Crypto Engineering TP - Generic second preimage attacks on long messages for narrow-pipe Merkle-Damgard hash functions

Barrois. F, Duverney. T

[2018-11-13 mar.]

Pratical infos :

To compile, just use the Makefile: `make`. The traces show the good functioning of all the questions except the very last one. The call to the function attack was commented for you to see quickly the response to the previous questions. To run the attack, just uncomment the call in the main function. Some possible second preimages have been computed and are displayed in `test_second_preim_results()`.

1 Part one: Preparatory work

1.1 Question 1

For this function, we based our implementation on the following representation:

As the function takes as input a 48-bit message and a key of 96 bits, the values α and β are respectively set to 8 and 3. So $S^{-\alpha}$ performs a circular shift of 8 bits to the right, thus it can be done using `ROTL24_16()` since this function shifts a value of 16 bits to the left, which is the same as shifting it of 8 bits to the right over 24bits. Similarly, S^{β} proceeds to a circular shift of 3 bits to the left and is implemented via a call to `ROTL24_3()`.

Besides, the only difficulty here lies in the order of the operations. According to the picture above, we firstly do the shift on p[1], then we perform the modular addition with p[0]. Afterwards, we xor p[1] to the key value associated to the current iteration, and finally p[0] is updated by the circular shift and the obtained value is xored to p[1].

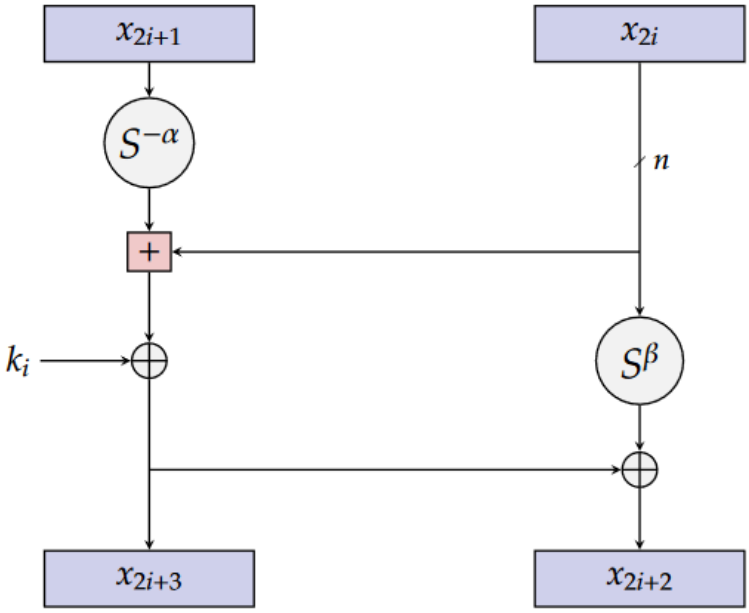

Figure 1: SPECK round function

1.2 Question 2

The `speck48_96_inv` process only consists in performing almost the same operations in the reverse order. There exists a few differences though:

- Because we want to invert the ciphering process, we need to apply $S^{-\beta}$ on c[0] so we call `ROTL24_21()` (21 being the complement of 3 to 24), and symmetrically, we use `ROTL24_8()` for S^{α} on c[1].
- As a modular addition is done during the ciphering, in the case where the value of the modulo was reached, computing the substraction from the cipher will not give back then original value. In order to bypass this case, we add the value of the modulo before computing the substraction so that in any case the obtained value for c[1] will be such that 0 <= c[1] < modulo.

Remark: Actually, the value is stored in an unsigned integer so the potential negative values due to the modular operation would be arranged to the right range of values, but we left the addition with the modulo value as it would be useful in a general case with a different implementation.

1.3 Question 3

The function `cs48_dm` takes a 64-bit h as input so we need to format it to use our cipher `speck48_96`. We compute both halves of it by applying a 24-bit mask on the least significant part and by shifting of 24 bits and applying the same mask to the most significant part (here the mask is a security measure for the case where the two most significant bytes of h are non-zero). Then we call our cipher and reformat the result in one 64-bit variable before xoring it with h, according to the Davies-Meyer function: $\epsilon(m, h) \oplus h$

1.4 Question 4

Finding a fixed point means that the hash value stays constant starting from a certain rank. Thus it corresponds to finding a solution to the equation $\epsilon(m, h) \oplus h = h$. We can simplify it to $\epsilon(m, h) = 0$. So we want to find h such that the ciphering cancels out. Consequently we fix h to the value $\epsilon^{-1}(m, 0)$. Then the fixed-point value is set back on 64 bits by applying shifting and masking operation as explained in the previous questions. The last line of the function ciphers once again the fixed-point and was used in order to check the obtained result, but is not useful technically speaking.

2 Part two: The attack

2.1 Question 1

For this function, we have opted for a hash table as a data structure, taken from the library uthash. This choice seemed to be a good one since the search operation is perfomed in constant time. We have split its use in many functions:

- The function `generateRandomMsg()` generates a new random message using the provided function `xoshiro256starstar_random()` twice since we need four 32-bit blocks to compose a message.
- The function `randomMsgHash()` adds a new couple <hash, randomMessage>, using `generateRandomMsg()`, to a hash table. A check is done to avoid the duplicates.
- The function `randomMsgFixedPoint()` computes the fixed point of a randomly generated message.

The function `find_exp_mess()` firstly fills a hash table with random messages and their associated hash. Next, we generate random messages until the fixed point of one them matches a hash present in the table. We return the random message and the one in the table whose hash was collided.

2.2 Question 2

In the function `attack()`, we fill our hash table with the intermediate hashes got from the given message mess made of 2^{18} blocks. Then we find the fixed point, given by m2, of an expandable message and randomly look for a message block m3 such that the hash of m1||m2||m3 is one of the chaining values in the hash table. Finally, we reconstruct a full preimage mess2 that will be the concatenation of m1, the appropriate number of blocks m2 (found via a counter), m3 and the remaining blocks of the original message located after the collided value.

Analysis :

In order to estimate the number of messages to iterate to find a preimage, we considerate the number of all possibilities and the size of the actual message used in the algorithm: $2^{48}/2^{18} = 2^{30}$

In average, 2^{30} values should be tested before finding a second preimage.

In practice, we usually find one after about 500 000 000 iterations, which is comprised between 2^{29} and 2^{30}. 2^{30} values should allow to find a solution with probability close to 1, so 2^{29} iterations give a success rate of 0.5. So the algorithm is exactly in the average and behaves as expected. Besides, for such number of tested messages, our algorithm lasts about 5 minutes, which corresponds to a rate of 1 666 667 values tested per second. Moreover, the worst case we could have observed gave a solution after 12 minutes. So our attack seems to be quite efficient.