Report Lab 1: Hardware Security

Henning Schei

May 2016

1 Introductoin

The implementation of the permutation code, `p_permutation`, is a good target for a timing attack, because there is a strong dependence between the hamming weight of the value to be permuted and the running time used. This is because of the aditional 32 iteration loop that the permutation code enter if a bit is set.

From an attackers point of view, this can be used to build up a timing model where the correlation between hamming weight and time consumed are beeing exploited.

2 The algorithm

This is a breif overview of the operations of the algorithm:

- Calculate the output of all 8 S-boxes simutainiously with all possible keys, using an XOR operation with hexadecimal value of += `0x041041041041`.
- Used a masking pattern of `0xf0000000` to isolate each S-boxes bits and shifted the masking pattern appropriate to each S-boxes output.
- A multithreaded function calculated the hamming weight of all 8 S-boxes. The results with hamming weight HW = 0, 1 got stored in a *slow* list and those with HW=3,4 got stored in a *fast* list, respectivly. Based on this, the maximum timing difference between the average time for the slow and fast list on each sbox were used to estimate the most likely subkey.

3 Discussion

Although this attack uses the easiest possible statistical model by using average timing difference, the code completes finds the correct key using ˜3000 acquisitions on the generated ta.dat file. With the shared ta.dat file, it uses ˜8000 acquisitions. This could have been done a whole lot better by using more advanced statistical tool, for instance Pearson Correlation.

4 Countermasure

In order to reduce the posibility for a timing attack, it's necessary to improve the permutation function. My first thought was to remove all branching dependent of hamming weight or keyword, but I realized that the quickest solution was to write an identical 32-iteration loop, such that the permutation code will have the same complexity regardless of the input. However, this makes the permutation code even more inefficient than it already was. See the code below.

```
uint64_t
des_p_ta(uint64_t val) {
  uint64_t res;
  int i, j, k,fake;

  res = UINT64_C(0);
  k = 0;
  for(i = 1; i <= 32; i++) {
    if(get_bit(i, val) == 1) {
      for(j = 1; j <= 32; j++) {
        if(p_table[j - 1] == i) { /* C array index starts at 0, not 1 */
          k = j;
        }
      }
      res = set_bit(k, res);
    }

    else if(get_bit(i,val)== 0){ /* This is a dummy branch */
      for (j=1;j<=32; j++){
         if (p_table[j-1] ==i){
             fake = j;
         }
      }
    }
    fake = set_bit(1,1);
  }
  return res;
}
```

With this improved permutation function, the algorithm can't find the correct key with 100000 acqusitions. It has to be mentioned that if the code was implemented using a better statistical tool, it may have found the correct key.