Introduction to cryptology
TP

2018-W11/12

Introduction

The goal of this "TP" lab session is to implement a simple keyed cryptographic function and to experimentally compute collisions in its (truncated) output. An optional second part considers a few additional functions built from a single unkeyed permutation with small state.

Instructions & grading

All functions and programs must be written in C or C++

The first part of this TP (up to question 5 included) is graded as *contrôle continu*. You must send a written report (in a portable format) detailing your answers to the questions, and the corresponding source code **with compilation and execution instructions** by April 6 (2018-04-06T23:59+0200) to:

pierre.karpman@univ-grenoble-alpes.fr.

Working in teams of two is allowed (but not mandatory), in which case only one report needs to be sent, with the name of both students clearly mentioned.

By default, this mandatory contrôle continu grade (*MCC*) will count for one third of the final grade for this course, the other two thirds being the grade of a final exam (*FE*). The second part of this TP (question 6 to 11) is optional, and will be graded separately, starting from zero (for instance, only answering question 6 will only result in a very poor grade *for this part*). This optional contrôle continu grade (*OCC*) can *only increase* the final grade for this course, and will be incorporated using the formula:

$$\text{Final grade} = \max((MCC + 2FE)/3, (MCC + OCC + 2FE)/4)$$

Part 1 (mandatory)

Question 1

Write a function implementing SipHash-2-4 [AB12]. This function must have the following prototype:

```
uint64_t siphash_2_4(uint64_t k[2], uint8_t *m, unsigned mlen);
```

You may (and should) test your implementation by comparing its outputs with the ones provided in [AB12, Appendix A], and the following, with `k2 = {0, 0}` and `m2 = {0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7}`:

- `siphash_2_4(k, m2, 8)    == 0x93f5f5799a932462`
- `siphash_2_4(k, NULL, 0)  == 0x726fdb47dd0e0e31`
- `siphash_2_4(k2, NULL, 0) == 0x1e924b9d737700d7`

Question 2

Do you think that the key size of `sip_hash_2_4` is large enough to make an exhaustive search of the keyspace intractable? Can this function be considered to be collision-resistant?

The goal of the following questions is to experimentally compute collisions for a variant of SipHash that has its output truncated to its 32 least significant bits. While computing collisions for the original function would still be possible in a reasonable time, it would either require a large amount of memory or a memoryless implementation of collision search that is beyond the scope of this exercise.

Question 3

Write a function `sip_hash_fix32` that "simplifies" SipHash by truncating its output to 32-bit, by using a shortened 32-bit key and by having a fixed-size 32-bit message input. The prototype of this function is:

```
uint32_t sip_hash_fix32(uint32_t k, uint32_t m)
```

You are free to choose how to map the first argument `k` to a 128-bit key for the original SipHash and how to map `m` to a byte string, as long as those mappings are injective.

Question 4

Write a function `coll_search` of prototype:

```
uint64_t coll_search(uint32_t k, uint32_t (*fun)(uint32_t, uint32_t))
```

This function takes as input a function pointer `fun` for a function that has the same type as `sip_hash_fix32`, and a 32-bit key `k`. It must return as output the minimum size (minus 1) of the sequence `(fun(k,i))` (indexed by i) such that two of its element collide. For example, if `fun` and `k` are such that `fun(k,0)`, `fun(k,1)`, ..., `fun(k,74221)` are all distinct but `fun(k,74222)` = `fun(k,10394)`, then `coll_search(k, &fun)` must return 74222.

This function must be reasonably fast: it should not take more than a few dozen milliseconds to return. However, in order to simplify things, your implementation only needs to work for "random-looking" functions. For instance, it is fine if it fails (or even crashes) when `fun(k,.)` is a permutation.

Advice. Before starting an implementation, try to estimate what the expected output should be for a random function. Is that value small enough to let you store the entire sequence (for instance as a list) up to finding a collision? Would a quadratic search through an unsorted list of that size be fast enough?

One possible way (but not the only one) of storing such a sequence in this case is to use a hash table. Assuming that the sequence elements are random, how can you "hash them" very efficiently? Try to guess the number and the size of buckets that you need, and make adjustements if necessary based on experiments.

Note. When writing your report, be particularly careful when explaining your implementation choices for this question.

Question 5

Gather statistics about the expected time (as a number of function calls) necessary to get a collision for `sip_hash_fix32` "in counter mode" for a fixed key. In particular, give the average, minimum, and maximum time over 1000 (or more) distinct keys. Are these results consistant with your initial estimate?

Part 2 (optional)

Question 6

Write a function `twine_perm_z` that implements the permutation of TWINE given in [SMMK12, Algorithm 2.1] where all the round keys $\mathrm{RK}^i_{(j)}$ are set to zero[1]. This function must have the following prototype:

```
uint64_t twine_perm_z(uint64_t input);
```

We will use the convention that the 16-nibble input is parsed as following (you may use a different type for `X`):

```
unsigned X[16];
for (int i = 0; i < 16; i++)
{
        X[i] = (input >> (4*i)) & 0xF;
}
```

The reverse process is then used to create the output as a 64-bit string.

You may (and should) test your implementation by comparing its outputs to the following known values:

- `twine_perm_z(0x0000000000000000ULL) == 0xb0049660a2858d43`
- `twine_perm_z(0x123456789abcdef1ULL) == 0x00de04856ecd7ad0`
- `twine_perm_z(0xb4329ed38453aac8ULL) == 0xd0790f39b4d2ecab`

Question 7

Use the function `twine_perm_z` to define a small cryptographic function of prototype:

```
uint32_t twine_fun1(uint32_t k, uint32_t m);
```

defined as the truncation to the 32 least significant bits of the output of `twine_perm_z` called on the concatenation of `k` and `m`, where the former occupies the most significant 32 bits. You may (and should) test your implementation by comparing its outputs to the following known values:

- `twine_fun1(0x00000000, 0x00000000) == 0xa2858d43`
- `twine_fun1(0xcdef1234, 0xab123478) == 0x1b234285`

Question 8

Do you think that the key size of `twine_fun1` is large enough to make an exhaustive search of the keyspace intractable? Can this function be considered to be collision-resistant?

Question 9

Gather statistics about the expected time (as a number of function calls) necessary to get a collision for `twine_fun1` "in counter mode" for a fixed key. In particular, give the average, minimum, and maximum time over 1000 (or more) distinct keys. How do these results compare with `sip_hash_fix32`?

[1] This change has a side-effect that the resulting permutation is vulnerable to *slide attacks*. This is however not problematic in the case of this exercise.

Question 10

The previous function only takes a fixed-size message as input. We wish to extend it to messages of arbitrary length by splitting the message input in two, using one half for a chaining value.

More precisely, let $\mathrm{f} : \{0,1\}^{32} \times \{0,1\}^{16} \times \{0,1\}^{16} \to \{0,1\}^{32}$ and `iv = 0xFFFF`, one recursively defines $\mathrm{F} : \{0,1\}^{32} \times \{0,1\}^{16 \cdot \ell} \to \{0,1\}^{32}$ for any $\ell > 0$ by:

- $\mathrm{F}(k, m_0) = \mathrm{f}(k, \mathtt{iv}, m_0)$
- $\mathrm{F}(k, (m_0, m_1, \ldots, m_i)) = \mathrm{f}(k, \lfloor \mathrm{F}(k, (m_0, m1, \ldots, m_{i-1})) \rfloor_{16}, m_i)$

with $\lfloor \cdot \rfloor_{16}$ denoting truncation to the 16 least significant bits.

Write the following functions:

- `uint32_t twine_fun2(uint32_t k, uint16_t *m, unsigned mlen)`
- `uint32_t twine_fun2_fix32(uint32_t k, uint32_t m)`
- `uint32_t twine_fun2_fix16(uint32_t k, uint32_t m)`

where `twine_fun2` is an instantiation of the above construction for F, using `twine_fun1` for f as $\mathrm{f}(k, x, y) =$ `twine_fun1(k,(x << 16) ^ y)`, `twine_fun2_fix32` is a simplified instance with a fixed 32-bit message, and `twine_fun2_fix16` is a simplified instance with a fixed 16-bit message. Note that this last function still uses a type `uint32_t` for its message argument, despite the fact that it should only be 16-bit long. This is to allow using this function as argument to `coll_search`.

You may (and should) test your implementation by comparing its outputs to the following known values, with `m1 = {0x67FC}`, `m2 = {0xEF12, 0x5678}`, and `m3 = {0xEF12, 0x5678, 0x31AA, 0x7123}`:

- `twine_fun2(0x00000000, m1, 1) == 0xf844b17f`
- `twine_fun2(0x23AE90FF, m2, 2) == 0x7f870480`
- `twine_fun2(0xEEEEEEEE, m3, 4) == 0x96003194`

Question 11

Gather statistics about the expected time (as a number of function calls) necessary to get a collision for `twine_fun2_fix32` and `twine_fun2_fix16` "in counter mode" for a fixed key. In particular, give the average, minimum, and maximum time over 1000 (or more) distinct keys. How do these results compare with `sip_hash_fix32` and `twine_fun1`? How can you explain these results?

References

[AB12] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A Fast Short-Input PRF. In Steven D. Galbraith and Mridul Nandi, editors, *Indocrypt 2012*, volume 7668 of *Lecture Notes in Computer Science*, pages 489–508. Springer, 2012. Available at https://eprint.iacr.org/2012/351.

[SMMK12] Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka, and Eita Kobayashi. TWINE : A Lightweight Block Cipher for Multiple Platforms. In Lars R. Knudsen and Huapeng Wu, editors, *SAC 2012*, volume 7707 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2012. Available at https://www-ljk.imag.fr/membres/Pierre.Karpman/cry_intro_tp_twine.pdf.