Generic second preimage attacks on long messages for narrow-pipe Merkle-Damgård hash functions

Martinez Anthony, Vallet Nicolas

November 11, 2021

1 Introduction

To compile the program you can use the Makefile. We have added some options to the Makefile to compile the tests or compile the program in debug mode :

- make : Compile the program such that only the function *attack* will be executed
- make test=1 : Compile the program such that the test functions of all the questions will be executed
- make debug=1 : Compile with -g option

To execute the program, run the file "second_preim_48_fillme".

2 Part one : preparatory work

2.1 Question 1

We finished to implement the function *speck48_96* corresponding to the specifications and we have added a test in the function *test_sp48*. The test works properly therefore we can conclude that our function *speck48_96* has been correctly implemented.

2.2 Question 2

We finished to implement the function *speck48_96_inv* in order to inverse the function *speck48_96* and we have added a test in the function *test_sp48_inv* to check our implementation. To perform the test we cipher a text and decipher it to see if we have the text of the beginning. The test works properly therefore we can conclude that our function *speck48_96_inv* has been correctly implemented.

2.3 Question 3

We have implemented the function *cs48_dm* that performs a compression function using a Davies-Meyer construction such that $F(h,m) = E(m, h) \oplus h$ and we have added a test in the function *test_cs48_dm* to check our implementation according to the test given in the subject. For an all-zero input, we get the desired result, therefore we can conclude that our function *cs48_dm* has been correctly implemented.

2.4 Question 4

We have implemented the function *get_cs48_dm_fp* that returns a fixed point fp such that *cs48_dm(m, fp) == fp*. And we have added a test in the function *test_cs48_dm_fp* to check our implementation. The test works properly therefore we can conclude that our function *get_cs48_dm_fp* has been correctly implemented.

3 Part two : the attack

3.1 Question 1

We have implemented the function *find_exp_mess* that returns two one block messages *m1* and *m2* such that there exists $h = cs48\_dm(m1, IV) = get\_cs48\_dm\_fp(m2)$. For the implementation we used the meet-in-the-middle attack.
The algorithm starts by computing some hash for different values of *m1* such that $h = cs48\_dm(m1, IV)$ and store them (the hash and its associated message m1). After that, we compute the hash of different values of *m2* such that h=$get\_cs48\_dm\_fp(m2)$, and see if we have a collision with the previous hash computed.
To store all values such that we have an efficient way to check for collisions, we use a *hashtable* defined in the file *hashTableForAttack.c*.
To compute random values for our 2 messages *m1* and *m2* we used the function in the file *xoshiro256starstar.h*.
This is our algorithm :

```
table <- HashTable
for i in range 1 to 9000000 :
    m1 <- random message
    h <- cs48_dm(m1, IV)
    store(table, m1, h)

while true :
    m2 <- random message
    h <- get_cs48_dm_fp(m2)
    if found(table, h) :
        break
```

In theory our aim is to find a collision and with the birthday bound we need to compute approximately the square root of hash possibilities.
The hash is of size 2^{48} so we need to compute approximately $2^{24} = 16777216$ values. With the meet-in-the-middle attack, we stored $2^{24}/2 \approx 9000000$ different messages *m1*.
And then we'll need to compute approximately the same number of *m2* messages.
It takes less than 1 minutes to find a collision.

3.2 Question 2

We have implemented the function *attack* that returns a second preimage for the message given in the subject.
This is our algorithm :

```
//First we compute the message we want to find a second preimage
//and store all hash for each part of the message in our hashtable
key <- IV
table <- HashTable
for submessage in message :
    key <- cs48_dm(test, key)
    store(table, m1, key)

//Now we find an expendable message
m1 <- uint32_t
m2 <- uint32_t
find_exp_mess(m1, m2)

//We compute the hash of the expandable message
fp <- cs48_dm(m1, IV)

//Find the second preimage
while true :
```

```
m3 <- random message
res <- cs48_dm(m3, fp)
if found(table, res) :
    break

//Compute the new message (depends of the message chose before)
```

This time the meet-in-the-middle attack is computed with the stored values of hash of our sub messages of our message. So if our message is big it will take less time but if it's too big it will take a lot of space to store it.
In our case we have a message of length $1 << 20 = 1048576$ which gives $1048576/4 = 262144$ sub messages so 262144 hash in our hashtable.
We will need to compute many more values of *m3* to find a message that gives a hash in our hashtable. Our implementation takes less than 10 minutes to compute.

In the folder *threadsTest* we have implemented the attack with some threads to make it more efficient. The compilation of the programs contained in *threadsTest* is done with the command *make* and the execution is done by running the file "second_preim_48_fillme" generated by the previous command. In average it seems to be more efficient but it is not always the case.
The implementation does not contain the tests from the previous part, it would have been possible if we had passed more time on it but the aim of this part was just to try to make the attack more efficient.