Lab 3: Hashing Fun

Due Date: Sunday 2/25/2018 11:59PM

This lab will introduce you to some real-world C programming and the art of hash function design.

The core programming component of the lab should not take you that long. The primary challenges to completing this lab will be (1) understanding the skeleton code and (2) getting a conceptual grasp on what your hash function should be doing.

Brief Overview of Hashing

The concept of hashing is one of the most important in all of practical computer science. There are countless applications. Just to give a few examples, we use them to build search data structures (hash tables), to construct digital signatures, to ensure data integrity, to build cryptosystems, to quickly perform file comparisons, and to store passwords securely.

A hash function is one that takes an arbitrary length input (e.g. a message) and produces a fixed-length output, or digest. The length of the digest depends on the hash function. Typical digest lengths are 128, 256, or 512 bits. Note that if the key space (the set of possible inputs) can be larger than the set of possible digests, then by the pigeon hole principle we know that the hash function is not one-to-one (injective). Therefore two (or more) keys can possibly map to the same output value. This is called a collision. Hash functions that do not have this property are called perfect hash functions.

Cryptographic hash functions are an important class of hashes that have the additional property that it is infeasible to derive the input key given only the output digest. Here infeasible means that there is no sound algorithmic technique to do this other than via brute force, i.e. hashing all possible keys until the outputs match. Cryptographic hashes are used to implement digital signing, password storage, and secure communication protocols.

Designing hash functions is a bit of an art, as a good hash function will trade off between several properties, discussed below.

- **Determinism**: One requirement of a mathematical function is that given the same input, the function should always produce the same output, i.e. it is said to be deterministic. The same is true of hash functions. Your hash function should always produce the same digest given the same input sequence.

- **Avalanche**: A good hash function (especially cryptographic hashes) will have the property that if a single bit of the input message flips, every bit of the output will flip with probability one half. This property makes the statistical analysis of the hash function’s outputs very difficult (which would lead to attacks against it) very difficult.

- **Uniformity**: Outputs of the hash should be uniformly distributed based on the inputs. That is, keys which produce the same output should not be similar, or close together. This property is useful for hash tables, where collisions cause buckets in the hash table
to include chains of values. The chains should be more or less the same length to keep average case lookup time small.

- **Speed**: Because hashing applications use the hash function very frequently (e.g. for every block of data, for every message, etc.), performing the hash should not be computationally expensive. However, speed tends to come with simplicity, and simpler hash functions tend to trade off the above two properties for performance. These days, many hardware devices will logic for a particular hash function baked into them to make it faster. Network cards are a good example of this.

**Coding Assignment (100 points total)**

Your goal here will be to implement a relatively efficient and secure hash function. You now have all the conceptual tools you need, (bit-level representations, bit ops, C programming), and it’s time to put them to work.

Get the lab3 code by logging into fourier and running the following:

```
$> cp /home/khale/HANDOUT/lab3.tgz .
$> tar xvzf lab3.tgz
$> cd 13
$> make
gcc -Wall -O1 -I. -c hash.c -o hash.o
gcc -Wall -O1 -I. -c measure.c -o measure.o
gcc -lm -lrt -o cs350hash hash.o measure.o
$> ls
cs350hash hash.c hash.h hash.o Makefile measure.c measure.h measure.h measure
```

The executable produced is cs350hash. You can use it like you would another hashing program like md5sum. For example:

```
$> ./cs350hash Makefile
00000000000000000000000000000000 Makefile

or

$> cat hash.c | ./cs350hash
00000000000000000000000000000000
```

The program is reading in the input files, computing the 128-bit hash digest of them using your hash function, then printing out the digest to the console.

Notice that the digest is all zeros currently. That’s because your hash function currently does nothing. You will change that. The main files of concern for your purposes are hash.c and hash.h. The former contains hashing code and the latter contains structure and function definitions.

The main three pieces of interest for you are the functions named hash_blk and encode and the hash_func_t structure. hash_blk() performs the hash on a single block of the input message. Here we are assuming 512-bit (64-byte) blocks. Therefore, if I have a message
of 1KB, it will be chunked into 4 blocks, and I will invoke hash_blk() on each of them. The catch, however, is that to ensure avalanche, we want the construction of our digest to depend equally on all the blocks of the message, not just one. To do this, we’re going to need our hash_blk() function to have memory. We do this by having a structure that holds the state of the hash. This structure is initialized when the first block of the message is hashed. It is then returned to the caller of our block hash function, and we presume that it will be passed back to us as an input parameter for the next block. Thus, each invocation of hash_blk() should mutate this state in some way, but it won’t produce a digest until the very last block is processed. What the state should be is up to you, the key is that it is somehow transformed by the bits of each message block. Your hash function should use all the bits in the message block to transform this state. You can do so using bit operations like shifts, XORs, ANDs, additions, subtractions, multiplications, etc. The world is your oyster here, but you should really think about how the operations you’re performing will affect the output.

At the end, we need to take our state and use it to produce the final hash output (the digest). How this is done, again, is up to you. The key is that the output must be 16 bytes of data.

The coding tasks therefore are for you to:

- Fill in the hash_func_t structure with variables to store state.
- Fill in the hash_blk() function to perform a hash. Your hash will be evaluated on the metrics mentioned above, and against the instructor’s simple solution.
- Fill in the encode() function, which transforms the hash state into a digest.

These are denoted in the code by comments that say FILL ME IN.

You might realize looking at this code that I’ve made essentially no effort to dumb down the programming style. While a gentle introduction to some C code might have been a more pedagogical approach, there are seldom gentle introductions in the real world. You’re often thrown at a codebase with which you are completely unfamiliar and which is written in a programming language (and style) that is foreign to you. I imagine both are somewhat true in this case. That’s alright though, don’t panic. I’ve commented the code pretty verbosely to guide you along.

### Evaluating and Optimizing your Code

Your goal is to produce a good hash function. How we measure good is, in this case at least, determined by the four properties listed in the previous section. If you run your code with the -s flag, you’ll see output like the following:

```
[khale@fourier]-/13% ./cs350hash -s
====== CS 350 LAB 3 | HASHING FUN ========
[anonymous]
===========================================
Hash Function Score Card:
Digests differ in trial 0:
1a00000000000000502092127f000
ffffffff00000000040f0b8d23a00000
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
Determinism Check: no
Hashing Rate: 38467.17 hashes/ms
```
This is the output of our measurement suite, which is evaluating the aforementioned metrics. The first test is checking for determinism (same input, same output) using random inputs. In this case we haven’t met that criteria yet, so the measurement tool is telling us where the outputs differ. Your hash function should show a “yes” output. The second test is determining the speed of your hash function (in hashes per millisecond). This one is very fast because it isn’t doing anything at all yet. The third test is using your hash function to generate indexes into a hash table and counting the collisions. A good hash function will have roughly the same (uniform) number of buckets chained onto each hash index. Bad ones will be heavily skewed. We’re producing a distribution of bucket lengths and measuring how skewed they are by looking at the standard deviation ($\sigma$), which is the number you see above. A $\sigma$ of 4524 is very high, so this hash function is awful. Finally, the last measurement is the avalanche. Here we’re generating a bunch of random keys and flipping each bit of the key, then measuring how many of the output bits flip. The number you end up with should be very close to half of them. The number you see above is a rough measure of how far away we are from half of the output bits flipping, so closer to zero is better.

When you optimize your hash function, you should try to increase the hashing rate by making your hash function perform simpler operations, you should try to decrease the avalanche number by making close to half of the output bits flip for any input bit, and you should try to decrease the uniformity measure by making sure that similar keys don’t produce similar outputs.

**Hand-in Instructions**

Make sure to put your name on your submission. Submissions without names will be given zero points! For code, this means put a comment at the top of your C file with your name on it.

For the code, you must hand it in digitally. Once you’re happy with your code, in the directory where your code is, run the following:

```bash
$> make handin
```