Cryptography - Project Report

Man In The Middle attack on 2PRESENT24

by

Gabriel Dos Santos
Theophile Molinatti

Department of Computer Science
University of Versailles - Saint-Quentin-en-Yvelines

Supervised by
Christina Boura
Axel Mathieu-Mahias
Yann Rotella

May 2021

Abstract

This report introduces the implementation choices made for the Man In The Middle attack on `2PRESENT24`. It is an appendix to the main report and discusses what was done to optimize performance.

1 Introduction

The project consists in a C implementation of a Man In The Middle attack on a 24 bits version of the ultra-lightweight `PRESENT` block cipher. It aims at finding couples of master keys $(k1, k2)$ using two pairs of plain/cipher texts $(m1, c1)$ and $(m2, c2)$.
The implemented cipher is similar to the original paper, but performs 11 rounds rather than 31.

2 Overview of the attack

The attack on `2PRESENT24` is organized in two main parts:

1. Computing the encryption of $m1$ and the decryption of $c1$ for every 24 bits key possible (2^{24} possibilities), and storing their output in arrays (dictionaries).
2. Searching for collisions between the two dictionaries and asserting that the possible pairs of keys $(k1, k2)$ are valid by repeating the first step using the plain/cipher texts $(m2, c2)$ instead.

3 Identifying and optimizing the program's hotspots

3.1 Using the compiler's flags

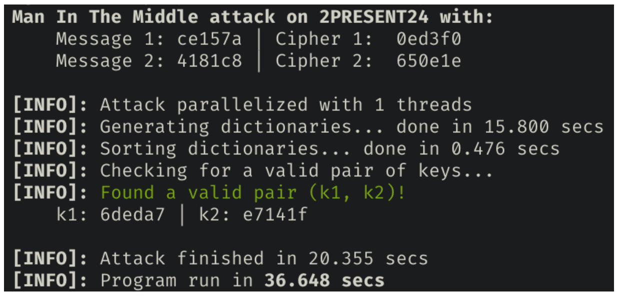
For this project, we used the widely available GNU C Compiler (`gcc`) which provides numerous optimization flags that helped us get faster execution times. We will hereafter describe some of them:

- `-O3` enables all of the optimizations that `gcc` provides, apart from floating point arithmetic (which is not relevant in our case). The compiler cannot guarantee that the requested flags will be applied as the source code may not be optimized enough to allow `gcc` to recognize patterns. However, the code outputting the `-O3` option may be suitable for enhancements that were not available before.

- `-march=native` and `-mtune=native` enable architecture specific optimizations that will allow the compiler to generate code that uses specific instructions (e.g. SSE and/or AVX instructions used for vectorization, if they are available).

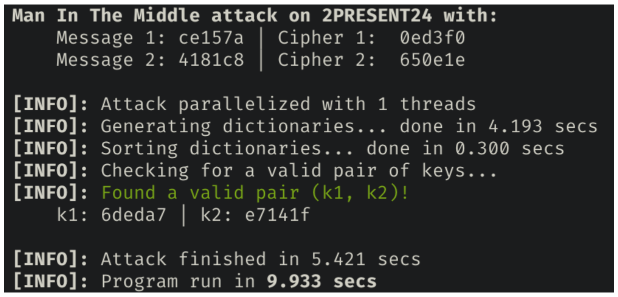
- `-finline-functions` enables function in-lining whenever it is possible which gives better execution times as it removes the call to the function by copying its code when needed. This works especially well for small functions that are called often or in a loop.
  The use of this flag is paired with the presence of the keyword `inline` in the C source code which helps the compiler do better work.

- Loop optimization flags such as `-funroll-loops` or `-ftree-loop-vectorize` are requested to ensure that the code is as optimized as possible, in case the compiler could not apply the on the first pass with `-O3`.



```
Man In The Middle attack on 2PRESENT24 with:
    Message 1: ce157a | Cipher 1:  0ed3f0
    Message 2: 4181c8 | Cipher 2:  650e1e

[INFO]: Attack parallelized with 1 threads
[INFO]: Generating dictionaries... done in 15.800 secs
[INFO]: Sorting dictionaries... done in 0.476 secs
[INFO]: Checking for a valid pair of keys...
[INFO]: Found a valid pair (k1, k2)!
    k1: 6deda7 | k2: e7141f

[INFO]: Attack finished in 20.355 secs
[INFO]: Program run in 36.648 secs
```

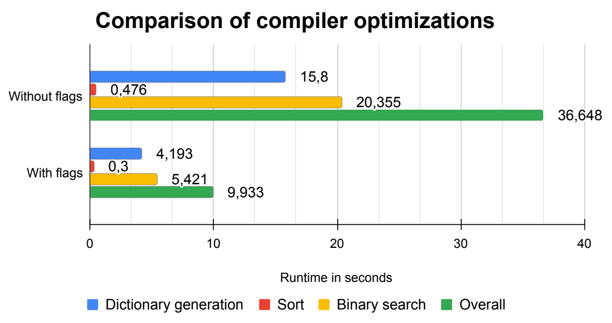
(a) Without optimization flags



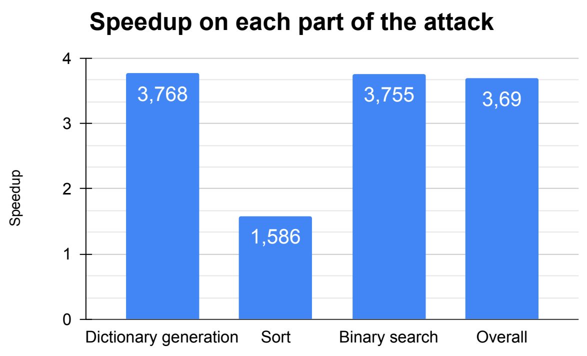
```
Man In The Middle attack on 2PRESENT24 with:
    Message 1: ce157a | Cipher 1:  0ed3f0
    Message 2: 4181c8 | Cipher 2:  650e1e

[INFO]: Attack parallelized with 1 threads
[INFO]: Generating dictionaries... done in 4.193 secs
[INFO]: Sorting dictionaries... done in 0.300 secs
[INFO]: Checking for a valid pair of keys...
[INFO]: Found a valid pair (k1, k2)!
    k1: 6deda7 | k2: e7141f

[INFO]: Attack finished in 5.421 secs
[INFO]: Program run in 9.933 secs
```

(b) With optimization flags

Figure 1: Comparison of compiler optimizations on 2PRESENT24



(a) Attack parts comparison

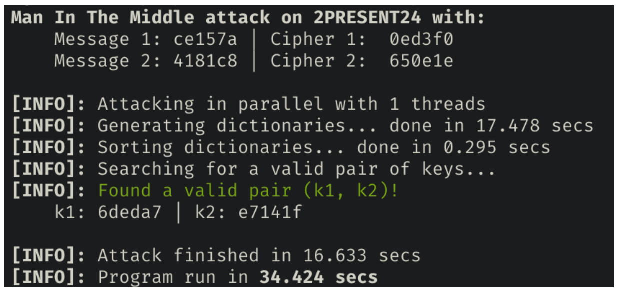


(b) Speedup

Figure 2: Comparison of compiler optimizations on 2PRESENT24

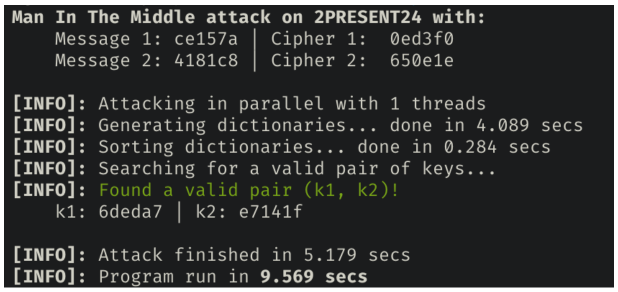
3.2 The permutation layer

The permutation box (P-box) is arguably the optimization that has the most significant impact on performance. Because it is so heavily used throughout the attack, both during encryption and decryption, it can become the bottleneck of the application if not carefully implemented.
The naive approach we first took was to use a static array that transposes every bits, ensuring Shannon's property of confusion. We then used a `for` loop to permute each bit of the input text to its final position. However, using a loop means that we need to perform as many iterations as they are bits in the block, each of them having to compute multiple memory addresses which are known to be very expensive operations, as we will demonstrate hereafter. Using the performance analysis and optimization framework developed at UVSQ, MAQAO (Modular Assembly Quality Analyzer and Optimizer), we noticed that over 50% of the execution time was spent in the P-box layer, so we decided to take another approach.
Instead of using of arrays and loops, we implemented the entire P-box manually by directly placing the bits of the input message to their final position using masks and shits. This method also reduces the number of memory accesses, thus drastically improving performance. As you can see below, we achieved a speedup of nearly 360% compared to our naive approach.



```
Man In The Middle attack on 2PRESENT24 with:
    Message 1: ce157a | Cipher 1:  0ed3f0
    Message 2: 4181c8 | Cipher 2:  650e1e

[INFO]: Attacking in parallel with 1 threads
[INFO]: Generating dictionaries... done in 17.478 secs
[INFO]: Sorting dictionaries... done in 0.295 secs
[INFO]: Searching for a valid pair of keys...
[INFO]: Found a valid pair (k1, k2)!
    k1: 6deda7 | k2: e7141f

[INFO]: Attack finished in 16.633 secs
[INFO]: Program run in 34.424 secs
```

(a) P-Box using static array and loop



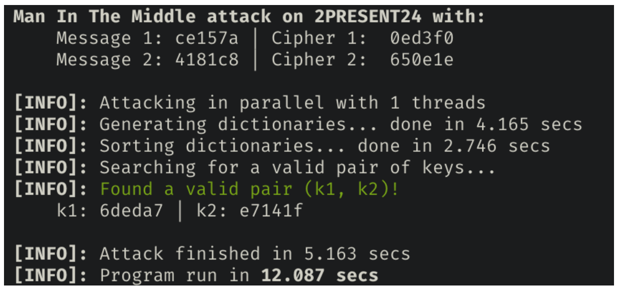
```
Man In The Middle attack on 2PRESENT24 with:
    Message 1: ce157a | Cipher 1:  0ed3f0
    Message 2: 4181c8 | Cipher 2:  650e1e

[INFO]: Attacking in parallel with 1 threads
[INFO]: Generating dictionaries... done in 4.089 secs
[INFO]: Sorting dictionaries... done in 0.284 secs
[INFO]: Searching for a valid pair of keys...
[INFO]: Found a valid pair (k1, k2)!
    k1: 6deda7 | k2: e7141f

[INFO]: Attack finished in 5.179 secs
[INFO]: Program run in 9.569 secs
```

(b) Optimized, in-place P-box

Figure 3: Comparison of P-box implementations on 2PRESENT24

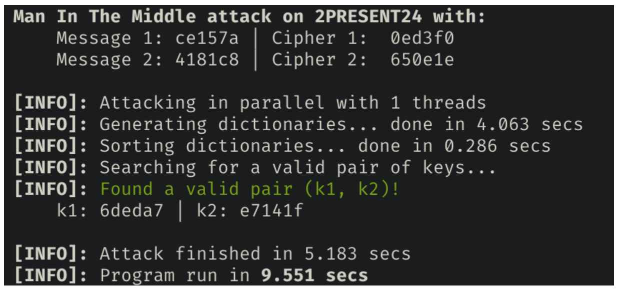
3.3 Choosing the right sorting algorithm

To be able to perform a binary search on the dictionaries, we first needed to sort them. As advised by Mrs Boura, we started by implementing a quick-sort algorithm as it often yields very good results on any type of data, averaging a time complexity of $O(nlog(n))$. Nonetheless, to bring performance even further, we decided to switch to a sorting algorithm that was more efficient on the type of data we dealt with, a radix sort. It is noticeably better at sorting numbers, with an average time complexity of $(O(nk))$, making it very fitting in our case. Thanks to it, we have achieved execution times, on the sorting part, up to 9.6 times faster than with quick sort (speedup of 1.26 overall).



```
Man In The Middle attack on 2PRESENT24 with:
    Message 1: ce157a | Cipher 1:  0ed3f0
    Message 2: 4181c8 | Cipher 2:  650e1e

[INFO]: Attacking in parallel with 1 threads
[INFO]: Generating dictionaries... done in 4.165 secs
[INFO]: Sorting dictionaries... done in 2.746 secs
[INFO]: Searching for a valid pair of keys...
[INFO]: Found a valid pair (k1, k2)!
    k1: 6deda7 | k2: e7141f

[INFO]: Attack finished in 5.163 secs
[INFO]: Program run in 12.087 secs
```

(a) Using a quick-sort algorithm



```
Man In The Middle attack on 2PRESENT24 with:
    Message 1: ce157a | Cipher 1:  0ed3f0
    Message 2: 4181c8 | Cipher 2:  650e1e

[INFO]: Attacking in parallel with 1 threads
[INFO]: Generating dictionaries... done in 4.063 secs
[INFO]: Sorting dictionaries... done in 0.286 secs
[INFO]: Searching for a valid pair of keys...
[INFO]: Found a valid pair (k1, k2)!
    k1: 6deda7 | k2: e7141f

[INFO]: Attack finished in 5.183 secs
[INFO]: Program run in 9.551 secs
```

(b) Using a radix sort algorithm

Figure 4: Comparison of sorting implementations on 2PRESENT24

3.4 Making the attack parallel

The last and probably most important optimization we made is to make the attack parallel. We did not multi-threaded the entire application as it would have brought more constraints to benefits. For this reason, we only parallelized the dictionary generation and the binary search.
Thanks to Salah IBN AMAR (teacher-researcher at the Li-PaRAD and Exascale Computing Research labs), we had the opportunity to measure the performances of our program on super-calculators, comparing the execution time with different numbers of threads. We ran the benchmarks on two different machines: a 64 cores AMD Zen2 and a 52 core Intel Skylake. The following chart compares the scalability of our program on these machines as well as on a modern laptop CPU.

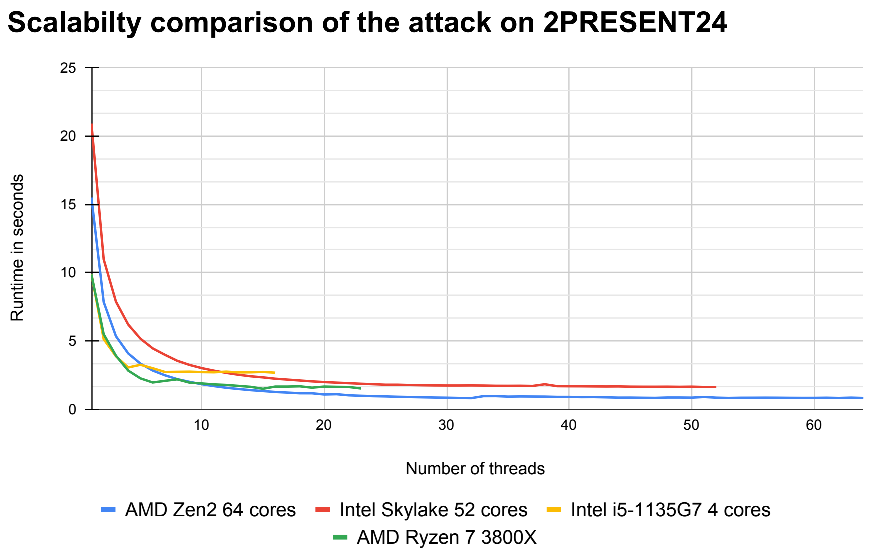


Figure 5: Scalability comparison of the attack on 2PRESENT24