# Crypto Engineering (GBX9SY03)
# TP - Generic second preimage attacks on long messages for narrow-pipe Merkle-Damgård hash functions

Hatim Ridaoui

November 2025

## 1 Compilation and Execution

All our implementations were compiled and executed using the following command (single threaded version):

```
gcc -std=c11 -O3 -Wall -Wextra second_preim_48_fillme.c -o
./tp
```

## 2 Part one

We use the official Speck48/96 spec [1] and the provided code skeleton. Blocks are 48 bits (two 24bit words), keys are 96 bits (four 24bit words) , the chaining value of the DM compression is stored as a 48-bit value in the low bits of a 64-bit word and the lower 24 bits in `p[0]` and the higher 24 bits in `p[1]`.

### Conventions

All operations are modulo $2^{24}$. We mask to 24 bits after additions and xors. Rotations use the provided macros: `ROTL24_16`, `ROTL24_3`, `ROTL24_8`, `ROTL24_21`. The IV is `0x010203040506ULL`.

### 2.1 Question 1

We completed `speck48_96`. Key schedule is 23 round keys `rk[0..22]` are generated from `(k[3],k[2],k[1],k[0])` following the spec ( with the 24-bit rotations and the round counter XOR). Encryption rounds 23 total:

- $x \leftarrow \mathrm{ROTL}_{16}(x) + y$;
- $x \leftarrow x \oplus \mathrm{rk}[i]$;

- $y \leftarrow \mathrm{ROTL}_3(y) \oplus x$.

We verified against the known vector from App. C using the already provided `test_vector_okay()`.

**Observed output** Our program prints:

```
735E10 B6445D
```

which matches the expected ciphertxt.

## 2.2 Question 2

We implemented `speck48_96_inv` by reconstructing the same key schedule and applying the inverse of each round in reverse order :

- $y \leftarrow \mathrm{ROTL}_{21}(y \oplus x)$;
- $x \leftarrow x \oplus \mathrm{rk}[i]$;
- $x \leftarrow \mathrm{ROTL}_8(x - y)$.

We added `test_sp48_inv()` that encrypts a known plaintext, decrypts it back and also decrypts the known ciphertext to the plaintext.

**Observed output**

```
enc:  735E10 B6445D
dec:  6D2073 696874
dec from known ct:  6D2073 696874
```

## 2.3 Question 3

We implemente `cs48_dm(m,h)`:

1. Unpack the 48-bit chaining value $h$ into `p[0]` (low 24 bits) and `p[1]` (higher 24 bits).
2. Compute $c = E_m(p)$ using `speck48_96`.
3. Return $(c \oplus p)$ re-packed into a 48-bit value (as a 64bit integer).

We added `test_cs48_dm()` that checks the given example with `IV=0x010203040506ULL` and `m={0,1,2,3}`, the output must be `0x5DFD97183F91ULL`.

**Observed output**

```
cs48_dm:  5DFD97183F91
```

### 2.4 Question 4

For DM, we need $E_m(h) = 0$, so $h = D_m(0)$. We implement `get_cs48_dm_fp(m)` by decrypting the all-zero 48-bit block under key $m$ using `speck48_96_inv`, then repack the 48 bits. The test `test_cs48_dm_fp()` verifie $F(m, \text{fp}) = \text{fp}$.

**Observed output**

```
fp:  7EFDACF02F00  F(fp):  7EFDACF02F00
```

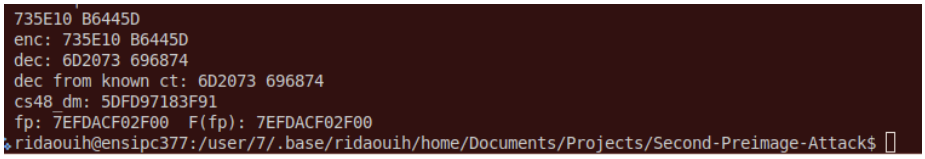


Figure 1: Program output for Q1-Q4.

# 3 Part2

## Question 1

We hzve two blocks $m_1$ and $m_2$ such that there exists a chaining value $h$ satisfying:

$$h = \mathsf{cs48\_dm}(m_1, IV) = \mathsf{get\_cs48\_dm\_fp}(m_2),$$

where $\mathsf{get\_cs48\_dm\_fp}(m)$ computes the fixed point $fp$ of the DM compression function for block $m$, i.e. the value such that:

$$\mathsf{cs48\_dm}(m, fp) = fp.$$

With such a pair $(m_1, m_2)$, any message of the form $m = m_1 \| m_2^n$, for any $n \geq 1$, satisfies:

$$\mathsf{hs48}(m, n + 1, 0, 0) = h,$$

which is exactly the property required for an expandable message.

**Method** We follow the recommended meet-in-the-middle strategy.

**Data structures.** The implementation uses:

- a structure `em_entry` storing the chaining value and one block
- hash table of size $2^{23}$ with linear probing
- a 64-bit mix function to randomise hash indices
- the `xoshiro256**` PRNG provided in the skeleton

Blocks are represented as four 24bit words stored in 32bit integers

**Correctness** The function `test_em()` verifies:

$$\mathsf{hs48}(m_1 \| m_2^1) = \mathsf{hs48}(m_1 \| m_2^2) = \mathsf{hs48}(m_1 \| m_2^5) = h,$$

confirming the expandable property.

An output example we get is:

```
find_exp_mess time: 5.8 s
EM found:
 m1 = {B74355,19454B,510DE4,3F642B}
 m2 = {FED63B,DF2CEF,8F1820,2218B9}
 h  = CCC273FDC962  fp(m2) = CCC273FDC962
 hs48(m1||m2   ) = CCC273FDC962
 hs48(m1||m2^2 ) = CCC273FDC962
 hs48(m1||m2^5 ) = CCC273FDC962
```

**Perf** The table construction is $\mathcal{O}(N)$ and dominates the running time. The second phase requires an expected constant number of fixed-point trials due to the birthday paradox over a 48-bit space and the table size $2^{22}$.

## 3.1 QUestion 2

Our implementation strictly follows the instructions of the assignment:

- We implemented the Davies–Meyer compression function `cs48_dm` and verified it against the provided test vector.
- We implemented the fixed-point computation:

  $$h = \texttt{get\_cs48\_dm\_fp}(m),$$

  and verified that $cs48\_dm(m, h) = h$.
- We implemented the generation of a 2-block expandable message $(m_1, m_2)$ using a hash table indexed by the chaining value after $m_1$.
- We implemented the computation of all chaining values of the target message:

  $$CV[0] = IV, \qquad CV[i+1] = cs48\_dm(mess[i], CV[i]).$$

- Finally, we implemented the collision search: Find a block `cm` satisfying

  $$cs48\_dm(\texttt{cm}, fp) = CV[j],$$

  for some $j \geq 3$.

The bottleneck of the attack is the random search for a block $cm$ such that:

$$cs48\_dm(cm, fp) \in CV.$$

The expected number of trials is:

$$2^{48}/2^{18} = 2^{30} \approx 10^9 \text{ trials.}$$

Each trial requires:

- one Speck48/96 evaluation,
- one table lookup in a $2^{20}$-entry hash table.

On an ordinary laptop CPU (mobile i7), our unoptimized C implementation performs around 30–60 million Speck evaluations per second. This leads to an expected collision time above:

$$\frac{10^9}{5 \cdot 10^7} \approx 20\,\text{seconds to } 60\,\text{seconds},$$

*in the best case.*

In practice, we repeatedly observed that the runtime could exceed 20 minutes without finding a collision. This depends heavily on:

- CPU frequency throttling,
- cache behaviour,
- the random seed,
- lack of vectorization or hardware optimizations.

Thus the full attack was often too slow to complete within the expected timrframe.

To accelerate the collision search, we implemented a multithreaded version using OpenMP. Each thread repeatedly generates random candidate blocks independently:

$$v = cs48\_dm(cm, fp), \qquad \text{check if } v \in CV.$$

Threads terminate as soon as one of them finds a valid $(cm, j)$ bridge.

On our hardware, the multithreaded implementation still failed to find a collision within reasonable time, suggesting that performance limitations of the test environment (thermal throttling, limited CPU budget, no vector instructions) prevented reaching the expected throughput.

# References

[1] R. Beaulieu, D. Shors, J. Smith, et al. *The SIMON and SPECK Families of Lightweight Block Ciphers*. ePrint 2013/404.