CS-E4320 Assignment 2: Ketje

Filippo Bonazzi
filippo.bonazzi@aalto.fi

November 1, 2016

Contents

1 Introduction

The assignment is to implement Ketje, an authenticated encryption primitive which is a third-round candidate in the CAESAR competition[1]. The specification of Ketje is available

[1] `competitions.cr.yp.to/caesar.html`

in the competition submission document ("the document"), which can be freely downloaded from the authors' website (ketje.noekeon.org/Ketjev2-doc2.0.pdf). The assignment is to implement **version 2.0 of Ketje**, which is described in the document above.

2 Definition of the Assignment

The task is to implement the Ketje Major authenticated encryption function, as defined in Section 5.1 of the document; implementation of the authenticated decryption function is not required.

In order to implement the Ketje Major authenticated encryption operation, you will need to implement the *initialize()* and *wrap()* functions of the MonkeyWrap construction, described in Section 4 of the document. You do not need to implement the *unwrap()* authenticated decryption function. To implement Ketje Major, you will need to use the following parameters for the MonkeyWrap construction:

- f, the inner permutation, equal to Keccak$-p^\star[b, n_r]$
- ρ, the block size, equal to 256
- n_{start}, the number of rounds for the *start* operation, equal to 12
- n_{step}, the number of rounds for the *step* operation, equal to 1
- n_{stride}, the number of rounds for the *stride* operation, equal to 6

To implement the required functions of the MonkeyWrap construction, you will need to implement the MonkeyDuplex construction, described in Section 3 of the document, with parameters:

- b, the width of the state, equal to 1600
- f, the inner permutation, equal to Keccak$-p^\star[b, n_r]$
- r, the rate, passed by MonkeyWrap as a parameter
- n_{start}, passed by MonkeyWrap as a parameter
- n_{step}, passed by MonkeyWrap as a parameter
- n_{stride}, passed by MonkeyWrap as a parameter

For use as the MonkeyDuplex inner permutation, you will need to implement the Keccak$-p^\star[b, n_r](s)$ permutation, described in Section 2 of the document, with parameters:

- b, the width, equal to 1600
- n_r, the number of rounds, an arbitrary value

- s, the input string, an arbitrary bit string of length 1600 bits

The KECCAK$-p^\star[b, n_r]$ permutation can be almost completely implemented by reusing code developed in Assignment 1: the only function you will need to implement from scratch will be the π^{-1} permutation, described in Section 2.1 of the document and in Section 8 of these instructions.

3 Provided code

In order to simplify the Assignment, we provide an implementation of several functions: `rc`, `pad10*1`, `cpynbits`, `concatenate`, `concatenate_00`, `concatenate_01`, `concatenate_10` and `concatenate_11`. The functions are not necessarily optimized for performance: they implement functionality as it is laid out in the document, 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`, and a macro to convert bit lengths in byte lengths, `BYTE_LEN`: you may find these useful in your implementation.

3.1 `rc`

The `rc` function implements the *rc(t)* algorithm defined in Section 3.2.5 of the SHA-3 standard. This function is the same which was provided in Assignment 1. The function implements a binary linear feedback shift register (LFSR) with outputs in $\mathbf{GF}(2)$. 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 1.

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

Figure 1: `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.

3.2 pad10x1

The `pad10x1` function implements the pad10*1(x, m) algorithm defined in Section 1.3 of the document. This function is the same which was provided in Assignment 1, **except for a bug that has been fixed in this new version**. 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 2.

```
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 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 equal to 8.

3.3 cpynbits

The `cpynbits` function implements the bit copy operation. The function copies n bits from a source buffer to a destination buffer; it is conceptually similar to the `memcpy` POSIX function, but it targets bits instead of bytes. The function has the following prototype:

```
void cpynbits(unsigned char *dst, unsigned int dst_o,
              const unsigned char *src, unsigned int src_o,
              unsigned int n);
```

dst points to the destination buffer allocated by the caller.

dst_o is the starting bit offset in the destination buffer.

src points to the source buffer allocated by the caller.

src_o is the starting bit offset in the source buffer.

n is the number of bits to copy.

The function can copy an arbitrary number of bits: `n` does not need to be a multiple of 8. The function can read and write starting at arbitrary bit offsets: `dst_o` and `src_o` do not need to be a multiple of 8.

A sample usage of the function is shown in Figure 3.

```
unsigned char dst[] = { 0x00, 0x00, 0x00 };
unsigned int dst_len = 24;
unsigned char src[] = { 0x01, 0x23, 0xff };
unsigned int src_len = 24;
cpynbits(dst, 8, src, 0, 8);
// dst is now { 0x00, 0x01, 0x00 }
cpynbits(dst, 16, src, 20, 4);
// dst is now { 0x00, 0x01, 0xf0 }
cpynbits(dst, 0, src, 16, 3);
// dst is now { 0x03, 0x01, 0x23 }
```

Figure 3: The `dst` array contains a bit string of length 24; after calling `cpynbits`, the content of `dst` will change as shown.

3.4 concatenate

The `concatenate` function implements the bit string concatenation operation, defined as $X||Y$ in Section 2.3 of the SHA-3 standard. This function is the same which was provided in Assignment 1, refactored to use the above `cpynbits` function. 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 4.

```
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 4: 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.

The concatenation is shown graphically in Figure 5.

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

Figure 5: Behaviour of the `concatenate` function

3.5 concatenate_xx

The `concatenate_00`, `concatenate_01`, `concatenate_10` and `concatenate_11` functions are special cases of the `concatenate` function described in the previous Section, where the second string is the 2-bit string `00`, `01`, `10` or `11` respectively. The functions have the following prototype:

```
unsigned long concatenate_xx(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 `concatenate_01` function is shown in Figure 6.

```
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 6: 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.

3.6 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 7.

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

Figure 7: `a` and `b` are two 64-bit unsigned integers. After the rotation, `b` will contain 128 (`0x80`).

3.7 BYTE_LEN

The BYTE_LEN macro converts a bit length in the corresponding byte length, rounding up: it produces the minimum integer number of bytes necessary to represent the given bits. The macro has the following definition:

```
#define BYTE_LEN(x) ((x/8)+(x%8?1:0))
```

The macro is substituted by an expression which evaluates to $x/8$ if the length of x is a multiple of 8, and $ceiling(x/8)$ otherwise. A sample usage of the macro is shown in Figure 8.

```
int a = 17, b = 80, c;
c = BYTE_LEN(a);
// c contains 3
c = BYTE_LEN(b);
// c contains 10
```

Figure 8: a, b and c are integers. After the first assignment, c will contain $ceiling(17/8) = 3$; after the second assignment, c will contain $80/8 = 10$.

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 ketje.c, keccak.c, ketje.h and keccak.h files. We will discard any modification made to the ketje_driver.c file.

You must implement the Ketje Major authenticated encryption function according to the prototype provided in the ketje.h file. You must not modify the provided ketje_mj_e() 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 5 (included) of the document, 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 `ketje.c`, `keccak.c`, `ketje.h` and `keccak.h` files, which should contain all your code. You may also submit the `ketje_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_assignment2.zip`).

The deadline for the assignments is strict: this assignment is due on **Monday 28th of November at noon**, Helsinki time (UTC+2).

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

[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

in the `ketje_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.

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: How can I implement the π^{-1} permutation?

The π^{-1} permutation required by KECCAK$-p^\star[b, n_r]$ is the inverse of the π permutation which you implemented in Assignment 1. It can be realized by inverting the matrix which describes the permutation: this is described in Section 2.1 of the document. In pseudocode, this means that the transformation shown in Step 1 of Algorithm 3 (π) in Section 3.2.3 of

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

the SHA-3 standard

$$\pi: \quad \mathbf{A}'[x, y, z] = \mathbf{A}[(x + 3y) \bmod 5, x, z]$$

will become

$$\pi^{-1}: \quad \mathbf{A}'[x, y, z] = \mathbf{A}[y, (2x + 3y) \bmod 5, z]$$

Q: What can help me in my implementation?

The C language has bitwise operators[8] which can be used for manipulating bits.

You can find several test vectors for the Ketje Major authenticated encryption operation in the provided `KetjeMj.txt` file. These describe the key, nonce, add. plaintex and plaintext tuple which result in a tag and ciphertext pair.

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 3.6 of this document.

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.

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