TP – Merkle-Damgård second preimage attack using fixed points

We implement the second preimage attack of Kelsey and Schneier against a Merkle-Damgård hash function, using fixed points. The compression function used to build the hash function is a Davies-Meyer compression function based on a block cipher from the SPECK family.

General instructions.

- A tarball with the required files, that you shouldn't modify, is here: `https://membres-ljk.imag.fr/Bruno.Grenet/IntroCrypto/26/fp.tar.bz2`.
- For you own implementations, **do not use any external library beyond the C standard library** (`stdxxx.h` and a few others such as `math.h`, `string.h`, ...).[1] In particular, `memcpy` and `memcmp` from `string.h` can be useful.
- 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.

LLMs are most probably able to produce the code for this assignment since there are solutions to these or very close exercises on the internet. It is the same as copy-pasting: This is silly and you won't learn anything! The (very limited) bonus on the final grade is not worth it...

Content of tarball.

- The files `utils.h` and `utils.c` contain functions and macros to manipulate byte arrays:
  - type definition `byte` for `uint8_t`;
  - `void random_bytes(byte* array, size_t len)` fills `len` bytes of `array` with (pseudo-)random values;
  - `void print_bytes(const byte* array, size_t len)` prints the `len` first bytes of `array`, in big-endian order: `array[len-1]array[len-2]...array[0]`;
  - `void read_bytes(char* str, byte *array, size_t len)` fills `len` bytes of `array` with the hexadecimal string `str` (adding zeroes if `str` is too short).
  - `void speck_enc(const byte k[KLEN], const byte m[MLEN], byte c[MLEN])`
    `void speck_dec(const byte k[KLEN], byte m[MLEN], const byte c[MLEN])`
    implement three variants of the block cipher SPECK, with block size $2n$ and key size n, for $n = 32$, 48 and 64 respectively. The variant is chosen at compilation time using the macro `BLOCKSIZE` (*cf.* Exercise 1). The macros `KLEN` and `MLEN` contain respectively the byte-lengths of the keys (8, 12 or 16) and of the messages (4, 6 or 8).
- `test_speck.c` is a test file, that can serve as an example to write your own test files: It creates a random key and a random message block, encrypt the message block, and then decrypts it.
- Header files for the exercises are also provided: `hash.h` and `attack.h`.
- A `Makefile` contains some rules.

Exercise 1. *Warm up*

1. Download and extract the tarball.
2. Compile and execute `test_speck.c` using the `Makefile`:

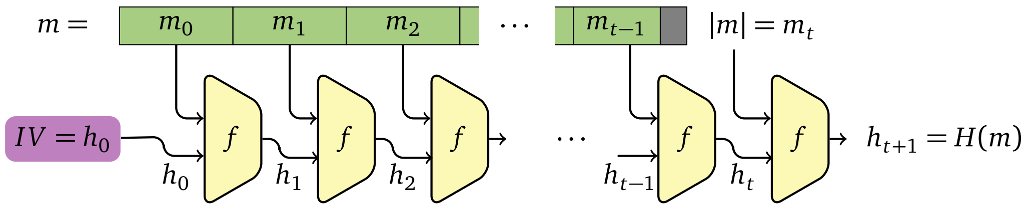
```
make test_speck BLOCKSIZE=<n>
```

   where `<n>` is 32, 48 or 64 depending on the variant of SPECK to use. Try the three variants.

[1] Full list on Wikipedia: `https://en.wikipedia.org/wiki/C_standard_library`.

Exercise 2. *Construction of the hash function*

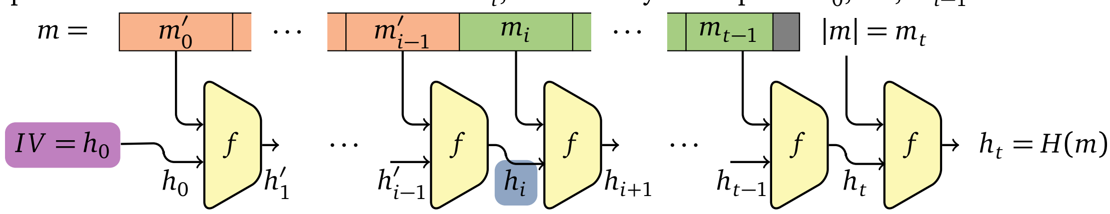
- We work with a *block cipher* $E : \{0,1\}^{2n} \times \{0,1\}^n \to \{0,1\}^n$ from the Speck family, where $n = 32$, 48 or 64. It has key size $2n$ and block size n.
- The block cipher is used to build a *compression function* $f : \{0,1\}^n \times \{0,1\}^{2n} \to \{0,1\}^n$ using the Davies-Meyer construction: $f(h,m) = E_m(h) \oplus h$. *Beware: The message block in the compression function is used as the key in the block cipher!*
- The compression function is used to build a *hash function* $H : \{0,1\}^* \to \{0,1\}^n$ using the Merkle-Damgård construction with IV `0x03020100` ($n = 32$), `0x050403020100` ($n = 48$) or `0x0706050403020100` ($n = 64$):



1. Write the file `hash.c`.
   - **i.** Implement a function `void compression(byte h[HLEN], const byte m[BLEN])` that computes $f(h,m)$, overwriting the input `h` with the output.
   - **ii.** Implement a function `void hash(const byte *m, size_t len, byte h[HLEN])` that computes $H(m)$ into `h`. *Do not forget to pad the input message m! The last block is the bit-length of* `m`*.*
   - **iii.** Implement a function `void intermediate_digests(const byte *m, size_t len, byte *h)` that computes $H(m)$ and *stores all the intermediate digests* $h_1, \ldots, h_{t+1}$ *in* `h`.
2. Write the file `test_hash.c`, to test each of the three functions.
   - Try with the three possible block sizes $n = 32$, 48 and 64.
   - Expected results are provided beginning on page 3.

Exercise 3. *Second preimage attack on the hash function*

Given a message $m = m_0 \| \cdots \| m_{t-1}$, the second preimage attack of Kelsey and Schneier computes a message $m' = m'_0 \| \cdots \| m'_{i-1} \| m_i \| \cdots \| m_{t-1}$ such that $H(m') = H(m)$. The blocks $m'_0, \ldots, m'_{i-1}$ are computed to reach the intermediate result h_i, so that they can replace $m_0, \ldots, m_{i-1}$ in m as follows:



The attack works as follows:

1. Find two blocks m_s and m_f such that $f(h_0, m_s) = E^{-1}_{m_f}(0) =: h_f$;
2. Find a block m_ℓ such that $f(h_f, m_\ell) = h_i$ for any $i \in \{1, \ldots, t-1\}$;
3. Build $m'_0 \| \cdots \| m'_{i-1} = m_s \| m_f \| \cdots \| m_f \| m_\ell$.

Justification. By construction, $f(h_0, m_s) = f(h_f, m_f) = h_f$ (check!) and $f(h_j, m_\ell) = h_i$. Therefore, $H(m') = H(m_s \| m_f \| \cdots \| m_f \| m_\ell \| m_i \| \cdots \| m_{t-1}) = H(m)$.

1. Write the file `attack.c` (starting with Step 2, easier than Step 1). *The three functions return the number of blocks sampled during the computation.*
   - **i.** Step 2: To compute m_ℓ, the technique is to sample blocks until one is found such that $f(h_f, m_\ell) = h_i$ for some i. Implement this as a function `double linkmsg(byte ml[BLEN], int *i, const byte hf[HLEN], const byte *h, size_t len)`.

ii. Step 1: To compute h_f, m_s and m_f, the technique is to sample blocks m_s and m_f until a collision $f(h_0, m_s) = E_{m_f}^{-1}(0)$ is found for some m_s and m_f. Implement this as a function `double collision(byte ms[HLEN], byte mf[HLEN], byte hf[HLEN])`. *Think carefully at the data structure(s) to use. Don't rely on external library, implement the data structure by yourself!*

iii. Step 3: Write a function `double attack(const byte *m, size_t len, byte *m2)` that computes a message m_2 such that $H(m) = H(m_2)$.

2. Write the file `test_attack.c` to test each of the three functions.

- Why do you need large messages to make the attack work in reasonable time? And what is *large* more precisely?
- Print the approximate number of samples used as power of 2 (*cf.* examples).
- Reasonably well-written code should run in (much) less than a second for $n = 32$ and around a minute for $n = 48$. The case $n = 64$ is doable but requires some clever implementations: *the challenge is to produce a second preimage for $H(0^{32})$ in this case.*
- Examples of outputs are provided beginning on page 5.

Examples of outputs for the hash function

```
==========================
Message size: 32 (4 bytes)
Key size: 64 (8 bytes)
==========================

*** SPECK
Key:             k = 0123456789abcdef
Message block:   m = db608390
Ciphertext block: c = 61cbb984

*** COMPRESSION
m        = 0123456789abcdef
h0       = 03020100
f(h0,m)  = 40fd37ca

*** HASH
Message: m = 01
Digest:  h = 785d2349
Intermediate digests:
h[0]  = 03020100
h[1]  = 0ee32567
h[2]  = 785d2349

Message: m = 0123456789abcdef
Digest:  h = c07a5a89
Intermediate digests:
h[0]  = 03020100
h[1]  = 40fd37ca
h[2]  = c07a5a89

Message: m = 0123456789abcdef0123456789abcdef
```

```
Digest:  h = 3572eb4f
Intermediate digests:
h[0] = 03020100
h[1] = 40fd37ca
h[2] = e0837e37
h[3] = 3572eb4f

==========================
Message size: 48 (6 bytes)
Key size: 96 (12 bytes)
==========================

*** SPECK
Key:              k = 0123456789abcdef01234567
Message block:    m = dad97e7053ea
Ciphertext block: c = 0858f63c61e4

*** COMPRESSION
m       = 0123456789abcdef01234567
h0      = 050403020100
f(h0,m) = 4e33de573ce0

*** HASH
Message: m = 01
Digest:  h = a542af392656
Intermediate digests:
h[0] = 050403020100
h[1] = 77fbcc225c1b
h[2] = a542af392656

Message: m = 0123456789abcdef01234567
Digest:  h = 9310a0fa68ca
Intermediate digests:
h[0] = 050403020100
h[1] = 4e33de573ce0
h[2] = 9310a0fa68ca

Message: m = 0123456789abcdef0123456789abcdef0123456789abcdef
Digest:  h = 08af35ed77c4
Intermediate digests:
h[0] = 050403020100
h[1] = 1cd0a5cb710e
h[2] = eeae6729cdae
h[3] = 08af35ed77c4

==========================
Message size: 64 (8 bytes)
Key size: 128 (16 bytes)
==========================
```

```
*** SPECK
Key:              k = 0123456789abcdef0123456789abcdef
Message block:    m = 8c8f4a901e18661a
Ciphertext block: c = 1a1a73e7a6467b53

*** COMPRESSION
m       = 0123456789abcdef0123456789abcdef
h0      = 0706050403020100
f(h0,m) = 2364af5f1011c178

*** HASH
Message: m = 01
Digest:  h = 73d0d70785c8e734
Intermediate digests:
h[0] = 0706050403020100
h[1] = 9e18a5e2d511f2f9
h[2] = 73d0d70785c8e734

Message: m = 0123456789abcdef0123456789abcdef
Digest:  h = 33072820a622e226
Intermediate digests:
h[0] = 0706050403020100
h[1] = 2364af5f1011c178
h[2] = 33072820a622e226

Message: m =
 ↪ 0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef
Digest:  h = cb428f9bf76088fa
Intermediate digests:
h[0] = 0706050403020100
h[1] = 2364af5f1011c178
h[2] = 2e80f1a1430ed47e
h[3] = cb428f9bf76088fa
```

Examples of outputs for the attack

```
===========================
Hash size: 32 (4 bytes)
Block length: 64 (8 bytes)
==========================

*** Collision
Collision found using approx. 2^13.646559 samples:
  ms = a187d954b5aba524
  mf = 8e391d08e015aff6
  hf = fe335fd0

*** Link message
Link msg found using approx. 2^15.776253 samples:
```

```
  m = 7d4f956d15e8440c
f(fe335fd0,7d4f956d15e8440c) = 0eb4d05a
Intermediate digest h[3642] = 0eb4d05a

*** Full attack
Attack using approx. 2^19.317640 samples:
H(m) = 97fa6890
H(m2)= 97fa6890
m = 4c2cb0be5eedf79be3a6192386bed7b4d387fb87206b772772d19acf5135bdc3b55c7
 ↪  540faa74cca ... (65526 more bytes)...
 ↪  26bb667f561a1462d0ac6081f11361846573fb9509ce796b8da5dc25b667aedc33e06
 ↪  95e9f1dcd98
m2= eab248c42ea90067eab248c42ea90067eab248c42ea90067eab248c42ea90067d3cc4
 ↪  4a2e93317ef ... (65526 more bytes)...
 ↪  26bb667f561a1462d0ac6081f11361846573fb9509ce796b8da5dc25b667aedc33e06
 ↪  95e9f1dcd98


===========================
Hash size: 48 (6 bytes)
Block length: 96 (12 bytes)
===========================

*** Collision
Collision found using approx. 2^24.214187 samples:
  ms = 372fff707022b7bac0437771
  mf = b687684f030fa3b17cda457a
  hf = 76a91a64f387

*** Link message
Link msg found using approx. 2^25.656222 samples:
  m = e90644dbdd32994b68a44658
f(76a91a64f387,e90644dbdd32994b68a44658) = 50fc724bcf94
Intermediate digest h[199442] = 50fc724bcf94

*** Full attack
Attack using approx. 2^25.892416 samples:
H(m) = ebdbb16f8e6e
H(m2)= ebdbb16f8e6e
m = 53497c2b0d753a88a7bc789fd38d476eb0835b209dba5e14c5d6c8f14e5bf6644f40a
 ↪  621746e80eb0dc612263612690588c0cc76bde105bbc11b8cd5 ... (16777206 more
 ↪  bytes)... 17d20219c7435d744bdf7f8d61de7217f2485d0702adb3456d5bf41abac
 ↪  ef91e235f7a94d61bf86fff39c20448ac232b47a0386969ba22b6f79bd654
m2= db51d1677ec1a55ef9bb7e36db51d1677ec1a55ef9bb7e36db51d1677ec1a55ef9bb7
 ↪  e36db51d1677ec1a55ef9bb7e3651f12b2f4459b1750d349c6a ... (16777206 more
 ↪  bytes)... 17d20219c7435d744bdf7f8d61de7217f2485d0702adb3456d5bf41abac
 ↪  ef91e235f7a94d61bf86fff39c20448ac232b47a0386969ba22b6f79bd654
```