Report on Rainbow Tables

Fu Tianwen

April 26, 2019

1 Introduction

1.1 Background

It is quite common nowadays to provide passwords to websites. When the database of website become compromised, attackers may achieve user's password and use it to crack the user's account at other websites (if the user has a bad habit of using the same password everywhere).

Therefore, most websites use cryptographic hash functions to validate passwords. For every input, such function gives a deterministic fixed-size hash and it is mostly infeasible to retrieve the original value from the message digest[1]. This approach can validate whether the user provided the correct password without the vulnerability to leak the user's original password.

However, such security belief depends on the invertibility of cryptographic hash functions. Once it becomes easy to create a collision of the hash digest, attackers can pretend to be the legitimate user. Rainbow tables provide a way to crack the hash functions.

1.2 Previous Work

Before data structures similar to rainbow tables were invented, there had been two brute-force ways to crack the password. One is to try all possibilities until one gets the same hash. This method is overwhelmingly time-consuming and requires recomputation each time cracking another password. The other is to pre-compute all the hashes for a rather large set of input, then create a dictionary-like data structure (such as hashtables or trie trees) to look up the plain text in $O(1)$ time. The dictionary can also be downloaded and shared to avoid the long pre-computation time. However, such dictionaries occupies a huge space and has difficulties to share, store and reuse.

To solve such problems, time-memory tradeoff techniques were introduced and varieties of optimizations were published, whose ideas and implementations will be discussed in the following sections. With these techniques, my implementation could generate and solve a 6-letter alphabetic string with only 6MB text file[1] of table in less than 10 minutes as measured on my Intel i5-6300HQ computer. It is believed with further code-level optimization[2] and GPU usage there is still much improvement space for efficiency, which will also be discussed in following sections.

2 The Original Time-Memory Tradeoff Method

2.1 Blocks and Chains

A similar idea to this method is making blocks and chains to maintain a sorted linked list. Suppose there is sorted list and one has to insert and delete a few elements. Also suppose that you do not want to write a complicated data structure such as a balanced tree or a probabilistic data structure such as a skip list. Also you do not want to suffer from $O(n)$ each time.

With blocks and chains, one can easily make the time complexity $O(\sqrt{n})$. Consider separate the original data into $\sqrt{n}$ blocks and each block consists of a $\sqrt{n}$-long chain. Rebuild the whole structure when the blocks becomes too imbalanced. This can allow insertion and deletion with $O(\sqrt{n})$ time to find the correct block and $O(\sqrt{n})$ time to operate within the block.

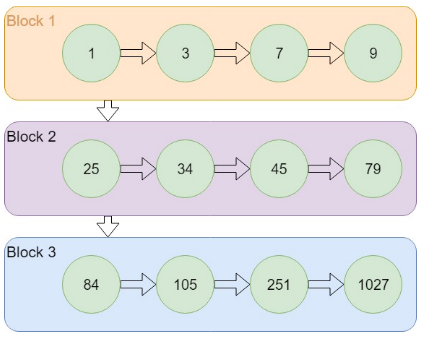

Figure 1: Blocks and Chains Linked List

[1]File size could be further reduced by using a binary file.

[2]My implementation involves a great number of function pointers and structures to provide clear framework and generality; if one can restrict the specific usage the performance can be greatly improved.

2.2 The Hash-Reduce Method

Just as the idea above, Hellman[2] proposed a kind of chain that can help make a tradeoff between time and memory. To achieve this goal, we first define the hash by

$$H : D \to G, C_0 = H(P_0)$$

where D is the domain of plain text of interest and G is the domain of all possible cipher digests of the hash function.

What we want to do is to build a chain of successive plain texts and hashes. Therefore we introduce a reduction function

$$R : G \to D, P_1 = H(C_0)$$

The function R reverses the domain and codomain of H, but it needs not be the inverse of H; otherwise we don't need all these data structures. It can be any arbitrary function[3] that gives a deterministic valid string output in D for each hashed value so that we can hash and reuse the output.

Once we have these functions, we can then pick some random strings and use them to start building a structure with n chains, each chain alternating functions H and R with total m hashes, as shown in the following graph:

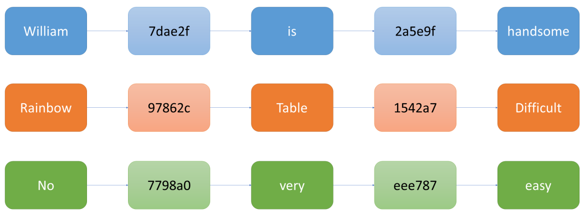

Figure 2: Hash and Reduce Chain

Notice that with the start of each chain we can rebuild the whole chain; also to check whether a hash is in a chain, we can simply apply the reduction function, and then hash and reduce until we find the result same as the end of some chain. Therefore, we only need to store the start and end of each chain, so the space complexity is only $O(n)$. Although the time complexity of generating a table is $O(nm)$, the algorithm suits parallelism well and therefore the process can be done in $O(m)$. Also one may simple download the table from the Internet, which is only $O(n)$ in space, and the time complexity of cracking is $O(m)$.

The implementation of this original method will not be given here since the optimized and more powerful implementation will be given in the following sections.

2.3 Details and Drawbacks

2.3.1 Merges

It may be implied that in the best case the chains can provide the plain text of $O(nm)$ hashes with $O(n)$ space and $O(m)$ time. However, merges can happen and greatly impact the efficiency of this algorithm. Consider the following table:

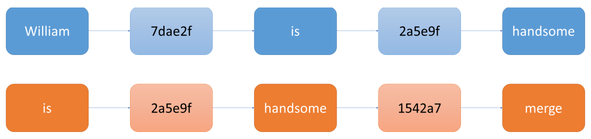

Figure 3: Merge

The two chains are identical starting from the first occurrence of the same text. This may happen at any place, making it almost infeasible to prevent it; when n and m become larger in proportion to the total possible plain text N, merging also become more frequent and waste a lot of computational resources.

2.3.2 False Alarms

Another problem that may occur during the process of cracking is the raise of false alarms. When the hashed and reduced result of some hash matches the end of some chain, other than having a success, it may also be possible that we have a collision in reduction functions, as shown in Figure 4:

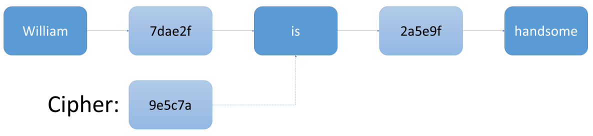

Figure 4: False Alarm

3 The Rainbow Table

3.1 The Algorithm

To cope with the decay of success rate when n, m become large and to reduce false alarms, Oechslin[3] proposed a powerful and easy optimization for the method discussed above. Instead of using only one reduction function, it takes a series of reduction functions

$$R_k : G \to D, k = 0, 1, \cdots, m$$

as shown in Figure 5:

[3]In fact, function R does have to satisfy some requirements to guarantee the performance and probability of success; this will be discussed in the following sections.

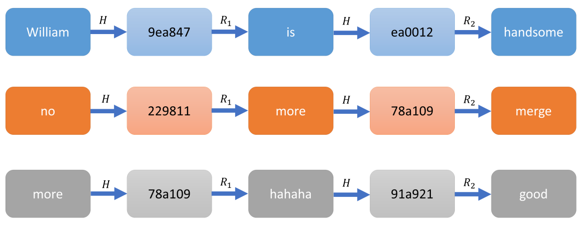

Figure 5: Illustration of Rainbow Table

3.2 Analysis

3.2.1 Reduction Functions

In Figure 5, note that there are two occurrences of *more* but they do not end up in identical chains because they were applied to different reduction functions.

Therefore, to minimize the probability of merging, the series of reduction functions need to give completely different outputs for the same input; the reduction functions also need to cover most of the possible plain text with almost uniform probability so that the rainbow table can contain as many plain texts as possible.

3.2.2 Startings of the List

The starting of each list mostly comes from independently random strings. The random function needs to generate uniformly equally likely strings in domain D so that we can cover most texts and avoid merging.

4 Implementation

To keep our focus on the algorithm, we assume the input are all hashes from 6-letter alphabetic strings. Although the algorithm is easy, a large amount of supplementary codes are involved. In the following we will only show some code snippets of the core parts, while the whole detailed code can be achieved at `https://github.com/kevin-futianwen/rainbow-table`.

4.1 Random Strings

As is discussed above, starting strings needs to be uniformly random among all possible inputs of interest. However, the c standard *rand()* function has limited randomness and the constant *RAND_MAX* is only guaranteed to be at least 32767, which is far from enough.

Therefore, we adopted the Tiny Mersenne Twister[4] which can generate random uniform 64-bit integer or double precision floating point number on 64-bit CPU. The random sequence has a period of 2^{127-1} while the performance is fast enough for our use.

With the random number generator above, we generate random strings by the following method:

```
const double stringMax =
            pow(charsetSize, maxSize);
uint64_t result = (uint64_t) round(stringMax *
     tinymt64_generate_double01(&random));
for (int i = 0; i < maxSize; i++) {
  if (!result) {
    str[i] = 0;
    break;
  }
  str[i] = charset[result % charsetSize];
  result /= charsetSize;
}
str[maxSize] = 0;
```

4.2 Generating the Rainbow Table

The algorithm suits parallelism well; to maintain simplicity, we use CPU-level parallelism (but it can be transformed to GPU parallel program easily), the core code is given by[4]:

```
for (int i = 0; i < listCount; i++) {
  randomString(listStart[i],
    config->maxPlainLength,
    config->charset,
    config->charsetSize);
}
for (unsigned i = 0; i < listCount; i++) {
  pthread_create(threadID + i, NULL,
      generateRainbowTableKernel,
      (void *) i);
}
for (int i = 0; i < listCount; i++)
  pthread_join(threadID[i], NULL);
```

The kernel in each thread is given by:

```
int i = (int) id;
strcpy(listEnd[i], listStart[i]);
for (int c = 0; c < reduction; c++) {
  hash(listEnd[i], digest);
  reduce(count, digest, listEnd[i]);
}
pthread_exit(NULL);
```

4.3 The Reduction Function

As is discussed above, reduction functions needs to be different from each other and cover as many plain texts of interest as possible. In this implementation, we create a new string by concatenating k, the number of reduction function, with first 7 bytes

[4]Some details are removed to keep the report short and concentrated; see the github site for the full code in *generator.c*

of hash, then get the SHA1 of this new string. From the SHA1 we reconstruct a plain text of interest.

By avalanche effect[5], the output of our reduction function will be different to a great extent for different k; properties of cryptographic functions can also give us pretty much coverage of all plain texts. The simplified core parts of the reduce function is given by[5]:

```
modifiedPlain[0] = k + 33;
memcpy(modifiedPlain + 1,
        cipher, 7);
SHA1(modifiedPlain, 8, digest);
for (int i = 0; i < 8; i++) {
  rank *= 256;
  rank += digest[i];
}
rank = (uint64_t) round(
  pow(26, SAMPLE_PLAIN_SIZE)
  * (rank / pow(256, 8)));
for (int i=0;i<SAMPLE_PLAIN_SIZE;i++){
  reduced[i] = rank % 26 + 'a';
  if (!rank) {
    reduced[i] = 0;
    break;
  }
  rank /= 26;
}
reduced[SAMPLE_PLAIN_SIZE] = 0;
```

4.4 Storage of Generated Table

In this implementation, we use the Trie data structure to store the (listStart, listEnd) pair with guaranteed constant-time time-complexity for fixed-length input. One may also use hashtables or other similar data structures for the same purpose. Implementations of this part has little relation to the rainbow tables and are therefore omitted. See the online full code for details.

4.5 Cracking a Hash

Just as is shown above, we try each position that the hash might be at, look up the ending string in the table, and try to retrieve the string that hashes to the hash:

```
for(int trial=reduction-1;trial>=0;trial--){
  reduce(trial, cipher, reducedCipher);
  reduceAndHash(config, reducedCipher,
    listEnd, trial + 1, reduction);
  if ((listStart=trieFind(table,listEnd))){
    reduceAndHash(config, listStart,
      plain, 0, trial);
    hash(plain, cipherTest);
    bool success = true;
    for (int i = 0; i < digestLength; i++) {
      if (cipherTest[i] != cipher[i]) {
        success = false;
        break;
      }
    }
    if (success) return true;
  }
}
```

[5]see *configsamples.c* at github repository for details

where the function *reduceAndHash* performs multiple times of reduction and hashing and the function *trieFind* returns NULL if not found.

5 Possible Improvements

5.1 Separated Generation and Cracking

In our implementation we generate the whole table and use it to crack one hash; however this could be even slower than using the direct brute force. The advantage of pre-computation is that it can be reused. In fact, we can first generate a huge rainbow table, save it to a file (possibly a database like SQLite), and then each time we want to crack a password we just use the file.

5.2 GPU acceleration

The rainbow table generation process is very friendly to parallel programming. In this implementation, we use multi-threading, which can make the best use of our CPU cores and threads. However, spawning thousands of threads does not make a great optimization on CPU which has merely 4 or 6 cores. NVIDIA CUDA library can allow such simple parallel computations on GPU, just by changing the kernel to global code and write simply

```
generateRainbowTableKernel<<<listCount,1>>>();
```

References

[1] Wikipedia contributors, "Cryptographic hash function — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Cryptographic_hash_function&oldid=892749881, 2019. [Online; accessed 25-April-2019].

[2] M. Hellman, "A cryptanalytic time-memory trade-off," *IEEE transactions on Information Theory*, vol. 26, no. 4, pp. 401–406, 1980.

[3] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," in *Annual International Cryptology Conference*, pp. 617–630, Springer, 2003.

[4] S. Mutsuo and M. Makoto, "Tiny mersenne twister (tinymt): A small-sized variant of mersenne twister." `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/`, 2015. [Online; accessed 26-April-2019].

[5] Wikipedia contributors, "Avalanche effect — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Avalanche_effect&oldid=892826167`, 2019. [Online; accessed 26-April-2019].