CS4236 Assignment Rainbow Table Attacks

Some considerations:

1. **Size of rainbow table:**
   - Storing the full digest would be too space consuming as each digest is 20 bytes which will result in too large tables. Storing last 4 bytes of the digest would be sufficient as 32-bit field is already much larger than 24-bit field (collisions are thus negligible). However, it is even better if only 3 bytes are stored rather than 4 bytes, though it is not advisable to store 3 bytes of the end digest as it will result in too many collisions. Therefore, maybe words rather than digests should be stored.
2. **Collision resolution:**
   - With big chain length, collisions will be usual. To reduce collision, merging chains must be dropped. However, if the chain heads are generated deterministically using a random seed, some tracking must be done to recover the chain heads when reading the table. Also, it is possible to generate chain heads deterministically and uniquely, but more consideration is required to reduce collisions as they are not randomly generated.
   - Also, different reduction functions should be used in a chain. Moreover, having multiple tables will allow different reduction functions in each table, thus further reducing collisions.
3. **Cost of locating a word:**
   - To minimize the number hashes needed to obtain a word from based on digest from query, it is crucial that we do not do redundant hashes, i.e. hash to the position where the next digest is the query digest and one more hash is needed to get the required word.
4. **Fast look-up:**
   - Besides reducing the number hashes used to locate a specific word for a given digest, getting to the right chain with less searching time is also necessary to perform the attack fast. Preferentially, hash map can be used.
5. **Time-space trade off:**
   - It is clear that there is a trade-off between time and space. More hashes will be needed with longer chains, but longer chains will give smaller tables. Also, more table will give less collisions and thus potentially less space required, but will increase the number of hashes required in each loop up. Therefore, it is important to have the right set of parameters to achieve optimal result. This can be done with trial and error.

Actual Implementation:

1. Generating the chain head deterministically:

Chain heads (words) are generated deterministically to ensure its uniqueness (as collision is concerned) and it can be later re-generated easily in Ftemplate.cpp. Regenerated heads are important later in traversing each chain to reach the correct word which we are looking for. This is done by simply incrementing a counter and use it value as the word. Masking and shifting are needed to generate the word in the char array format.

2. Last word in each chain is stored:

This is to reduce the storage requirements as only 3 bytes are used for each chain. To me, this seems to be least storage required for each chain. Moreover, by storing the last word, the last

hash can be generated with little cost so that given a query digest, we can easily compare it with the last digest with only 1 hash.

3. Drop merging chains:

Merging chains are chains which collide at same positions which leads to lots of collisions. As the operations done in the head-tail orientation are deterministic in nature. We should check the collision at the end of the chain to achieve optimal effect. Therefore, hash map is used to store unique last digest and repetitive last digest can be detected, and the merging chain can be dropped subsequently.

4. Keeping track the skipped chains by using 4 bit numbers:

As we drop merging chains to reduce collisions, without some tracking, we cannot later regenerate the chain heads any more as it is no longer deterministic. To track the skipped chains, we make some space trade-off by using a 4-bit number to recode the number of chains skipped at each row of the rainbow table. These numbers are stored and later can be used to regenerate the chain heads in a deterministic way. However, the space trade-off is minimized using 4-bit numbers, if more than 15 rounds of regenerations of chain heads does not give us a unique word, we will just accept it. Luckily, this does not happen too frequently. As the OS only allows memory allocation in at least 1 byte, we pack 2 4-bit numbers into one byte. This is done by shifting to pack two 4-bit number in an unsigned char type. For two 4-bit number A and B, we do (A + (B << 4)) and store the result in unsigned char C.

5. Fast look-up:

To reduce the number hashes needed, we hash minimally until we reach the correct position in a particular chain and switch to a different chain immediately if we do not get the word needed. A hash map is used which maps the stored words in rainbow tables to a vector of indices of their occurrences in the tables so that the look up can be done more efficiently. For example, word “abc” may appear multiple times due to collision, we store [1, 45, 67] as the mapped index-vector in the hash table if “abc” occurs at index 1, 45, and 67 in the rainbow table.

Parameters and results:

Number of tables: 2

Chain length: 220

Total number of words covered (include collisions): $2.36 * 2^{24}$

Number of chains: 179973

Total storage: two tables + one skip list file = 2 * 264KB (unzipped) + 51KB (zipped using Winzip) = 579 KB

Final results by testing with different seeds in generating the queries:

$$t = 104857600,\ C \geq 90.03\ \%,\ S = 579KB,\ F \geq 390$$

Usage of the program:

Unzip the submission file.

1. **Compile source code and use in the "src" subfolder:**
   $ cd src
   - Generate rainbow tables and skiplist file:

   $ g++ -std=c++11 Btemplate.cpp sha1.cpp -o Btemplate

   $ Btemplate (may take a few minutes)

   - Compilte Ftemplate.cpp

   $ g++ -std=c++11 Ftemplate.cpp sha1.cpp -o Ftemplate

   - Generate query and perform rainbow table attack

   $ g++ -std=c++11 Generate.cpp sha1.cpp -o Generate

   $ Generate [seed] [number of queries] | Ftemplate

2. **Use the executables and precomputed tables in the "executables" subfolder directly:**
   - $ cd executables
   - Unzip "skiplist.txt"
   - $ Generate [seed] [number of queries] | Ftemplate