Brandon Vo

Homework 2

The hash function provided is a modified version of the lightweight Jenkin's hash function where the size of an input is taken in terms of bytes. The hash is generated by bit-shifting the input through a loop based on the size of the input then bit shifting the end result a few more times by a fixed amount in order to obscure the hash output string[1].

The original Jenkins hash function doesn't use the input string itself as the key rather it uses the address space location of the input as the key to create a hash. This means that any two strings or integers that have the same size in length will output the same hash regardless of having different values.

In order to fulfill the conditions of a cryptographic hash function, the Jenkins function taken from Wikipedia has been modified. In order to make the function deterministic, the address space is no longer used as the key. Instead, it takes the 8-bit integer variant of the input as the key. This makes it so that the same input will always yield the same hash output.

It is also uncertain if the Jenkin's hash function can be considered uniform as the expected inputs are based on the size length, and the inputs are being truncated into their 16-bit forms. A script was used to automatically generate hash functions from size 0 to 255 to see if there were any matching collisions based on string length, but none were found. This could possibly mean that there is enough uniformity in the hash function to avoid a collision within range, but the modified hash function hasn't been tested enough to make any conclusions. There could be more done to enforce uniformity such as increasing the bit-size or not truncating the input at all.

Another issue with the Jenkins hash function is that there is no method to ensure a hash of a fixed size. While the hash function outputs a hash of 32-bits, there is nothing to ensure that the output will always have a 32-bit size because of the simplistic method of hashing. The modified hash function provided reduced the size of the hash from 32-bit to 16-bits to reduce the maximum possible number of hash outputs, and the function includes a loop which repeatedly bit shifts the hash value until it reaches the specified length of 5. This makes the hash function fixed length.

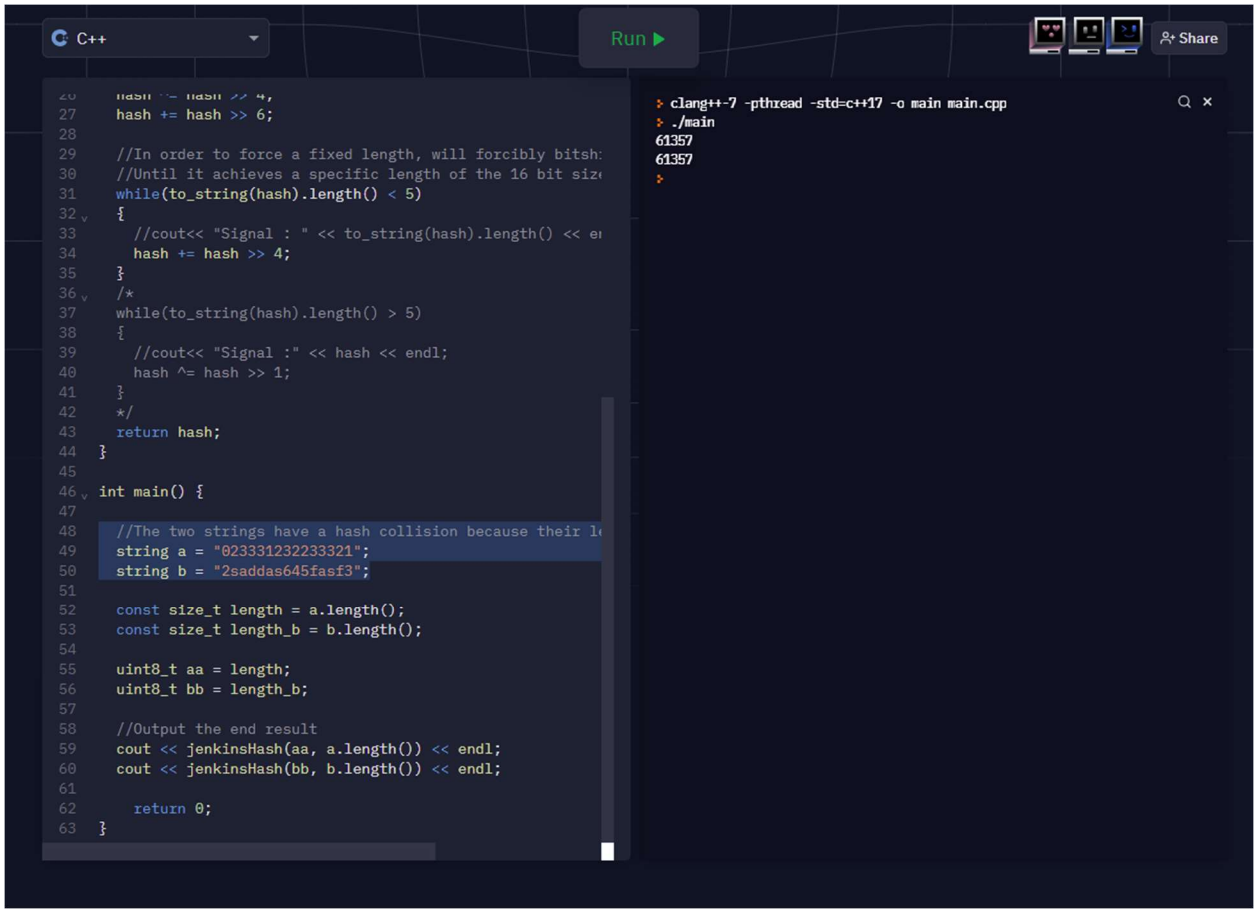
The final issue of the Jenkins hash function is that there isn't any collision-detection or avoidance. As stated previously, the Jenkin's hash function only uses the size of the input to generate a hash. This means that a collision is guaranteed as long as the two inputs have the same size. The example below demonstrates the issue with the Jenkins hash function provided. Because the hashing method is determined based on input length, then the hashing method won't change for any two string inputs that have the same length. Both string inputs that have the same size means they will use the same key values which means they will both lead to the same hash output, leading to a hash collision as shown below.

[1] https://en.wikipedia.org/wiki/Jenkins_hash_function

Brandon Vo

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
Input first hash:
ThisIsALargeInput

Input second hash:
ThisIsTheSameSize
27127
27127
```



Brandon Vo

Possible solutions to the current collision problem and other collision avoidance methods:

- Use the string inputs themselves by taking their bit counterparts and use the entire string as input for hashing rather than the size itself. This avoids the issue of inputs with the same length yielding the same results, but the modified Jenkin's hash function takes only the first 16-bits from a string input, meaning a collision is still possible if two inputs share the same first 16-bits.
- While reverting the maximum hash bit from 16-bit to 32-bit would increase the possible number of potential hash outputs, it only adds collision avoidance by increasing the complexity needed to brute force a hash collision. It means that an attacker has to check out of 32-bits rather than 16-bits to find a hash collision. If the attacker can find and reverse engineer the hashing method, the attack can identify the pattern of creating a hash and use that pattern to create a hash collision, bypassing the need to brute force a hash collision.
    - However, this would put the hash function at risk of not being uniform anymore.
- We can take a value from a specific byte in the input and use it as an additional offset for the hash. This would make it more difficult to reverse engineer the hashing method because a collision would now require two inputs either to have a matching offset or their offsets would both have to lead to the same output hash.
    - Matching the same offset would require figuring out which byte from the input is being read and is being used as the offset. A hash collision would require both inputs to share that same byte value to yield matching offsets. However, if an attacker knows what offset is being used, then it would reduce the time complexity of an attacker's brute force attack because the attacker now knows what specific values one section of the inputs must have.
    - Two inputs could also just have an offset that is just enough to lead to the same hash output. While this could add an extra challenge for an attacker trying to reverse engineer the hashing method, if the attacker can figure out how to manage the offset, then he can use that to easily figure out a pattern to create a collision, meaning the offset that was previously used as a method to avoid a collision can be abused to force a collision instead.

You can't necessarily avoid a collision in a cryptographic hash function, only reduce the chances of a collision happening. This is due to the Pigeonhole Principle, an issue due to hash algorithms having a fixed output length but infinitely many possible inputs. The input for a hash function can be of any length and size and any combination of inputs, and because hashing is deterministic, there will always be a pattern of inputs that lead to a specific output. As a result, there will inevitably be a specific pattern of inputs that will lead to the same hash due to the cryptographic hashing algorithm being unable to account for the infinite variety of inputs[2]. The only thing we can do is implement collision avoidance methods to keep reducing the chances of a collision from ever occurring and reduce the probability of a brute force attack from ever being viable.

[2] https://coincentral.com/hashing-basics-history/