CSC 230 Project 5

SHA1 Hash + HMAC Utility

For this Project, you will implement a program that computes the SHA1 (usually referred to as SHA-1) hash (also known as a digest) of a message, and then using this hash, compute the Hashed Message Authentication Code or HMAC for a given input file. What is SHA1? It's an example of a cryptographic hash. It's like the hash function you might use to build a hash table except, for cryptographic purposes, it is expected to be collision resistant. It is computationally straightforward to compute the SHA1 hash for an input file, but it should be really, really hard to find any other file that hashes to exactly the same thing. This kind of algorithm is used for things like authentication and digital signatures.

Note: SHA1 is considered today to be computationally weak, and is deprecated (translation: no longer recommended for use). The combination of SHA1 and HMAC however is still strong enough for most uses, and if you can implement SHA1, you should have no trouble understanding and implementing SHA2.

If that's a SHA1 hash, then what is a SHA1 HMAC? Like other cryptographic hash functions, SHA1 operates without a secret key. If there is no key, and the algorithm is known (a good thing for any cryptographic algorithm), then anyone can create a SHA1 hash. Let's say Alice wants Bob to download a file containing message **M**. You might think Alice could host two files on her Web server, one containing **M** and the other containing SHA1(**M**), and then have Bob determine if **M** has been modified by calculating SHA1(**M**) locally and seeing if it matches the second file. However, this doesn't work if an attacker replaces both files, storing **M'** and SHA1(**M'**). The attacker can create SHA1(**M'**), because the SHA1 algorithm is public. To fix this problem, Alice stores HMAC-SHA1(key, **M**) in the second file and shares the key with Bob out-of-band (e.g., over the phone). Since the attacker doesn't know the key, she can't create HMAC-SHA1(key, **M**). An HMAC is a special way of combining a key with a cryptographic hash function. We'll discuss this more below.

You'll be writing a program named hash. This program can be run in two ways: without and with the HMAC. To run the program without the HMAC, you simply give the name of an input file on the command line. The program computes and prints out the SHA1 hash for the input file, in hexadecimal. A SHA1 hash is 160 bits, so the program's output will always be 40 hexadecimal digits.

```
$ ./hash input-3.txt
474E8F1A64171D240A3EE95A07236DA2CFA3810A
```

There are already command-line tools that compute a SHA1 hash for an input file, so you can compare your output against these programs. The output format may be a little different, but, since we're implementing the real SHA1 algorithm, the hash values you get should match what these standard tools report. For example:

```
$ openssl sha1 input-3.txt
SHA1(input-3.txt)= 474e8f1a64171d240a3ee95a07236da2cfa3810a
```

To run the program with the HMAC, you pass the `-hmac <key>` command line argument *before* the filename. This should match existing command line tools. To be safe, your key value argument, which is just a sequence of ASCII characters, should be enclosed in quotes, as the examples show.

```
$ ./hash -hmac "mykey" input-3.txt
DE88B667B89D9B349864691932513F993DB367D1

$ openssl sha1 -hmac "mykey" input-3.txt
HMAC-SHA1(input-3.txt)= de88b667b89d9b349864691932513f993db367d1
```

Tip: Start out by building and testing plain-old SHA1 first. Then continue on to implementing the HMAC feature.

As with recent assignments, you'll be developing this project using git for revision control. You should be able to just unpack the starter into the p5 directory of your cloned repo to get started. See the Getting Started section for instructions.

This Project supports a number of our course objectives. See the Learning Outcomes section for a list.

Rules for Project 5

You get to complete this project individually. If you're unsure what's permitted, you can have a look at the academic integrity guidelines in the course syllabus.

In the design section, you'll see some instructions for how your implementation is expected to work. Be sure you follow these rules. It's not enough to just turn in a working program; your program has to follow the design constraints we've asked you to follow. For this project, we're putting some constraints on the functions you'll

need to define, and on one data structure you'll need to use. Still, you will have lots of opportunities to design parts of the project for yourself.

Requirements

This section says what your programs are supposed to be able to do, and what they should do when something goes wrong.

Running the program

The `hash` program expects a filename as its last command-line argument. This tells it the file for which it should compute the SHA1 hash. For example, you can run it as follows to get it to compute the hash of input file 3:

```
./hash input-3.txt
```

The program also accepts an optional command-line option, `-hmac` followed by a key to use in the HMAC computation. For example, if we run the program as follows, we're asking it to report a HMAC for input file 3, with the key "abc123".

```
./hash -hmac abc123 input-3.txt
```

If the user runs the command with invalid command-line arguments, the program should print the following usage message to standard error and terminate with an exit status of 1.

```
usage: hash [-hmac <key>] <filename>
```

If the command line arguments are valid but the given input file can't be opened, the program should report the following error message to standard error, where filename is the is the command-line argument given by the user. Here also, the program should terminate with an exit status of 1.

```
Can't open file: filename
```

SHA1 Computation

The SHA1 algorithm is a well know algorithm for cryptographic applications. It computes a 160-bit hash value for an arbitrarily large file. The SHA1 algorithm is documented in a lot of places. The Wikipedia Page is recommended for a well written explanation.

Also, be sure to look at the design section. It says something about how to organize your implementation so it works with our test driver code.

Input File Padding

The SHA1 algorithm works on 64-byte blocks from the input file. Before the file is processed, some extra bytes are added to the end, to bring the input size up to a multiple of 64 bytes and to put a little bit of information at the end. First, a byte with the value of `0x80` is added to the end of the input. Then, enough zero bytes are added to bring the total length of the input up to 8 bytes short of a multiple of 64. So, for example, if the original input is 1097 bytes long, after the 0x80 is added to the end, 46 more byte values of zero will need to be added so that the total length is 8 bytes less than a multiple of 64 bytes.

Then 8 more bytes will be added to the input file. These bytes should be the original length of the input, measured in **bits**. This value should be written out as an unsigned 64-bit quantity. For example, if the original input length was 1097 bytes, before any padding, then that's 8776 bits, the value of the 64-bit, unsigned integer length. Since message blocks consist of 32-bit words, the most significant 32 bits of the length will be in the next to last 32-bit word (4 bytes) of the last block, after the padding, and the least significant 32 bits of the message length will be in the last 32-bit word (4 bytes) of the last block to be processed. This is a reversal of the most significant half and least significant half of the normal little-endian ordering of a 64-bit quantity. (More on little-endian vs big-endian byte reordering within 32-bit words below.)

For a very good set of two SHA-1 examples, see the official NIST resource here.

SHA1 State and Hash Result

SHA1 operates on messages one block at a time, generating a new intermediate hash result after each block. The final hash result is simply the last intermediate hash from the final block of input (which includes padding and the message length field).

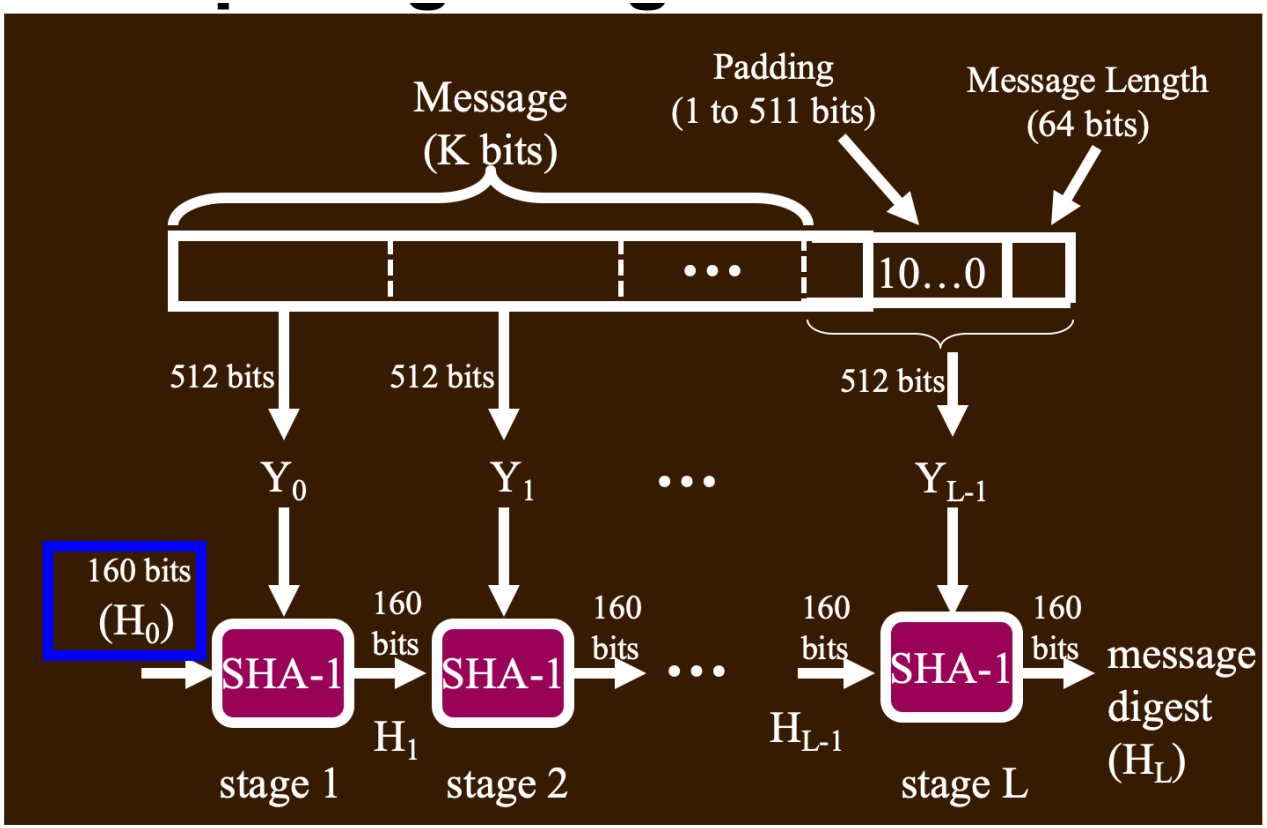
The SHA1 algorithm maintains a 160 bit state, represented as five unsigned, 32-bit integer fields. We'll call these five fields h0, h1, h2, h3, and h4. As it processes the input, the algorithm updates these five fields based

on each block of 64 bytes from the padded input. The state is initialized with random-looking constant values in h0, h1, h2, h3, and h4.

After processing all the input blocks, the contents of these fields, concatenated in order, is the 160-bit hash of the input. The hash is reported by printing out the values for h0, h1, h2, h3, and h4, in order. Each is printed as an 8-digit value in hexadecimal.

Block Processing

The SHA1 algorithm processes the input in blocks of 64 bytes. That's why the input size needs to be padded to a multiple of 64 bytes before processing. After processing an individual block, it updates the five fields in the state, and then goes on to process the next block from the input (so, it processes the first 64 bytes of the file, updates h0, h1 h2, h3, and h4, then the next 64 bytes, updates h0, h1, h2, h3 and h4, and so on). The initial values of h0, h1, h2, h3, and h4, input to the processing of the first block of the message, are the predefined constants 0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476, and 0xC3D2E1F0, respectively.



Overall execution flow of SHA1

Each 64-byte block of the input is interpreted as a block of 16 unsigned integers. This is accomplished by casting the array of 64 chars into an array of 16 unsigned ints.

Before processing the block, the five values in the SHA1 state are copied from the h0, h1, h2, h3, and h4 fields to local variables, a, b, c, d, and e. The block is processed in a series of 80 iterations (numbered iteration 0 through iteration 79). In each iteration, the value of the local variables, a, b, c, d, and e are updated. After all 80 iterations, the five fields of the SHA1 state are updated by by adding the corresponding variables a, b, c, d, and e to them. So, for example, the h0 field in the state is updated by adding the final value of a to it. These additions might overflow the unsigned integer. That's OK. You're just expected to keep the low-order 32 bits of the result (which is what you'll get when an unsigned overflow occurs).

The 16 words (32 bit unsigned ints) of the message block being processed are assigned to the first 16 words w[0]...w[15] of an 80-word array w. These 16 words are then used to derive 64 other words w[16]...w[79] as

shown below. This array of words is used in the computation of the output state for this message block, one word per iteration. In this figure, ⊕ means bitwise exclusive or, and lrot(exp, s) means left rotate the 32 bit unsigned int computed by the expression, for s positions. Because the C language does not include an operator for left rotation, you will have to implement this yourself. If you are unfamiliar with the left rotation operation for unsigned integers, see this for an explanation and example.

- First sixteen (32-bit) words w[0]..w[15] = 512-bit message block M_i
- for $16 \leq i \leq 79$
  w[i] = lrot((w[i-16] ⊕ w[i-14] ⊕ w[i-8] ⊕ w[i-3]), 1)

- Ex.: w[25] = lrot((w[9] ⊕ w[11] ⊕ w[17] ⊕ w[22]), 1)

Words computation

The 80 iterations performed on a block are broken into four *rounds*, each consisting of twenty iterations. So, iterations 0 to 19 are round 0, iterations 20 to 39 are round 1 and so on. In each iteration i, the value of w[i] described above will be used. Additionally, each iteration requires values for a constant k[i], and for a function f that combines the values of b, c and d that are input to this iteration, using bitwise operations. The definition of f and k[i] depends on the round. The following table shows the definition of f in each round, where ∧ means bitwise and, ∨ means bitwise or, ⊕ means bitwise exclusive or and ~ means bitwise complement. The predefined constant k[i] is also combined with the results of each iteration. The table also shows the values for k[i] to be used, as defined by the standard.

Iterations	Function f(i,b,c,d)	k[i]
$0 \le i \le 19$	$(b \wedge c) \vee (\sim b \wedge d)$	0x5A827999
$20 \le i \le 39$	$b \oplus c \oplus d$	0x6ED9EBA1
$40 \le i \le 59$	$(b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$	0x8F1BBCDC
$60 \le i \le 79$	$b \oplus c \oplus d$	0xCA62C1D6

Values for f and k

SHA1 Iteration

A SHA1 iteration is shown in the following figure. In each iteration, the values of a, b, c, d, and e are updated, by combining the previous values of a, b, c, d, and e with the appropriate values for f, k[i], and w[i], as shown below.

```
for i = 0 to 79
    temp = e + lrot(a, 5) + w[i] + k[i] + f(i,b,c,d)
    e = d
    d = c
    c = lrot (b, 30)
    b = a
    a = temp
endfor
```

One SHA1 Iteration

After completing all 80 iterations, the values of a, b, c, d, and e calculated by the last iteration are added respectively to the 5 values of the state (h0...h4) that was input to the processing for this block. This addition also ignores any overflow that may occur. (Note: in the Wikipedia pseudocode, indentation is used to indicate nesting; there is no close brace, or endfor, or other delimiter. The update of h0...h4 is not part of the for loop that computes each iteration of results; it comes after all iterations are complete.)

HMAC Computation

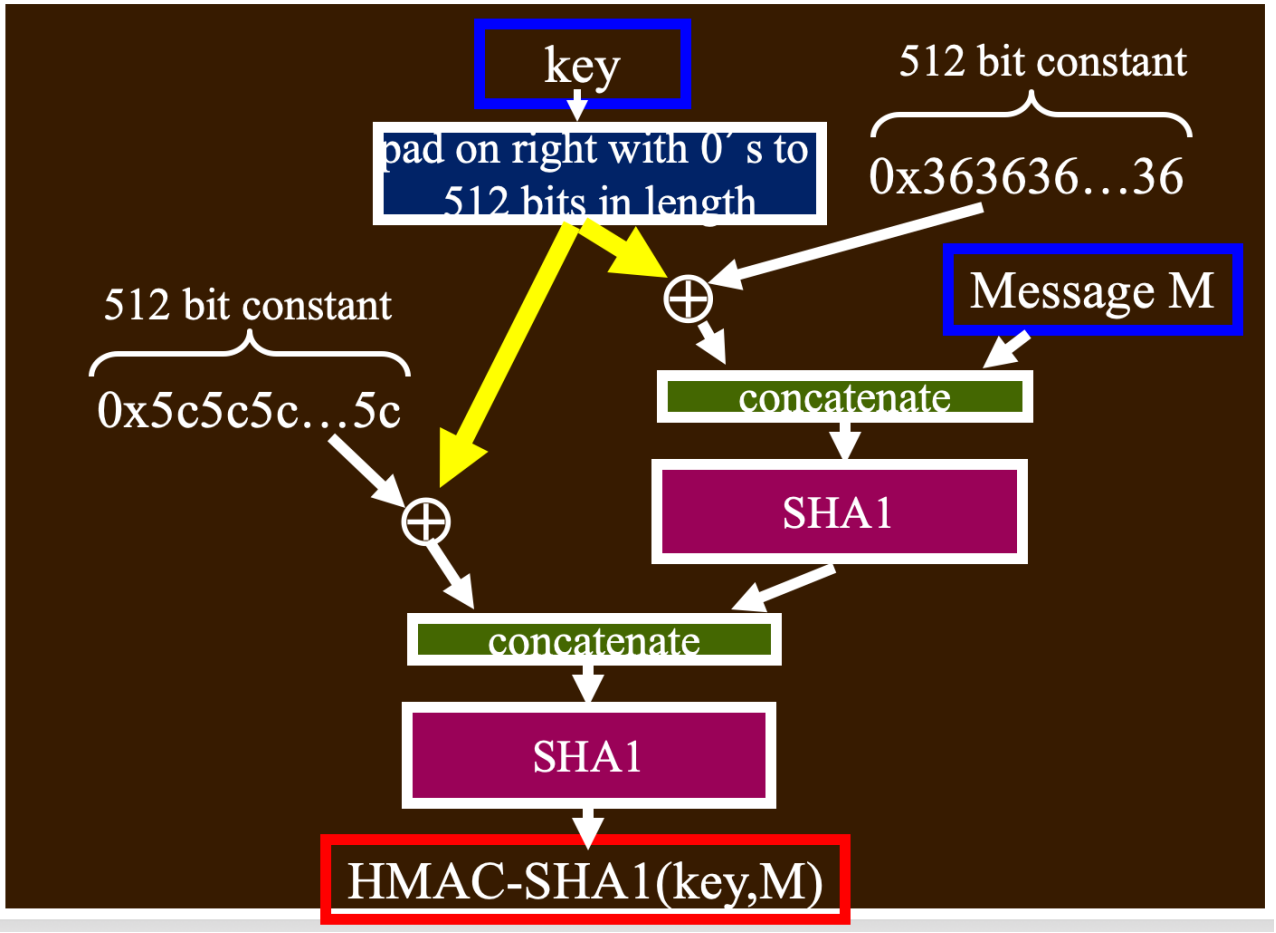
The HMAC standard (RFC 2104) safely combines a key with a cryptographic hash function and is designed to work with *any* cryptographic hash function. The discussion below describes HMAC using a generic function H(.). Of course, in our solution the hash function will be our SHA1 implementation. HMAC is well documented in many places. You may find the Wikipedia page helpful.

The goal of HMAC is to create a Message Authentication Code (MAC). This is a value for a message M that cannot be created (forged) without knowledge of a key K. This is typically denoted MAC(K,M). Simply hashing the key concatenated (||) with a message is not safe. For example, if we defined MAC(K,M) = H(K||M), there is a subtle attack called a *message extension attack* that impacts most known cryptographic hash functions. Think about how you have implemented SHA1. Each block iteration just extends the previous iteration. Therefore, if an attacker knows M and H(K||M), then she can replace M with M||M' and create H(K||M||M') without knowing the key K!

The HMAC algorithm is a safe way to combine a key with a hash function. Conceptually, HMAC = H(K || H(K||M)). That is, there is an inner hash just like the above vulnerable MAC function, but then a second outer hash that combines the key again. This prevents the attacker from extending the message and computing an updated value without knowing the key K.

The actual HMAC algorithm is a little more complex. The inner and outer hashes don't use the key directly. Instead, they XOR the key with a pad. The inner pad (ipad) is 0x36 repeated to the size of the hash input block (64 bytes for SHA1). The outer pad (opad) is 0x5c repeated to the size of the hash input block. Thus, HMAC(K,M) = H((K⊕*opad*) || H((K⊕*ipad*) || M)). Note that the key is often shorter than the hash input block. To handle this, shorter keys are padded with zeros until they're as long as the blocksize uzed by the hash. You can do this by first creating an array that's the same size as a block used by the hash function (64 bytes). You can fill it with zeros then copy the key into this array. That will give you the same characters as the key, followed by zeros up to the block size.

A diagram of HMAC is shown below.



HMAC

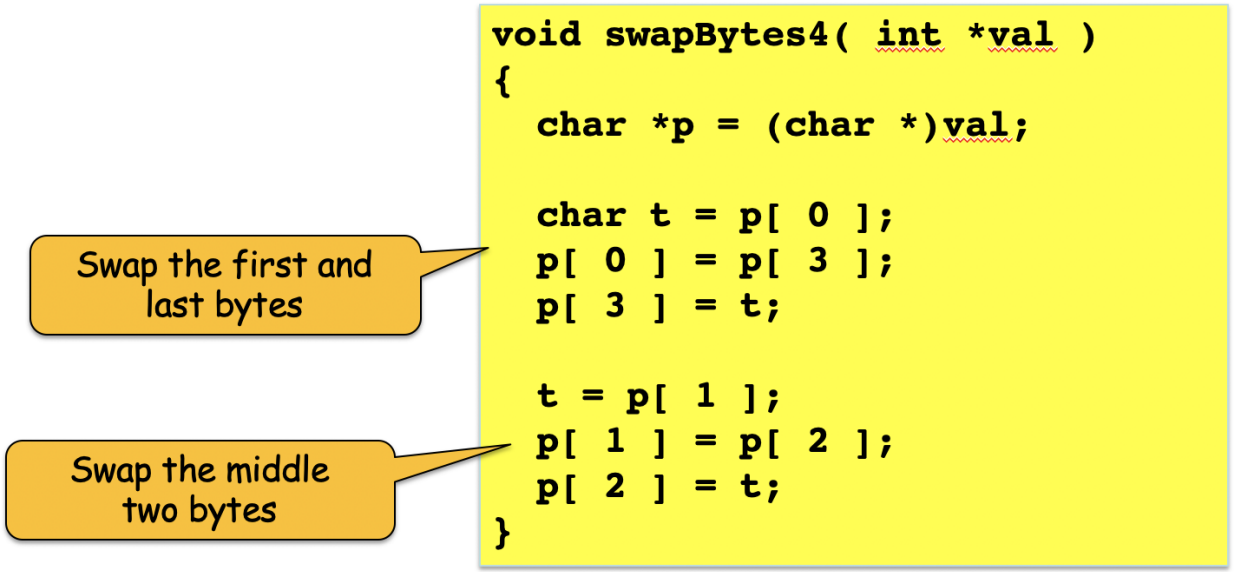
One last thing. If the key is *longer* than the hash function block size (64 bytes for SHA1), then the HMAC standard will hash the key before use in the above equation, just to reduce it to a smaller size. That's officially how the algorithm is supposed to work, but we won't require you to implement this part; you can assume the key is shorter than SHA1's block size. If you want to implement this part, just for fun, that's fine, but we will not test it.

A Final Complication: Big-Endian vs Little-Endian

The SHA1 algorithm is described for a big-endian architecture. It is almost a certainty that your program will run on a little-endian architecture (Intel processor). It is not required that your program constantly convert

back and forth between these two. The only required conversions are the following:

1. Before you execute SHA1, each 32-bit word of the the input message, block by block,including padding and message length at the end, must be converted from big-endian to little-endian, one time only.
2. Before outputting the result, your program must convert the results of the hash from little-endian to big-endian. This is done for each of the five 32-bit words of the hash. The code for doing so was discussed in the lecture on the Standard Library, shown below. The important part is the cast of a pointer to a 32-bit int to an array of 4 8-bit characters, which can then be reorderd as shown.



Byte reordering (big-endian / little-endian)

In the algorithm / code itself, nothing else needs to be changed. No intermediate values at any stage of hashing or computation of the HMAC, needs to be converted. All constants can be written in the form shown, all computations are executed in the form shown, etc. However, if you want to output intermediate results to the user from any step of computation, you will need to convert each word output from little-endian to big-endian order before printing, in order for the results to agree with any other source.

Design

Program Organization

Your source code will consist of four components, three of them with header files.

- `buffer.c` / `buffer.h`
  This component is responsible for reading and storing the input file. It uses a struct named Buffer to store the entire file contents in memory. This makes it easy to compute with and to add padding at the start of the SHA1 computation. The starter includes a partial implementation of this component, including a definition of the Buffer struct.

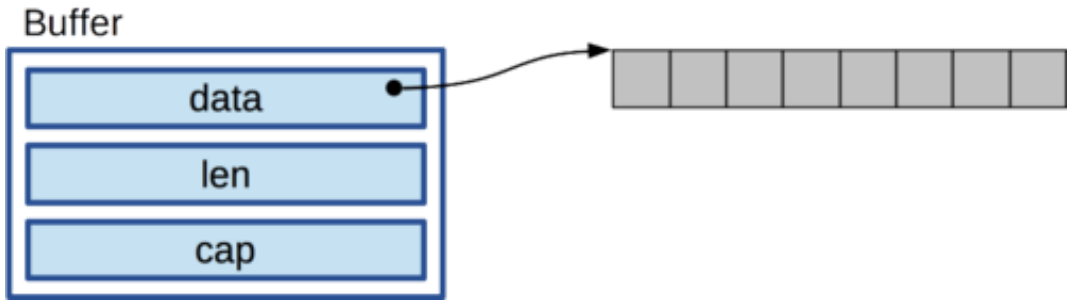
- `sha1.c` / `sha1.h`
  This component contains a struct to store the SHA1 state and functions to compute various parts of the SHA1 algorithm. The starter includes a partial implementation of this component, including a definition of the SHA1State struct.

- `hmac-sha1.c` / `hmac-sha1.h`
  This component defines a single utility `hmacSHA1()` for performing the HMAC operation for a given key and input.

- `hash.c`
  This is the main component. It contains the main function. It's responsible for parsing the command-line arguments and using the other components to read the input file, to perform the SHA1 computation and to compute the HMAC if requested.

Buffer Representation

Before starting the SHA1 / HMAC computation, the program reads the entire input file into a Buffer. This is implemented as a Buffer struct, defined in the buffer.h file. A buffer is just a container for a resizable array of bytes. The resizable array should start out with an initial capacity of 64 bytes (the size of one message block), doubling the capacity whenever we run out of room. Functions from the buffer component make it easy to create a buffer, add bytes to it and create a buffer holding the contents of any file. Other functions will use the data field directly to process the bytes of the file.



Buffer representation

Keep in mind, we can't store the input file as an ordinary C string. The input file could be binary, so it could have bytes with a value of zero inside it. The string functions would think these zeros are null terminators for a string. Fortunately, the Buffer struct keeps a sequence of bytes (the contents of the input) along with a separate length field, so we don't need to depend on any kind of end marker to know how long the data in a buffer is.

SHA1 State

The SHA1 component defines a struct called SHA1State. This keeps up with the current state of the SHA1 hash. It's just a collection of five unsigned integers named h0, h1, h2, h3, and h4. This state is initialized based on the 5 initialization constants given in the SHA1 algorithm, and it is updated for each 64-byte block in the input file. Once we finish processing all blocks in the input, we can print out the final value of the h0, h1, h2, h3, and h4 fields to report the resulting hash value.

Required Functions

As part of your implementation, you will need to define and use certain functions we're expecting. We're providing a unit test driver (described below) that will help you make sure your functions are doing the right thing.

The buffer component is expected to have at least the following four functions. You can add more functions if you want, but be sure to mark them as static if they're only intended to be used within the buffer component (e.g., if they're just helper functions that other components wouldn't need to use).

- `Buffer *makeBuffer()`
  This function dynamically allocates a Buffer struct, initializes its fields (a typical representation for a resizable array).

- `void appendBuffer( Buffer *b, unsigned char byte )`
  This function adds a single byte to the end of the given buffer, enlarging the data array if necessary.

- `void freeBuffer( Buffer *b )`
  This function frees all the memory for the given buffer.

- `Buffer *readFile( const char *filename )`
  This function creates a new buffer, reads the contents of the file with the given name, stores it in the buffer and returns a pointer to the buffer. If the file can't be opened, it just returns NULL.

The sha1 component needs to have the following functions. As with buffer.c/buffer.h, you can add other functions if you want to, but consider marking them static where possible. The test_static on some of these functions is a preprocessor macro. It lets us make some functions static when we compile for production, but we can can disable the static designation when we need to test these functions from a separate test driver.

- `void initState( SHA1State *state )`
  Given the address of a SHA1State, this function initializes its fields, filling them in with the five constant values given in the SHA1 algorithm.

- `void padBuffer( Buffer *b )`
  This function pads the given buffer, bringing its length up to a multiple of 64 bytes, adding byte values as described in the SHA1 algorithm.

- `void sha1Block( unsigned char data[ SHA1_BLOCK ], SHA1State *state )`
  This function performs 80 SHA1 iterations on the given block of bytes, updating the given state.

- `void sha1Encode( unsigned char digest[ SHA1_DIGEST ], SHA1State *state);`
  This function is used to create the final hash value (also known as a "digest"). It transfers the 20 bytes in the h0, h1, h2, h3, and h4 state variables into a 20 byte unsigned char array, in big-endian order. i.e., digest[0] through digest[3] contains the value of h0 in big-endian order, digest[4] through digest[7] contains h1 in big-endian order, etc.

- `test_static unsigned int fVersion0( unsigned int b, unsigned int c, unsigned int d )`
  `test_static unsigned int fVersion1( unsigned int b, unsigned int c, unsigned int d )`
  `test_static unsigned int fVersion2( unsigned int b, unsigned int c, unsigned int d )`
  `test_static unsigned int fVersion3( unsigned int b, unsigned int c, unsigned int d )`
  The SHA1 algorithm uses four different versions of a function named f, a different version for each round. Each of these implements one of the versions.

- `test_static unsigned int rotateLeft( unsigned int value, int s )`
  This function implements the rotate left operation from the SHA1 algorithm, shifting the given value to the left by s bits, with wraparound. It returns the resulting value.

- `test_static void sha1Iteration( unsigned int data[ 16 ], unsigned int *a, unsigned int *b, unsigned int *c, unsigned int *d, unsigned int *e, int i )`
  This function implements an iteration of the SHA1 algorithm on a 64-byte block (interpreted as 16 unsigned integers). The first parameter is the data block, the next five parameters are the a, b, c, d, and e values from the SHA1 algorithm description, passed by reference so the function can change them. The last parameter is the iteration number, a value between 0 and 79.

In the `sha1.h` header, only give prototypes for the functions that have external linkage. Don't give prototypes for the ones marked as test_static. The test_static functions will be static (except when we're compiling for test), so other components couldn't call them anyway. Plus, the header doesn't contain a definition of the test_static macro, so trying to use it in the header will cause compilation problems.

The hmac-sha1 component provides a simple interface to perform the HMAC-SHA1 operation using the interface provided by the sha1 component. The hmac-sha1 component needs the following function. As with the other components, you can add other fucntions if you want to, but consider making them static where possible.

- `void hmacSHA1( char *kstr, Buffer *b, unsigned char digest[ SHA1_DIGEST ]);`
  This function performs the HMAC-SHA1. It takes a key as a string of characters, a pointer to a Buffer struct, and a pointer to an area of memory to store the digest (using `sha1Encode()`). Note that you will likely *not* want the Buffer to have the padding, as the computation requires you to put the *ipad* before the Buffer contents (e.g., by making a new buffer and copying from the passed Buffer).

f Function Pointer and k Constants Lookup

In each of its 4 rounds, the SHA1 algorithm needs to use a different version of a function named f, and a different constant called k. You will be implementing each version of f as a separate C function. Create an array of four function pointers, each element pointing to a different version of the f function. Also make an array for the k values. Then, during a SHA1 Iteration, you can look up the right version of f and k in these two arrays, based on the current round of the SHA1 calculation.

Magic Numbers

The SHA1 algorithm uses a lot of constants, some of which are just used as a source of pseudo-randomness in the algorithm. You're not expected to make named constants for these values (it's not really clear what you'd call these constants anyway). Jenkins may still complain about them, but these values won't be charged against you when we look for magic numbers in your source code.

There are some values used by SHA1 that would still be good opportunities to use a preprocessor symbol, like the number of iterations applied to each block, the number of iterations in each round, or maybe some of the bit masks you might need. Be sure to use named constants when they help to explain what the value is for, when they represent a configurable parameter, or when you need to use the same value in multiple places.

Build Automation

You get to implement your own Makefile for this project (called `Makefile` with a capital 'M', no filename extension). Its default target should build your hash executable. Be sure to use our standard gcc options, `-Wall`, `-std=c99` and `-g`. As usual, your Makefile should correctly describe the project's dependencies, so targets can be rebuilt selectively, based on what parts of the project have changed. It should also have a clean rule, to let the user discard temporary files as needed.

For this project, you'll also need to be able to build a testdriver target, by compiling the testdriver.c program and linking it with your buffer and sha1 components. Here, we won't worry about being able to compile the program selectively. You can use a single make rule, with testdriver as the target and all of the implementation and header files it depends on as prerequisites. To rebuild the target, just use a gcc command to compile and link all the source files. This isn't as efficient as what we'd normally do, but we need to re-compile the sha1 component with the -DTESTABLE flag (see below) and rebuilding everything is an easy way to do this. In your makefile, when the user asks you to rebuild the testdriver target, just use the following command:

```
gcc -Wall -std=c99 -g -DTESTABLE testdriver.c sha1.c buffer.c -o testdriver
```

Testing

The starter includes a test script, along with test input files and expected outputs. It also has a test driver to examine the behavior of your individual functions. When we grade your program, we'll test it with this script, along with an extended test driver and a few other test inputs we're not giving you. To run the automated test script, you should be able to enter the following:

```
$ chmod +x test.sh # probably just need to do this once
$ ./test.sh
```

This will automatically build your program using your makefile, see how it does against all the tests and try it out with the test driver.

As you develop and debug your programs, you'll want to be able to run the tests yourself. For the test driver, you can just build it using your makefile, then run it from the command line. As you run the test script, you'll see it reports on how it's running your program for each test. You can copy this command to run your program directly to get a better idea of how it's behaving.

Test Driver

Along with the usual test inputs and expected outputs, this project comes with a test driver program for testing individual functions in buffer.c and sha1.c. It's named testdriver.c.

If you look inside this file, you can see it uses the Buffer and SHA1State types, and it calls functions in the buffer and sha1 components, checking their results to see if it looks like they worked correctly.

For the test driver, we use some preprocessor tricks. Since you have to implement all the functions the test driver needs, lots of these test won't even work until you get some of your implementation done. If you look in the driver, there's some conditional compilation code that, initially, just leaves all the tests out. As you implement parts of your program, you'll be able to try out these tests. You can move the `#ifdef NEVER` line down past blocks of tests, enabling each block of code as you fill in more and more of your implementation. When you're done, you should just be able to remove this `#ifdef NEVER / #endif` to enable all the tests.

In the test driver, we use a preprocessor macro, TestCase() to run each test. This macro checks a condition to see if the test works, and it automatically reports an error message giving the line of code where each test failed. It's kind of like the assert() macro, but it lets the program continue running even after a failed test. If your test driver makes it all the way to the end (without, say, crashing), you get a report of how many tests passed and how many failed.

We use one other preprocessor trick for the test driver. In the sha1 component, there are some functions that should normally be static; they're implementation details of the sha1 component, and other code shouldn't be able to access them directly. However, it's nice to be able to these these functions from the test driver, so we can test them. Comparing this to Java, it's kind of like wanting to test private methods in a class. To permit this, the sha1 component uses a preprocessor macro named `test_static`. This macro lets us mark selected functions as static when we're building the `hash` executable, but we can mark them as non-static when we're compiling for test. That's what the `-DTESTABLE` compiler flag is used for. If we define TESTABLE preprocessor symbol, then selected functions (those marked with test_static) get exposed to the linker, so we can test them. Otherwise, they get marked as static, so other components can't see them. That's why you need to use the `-DTESTABLE` flag when you compile for test, but not when you compile the regular `hash` executable.

We've come up with a few additional unit tests and wrapped these up in a new test driver. It's called testdriver2.c. The original test driver didn't test all four rounds for the sha1Iteration() function, so there's a new block of tests for this. There's also a block of tests that corresponds to what should happen when computing a sha1 hash for input-1.txt. It shows what the buffer should contain after this file is read in, padded and after its byte order is reversed. You should be able to compile this test driver using a similar gcc command to testdriver.c. You don't need to add support for this driver to your Makefile (but you can if you want to).

Memory Error and Leaks

When it terminates successfully, your program is expected to free all of the dynamically allocated memory it allocates and close any files it opens. When you're programs have to exit unsuccessfully, they aren't expected to be free of leaks. Valgrind can help you find memory errors or leaked files.

To get valgrind to check for memory errors in one of your programs, you can run your program like this:

```
$ valgrind --tool=memcheck --leak-check=full ./hash input-3.txt
-lots of valgrind output-
```

To get it to look for file leaks instead, you can run it like the following. You'll get a report that file descriptors 0, 1 and 2 are still open. That's normal; those are standard input, standard output and standard error. If you see others, that's probably a file leak.

```
$  valgrind --track-fds=yes ./hash input-3.txt
-lots of valgrind output deleted-
```

Remember, valgrind will be able to give you a more useful report if you compile with the `-g` flag, so don't forget it.

Static Program Analysis (Optional, if you wish to try)

The Clang Static Analyzer is a static source code analysis tool that finds bugs in C, as well as C++ and Objective-C. It contains a suite of checks, as well as a number of alpha checkers. Running the Clang Static Analyzer in its default configuration is fairly simple:

```
$ scan-build make
```

This will wrap your build process within the Clang Static Analyzer. While we will not be grading based on Clang output, you are encouraged to experiment with it. Start out by running scan-build with the default options, and then explore the other checkers. Since enabling checks is a bit verbose, you can make a small script (e.g., `clang-test.sh`) with something similar to the following:

```
#!/bin/sh

CHECKERS="\
    -enable-checker alpha.security.ArrayBoundV2 \
    -enable-checker alpha.security.MallocOverflow \
    -enable-checker alpha.security.ReturnPtrRange \
    -enable-checker alpha.unix.cstring.BufferOverlap \
    -enable-checker alpha.unix.cstring.NotNullTerminated \
    -enable-checker security.insecureAPI.strcpy
"

scan-build ${CHECKERS} make
```

By modifying the checkers listed in the CHECKERS shell variable, you can control which tests you want to use. Explore the different security (and non-security) checks.

Installing scan-build: The Clang Analyzer (and scan-build) may be on some but not all of the university-provided common platform machines. It is on remote-linux.eos.ncsu.edu, for one. If you are running a CentOS or Fedora Linux distribution, you can use `dnf` (or `yum`) to install the `clang-analyzer` package. On Debian / Ubuntu based systems, scan-build is part of the `clang-tools` package.

Test Inputs

We use a few input files to test your program. The test script and the test driver both use these to see if your program works. We also have some invalid tests that try out your program with bad command-line arguments or with an input file that doesn't exist.

- input-1.txt : This is a short file that fits in a single block for SHA1, even after padding.
- input-2.txt : This is another short file that just barely fits in a SHA1 block. After adding the padding at the end, you'll have two blocks to process.
- input-3.txt : This is a longer input file. It will take several calls to sha1Block() to hash it. It's from the Wikipedia page on the Okapi.
- input-4.txt : This is an even longer file, where your input buffer will need to do some resizing a few times. It's a copy of the RFC for SHA1.
- input-5.bin : This is a binary file. Your program should be able to handle it, but you can't treat it like a C string, since it contains some zero bytes.

Grading

The grade for your program will depend mostly on how well it functions. This time, we'll be looking at behavior of the program as a whole (what we might call acceptance tests) and behavior of individual functions (what we might call unit tests). We'll also expect your code to compile cleanly, to follow the style guide and to follow the design given in this assignment. We'll also try your programs under valgrind, to see if it can find anything to complain about.

- Compiling cleanly on the common platform: **10 points**
- Working Makefile: **8 points**
- Behaves correctly on all tests (unit tests and acceptance tests): **80 points**
- Program follows the style guide: **20 points**
- Deductions
  - Up to **-60 percent** for not following the required design.
  - Up to **-30 percent** for failing to submit required files or submitting files with the wrong name.
  - Up to **-30 percent** penalty for file leaks, memory leaks or other memory errors.
  - **-20 percent** penalty for late submission.

Getting Started

To get started on this project, you'll need to clone your NCSU github repo and unpack the given starter into the p5 directory of your repo. You'll submit by checking files into your repo and pushing the changes back up to the NCSU github.

Clone your Repository

You should have already cloned your assigned NCSU github repo when you were working on project 2. If you haven't already done this, go back to the assignment for project 2 and follow the instructions there for cloning your repo.

Unpack the starter into your cloned repo

You will need to copy and unpack the project 5 starter. We're providing this file as a compressed tar archive, starter5.tgz. You can get a copy of the starter by using the link in this document, or you can use the following curl command to download it from your shell prompt.

```
$ curl -O https://www.csc2.ncsu.edu/courses/csc230/proj/p5/starter5.tgz
```

Temporarily, put your copy of the starter in the p5 directory of your cloned repo. Then, you should be able to unpack it with the following command:

```
$ tar xzvpf starter5.tgz
```

Once you start working on the project, be sure you don't accidentally commit the starter to your repo. After you've successfully unpacked it, you may want to delete the starter from your p5 directory, or move it out of your repo.

```
$ rm starter5.tgz
```

After this project was published, we added a source file for the test driver to the starter. We also made a small change to the sha1.h header file. The starter has been updated to include these changes. If you've already started on the project, you may want to download the project 5 update file instead, update5.tgz. The update file just contains the files that have changed since the project was first posted.

Instructions for Submission

If you've set up your repository properly, pushing your changes to your assigned CSC230 repository should be all that's required for submission. When you're done, we're expecting your repo to contain the following files. You can use the web interface on github.ncsu.edu to confirm that the right versions of all your files made it.

- `hash.c` : Top-level implementation file, created by you.
- `sha1.c` and `sha1.h` : Component implementing the sha1 algorithm, provided with the starter but extended considerably by you.
- `hmac-sha1.c` and `hmac-sha1.h` : Utility component for performing HMAC-sha1 using the functions in the sha1 component. Created by you.
- `buffer.c` and `buffer.h` : Component implementing a resizable array of bytes. A partial header is provided with the starter, but this component will need to be extended considerably by you.
- `testdriver.c` : Unit test driver, provided with the starter, but modified by you as you enable more and more of the tests.
- `Makefile` : The project's makefile, created by you.
- `input-*.txt` : Test input files.
- `expected-*.txt` : Expected output files.

- `estderr-*.txt` : expected error output for a few of the tests.
- `test.sh` : test script, provided with the starter.
- `.gitignore` : a file to prevent executables, and temporary files from being submitted to the repo. Created by you.

Pushing your Changes

To submit your project, you'll need to commit your changes to your cloned repo, then push them to the NCSU github. Project 2 has more detailed instructions for doing this, but I've also summarized them here.

Whenever you create a new file that needs to go into your repo, you need to stage it for the next commit using the `add` command:

```
$ git add some-new-file
```

Then, before you commit, it's a good idea to check to make sure your index has the right files staged:

```
$ git status
```

Once you've added any new files, you can use a command like the following to commit them, along with any changes to files that were already being tracked:

```
$ git commit -am "<meaningful message for future self>"
```

Of course, you haven't really submitted anything until you push your changes up to the NCSU github:

```
$ git push
```

Checking Jenkins Feedback

Checking jenkins feedback is similar to previous projects. Visit our Jenkins system at http://go.ncsu.edu/jenkins-csc230 and you'll see a new build job for project 5. This job polls your repo periodically for changes and rebuilds and tests your project automatically whenever it sees a change.

Learning Outcomes

The syllabus lists a number of learning outcomes for this course. This assignment is intended to support several of theses:

- Write small to medium C programs having several separately-compiled modules
- Explain what happens to a program during preprocessing, lexical analysis, parsing, code generation, code optimization, linking, and execution, and identify errors that occur during each phase. In particular, they will be able to describe the differences in this process between C and Java.
- Correctly identify error messages and warnings from the preprocessor, compiler, and linker, and avoid them.
- Find and eliminate runtime errors using a combination of logic, language understanding, trace printout, and gdb or a similar command-line debugger.
- Interpret and explain data types, conversions between data types, and the possibility of overflow and underflow
- Explain, inspect, and implement programs using structures such as enumerated types, unions, and constants and arithmetic, logical, relational, assignment, and bitwise operators.
- Trace and reason about variables and their scope in a single function, across multiple functions, and across multiple modules.
- Allocate and deallocate memory in C programs while avoiding memory leaks and dangling pointers. In particular, they will be able to implement dynamic arrays and singly-linked lists using allocated memory.
- Use the C preprocessor to control tracing of programs, compilation for different systems, and write simple macros.
- Write, debug, and modify programs using library utilities, including, but not limited to assert, the math library, the string library, random number generation, variable number of parameters, standard I/O, and file I/O.
- Use simple command-line tools to design, document, debug, and maintain their programs.

- Use an automatic packaging tool, such as make or ant, to distribute and maintain software that has multiple compilation units.
- Use a version control tools, such as subversion (svn) or git, to track changes and do parallel development of software.
- Distinguish key elements of the syntax (what's legal), semantics (what does it do), and pragmatics (how is it used) of a programming language.