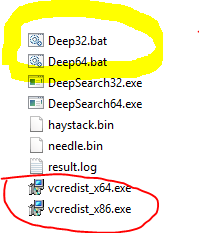
**TEST Administered instructions for the reviewer.**

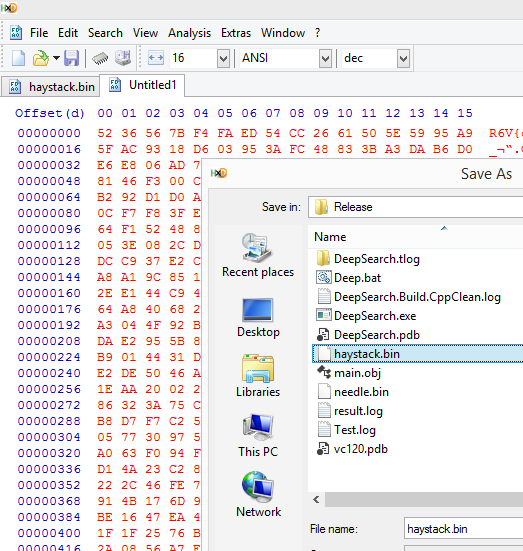
**The test has produced the following files which are delivered together with this document:**

****

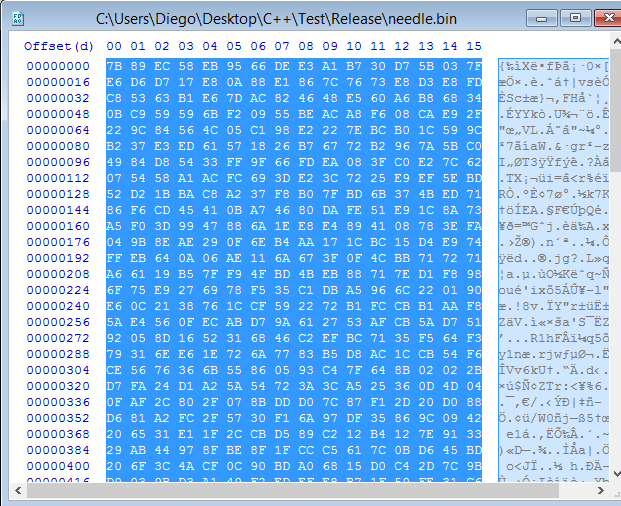
There are two .bat files which allow to run immediately the application either 32 or 64 bit.  
editing the bat file it is possible to change the input parameters which are haystack file name, needle file name and threshold.  
The sample run in windows 8.0/8.1 and it might be necessary to install the corresponding redistributables.  
For example on a64 bit machine it is appropriate to install both redistributable so that both versions can be tested.  
Once the application has finished the result can be seen in the log file.

**Example of creation of haystack.bin and needle.bin.**The Example is based on two files which I have manually created as follows:  
Haystack.bin and needle.bin

The Haystack.bin has random byte values inside:

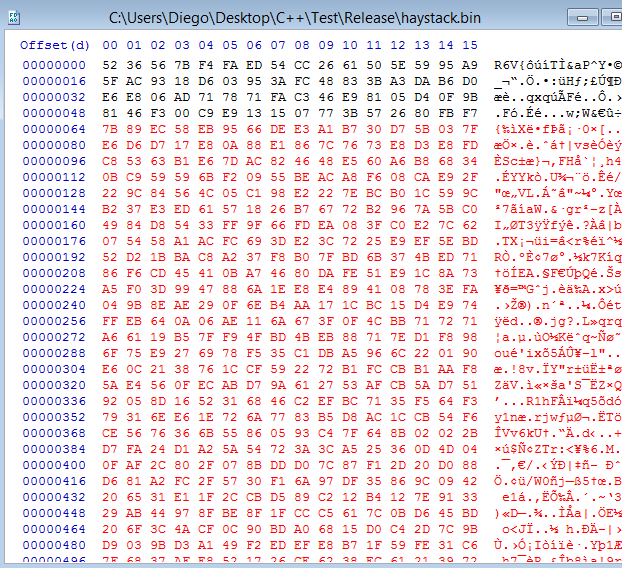
****

**The needle.bin file is a file of 1000 random bytes :**

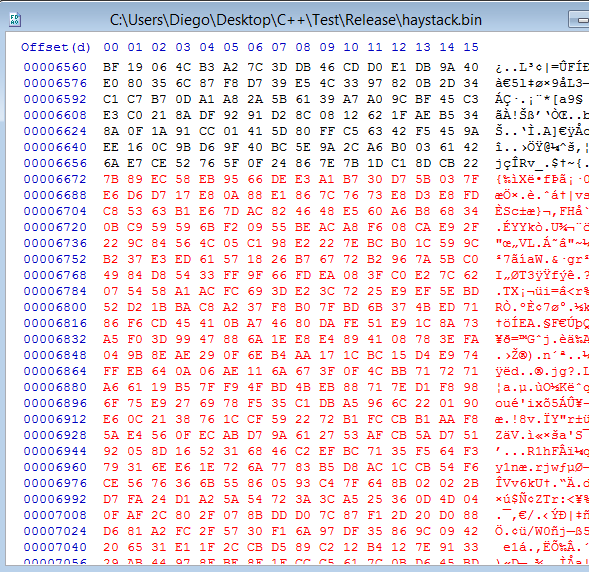
****

**In order to test the application I have written the pattern of 1000 bytes of needle into haystack at different position leaving unaltered the file size (past write) at the following locations:**

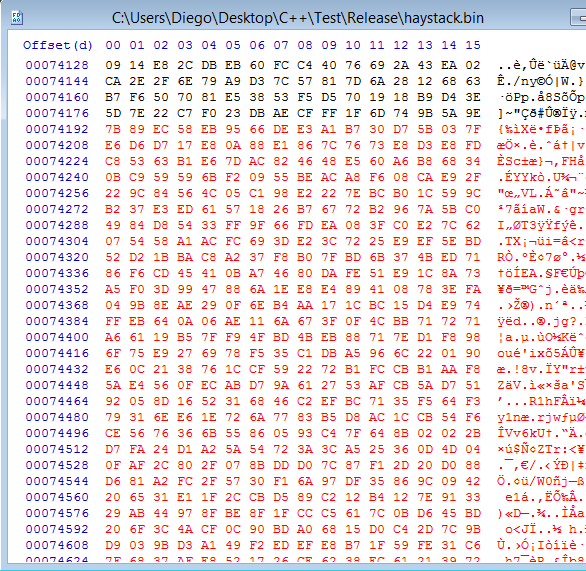
1. **Offset 64 from origin 1000 bytes**

****

1. **Offset 6672 from origin 1000 bytes**

****

1. **Offset 74192 from origin 1000 bytes**

****

Then we can run the tool and after the due time we are prompted to inspect the file result.log which has been created in the same folder where the tool and the data files reside.

The file deep32.bat or deep64.bat contains the command line that fires the tool. As follows:

**deepsearch.exe haystack.bin needle.bin 1000**

The result log content will be:

Reading File......  
Reading File......  
Threshold: 1000  
Offset of needle: 0  
Element of Size 1000 found at position 64  
Element of Size 1000 found at position 6672  
Element of Size 1000 found at position 74192

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Search terminated \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Time elapsed in seconds: 0.0000

If the threshold is modified to a value that is inferior to the length of the needle multiple results will be written in the log file accordingly.

**Implementation of multiple strategies for different trade-offs between speed and memory requirements.**

The Algorithm of Search is implemented with pointers in memory.  
In this way only a limited amount of data can be inspected and the amount itself is limited to the memory installed on the machine.  
A better solution in terms of memory saving could be to create a view of the Haystack file on disk and use memory mapped files although the performance might be inhibited.   
Since I didn’t want to use a specific platform I decided not to implement memory mapped files which requires a different approach on different platforms.

I want to give a simple explanation here of the Mapping in Windows and I want also to indicate that file mapping has its corresponding methods in Linux with different function names and slightly a different procedural approach.

* Windows Approach

HANDLE HN = CreateFile(FileName,GENERIC\_ALL ,0,NULL,OPEN\_EXISTING,FILE\_ATTRIBUTE\_NORMAL,NULL);  
HANDLE HP = CreateFileMapping(HN,NULL,PAGE\_READWRITE,0,0,NULL);   
HANDLE HX =MapViewOfFile(HP,FILE\_MAP\_ALL\_ACCESS,0,0,0);  
**unsigned char \* pt = (unsigned char \*)HX;**DWORD fileSize = GetFileSize(HN, NULL);  
 // Here goes file manipulation for example  
**for(int j =0; j< fileSize; j++)  
pt[j] = pt[j] ^ 100;**// then to deallocate  
UnmapViewOfFile(HX);  
CloseHandle(HP);  
CloseHandle(HN);

**Performance test.**

The tool has been tested with an haystack of 512,000,000 bytes and a needle of 1,000,000 bytes with the following report reading result.log:  
tool compiled 32 bit.  
**Only one occurrence placed in haystack manually using Hxd editor**

Reading File......  
Reading File......  
Threshold: 1000000

Offset of needle: 0  
Element of Size 1000000 found at position 427653376

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Search terminated \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Time elapsed in seconds: 1.9850 sec.

The application has been tested with the same data but with a threshold of (1,000,000 -10). This implies multiples occurrences at different threshold levels. The following results:

Results from result.log

Reading File......  
Reading File......

Threshold: 999990  
Offset of needle: 0  
Element of Size 999990 found at position 427653376  
Offset of needle: 1  
Element of Size 999990 found at position 427653377  
Offset of needle: 2  
Element of Size 999990 found at position 427653378  
Offset of needle: 3  
Element of Size 999990 found at position 427653379  
Offset of needle: 4  
Element of Size 999990 found at position 427653380  
Offset of needle: 5  
Element of Size 999990 found at position 427653381  
Offset of needle: 6  
Element of Size 999990 found at position 427653382  
Offset of needle: 7  
Element of Size 999990 found at position 427653383  
Offset of needle: 8  
Element of Size 999990 found at position 427653384  
Offset of needle: 9  
Element of Size 999990 found at position 427653385  
Offset of needle: 10  
Element of Size 999990 found at position 427653386  
Offset of needle: 0  
Element of Size 999991 found at position 427653376  
Offset of needle: 1  
Element of Size 999991 found at position 427653377  
Offset of needle: 2  
Element of Size 999991 found at position 427653378  
Offset of needle: 3  
Element of Size 999991 found at position 427653379  
Offset of needle: 4  
Element of Size 999991 found at position 427653380  
Offset of needle: 5

Element of Size 999991 found at position 427653381

Offset of needle: 6

Element of Size 999991 found at position 427653382

Offset of needle: 7

Element of Size 999991 found at position 427653383

Offset of needle: 8

Element of Size 999991 found at position 427653384

Offset of needle: 9

Element of Size 999991 found at position 427653385

Offset of needle: 0

Element of Size 999992 found at position 427653376

Offset of needle: 1

Element of Size 999992 found at position 427653377

Offset of needle: 2

Element of Size 999992 found at position 427653378

Offset of needle: 3

Element of Size 999992 found at position 427653379

Offset of needle: 4

Element of Size 999992 found at position 427653380

Offset of needle: 5

Element of Size 999992 found at position 427653381

Offset of needle: 6

Element of Size 999992 found at position 427653382

Offset of needle: 7

Element of Size 999992 found at position 427653383

Offset of needle: 8

Element of Size 999992 found at position 427653384

Offset of needle: 0

Element of Size 999993 found at position 427653376

Offset of needle: 1

Element of Size 999993 found at position 427653377

Offset of needle: 2

Element of Size 999993 found at position 427653378

Offset of needle: 3

Element of Size 999993 found at position 427653379

Offset of needle: 4

Element of Size 999993 found at position 427653380

Offset of needle: 5

Element of Size 999993 found at position 427653381

Offset of needle: 6

Element of Size 999993 found at position 427653382

Offset of needle: 7

Element of Size 999993 found at position 427653383

Offset of needle: 0

Element of Size 999994 found at position 427653376

Offset of needle: 1

Element of Size 999994 found at position 427653377

Offset of needle: 2

Element of Size 999994 found at position 427653378

Offset of needle: 3

Element of Size 999994 found at position 427653379

Offset of needle: 4

Element of Size 999994 found at position 427653380

Offset of needle: 5

Element of Size 999994 found at position 427653381

Offset of needle: 6

Element of Size 999994 found at position 427653382

Offset of needle: 0

Element of Size 999995 found at position 427653376

Offset of needle: 1

Element of Size 999995 found at position 427653377

Offset of needle: 2

Element of Size 999995 found at position 427653378

Offset of needle: 3

Element of Size 999995 found at position 427653379

Offset of needle: 4

Element of Size 999995 found at position 427653380

Offset of needle: 5

Element of Size 999995 found at position 427653381

Offset of needle: 0

Element of Size 999996 found at position 427653376

Offset of needle: 1

Element of Size 999996 found at position 427653377

Offset of needle: 2

Element of Size 999996 found at position 427653378

Offset of needle: 3

Element of Size 999996 found at position 427653379

Offset of needle: 4

Element of Size 999996 found at position 427653380

Offset of needle: 0

Element of Size 999997 found at position 427653376

Offset of needle: 1

Element of Size 999997 found at position 427653377

Offset of needle: 2

Element of Size 999997 found at position 427653378

Offset of needle: 3

Element of Size 999997 found at position 427653379

Offset of needle: 0

Element of Size 999998 found at position 427653376

Offset of needle: 1

Element of Size 999998 found at position 427653377

Offset of needle: 2

Element of Size 999998 found at position 427653378

Offset of needle: 0

Element of Size 999999 found at position 427653376

Offset of needle: 1

Element of Size 999999 found at position 427653377

Offset of needle: 0

Element of Size 1000000 found at position 427653376

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Search terminated \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Time elapsed in seconds: 129.3960

**More on performance:**

It is worth mentioning that in multiprocessor machines like the common laptops the power of calculus is divided among its processors, and every processor manages one thread in an application.  
The tool developed is using only one thread at present and on a machine with 4 processor s only roughly 20% of the cpu will work to search in the haystack.

If required it is possible to use the power of other processors and divide among them the workload.

For example we can fill a queue of possible thresholds using one thread and at the same time inquire the queue for the threshold in order to perform the search in another thread.

**A step further can be to assign to each processor a range of thresholds which is a very performing solution because doing so every thread can work without synchronization.**The cost of a multithreaded choice can be important and it must be devised wisely taking into consideration the target, the amount of data to be searched and the time at disposal.

**Search Algorithm**.

The choice to compare the needle to portions of haystack using **memcmp** crt function is justified by its high speed and by its portability in Linux and Unix.  
It seems that internally the **memcmp** function uses Assembler **Boyer Moore algorithm.**  
The application source code can be compiled either in Windows and Linux because I wanted not to use third part libraries which can prevent portability.

inline void coreSearchTurbo(byte\* haystack, int lenHaystack, byte \* elementTosearch, int sizeOfelement, FILE \* logger)

{

int limit = (lenHaystack - sizeOfelement);

for (int q = 0; q <= limit; q++) {

// we check the first and last element of the needle. It should be beneficial for long needles.(..To test)

if ((haystack[q] == elementTosearch[0]) && (haystack[q + sizeOfelement - 1] == elementTosearch[sizeOfelement - 1]))

{

if (memcmp(elementTosearch, &haystack[q], sizeOfelement) == 0)

{

fprintf(logger, "Element of Size %d found at position %d\n", sizeOfelement, q);

q += sizeOfelement;

}

}

}

}  
  
**Further improvements can be added to the algorithm using inspiration on sources like the following link although ,for reasons of time I wanted to create my own solution from scratch and I didn’t copy any pattern on the web.**

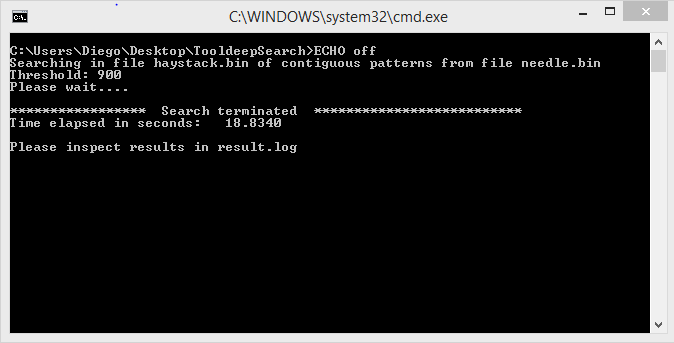
<https://github.com/FooBarWidget/boyer-moore-horspool/blob/master/Horspool.cpp>

**Test on 32bit and 64bit Architecture**

Using the same machine it has been noted a conspicuous difference in terms of performance using the 32 bit version of the tool or the 64 version of the tool.

The result.log for both 32 and 64 implementations gives the following results:

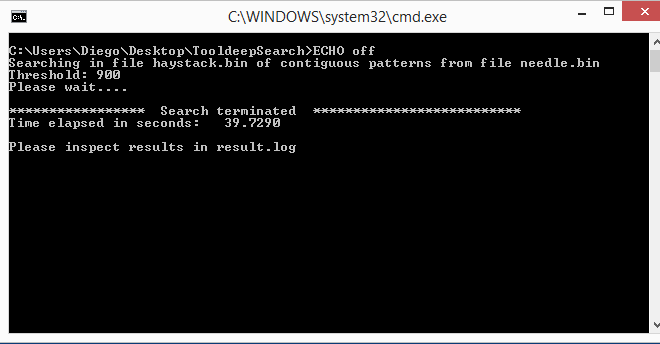
**64 bit version of the tool:**



And its corresponding usage in terms of resources



**32 bit version of the tool:**



And its corresponding usage of resources



**Using the tool to benchmark Amazon AWS Micro Instances:**

The following screenshot demonstrates that the application has been tested on an Amazon AWS Micro instance with architecture 64 bit.

