CS-E4320 Assignment 1: SHA-3

Filippo Bonazzi
filippo.bonazzi@aalto.fi

September 9, 2016

Contents

1 Introduction

The assignment is to implement SHA-3, a cryptographic hash function released by NIST in 2015[1]. The specification of the SHA-3 standard is available as FIPS publication 202 [1], which can be freely downloaded from the NIST website (`http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf`).

[1] `www.nist.gov/news-events/news/2015/08/nist-releases-sha-3-cryptographic-hash-standard`

2 Definition of the Assignment

The task is to implement the SHA3-256 cryptographic hash function, as defined in Section 6.1 of the standard; the other SHA3 functions (SHA3-224, SHA3-384, SHA3-512) are not required.

In order to implement SHA3-256, you will need to implement KECCAK[c](N,d), described in Section 5.2 of the standard, with parameters

- c, the capacity, equal to 512
- N, the message, a bit string of arbitrary length
- d, the digest length, equal to 256

This will lead you to implement the KECCAK-p[1600, 24] permutation, with width 1600 bits and 24 rounds. The KECCAK-p permutation is described in Section 3 of the standard. You do not need to implement KECCAK-p permutations with width and round number other than 1600 and 24, at this time.

3 Provided code

In order to simplify the Assignment, we provide an implementation of four functions defined by the standard: `pad10*1`, `concatenate`, `concatenate_01` and `rc`. The functions are not optimized for performance: they implement functionality as it is laid out in the standard, and as such they are optimized for clarity and readability. You may use these functions in your code as they are, or freely adapt them to suit the structure of your code. It is not required that you use these functions.

For convenience, we also provide a macro to rotate 64-bit words to the left, `ROL64`: you may find this useful in your implementation.

3.1 `pad10x1`

The `pad10x1` function implements the pad10*1(x, m) algorithm defined in Section 5.1 of the standard. The function creates a padding bit string of suitable length such that, by concatenating it to a m-length bit string, the length of the resulting string will be a multiple of x. The function has the following prototype:

```
unsigned long pad10x1(unsigned char **P, unsigned int x,
                      unsigned int m);
```

P is a pointer to a `unsigned char` pointer. The function dynamically allocates an array: P is used to return the pointer to this array to the caller. It is responsibility of the caller to free this array after use.

x is the alignment value.

m is the length of the complementary string.

The function returns the length of the padding array P in bits. A sample usage of the function is shown in Figure 1.

```
unsigned char sample[] = { 0x00, 0xff, 0x00 };
unsigned int sample_len = 24;
unsigned char *padding = NULL;
unsigned long pad_len;
pad_len = pad10x1(&padding, 32, sample_len);
...
free(padding);
```

Figure 1: The `sample` array contains a bit string of length 24; after calling `pad10x1`, `padding` will point to an array containing a bit string of length 8, and `pad_len` will be equal to 8.

3.2 concatenate

The `concatenate` function implements the bit string concatenation operation, defined as $X||Y$ in Section 2.3 of the standard. The function creates a bit string which is the result of the concatenation of the X and Y bit strings. The function has the following prototype:

```
unsigned long concatenate(unsigned char **Z,
        const unsigned char *X, unsigned long X_len,
        const unsigned char *Y, unsigned long Y_len);
```

Z is a pointer to an `unsigned char` pointer. The function dynamically allocates an array: Z is used to return the pointer to this array to the caller. It is responsibility of the caller to free this array after use.

X is a pointer to the array containing the first bit string.

X_len is the length of X in bits.

Y is a pointer to the array containing the second bit string.

Y_len is the length of Y in bits.

The function returns the length of the concatenated array `Z` in bits. The function can concatenate bit strings of arbitrary length: `X_len` and `Y_len` do not need to be multiples of 8, nor do they need to add up to a multiple of 8.

A sample usage of the function is shown in Figure 2.

The concatenation is shown graphically in Figure 3.

```
unsigned char sample[] = { 0x00, 0xff, 0x00 };
unsigned int sample_len = 24;
unsigned char *padding = NULL, *concat = NULL;
unsigned long pad_len, concat_len;
pad_len = pad10x1(&padding, 32, sample_len);
concat_len = concatenate(&concat, sample, sample_len,
                         padding, pad_len);
...
free(concat);
free(padding);
```

Figure 2: The `sample` array contains a bit string of length 24; after calling `pad10x1`, `padding` will point to an array containing a bit string of length 8, and `pad_len` will be 8. After calling `concatenate`, `concat` will point to an array containing a bit string of length 32, and `concat_len` will be 32.

```
            sample: 00000000 11111111 00000000
           padding: 10000001
sample || padding: 00000000 11111111 00000000 10000001
```

Figure 3: Behaviour of the `concatenate` function

3.3 `concatenate_01`

The `concatenate_01` function is a special case of the `concatenate` function described in the previous Section, where the second string is the `01` string of length 2: it implements the operation $X||01$. The function has the following prototype:

```
unsigned long concatenate_01(unsigned char **Z,
        const unsigned char *X, unsigned long X_len)
```

Z is a pointer to an `unsigned char` pointer. The function dynamically allocates an array: Z is used to return the pointer to this array to the caller. It is responsibility of the caller to free this array after use.

X is a pointer to the array containing the user-provided bit string.

X_len is the length of X in bits.

The function returns the length of the concatenated array `Z` in bits (`X_len + 2`).

A sample usage of the function is shown in Figure 4.

```
unsigned char sample[] = { 0x00, 0xff, 0x00 };
unsigned int sample_len = 24;
unsigned char *concat = NULL;
unsigned long concat_len;
concat_len = concatenate_01(&concat, sample, sample_len);
...
free(concat);
```

Figure 4: The `sample` array contains a bit string of length 24. After calling `concatenate_01`, `concat` will point to an array containing a bit string of length 26, and `concat_len` will be 26.

3.4 rc

The `rc` function implements the *rc(t)* algorithm defined in Section 3.2.5 of the standard. The function implements a binary linear feedback shift register (LFSR) with outputs in **GF**(2). The function has the following prototype:

```
unsigned char rc(unsigned int t);
```

`t` is the number of rounds to perform in the LFSR.

The function returns a single bit, stored as the LSB of an unsigned 8-bit number (`unsigned char`). Note that on Intel x86-64 machines the LSB is the right-most bit in a byte.

A sample usage of the function is shown in Figure 5.

```
unsigned char RC=0;
for(int i = 0; i < 8; i++) {
        RC ^= (rc(i) & 1) << i;
}
```

Figure 5: `RC` is a 8-bit unsigned number, initialized to zero. For each iteration of the loop, the result of `rc(i)` is computed and stored in the i-th bit of `RC`. Note that on Intel x86-64 machines the LSB is the right-most bit in a byte.

3.5 ROL64

The `ROL64` macro implements rotation (to the left) of a 64-bit word. The macro has the following definition:

```
#define ROL64(a, n) ((((n)%64) != 0) ? ((((uint64_t)a) << ((n)
   %64)) ^ (((uint64_t)a) >> (64-((n)%64)))) : a)
```

The macro is substituted by a 64-bit word containing `a` rotated by `n` positions to the left. Note that rotation is periodic: rotating a 64-bit word by 64 positions will yield the original word. A sample usage of the macro is shown in Figure 6.

```
uint64_t a = 1, b;
b = ROL64(a, 7);
```

Figure 6: `a` and `b` are two 64-bit unsigned integers. After the rotation, `b` will contain 128 (0x80).

4 Requirements

You must write your own code. Academic dishonesty and plagiarism are serious offenses at Aalto University. Submissions will be compared to the work of other students and what is available on the Internet.

You must fully document your code. By reading the comments, we should understand what small chunks of code do. More importantly, it tells us that you understand the code.

Your code must compile and run on the computing center linux machines (Intel x86-64). We've made it simple by providing skeletons, makefiles, and drivers. You are free to develop on whatever platform you want - but your submission must compile and run on the computing center Linux machines (*e.g.* `kosh`).

You must not use any non-standard library. You may only use standard C headers present on the computing center Linux machines (see previous point).

You must add your code to the `sha3.c` and `sha3.h` files. We will discard any modification made to the `sha3_driver.c` file.

You must implement the SHA3-256 function according to the prototype provided in the `sha3.h` file. You must not modify the provided `sha3()` function prototype.

Not fulfilling these requirements may result in your Assignment submission being assigned a failing grade.

5 Recommendations

We recommend that you carefully read Sections from 1 to 6 (included) of the standard, and take notes; we then recommend that you read all the provided code carefully. You should do this before writing any code.

You may use any text editor, compiler, build automation tool, or IDE of your choice. We recommend that you use a text editor (*e.g.* Vim, Emacs, Nano, Gedit), the GCC compiler[2], and the `make` build automation tool[3]. We provide a minimal working Makefile, which you may edit if necessary; we will use our own Makefile when grading, so any modifications you make will not persist.

We recommend that you follow the Linux kernel coding style[4]. Please write tidy code, with proper spacing, indentation and formatting.

You may use features of C introduced by the C99 standard[5]; this is supported by the provided Makefile.

Following these recommendations is not required, but may help you implement better code.

6 Returning the Assignment

Once you have completed the implementation of the Assignment, you can submit it in the appropriate section of the course page on MyCourses.

You should return at least the `sha3.c` and `sha3.h` files, which should contain all your code. You may also submit the `sha3_driver.c` and `Makefile` files, but realize that **any modifications you have made there will be discarded** for grading.

All files must be returned in an archive (`zip` or `tar.gz`), whose name must start with your student number (*e.g.* `123456_assignment1.zip`).

The deadline for the assignments is strict: this assignment is due on **Monday 31st of October at noon**, Helsinki time (GMT+3).

7 Grading

After you have submitted the Assignment, we will grade it. We will verify that the Assignment submission passes all tests (all `PASS`, no `FAIL`); we will also use some tests not provided in the `sha3_driver.c` file. If your Assignment submission passes all tests, we will examine the code in detail and analyse its quality.

The Assignment submissions will then be assigned one of three possible scores, depending on criteria shown in Table 1.

[2] https://gcc.gnu.org/
[3] https://www.gnu.org/software/make/manual/make.html
[4] https://www.kernel.org/doc/Documentation/CodingStyle
[5] http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf

0	The Assignment submission does not pass all tests. **OR** The Assignment submission does not compile on the computing center Linux machines. **OR** You have not submitted the Assignment. **OR** You did not write your own code (disciplinary action may follow).
1	The Assignment submission passes all tests.
2	The Assignment submission passes all tests. **AND** The Assignment submission is well implemented and structured, has no memory leaks.

Table 1: Possible scores for an Assignment submission

8 F.A.Q.

Q: I have never programmed in C, where can I find some documentation?

Many books will teach you how to program in C; you may find several in the library or online. However, please note that some basic programming knowledge (independently of the language) is necessary to follow this course.

Q: I am getting "segmentation fault" error messages. How can I fix the bug?

You may use tools such as Valgrind[6] and GDB[7] to debug your program.

Q: What can help me in my implementation?

The 1600-bit KECCAK-p permutation is best realised with lanes as 64-bit words. The C language has bitwise operators[8] which can be used for manipulating bits.

You can find several test vectors for the KECCAK-p[1600, 24] permutation in the provided `KeccakF-1600-IntermediateValues.txt` file. These describe the output of the permutation and intermediate values of the algorithm State between the various steps, given various inputs.

You can find several test vectors for the SPONGE[f, pad, r](N, d) function in the provided `KeccakSpongeIntermediateValues_SHA3-256.txt` file. These show the input messages, suffixes, initial state, padding, input and output of the SPONGE internal permutation.

[6] http://valgrind.org/

[7] https://www.gnu.org/software/gdb/

[8] http://www.tutorialspoint.com/cprogramming/c_bitwise_operators.htm

Q: How can I implement the rotation?

Rotation (to the left) of a 64-bit word can be obtained using the `ROL64` macro described in Section 3.5 of this document.

Q: In the description, some table indices are negative, what does that mean?

The indices are defined modulo 5.

Q: When should I start doing the assignment?

Now! The assignments are laborious, especially if you have no previous experience on C programming or cryptosystems.

Q: I'm stuck, what to do?

If you have questions regarding the assignments, you can ask the course staff by e-mail to filippo.bonazzi@aalto.fi. The farther from the deadline you email, the more likely it is you will get a timely answer. Getting an answer during the last weekend before the deadline is very unlikely (but you can always try, of course).

If you need more help, you can reserve some face-to-face time during office hours by sending an email to filippo.bonazzi@aalto.fi. Please note that course staff time is finite: do not wait until the last second to ask for a meeting if you need it.

References

[1] FIPS, "Secure hash algorithm-3 (SHA-3) standard: Permutation-based hash and extendable-output functions," *National Institute for Standards and Technology (NIST)*, vol. 202, 2015, doi:10.6028/NIST.FIPS.202