position, pos, is set to min_len, and the block BL of last B characters ending at pos will be checked. If shift=$SHIFT(h_1(BL))>0$, the text pointer will move to pos+shift. Once shift=$SHIFT(h_1(BL))=0$, the first B characters of all patterns appended after the hash table entrance of $HASH(h_2(BL))$ will be compared with the prefix, prefix($t_{pos-min_len+1}t_{pos-min_len+2}\ldots t_{pos-min_len+1+B-1}$). If the first B characters of the substring for current text equals to the prefix of some pattern in $HASH(h_2(BL))$, the text will be compared with all the possible patterns.

3. String Matching Algorithm Based On Horizontal Instruction

We can know that when the amount of patterns increases there will be more patterns for HASH(j), and these patterns might share few prefix($t_0t_1\ldots t_{B-1}$). Then it will take more time to compare each prefix($t_0t_1\ldots t_{B-1}$) of the patterns. This part will introduce another hash-like procedure to make more filtration to the text to avoid impossible matching. The function of hash function is to make the data mixed up well by some complicated actions so that a well-mixed-up value can be achieved. The mixed-up value is called hash value or fingerprint. With the help of hash function, different patterns can

be distributed in to the address space of the hash table. When we want to know whether the substring of a text equals to some pattern, we can get the fingerprint of the substring by hash function. If the patterns' set is not empty in the hash table with the value fingerprint, one-by-one patterns comparison with the substring will be operated. In order to make the matching procedure run quickly, a fast hash function should be used. But even in the fastest hash function, SuperFastHash[20], there are several rounds of addition, shift and xor operations. Such complicated function will not result in good performance. In the following sections we will discuss a simple hash-like function using horizontal operation in SSE instruction set to speed up the matching procedure.

```
typedef struct{
    unsigned char char_set_size;
    unsigned char * char_set;
    pattern_list_leadedby_each_char;
}PATTERN_SUBSET;
```

Algorithm 1 SSEMatch Construct

Input: patterns set
Output: pseudo_hash_table

1. min_len=minimum length of all patterns
2. Let B as 2, i as 0
3. **WHILE** i<sizeof(patenrs set)
4. Fetch pat[i] ∈ patterns set
5. add shift value
6. load pat[0…15] into xmm1
7. PHADDW xmm1,xmm1
8. PHADDW xmm1,xmm1
9. entrance=xmm1[0-31]
10. add pat into pattern_subset of pseudo_hash_table[entrance]
11. i++
12. **End WHILE**

```
struct{
    PATTERN_SUBSET * pattern_subset;
}[256][256][256] pseudo_hash_table;
```

Algorithm 2 SSEMatch Text match

Input: text, pseudo_hash_table
Output: matched info

1. **WHILE** text set is not NULL
2. get one text
3. Fetch shift_table[min_len-2,min_len-1] as shift value
4. **IF** shift value!=0
5. text move forward shift value and continue
6. **End IF**
7. load text [0…15] into xmm1
8. PHADDW xmm1,xmm1
9. PHADDW xmm1,xmm1
10. entrance=xmm1[0-31]
11. compare each pattern in pattern_subset of pseudo_hash_table [entrance] with text
12. **End WHILE**

Figure 2. Data Structure and Algorithm

The horizontal operation PHADDW xmm1, xmm2 in SSE instruction set [21] can be written as following:

xmm1[0-15] = xmm1[0-15]+xmm1[16-31]
xmm1[16-31] = xmm1[32-47]+xmm1[48-63]
xmm1[32-47] = xmm1[64-79]+xmm1[80-95]
xmm1[64-63] = xmm1[96-111]+xmm1[112-127]
xmm1[64-79] = xmm2[0-15]+xmm2[16-31]
xmm1[80-95] = xmm2[32-47]+xmm2[48-63]
xmm1[96-111] = xmm2[64-79]+xmm2[80-95]
xmm1[112-127] = xmm2[96-111]+xmm2[112-127]

In PHADDW, 8 additions for 16 bits integer can be computed in just one instruction cycle. The results of xmm1[0-15], xmm1[16-31], xmm1[32-47] and xmm1[48-63] come from xmm1[0-15], xmm1[16-31], xmm1[32-47], xmm1[48-63], xmm1[64-79], xmm1[80-95], xmm1[96-111] and xmm1[112-127]. The value in xmm1[0-31] comes

only from xmm1, under the circumstance that the PHADDW instruction is operated twice, in the first round set value in xmm2 is the same as that in xmm1, and before the second round nothing in xmm1 and xmm2 is changed. By this way xmm1[0-31] will be a result of 6 additions of 128 bits number. After two rounds of operation, the value of xmm1[0-31] can be regarded as a mixed-up result. Although the mixed-up operation is not strong enough, it really can make the 128 bits number distributed into a 2^{32} space evenly. So our purpose is just to distribute the 128 bits number into a space with length 256×256×256 instead of making more complicated mixed-up value. By this character of PHADDW, we can design a multiple string matching algorithm suitable to patterns which are longer than 16.

3.1. Evenly Distribution of 2 rounds of PHADDW

Let $D=\{e|e\in Z, 0\leq e\leq 2^n-1\}$ represents a set of any n-bits data. From Figure 3(1) it can be seen that: (1) $\forall a,b\in D$, there are $2^n\times 2^n$ ways for computing $(a+b)\%2^n$. (2) $\forall r\in D$, if r is regarded as a computation's result, there are 2^n possible computations which satisfy $(x+y)\%2^n=r$.

Figure 3(2) shows us how a data in xmm register is distributed evenly after horizontal operations. After two rounds of PHADDW, $\forall x\in$ xmm1[112-127], there are 2^n possible computations which satisfy $(a+b)\mathrm{mod}2^n=x$. Also there should be 2^n possible computations which satisfy $(e+f)\mathrm{mod}2^n=a$ and $(g+k)\mathrm{mod}2^n=b$ respectively. After two rounds of PHADDW there are $2^n\times 2^n\times 2^n$ possible computations which satisfy $\{\{[(e+f)\mathrm{mod}2^n]+[(g+k)\mathrm{mod}2^n]\}\ \mathrm{mod}2^n\}=x$. xmm1[112-127] can represen 2^n different data, as a result there are $2^n\times 2^n\times 2n\times 2^n=2^{4n}$ possible computations. For the same reason, $\forall x\in$ xmm1[96-111] there are 2^{4n} possible computations. From Figure 3(1) it can be concluded that $\forall x\in$ xmm1[96-127] there are $2^{4n}\times 2^{4n}=2^{8n}$ possible computations. Let n=16, $2^{8n}=2^{128}$, that means any data from xmm1[96-127] is from the computation results from 128-bits register.

$\forall x\in$ xmm1[112-127][96-111], x can represent $2^{8n}/2^{2n}=2^{6n}$ computations because all possible ways of computations are distributed evenly. In this paper, only the last 24 bits of xmm1[112-127][96-111] is used (operation u32i_v&0x00ffffff in Figure 4). Any data composed of the last 24 bits of xmm1[112-127][96-111] can be decomposed as $2^{8n}/2^{24}=2^{104}$ computations in formula 1, where $\cdot$ means the conjunction of two data.

$$\{\{[(e+f)\mathrm{mod}2^n]+[(g+k)\mathrm{mod}2^n]\}\mathrm{mod}2^n\}\cdot\{\{[(e'+f')\mathrm{mod}2^n]+[(g'+k')\mathrm{mod}2^n]\}\mathrm{mod}2^n\} \quad (1)$$

With the help of (1), we can conclude that u32i_v can express 2^{104} possible data.

In our design there SHIFT and pseudo_hash_table are two major data structures used, and only the first 16 continuous characters of each pattern are used to locate the pattern's address in the pseudo_hash_table. Our design has the same computing complexity of O(n/m) with that of WM, because our SHIFT is the same with that of WM, and our pseudo_hash_table plays the same role as the HASH table does in WM. In order to make the computation more effective, we can retrieve data directly from the pseudo_hash_table with the last 24 bits of the number pseudo_hash_value(32). If non-empty information is found in the table entrance low24(pseudo_hash_value(32)), we can go on to retrieve the last 8 bits of pseudo_hash_value(32) from the link list appended to pseudo_hash_table [low24(pseudo_hash_value(32))]. The filtration procedure is the same with that of WM algorithm, but this is not discussed in this paper. It is necessary to point out that our design is only used for those patterns whose lengths are not shorter than 16.

Row 0: $(0+0)\%2^n = 0$, $(0+1)\%2^n = 1$, $(0+2)\%2^n = 2$, $(0+3)\%2^n = 3$ …… $(0+2^n-3)\%2^n = 2^n-3$, $(0+2^n-2)\%2^n = 2^n-2$, $(0+2^n-1)\%2^n = 2^n-1$

Row 1: $(1+0)\%2^n = 1$, $(1+1)\%2^n = 2$, $(1+2)\%2^n = 3$ …… $(1+2^n-4)\%2^n = 2^n-3$, $(1+2^n-3)\%2^n = 2^n-2$, $(1+2^n-2)\%2^n = 2^n-1$, $(1+2^n-1)\%2^n = 0$

Row 2: $(2+0)\%2^n = 2$, $(2+1)\%2^n = 3$ …… $(2+2^n-5)\%2^n = 2^n-3$, $(2+2^n-4)\%2^n = 2^n-2$, $(2+2^n-3)\%2^n = 2^n-1$, $(1+2^n-2)\%2^n = 0$, $(1+2^n-1)\%2^n = 1$

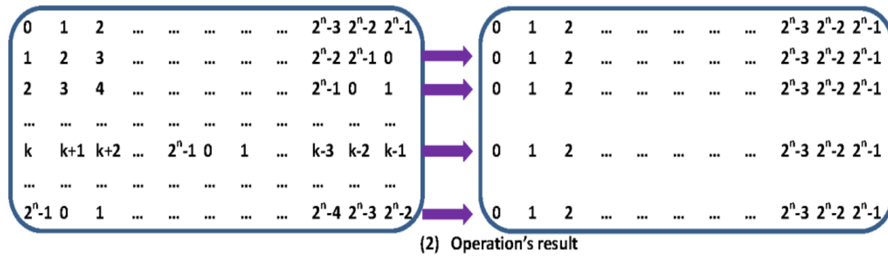
……

Row k: $(k+0)\%2^n = k$, $(k+1)\%2^n = k+1$, $(k+2)\%2^n = k+2$ …… $(k+2^n-k-1)\%2^n = 2^n-1$, $(k+2^n-k)\%2^n = 0$, $(k+2^n-k+1)\%2^n = 1$ …… $(k+2^n-3)\%2^n = k-3$, $(k+2^n-2)\%2^n = k-2$, $(k+2^n-1)\%2^n = k-1$

……

Row 2^n-1: $[(2^n-1)+0]\%2^n = 2^n-1$, $[(2^n-1)+1]\%2^n = 0$, $[(2^n-1)+2]\%2^n = 1$ …… $[(2^n-1)+2^n-3]\%2^n = 2^n-4$, $[(2^n-1)+2^n-2]\%2^n = 2^n-3$, $[(2^n-1)+2^n-1]\%2^n = 2^n-2$

(1) Possible operation's combination



(2) Operation's result

Figure 3. Evenly Distribution for 2n Modulo Addition Operation

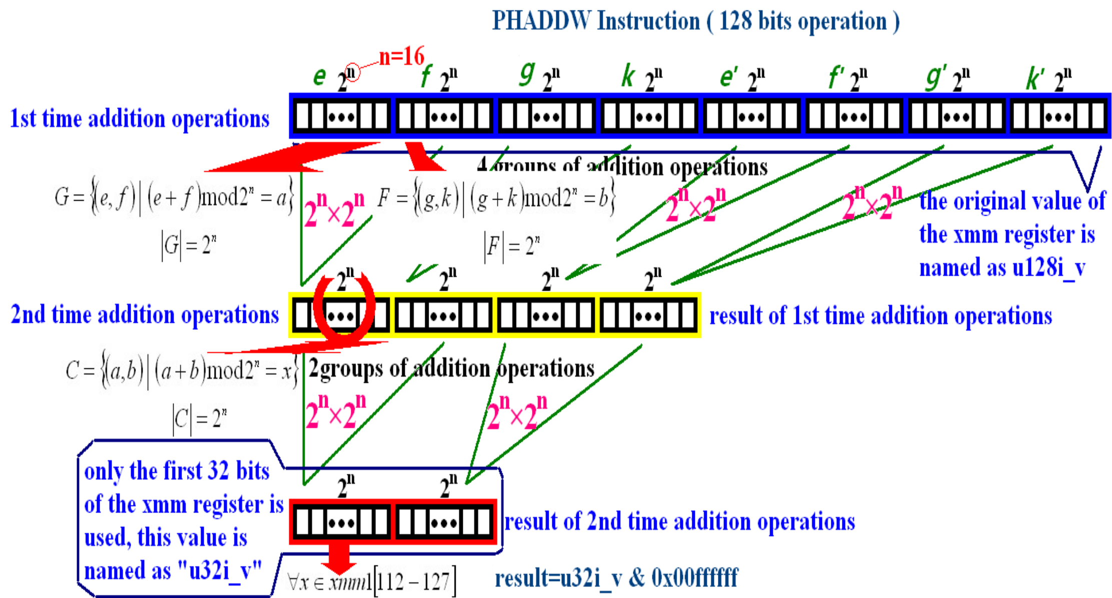


Figure 4. Two Rounds of PHADDW

4. Experimental Results

In this part some experiments considering demand for memory and matching speed are used to test the performance of the newly proposed SSEMatch algorithm. To make the test more impartial, WM and AC are chosen among string matching algorithms

because SSEMatch is similar to WM and AC which run steady during a matching procedure. The code of AC is from snort [22] while the code of WM is from a website [23]. In the website, the implementation of WM only employs one time of shift and xor operation for hash function h_1 and h_2 so that the cost for hash computation can be ignored during the test. We changed from reporting for each appearance of all possible matches in a text to reporting only the first appearance of the match in a text for both AC and WM codes.

The tests were run on a notebook equipped with 2.0 GHz Pentium Intel(R) Core(TM) 2 Duo CPU, 2 GB of memory, 32×2 KiB L1 data cache and 2048 KiB L2 cache. The computer was running Fedora 16 x86_64 Linux with kernel 3.1.0. All programs were written in C (except that the model reading text file was written in C++) and compiled using the optimization level -O3 with the gcc compiler 4.6.2. In the tests, we did not explicitly specify which core was used. All the sets of different amount patterns come from the same set of 5,000,000 URLs. And the text is a network data containing 1,000,000 URLs. Among the 5,000,000 URLs, there are only 4,000,000 patterns whose lengths are not less than 32, so for parameter 32 the patterns amount is up to 4,000,000. And for parameter 48 and 64 the patterns amount is up to 1,000,000 and 200,000 for the same reason.

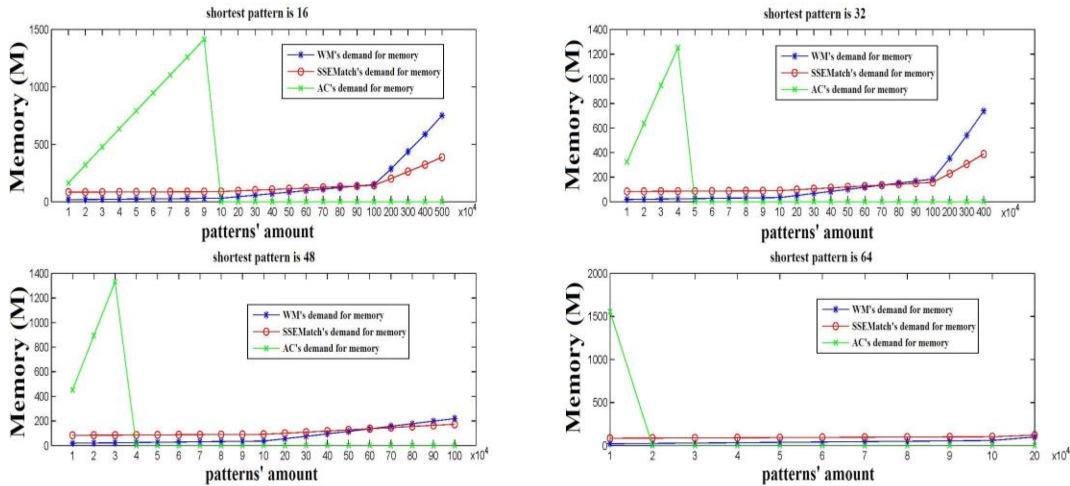


Figure 5. Demand for Memory with Different Patterns' Amount

4.1. The Memory's Demand of SSEMatch

It is shown in Figure 5 that AC demand for memory increases almost linearly while the patterns amount increases. This is due to the data structure used to store trie node. More patterns lead to more trie nodes while no shared prefixes among patterns are found. So AC demand for memory is nearly proportional to the pattern amount. In the test, the experimental data relating to AC demand for memory is not recorded in the result when the pattern amount is more than 90,000. This is because the system stops to respond to user's action when AC demand for memory goes up to 1.5G. Therefore the subsequent result for AC is set to 0, also the following results in this figure and those results in later figures for AC are set to 0 for the same reason. For WM, its demand for memory mainly depends on SHIFT table size and the patterns total bytes. In our test, B=2, so WM only requires 256×256 units to store the SHIFT table; for SSEMatch, a table with 256×256×256 units is employed; as a result SSEMatch demand for memory

is between WM and AC. From this test we can see that SSEMatch demand for memory is nearly steady when the patterns amount increases, and its demand for memory is acceptable.

4.2. The Amount of Patterns Needs to be Verified of SSEMatch and WM

In Figure 6 and Figure 7 the results respectively show the difference of maximal and average number of patterns (pattern) waiting to be verified for table entrance between WM and SSEMatch. For WM a table entrance is any element inside 256×256=65536 units when B=2, the number VerifyCntWM of patterns (pattern) waiting to be verified means how many patterns can be found from this element. For SSEMatch a table entrance is any element inside $256 \times 256 \times 256 = 256^3$ units from pseudo_hash_table, the number VerifyCntSSE of patterns (pattern) waiting to be verified means how many patterns can be found from this element of pseudo_hash_table.

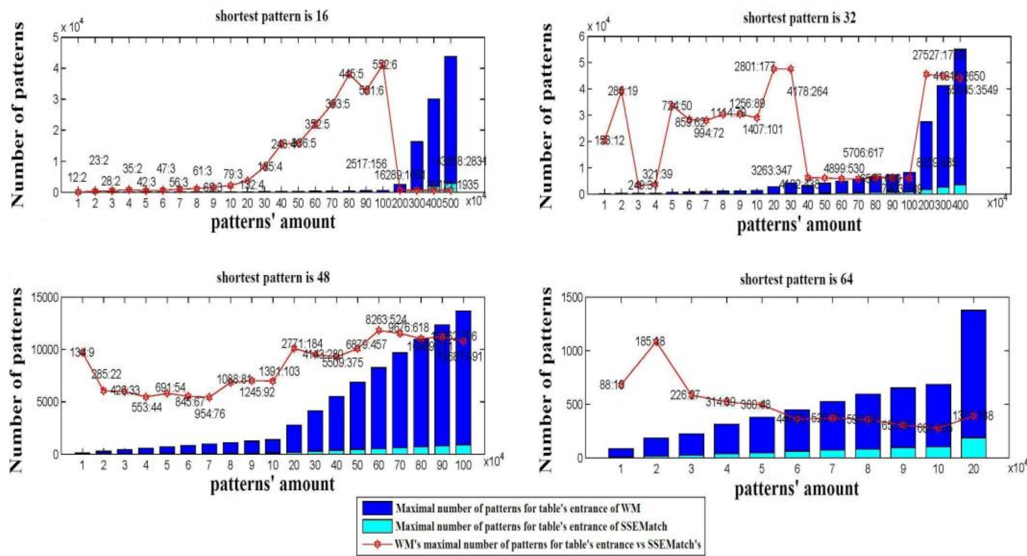


Figure 6. The Amount of Patterns Needs to be Verified of SSEMatch and WM (Maximal Value)

Once the shift value in SHIFT table is 0, the substring of current text should be compared with the patterns belonging to some entrance of the table (table for WM or pseudo_hash_table for SSEMatch). For WM the prefix of the substring needs to be compared with each prefix (prefix_judgement) listed for the table entrance before patterns comparison. For SSEMatch, after shift is 0, the only operation that needs to be done is to use 2 PHADDW instructions to locate current text substring entrance inside the pseudo_hash_table and compare the substring with the patterns belonging to this entrance, this procedure is named as SSE_judgement. When the patterns amount increases, the time for prefix_judgement is longer than SSE_judgement because pseudo_hash_table size is far more larger than hash table size for WM. So it will be a good choice to compare VerifyCntSSE with VerifyCntWM.

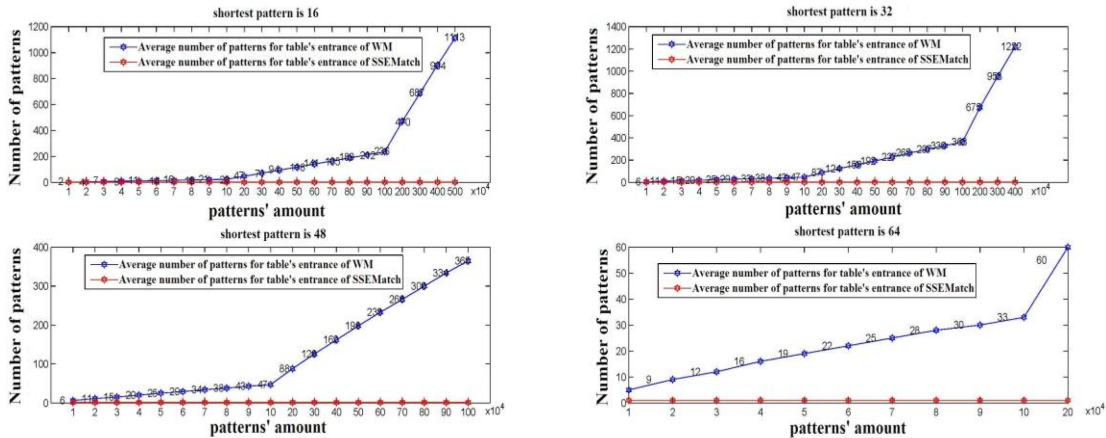


Figure 7. The Amount of Patterns Needs to be Verified of SSEMatch and WM (Average Value)

The results in Figure 6 and Figure 7 show that VerifyCntWM is always larger than VerifyCntSSE, such results are supposed to lead to a higher speed for SSEMatch than WM (results in Figure 9).

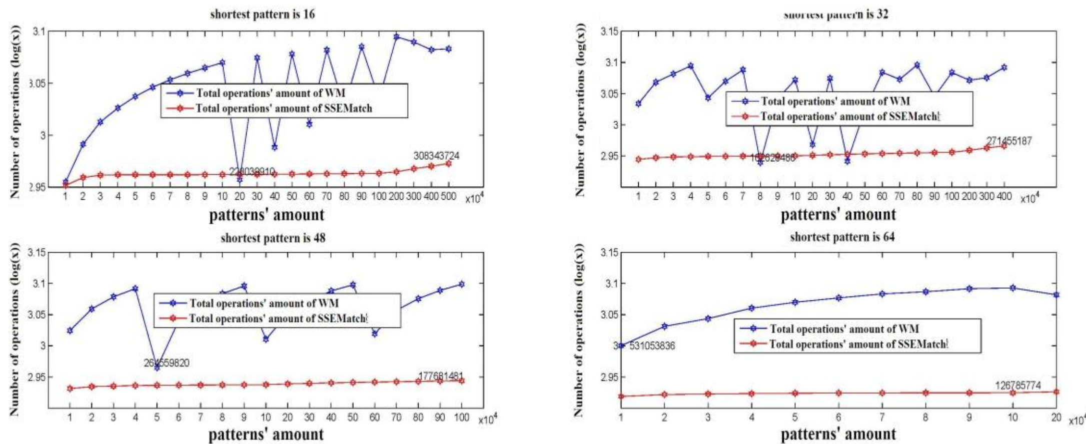


Figure 8. The Amount of Operations During Matching of SSEMatch and WM

For the test in Figure 8 the total number of operations during matching is compared between SSEMatch and WM. In SSEMatch, the operations include: locating the text substring entrance in the pseudo_hash_table by 2 PHADDW instructions noted as tssematch1; judging whether there are some patterns belonging to the located entrance noted as tssematch2; comparing the text substring with each of the patterns belonging to the located entrance noted as tssematch3. As a result the total operations for SSEMatch will be $t_{ssematch}=t_{ssematch}1+ t_{ssematch}2 + t_{ssematch}3$, and $t_{ssematch}1= t_{ssematch}2\times 2$. In WM the operations include: comparing the text substring prefix with each possible prefix belonging to the located entrance noted as $t_{wm}1$; comparing the text substring with each of the patterns belonging to the located entrance noted as $t_{wm}2$. So the total operations for WM will be $t_{wm}=t_{wm}1+t_{wm}2$. It needs to be pointed out that the operations of shift value computation for both WM and SSEMatch and the operation of locating some entrance inside the hash table of WM are not included because these operations only

employ simple shift and xor, they can be ignored compared with other complicated operations while discussing the performance of the two algorithms. The results show that during a matching procedure, SSEMatch needs fewer operations than WM. This is also an evidence to show that SSEMatch will take less time than WM (as shown in Figure 9).

4.3. The Matching Speed of SSEMatch and WM

From the results in Figure 9, it can be seen that the matching speed of all the three algorithms are likely to drop while the patterns amount increases. Compared with WM and SSEMatch, the speed of AC drops fastest because AC's demand for more memory increases more dramatically than the other two algorithms while the patterns amount increases. When the lengths of patterns are not longer than 16, WM runs faster than SSEMatch while the patterns amount is within 80,000. Once the patterns amount is over 80,000 WM falls behind SSEMatch. In the later results in Figure 9 regarding the length of patterns as 32, 48 and 64, SSEMatch surpasses WM in matching speed whatever the patterns amount is. Compared with WM, we can conclude that each factor of SSEMatch (such as max(VerifyCntSSE), ave(VerifyCntWM), tssematch and memory) possibly relating to the matching speed of the algorithm remains more steady when the patterns amount increases. As our conclusion of this part, SSEMatch will be a better choice than WM when the patterns amount is very large or the length of patterns differs from each other very much.

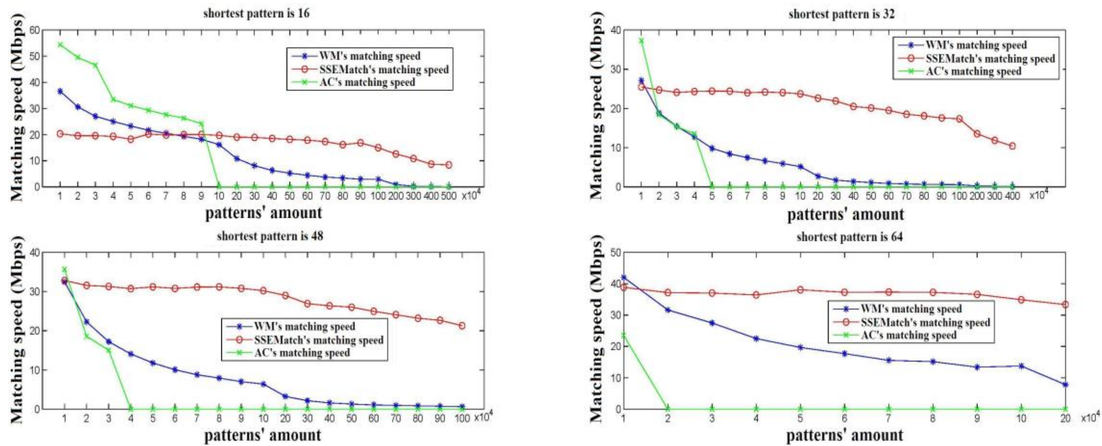


Figure 9. The Matching Speed of SSEMatch and WM

5. Conclusion

In this paper, a new data-mix-up method is proposed to reduce the time to mix data up. This kind of data-mix-up method depends on the PHADDW instruction of horizontal additions in SSE instructions set. With the help of PHADDW a new string matching algorithm SSEMatch is designed. Similar to WM, SSEMatch can match text fast by moving the text pointer fast. Different from WM, our design does not use prefixes to filter out unnecessary pattern verification. Instead our design depends on pseudo hash to filter out impossible matching. To use this kind of pseudo hash, the prefixes with length of 16 of all patterns should be extracted and then packed into a fingerprint of 4 bytes by two PHADDW instructions. Each fingerprint represents an entrance of the pseudo_hash_table. In the later matching, when the text pointer stops

going further a fingerprint for the prefix of the text substring will be computed by the same two PHADDW instructions. It means that the current text substring cannot be a pattern if there is no pattern belonging to the substring entrance in the pseudo_hash_table. So the current substring of the text can be filtered out and the text pointer can be moved forward for later matching. The limitation of SSEMatch is that it can only be used when all the patterns are not shorter than 16. Compared with WM, SSEMatch demands more memory, however the demand for memory can be kept within 200M and 500M. The advantage of SSEMatch is that its demand for memory is between AC and WM, and the matching speed of SSEMatch is faster and more steady than WM when the patterns amount increases. Especially when the patterns amount goes up beyond 20,000, SSEMatch will be a better choice.

Acknowledgements

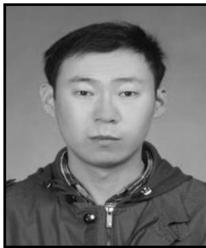
This research was partially supported by the National Basic Research Program of China (973 Program) under grant No. 2011CB302605; the National High Technolgy Research and Development Program of China (863 Program) under grants No. 2011AA010705, No. 2012AA012502 and No. 2012AA012506; the National Key Technology R&D Program of China under grant No. 2012BAH37B01. I really appreciate Ozoemena Ani's help for her work correcting this paper. The authors also gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the presentation.

References

[1] G. Navarro and M. Ra_not, "Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences", Cambridge University Press, **(2002)**.
[2] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search", Communications of the ACM, vol. 18, no. 6, **(1975).**
[3] C. Coit, S. Staniford and J. McAlerney, "Towards faster string matching for intrusion detection or exceeding the speed of snort 1", **(2001)**.
[4] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching", Technical Report TR-94-17, University of Arizona, **(1994)**.
[5] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture.
[6] Z. Qi, W. Feng-dan, D. Ke and W. Zhen-yang, "MMX/SSE/SSE2-based Optimization of the Key Algorithms of H.264 Decoder", Information and Electronic Engineering, vol. 1, **(2006)**.
[7] L. Xiao-hong, "Parallelization of Motion Estimate Algorithm in H.264 Based on SSE", Journal of Hefei University, vol. 3, **(2005)**.
[8] C. Yansong, D. Dagao and D. Zhongliang, "The Analysis of Transform and Quantization in H.264", Modern Transmission, vol. 5, **(2004)**.
[9] D. Da-gao, C. Yan-song and D. Zhong-liang, "Study on Optimization of H.264 Video Encoder", TV Engineering, vol. 3, **(2005)**.
[10] L. Yun-lin and W. Shu-dong, "Conversion of Color Space Between YUV and RGB on SSE2 Technique", Journal of Image and Graphics, vol. 1, **(2010)**.
[11] X. Zhiying, Z. Liping and C. Jing, "Recognition and tracking system design of image based on SSE instructions", Foreign Electronic Measurement Technology, vol. 2, **(2012)**.
[12] T. Yuelin, W. Biao, R. Yong, M. Deling, F. Peng and P. Yingjun, "Application of SSE to real-time frequency spectrum analysis for neutron pulse signal", Nuclear Techniques, vol. 1, **(2009)**.
[13] L. Cheng-jun, Z. Wei-feng and Z. Chong-guang, "Optimal 2D FFT algorithm based on intel SIMD instructions", Computer Engineering and Applications, vol. 43, no. 5, **(2007)**.
[14] Y. Quan, G. Zi-qi, Y. Qian and L. Cai-xia, "Highly Effective FFT Algorithm Based on Parallel Techniques", Science Technology and Engineering, vol. 16, **(2008)**.
[15] Z. Guang and H. Wen-bao, "Construct Random Number Generator Based on SSE2 Instruction", Journal of Information Engineering University, vol. 3, **(2008)**.

[16] D. Yong, F. Ruhuia and H. Tianjian, "SSE-based Linear Equation Parallel Calculation", Computer and Communications, vol. 1, **(2004)**.
[17] Z. Li-hong, Y. Wen-hua and Y. Xiao-ling, "New acceleration technique for parallel FDTD simulation", Chinese Journal of Radio Science, vol. 1, **(2012)**.
[18] S. Qi, L. Zhi-yu and C. Peng, "Evolution for ^{60}Co container CT image reconstruction based on SSE", Nuclear Electronics & Detection Technology, vol. 1, **(2007)**.
[19] L. Ruo-yu, L. Qiang and Z. Shao-qun, "Use MMX and SSE Instruction in Medical Image Processing", Application Research of Computers, vol. 1, **(2005)**.
[20] P. Hsieh, Hash functions (last accessed on Nov. 23rd, 2011) URL http://www.azillionmonkeys.com/qed/hash.html
[21] Intel64 and IA-32 Architectures Software Developers Manual volume2B-instruction set ref-2009.
[22] http://www.snort.org.
[23] http://www.zhiwenweb.cn/Category/Security/1261.htm.

Authors



Tianlong Yang, born in 1980, Ph. D. candidate. Email: coolskydragon@163.com. His main research interest is string matching.

Hongli ZHANG, born in 1973, professor, Ph. D. supervisor. Her research interests include network security and network measurement.