Applications of CRCs to TMTO

Author: Jason R. Davis
Site: TMTO[dot]ORG
Date: September 28th, 2005

Table of Contents:

Introduction

CRCs or Cyclical Redundancy Checks are used to verify data packets in transmission protocols such as, TCP/IP and many others. This type of math can be applied to TMTO as a space requirement reduction technique.

One of the largest restraints TMTO hash databases have is their sheer size. Usually reaching multi-gigabyte levels, they fill up huge volumes with no second thoughts. In order to make TMTO more feasible for everyone to develop, this large space requirement must be tamed.

Size Calculations

An MD5 hash results in 16 bytes of data, and the clear text counterpart takes one half its length in bytes. So, if I want to store the hashes and text for the numerical values of 0-9,999 (11,110 possibilities). The result can be calculated as follows:

Character Set: 0123456789
Length: 1-4

Possibilities: (10^1)+(10^2)+(10^3)+(10^4) = 11,110

Hashes: (16*11,110) = 177,760
Text: (.5*(10^1))+(1*(10^2))+(1.5*(10^3))+(2*(10^4)) = 21,605

Total: (177,760 + 21,605) = 199,365 bytes

Now that's 194kb, and is nearly 10,000 possibilities. Each row is approximately 18 bytes (199365/11,110). At that rate, it is not feasible to produce a 1-8 length alpha-numeric TMTO hash database.

Possibilities:

(62^1)+(62^2)+(62^3)+(62^4)+(62^5)+(62^6)+(62^7)+(62^8) = 221,919,451,578,090

Function CRC()

This is why a mathematical function, similar to that used to generate CRCs, had to be created. The function is as follows (In PHP):

```
function crc($hash) {
   if(strlen($hash) == 32) {
      $block[0] = hexdec(substr($hash,0,2));
      $block[1] = hexdec(substr($hash,2,2));
      $block[2] = hexdec(substr($hash,4,2));
      $block[3] = hexdec(substr($hash,6,2));
   return dechex(floor((($block[0]*256*256*256)+($block[1]*256*256)+ ($block[2]*256)+($block[3]))/2));
 }
}
```

The function has been critiqued for MD5, but can be easily adapted to other algorithms. The function is designed to return a hexadecimal value between 00000000-7FFFFFFF for all MD5 hashes.

For those privileged to programming, the hash is broken into 4 blocks. $block[0] being the first byte, $block[1] is the second byte, $block[2] is the thrid byte, $block[3] is the fourth byte, and the rest are disregarded.

The MD5 hash of "Hello World": "b10a8db164e0754105b7a99be72e3fe5", is run through the function. The blocks are created:

$block[0] = b1
$block[1] = 0a
$block[2] = 8d
$block[3] = b1

And translated from hexadecimal to decimal.

$block[0] = 177
$block[1] = 10
$block[2] = 141
$block[3] = 177

The mathematics are then applied to the blocks.

((177*256*256*256)+(10*256*256)+(141*256)+(177))/2 = 1485129432.5

The returned result is rounded down to the nearest whole number.

floor(1485129432.5) = 1485129432

And Finally, the resulting decimal number is converted to hexadecimal and returned.

588546CF

As you can see, the result is 4 bytes long, and is a 75% reduction compared to its MD5 equivalent.

Collision Compensation through Grouping

If there are 2^128 possible MD5 hashes, but only 2^32 possibilities for results, will there be collisions? The answer is yes, but to defeat collisions you can group them, and just check each text that resides at that collision.

If ten hashes all result in "588546CF", those ten hashes are grouped together. When searching for "588546CF", all ten hashes can be recalled and checked with minimal loss of time. Hence the paragraph title: Collision Compensation through Grouping. This will result in a table similar to the following:

Note: These numbers/text are hypothetical and being used for example, they are not exact.

0x588546CF : b10a8db164e0754105b7a99be72e3fe5
0x588546D0 : f0a3dce38fa16c2670af0ca54f690d9b 10ed339a9c5721d6ead40ac469f871b
0x588546D1 : b795c02a97323bf9aa4086bc016e4a40

As you can see at 0x588546D0, there are two MD5 hashes. They both resulted in the same value, and therefore were grouped. In a TMTO hash database, you would store the CRC value, and the clear text to make a table similar to the following:

Note: These numbers/text are hypothetical and being used for example, they are not exact.

0x588546CF : hello
0x588546D0 : swiss 199
0x588546D1 : w0rld

So if I submit the hash, "b10a8db164e0754105b7a99be72e3fe5", after being run through the CRC function it would return, "588546CF". I would then search the table for that value and return the text found at that row. In this instance it would be "hello". That is how CRCs can be used as a space reduction technique.

When comparing a non-CRC table to a CRC table the space difference is visibly noticed.

Non-CRC Table
B10a8db164e0754105b7a99be72e3fe5 : hello
f0a3dce38fa16c2670af0ca54f690d9b : swiss
f10ed339a9c5721d6ead40ac469f871b : 199
b795c02a97323bf9aa4086bc016e4a40 : w0rld

CRC Table
0x588546CF : hello
0x588546D0 : swiss 199
0x588546D1 : w0rld

And so, in this situation, collisions actually help in reducing the size of the table by allowing multiple values per row, as opposed to one value per row.

Conclusion

The latest sources @ http://www.md5lookup.com, v0.20b, utilize this technique for building an MD5 based TMTO hash database. This technique is not copyright, and should be distributed for use in the public domain. This is a stepping stone, and now TMTO hash databases can begin comparing to Rainbow Tables in the factor of overall size without the statistical in-accuracy.

Jason R. Davis
TMTO[dot]ORG