TP – Multicollisions and expandable messages

Notations.
Let $f : \{0,1\}^n \times \{0,1\}^w \to \{0,1\}^n$ be a compression function, with $n \leq w$, and let IV be some fixed initial value. For a message $\hat{m} = m_1 \| \cdots \| m_B$ of length $B \times w$, let $h_0 = IV$ and $h_i = f(h_{i-1}, m_i)$ for $i \geq 1$. Then we define F by $F(h_0, m_1 \| \cdots \| m_i) = h_i$ for all i. In particular, $F(h_0, m) = h_B$.
For a message $m \in \{0,1\}^*$, let $\text{pad}(m) = m \| 10 \cdots 0 \| \langle \text{length of } m \rangle$ be the padded version of m where the number of zeroes is adjusted to have $|\text{pad}(m)| = B \times w$ for some B. Then we define $H(m) = F(IV, \text{pad}(m))$.

Finding a multicollision.
To find a 2^t-multicollision for a hash function H, the method is as follows. Let h_0 be the IV of h. Then, for $i = 0$ to $t-1$ (in that order), find a collision $h_{i+1} = f(h_i, m_i^0) = f(h_i, m_i^1)$ by sampling random blocks m_i^0, $m_i^1 \in \{0,1\}^w$. Then, each message $m \in \mathcal{M} = \{m_0^{b_0} \| \cdots \| m_{t-1}^{b_{t-1}} : b_0, \ldots, b_{--1} \in \{0,1\}^t\}$ satisfy $F(h_0, m) = h_t$. Therefore, since they have the same number of blocks, they form a multicollision for H as well since $H(m) = F(h_0, \text{pad}(m))$.

Finding an expandable message.
To find an expandable message for F, we need the messages to have pairwise distinct lengths. We slightly modify the approach: At each step, instead of looking for a collision $f(h_i, m_i^0) = f(h_i, m_i^1)$, we look for a collision $h_{i+1} = f(h_i, m_i) = F(h_i, m^i)$ where m_i is one block and m^i is a message of ℓ_i blocks. To this end, we can fix the first $(\ell_i - 1)$ blocks of m^i and only sample its last block. Finally, we build the set $\mathcal{M}_{\text{exp}}$ of messages by choosing, for each i, either m_i or m^i. If $\ell_i = 1 + 2^i$, $\mathcal{M}_{\text{exp}}$ contains messages of lengths t to $t + 2^t - 1$.

Instructions for the implementation.

1. Download the tarball `https://membres-ljk.imag.fr/Bruno.Grenet/IntroCrypto/mc.tar.bz2`. It contains the following files:
   - `mc48.h` defines a compression function `tcz48_dm` with 128-bit message blocks and 48-bit hashes, and the associated Merkle-Damgård hash function `ht48`;
   - `mc48.c` contains the implementations;
   - `xoshiro256starstar.h` defines a pseudo-random number generator to be used in your program.

   You are not allowed to modify any of the given files.
2. Your file is supposed to compile without any warning, without esoteric invokations. Use the compiler that you prefer (`gcc`, `clang`, ...) as long as it sticks to the standard.
3. You can only use the C standard library. No other external library or software can be used.
4. Beyond the explicitly requested functions, you can (and are encouraged to) write any other function you need. In particular, it is good practice to avoid huge monolithic functions and to comment your code.

Exercise 1. *Warm-up*

1. Identify which functions implement f and H, and the values of n and w.
2. Write a function

   ```
   void iterated_tcz48_dm(const uint8_t *m, const size_t len, uint8_t h[6])
   ```

   that implements the function F, where `len` denotes the byte-length of `m`. The result is written in the variable `h` at the end of the function. You must check that the byte-length is correct, that is a multiple of w, and do nothing otherwise.
3. Write a function

   ```
   void print(const uint8_t *x, const size_t len, const char* s)
   ```

   that writes `x` to the standard input as a string of hexadecimal characters followed by `s`, where `len` is the byte-length of `x`. *Example of output:* `228E102A86FF`.

[1] This subject is mainly due to Pierre Karpman.

Exercise 2. *Multicollisions*

1. Write a function

   ```
   double collision(uint8_t h[6], uint8_t m1[16], uint8_t m2[16])
   ```

   that finds a collision of the form $f(h, m1) = f(h, m2)$, using the following remarks and instructions:

   i. According to the birthday bound, how many samples do you need to find a collision? Which space do you need? Choose a data structure to store the samples, and implement it. *You are not allowed to rely on external software or library for the data structure. Do it yourself!*
   ii. The parameter `h` contains the hash of the colliding blocks `m1` and `m2` at the end of the function. The function returns the number of samples used during the computation.
   iii. The function writes on the standard output the two colliding blocks, as well as $\log_2(N)$ where N is the number of samples before a collision is found. *Example of output:*

   ```
   5A2F65758D839026D00B5169DA59FD90 DC747E10D731C723724F10592FEEAFB3 22.096361.
   ```

   iv. This question should not require more than a few dozens of lines of codes. On my (average) laptop, the running time to find a collision is around 30 seconds. *Use optimization flags!*

2. Write a function

   ```
   double multicollision(int t)
   ```

   that computes a 2^t -multicollision for H:

   i. The function prints the colliding messages on the standard output, as well as the common hash value.
   ii. It returns the total number of samples used for the computation.
   iii. Test your function with $t = 1$ to 4 and compare the running times with the theoretical complexity.

Exercise 3. *Expandable message*

1. Write a function

   ```
   double unbalanced_collision(uint8_t h[6], uint8_t m1[16], uint8_t* m2, const size_t len)
   ```

   that finds a collision of the form $f(h, m1) = F(h, m2)$ where `m2` has `len` blocks:

   i. Recall that we can fix the first blocks of `m2`: this function should be really close to `collision`, and you can in particular reuse the data structure for standard collisions.
   ii. All the remarks for `collision` apply to the current function.

2. Write a function

   ```
   double expandable_message(int t)
   ```

   that computes an expandable message for F of 2^t messages of length t to $t + 2^t - 1$:

   i. The function writes the message on the standard output, ordered by size, as well as the common hash value.
   ii. It returns the total number of samples used for the computation.
   iii. Test your function with $t = 1$ to 4.