Address Access Incrementing Scheme

Diego Jurado

May 2, 2024

1 Introduction

The motivation behind the investigation is determining whether the following scheme for memory address access incrementing works effectively. Bytes 0-3 are dedicated to a 32-bit random integer or salt, and Bytes 4-127 are dedicated to a compressed string that when uncompressed creates an array of 128 address access counters. The purpose of the investigation was to determine whether a 10-bit, 12-bit, 14-bit or 16-bit data type serves as the best counter. However, as the investigation went on I determined the effort it would take to implement the 10-bit, 12-bit, and 14-bit counters would be wasted.

2 Methods

The generation of the secure string involves a predetermined random integer to be stored in the first 4 bytes of the 128B field, and the following 124B are allocated for a compressed string. I chose to compress using LZW compression as it is a lossless scheme that requires no additional data structure like Huffman encoding.

Algorithm 1: Generation of Secure 128B Field

Output: secureStr

unsigned int salt ← rand();

unsigned char arr[128];

compressedStr ← lzwCompress(arr);

secureStr ← fixedSizeOutputHash(salt + compressedStr);

return secureStr;

However, while these are the methods to be used in implementation the methods used for the simulation are reduced for the sake of testing. I remove the use and implementation of the random salt, and the reversible fixed size output hash function.

By simply using a random number generator to determine the address to increment, then compressing the array to determine the size of it, I was able to come to the following conclusion.

3 Results

The 1B data type permits more accesses than the 2B data type. Logically, this is a consequence of LZW-compression, where the quantity of 0s in the 2B data type far outnumbers the 1B one and

Algorithm 2: Memory Access Incrementation

Input: secureStr, address
Output: secureStr

combinedStr ← decodeHash(secureStr);
salt ← combinedStr[0:3];
compressedStr ← combinedStr[4:127];

unsigned char arr[128] ← uncompress(compressedStr);
arr[address]++;

compressedStr ← lzwCompress(arr);
secureStr ← fixedSizeOutputHash(salt + compressedStr);

return secureStr;

Table 1: Iterations and Accesses on 124B Max Compression

	1B (unsigned char)	**2B (unsigned short)**
Avg Increments per iteration	15008	825
Avg Median accesses	117	6
Avg Max accesses	147	11
Avg Min accesses	88	0
Avg Mean accesses	116	5

negatively impacts the compression size. By artificially inflating the compression size less accesses are possible before compressing too large.

4 Discussion

After 250 iterations, the following results have made it clear that given a maximum compressed size of 124B, the best strategy is to use a 1B unsigned char to represent the counter. The average maximum value found in 250 iterations was 147, which exceeds the 7-bit maximum but falls short of the 8-bit or 1B maximum. Therefore, any size smaller or larger than 8-bit or 1B would be suboptimal. Thus, for a maximum compression size of 124B, a data type of 1B is optimal.

However, testing the ratio of maximum compression size to access-counter data size, I found that a ratio of around 124:1 works effectively. For instance, I tried 2B with a size of $128B \times 2 = 256B \Rightarrow 256B - 4B(\text{random integer}) = 252B \Rightarrow 252B/2B \Rightarrow 252 : 2$, roughly equal to a 124:1 ratio. This resulted in 4974664 accesses before growing too large.

5 Conclusion

In short, a more sophisticated compression scheme might be beneficial in implementation, as well as considering a larger maximum compression size. Both of these would allow for easier implementation using a larger than 1B data type. Otherwise, the best performance to be expected is a 1B data type that gets an average of 15000 accesses before compressing to a size too large.