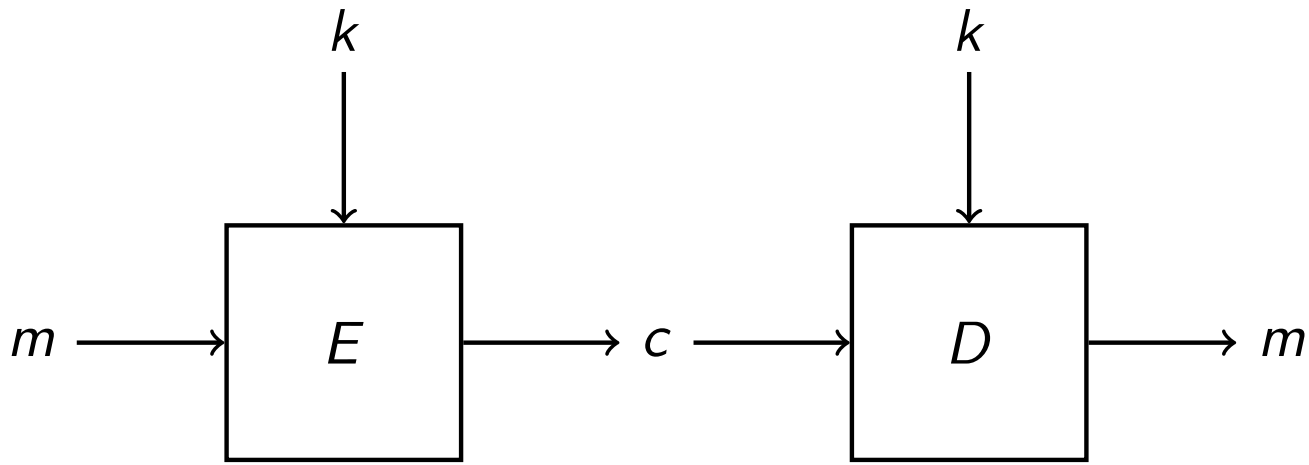
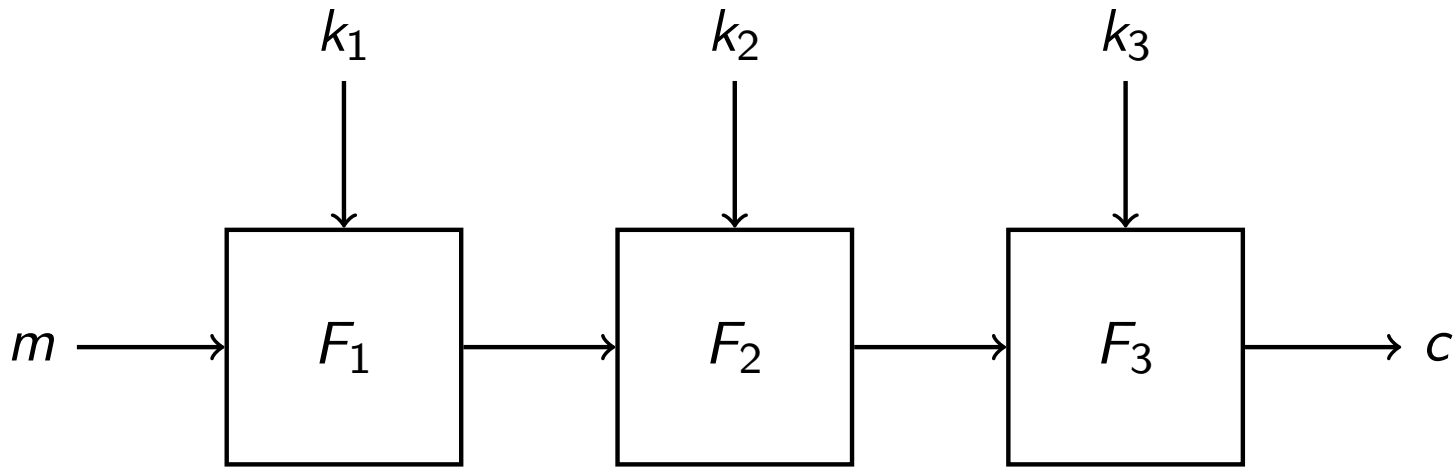
Brute Force

February 26, 2019

Symmetric Encryption



We often model encryption as a key alternating cipher:



Symmetric Encryption (2)

Two main constructions for 'practical' symmetric block ciphers:

- SPN (Substitution Permutation Networks) [i.e. AES]
- (Generalized) Feistel networks [i.e. DES]

In a few slides we look at AES (like) ciphers.

AES-128-128

- 10 rounds
- Each round consists of the following ‘layers’:
  - Sub Bytes (SB)
  - Shift Rows (SR)
  - Mix Columns (MC)
  - Add Round Key (ARK)
- Funny (and good) explanation of AES:
  `https://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html`

AES like ciphers

A lot of SPN ciphers reuse components and/or the structure of AES. So it is worthwhile to look at how to implement the basic components.

Disclaimer

Note that in this class we look at implementing ciphers for analysis purposes. This means that the code we write is not constant time and/or secure and should not be used in any system.

State representation

First we need to choose a representation for the state:

- 'State sliced'
- Row sliced
- Cell sliced
- Bit sliced

We mostly use State/Row/Cell sliced implementations, but for some ciphers a bit sliced implementation can be fast.

Sub Bytes

The Sub Bytes layer can be implemented in several ways:

- Sbox can be implemented as a boolean circuit (or as a inversion in a GF(2^8))
- We can use a lookup table
  - Combine multiple Sboxes into one big LUT
  - Combine the SC and SR/MC into one big LUT

Optimization

Always measure the speed of your implementation and the components. A LUT for example is not always beneficial depending on your system and how the cipher is used.

Shift Rows

The implementation for the Shift Rows layer depends a lot on the chosen state representation and the capabilities of the processor.

- State Sliced
  - Select row from state and rotate.
  - Reinterpret state as an array and rotate each element.
- Row Sliced
  - Rotate each row.
- Cell Sliced
  - We can omit the SR layer and combine it with MC.

Mix Columns

The implementation is again dependend on the chosen state representation. The implementation tactics can be deployed as with the Shift Rows.

Hint

One trick that we can often deploy is to combine multiple XOR's. E.g.:

$$
\begin{aligned}
x &= a \oplus b \\
y &= c \oplus b \oplus a \\
&\Longleftrightarrow \\
x &= a \oplus b \\
y &= c \oplus x
\end{aligned}
$$

Brute Force

- The 'most trivial' attack possible $\Rightarrow$ Try out all possible keys
- Often used as a subroutine in other attacks

Algorithm 1 BruteForce

Let $k = 0$ be the key
Given a known plaintext p and ciphertext c
while Encrypt(p, k) != c **do**
    Get next key k
end while
return k

Brute Force (2)

- We only need to implement one way (so encryption or decryption).
- Note that also the key schedule should be fast.
- Decrease the overhead as much as possible.
- Often only needs 1 or 2 plaintext ciphertext pairs.
- How far can we push?

Pushing Brute Force Attacks

Say we have a 100 cycles per encryption implementation of a cipher. Most desktops have a ≈3GHz processor. This means that we can do: $\frac{3 \cdot 10^9}{100} = 3 \cdot 10^7 \approx 2^{24.8}$ encryptions per second. Which leads to the following brute force times (for one processor):

Key bits	Time	AWS cost ($)
25	1.14 seconds	0.0000171
30	36.75 seconds	0.00055
32	147.0 seconds	0.0022
36	39.21 minutes	0.0351
40	10.46 hours	0.565
44	6.970 days	9.03
50	1.221 years	567.7
56	78.17 years	36977
64	$2.001 \cdot 10^4$ years	9460800
128	$3.694 \cdot 10^{23}$ years	$1.7 \cdot 10^{23}$
256	$1.257 \cdot 10^{62}$ years	A lot

Attacker Models

We consider the following attacker models:

- Known Ciphertext
- Known Plaintext
- Chosen Plaintext
- Chosen Adaptive Plaintext
- Related Key
- Known Key

Questions

- In what attacker models can we use a brute force attack?
- What representations can we use for the state in an implementation of a cipher?
- What is often the best method to implement Sub Bytes?

Measuring time

One very important part of optimizing implementations is measuring the time particular parts of the implementation take. Some pointers for measuring time:

- For counting cycles use rdtsc (there is a C header file on the website you can use)
- Always do multiple runs and take the median.
- Keep in mind that a compiler often tries to 'optimise' out dead code. I.e. if it sounds too good to be true it probably is ;)
- For things that take more time use `clock()` from `time.h`.
- Try different approaches, and write down your results for later.

For next week

- Create a nice handle for the exercise site.
- Do this week's exercises.
- Read the papers for next week (see website).