universität innsbruck

# Defending against power analysis by balancing binary values

## a compiler based approach

Alexander Schlögl, supervised by Univ.-Prof. Dr. Rainer Böhme

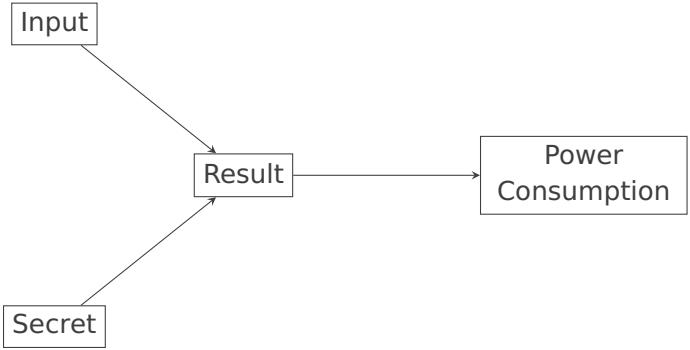
# Motivation



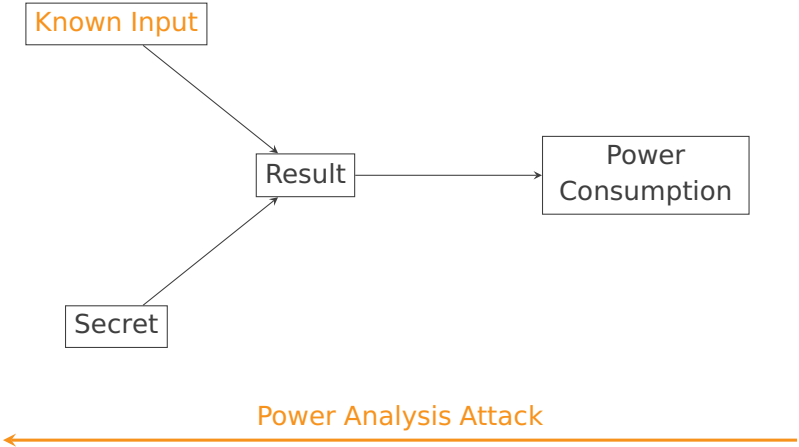
https://store.storeimages.cdn-apple.com/4982/as-images.apple.com/is/HJCC2?wid=1144&hei=1144&fmt=jpeg&qlt=95&op_usm=0.5,0.5&.v=0

https://www.liberaldictionary.com/wp-content/uploads/2019/01/bank-card-8123.jpg

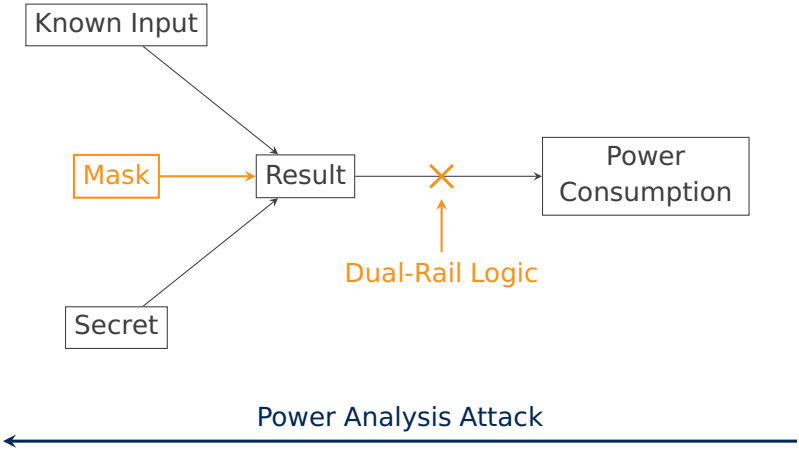
# Motivation



# Motivation


Known Input
Result
Power Consumption
Secret
Power Analysis Attack

# Motivation


Known Input
Mask
Result
Power Consumption
Dual-Rail Logic
Secret
Power Analysis Attack

# Motivation

## Masking

Increases analysis complexity

- + Runs on standard hardware
- – Built into algorithm
- – Requires expert knowledge

## Dual-Rail Logic

Balances power consumption

- + Can run any program
- – Requires specialized hardware

# Motivation

**Masking**

Increases analysis complexity

- + Runs on standard hardware
- – Built into algorithm
- – Requires expert knowledge

**Dual-Rail Logic**

Balances power consumption

- + Can run any program
- – Requires specialized hardware

**Best of both worlds?**

Apply balancing similar to Dual-Rail logic in software

# Overview

**Content**

- Motivation
- Balancing
- Arithmetic
- Code Transformation
- Results
- Conclusion

# Balancing

**Working assumption:**
Power consumption is proportional to Hamming weight (number of 1s)
$\rightarrow$ constant Hamming weight = constant power consumption

# Balancing

**Working assumption:**
Power consumption is proportional to Hamming weight (number of 1s)
$\rightarrow$ constant Hamming weight = constant power consumption

**Approach**

Extend register size, and store inverse along with actual value

# Balancing

**Working assumption:**
Power consumption is proportional to Hamming weight (number of 1s)
$\rightarrow$ constant Hamming weight = constant power consumption

**Approach**

Extend register size, and store inverse along with actual value

# Arithmetic

Regular operators will not work:

$$
\begin{array}{ccccccccc}
| & 0 & | & \overline{x} & | & 0 & | & x & | \\
32 & & 24 & & 16 & & 8 & & 0
\end{array}
$$

$$\vee$$

$$
\begin{array}{ccccccccc}
| & 0 & | & \overline{y} & | & 0 & | & y & | \\
32 & & 24 & & 16 & & 8 & & 0
\end{array}
$$

$$=$$

$$
\begin{array}{ccccccccc}
| & 0 & | & \overline{x} \vee \overline{y} & | & 0 & | & x \vee y & | \\
32 & & 24 & & 16 & & 8 & & 0
\end{array}
$$

$$\neq$$

$$\overline{x \vee y} = \overline{x} \wedge \overline{y}$$

# Arithmetic

Find replacements for:

- `ORR`
- `AND`
- `XOR`
- `ADD`
- `SUB`
- `MUL`
- `SHIFTS`
- `DIV`
- `REM`

# Arithmetic

Find replacements for:

- ORR
- AND
- XOR
- ADD
- SUB
- MUL
- SHIFTS
- DIV
- REM

```
int balanced_or(int lhs,
        int rhs) {
  int temp_or = lhs | rhs;
  int temp_and = lhs & rhs;
  int combined = (temp_and << 8)
          | temp_or;
  combined &= 0xff0000ff;
  return balanced_2_1(combined);
}
```

# Verifying the arithmetic

Perform exhaustive search of the input space:

**Test framework**

- Takes individual steps as lambdas
- Executes over all inputs
- Stores intermediate values
- Checks correctness
- Plots Hamming weight histograms

# Verifying the arithmetic

Perform exhaustive search of the input space:

**Test framework**

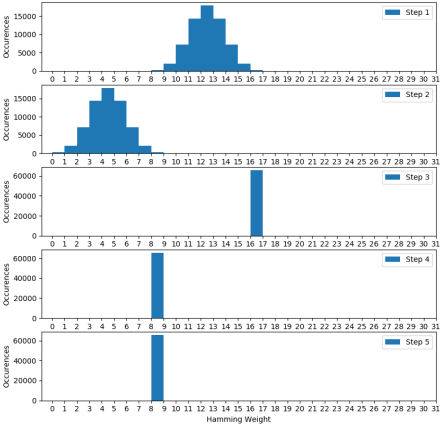
- Takes individual steps as lambdas
- Executes over all inputs
- Stores intermediate values
- Checks correctness
- Plots Hamming weight histograms

# Verifying the arithmetic

Perform exhaustive search of the input space:

**Test framework**

- Takes individual steps as lambdas
- Executes over all inputs
- Stores intermediate values
- Checks correctness
- Plots Hamming weight histograms



# Applying the changes

**Automatic balancing**

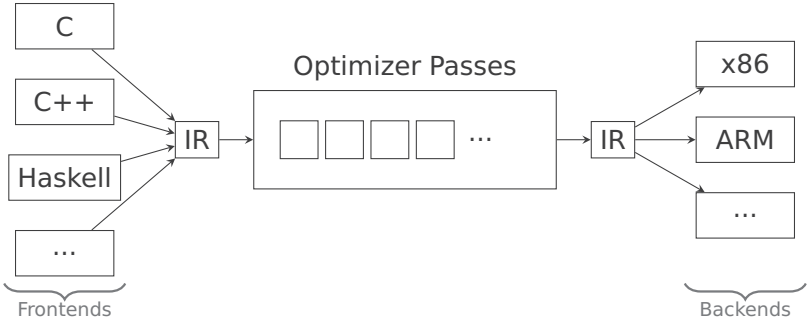
Rewrite code during compilation

# Applying the changes

**Automatic balancing**

Rewrite code during compilation

LLVM:

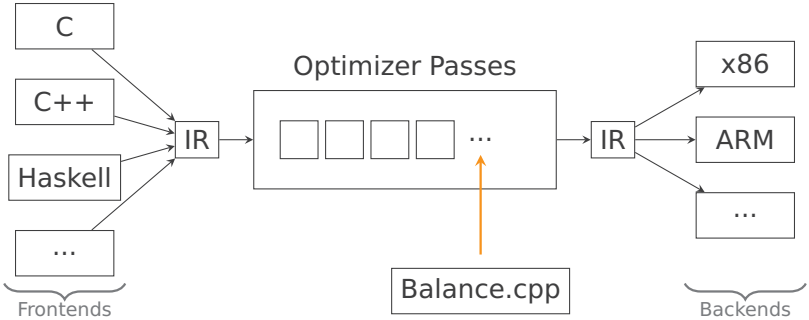


# Applying the changes

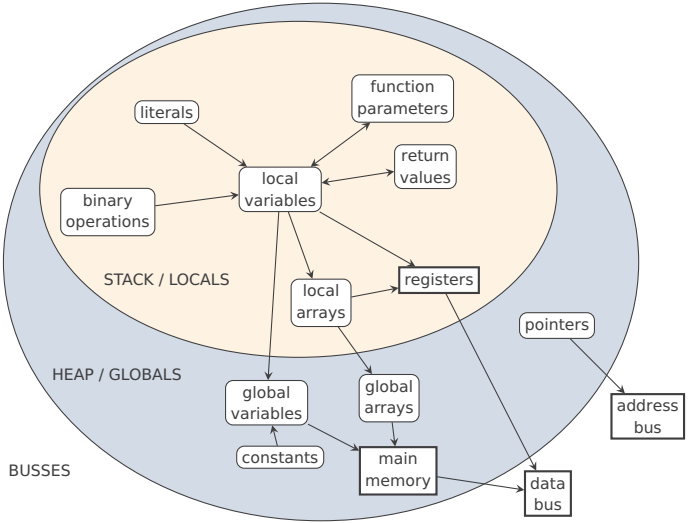
**Automatic balancing**

Rewrite code during compilation

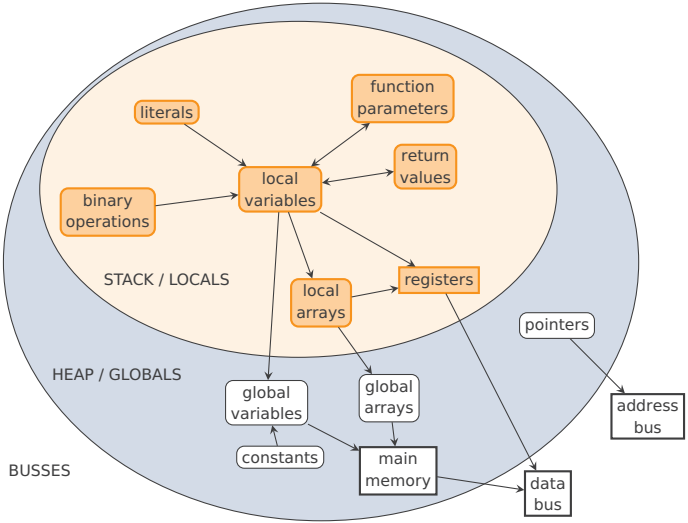
LLVM:



# Optimizer Pass



# Optimizer Pass



# Optimizer Pass

Additionally transform:

- stores
- loads
- casts
- array indexing
- compares
- function calls

# Evaluation

How to generate “virtual” power traces?

**Qemu alone**

+ fast
– incorrect view

# Evaluation

How to generate “virtual” power traces?

**Qemu alone**

- \+ fast
- – incorrect view

**Qemu + gdb**

- \+ correct view
- \+ includes program location information
- – **very** slow

Execute instruction by instruction, dump registers every time

# Results

**No attack**

No attack was mounted, instead performed statistical analysis

# Results

**No attack**

No attack was mounted, instead performed statistical analysis

| | AES | |
|---|---|---|
| | unbalanced | balanced |
| Executed instructions | 22 876 | 339 168 |
| Relative increase | 1 | 14.888 |
| Balanced operations | 20 571 | 334 521 |
| Balancedness | 0.903 | 0.986 |
| Unbalanced operations | 2211 | 4647 |

# Summary

- Arithmetic is *mostly* proven to be correct
- Works without programmer work
- Balances everything on stack
- Requires more powerful, but standard hardware
- Does not explode code size

# Limitations

- Works only on stack
- Only tested on some code samples
- Correctness of `REM` and `DIV` not proven
- Not attacked, only evaluated
- Greatly increased execution time

# Future work

Improve on thesis:

- Test on actual hardware
- Balance globals
- Improve operators
- Mark balancing targets

Similar ideas:

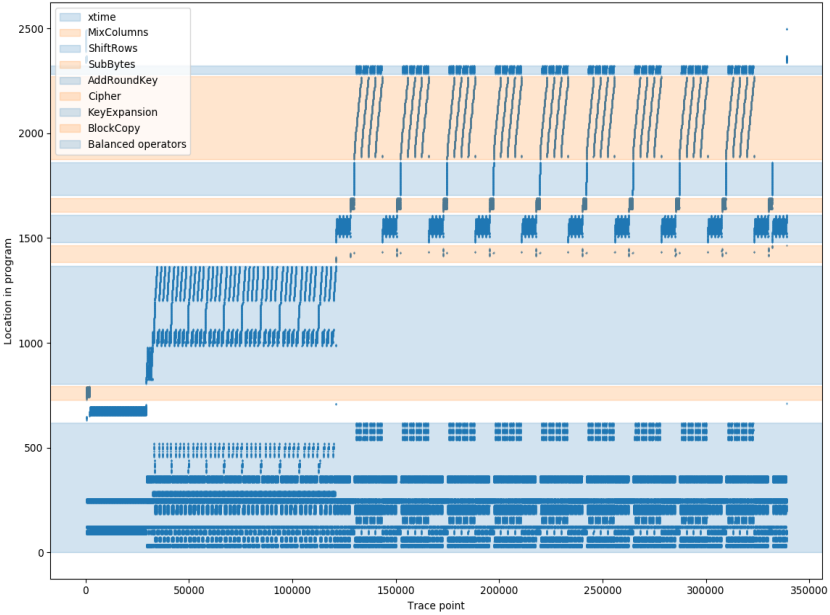
- Move balancing to type system
- Other power analysis defenses
- Control flow randomization
- Move more security tools to LLVM

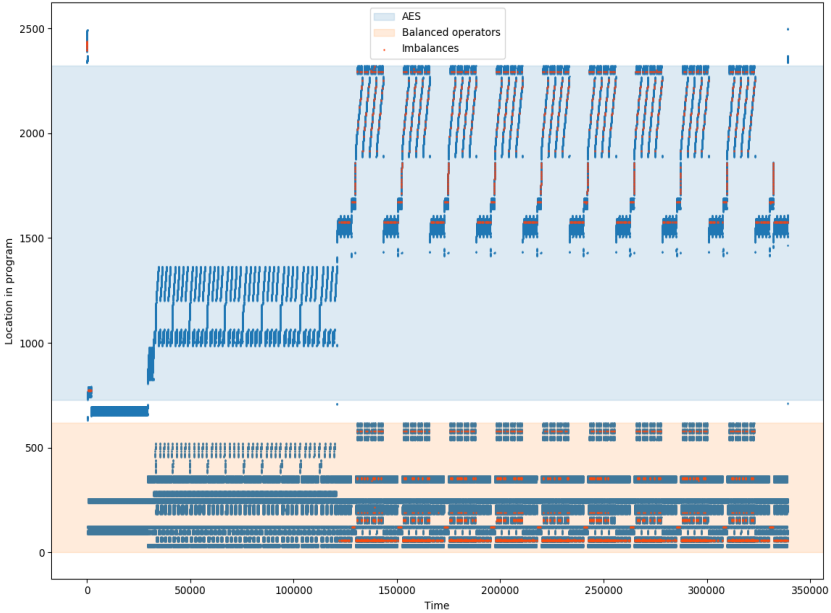
# Conclusion

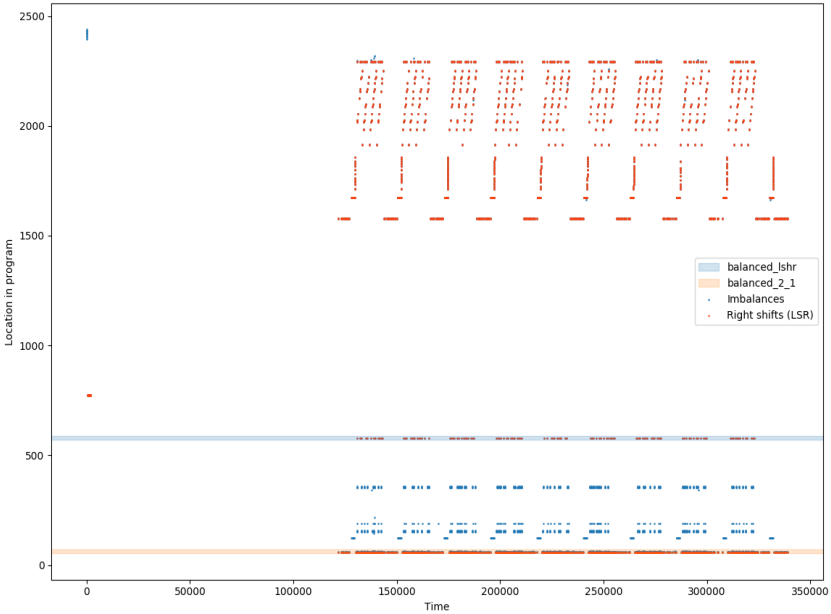
- Debugging optimizer passes is hard
- Security and performance likely mutually exclusive
- Backend cannot entirely be ignored
- Qemu is not a processor emulator

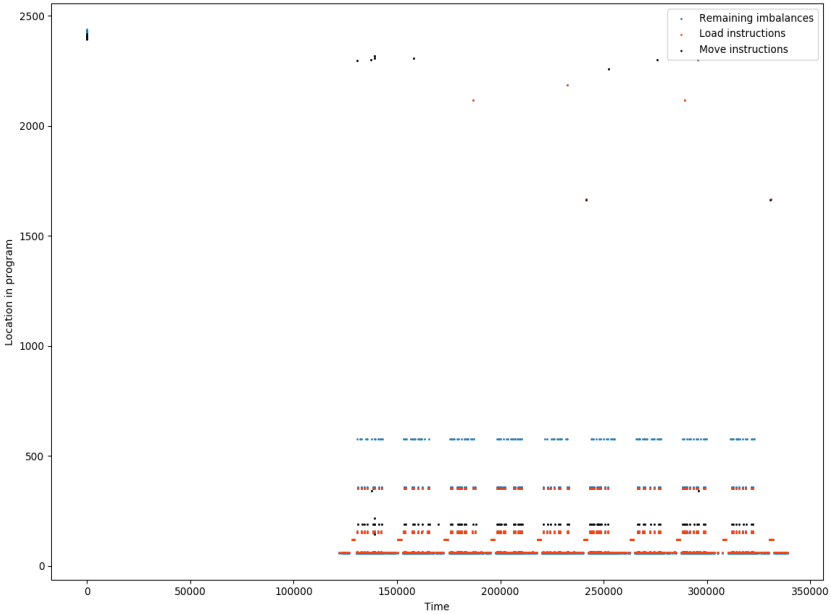
**LLVM IR**

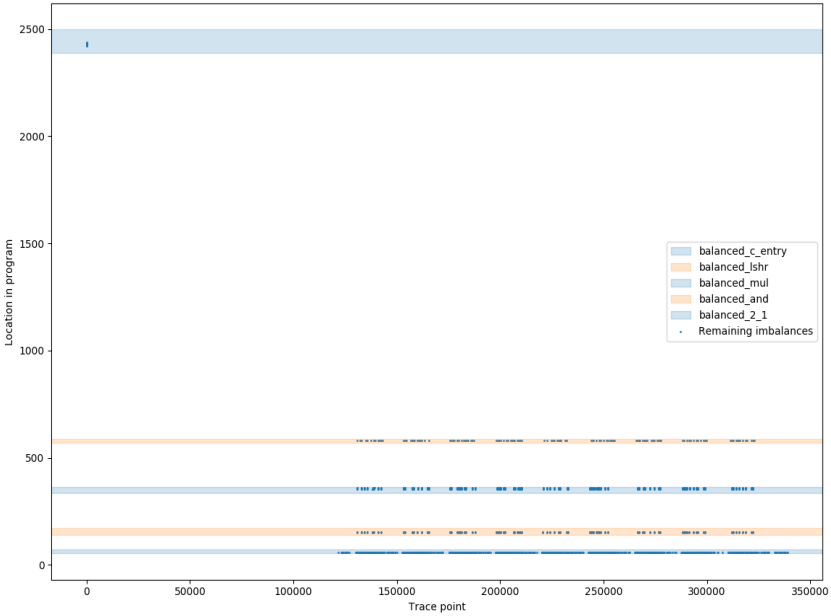
LLVM’s intermediate representation offers many avenues for future work, not only for optimizition, but also for security.


xtime
MixColumns
ShiftRows
SubBytes
AddRoundKey
Cipher
KeyExpansion
BlockCopy
Balanced operators
Location in program
2500
2000
1500
1000
500
0
0
50000
100000
150000
200000
250000
300000
350000
Trace point


AES
Balanced operators
Imbalances
Location in program
Time
2500
2000
1500
1000
500
0
0
50000
100000
150000
200000
250000
300000
350000


2500
2000
1500
1000
500
0
Location in program
0
50000
100000
150000
200000
250000
300000
350000
Time
balanced_lshr
balanced_2_1
Imbalances
Right shifts (LSR)


Remaining imbalances
Load instructions
Move instructions
Location in program
Time
0
500
1000
1500
2000
2500
0
50000
100000
150000
200000
250000
300000
350000


2500
2000
1500
1000
500
0
Location in program
0
50000
100000
150000
200000
250000
300000
350000
Trace point
balanced_c_entry
balanced_lshr
balanced_mul
balanced_and
balanced_2_1
Remaining imbalances

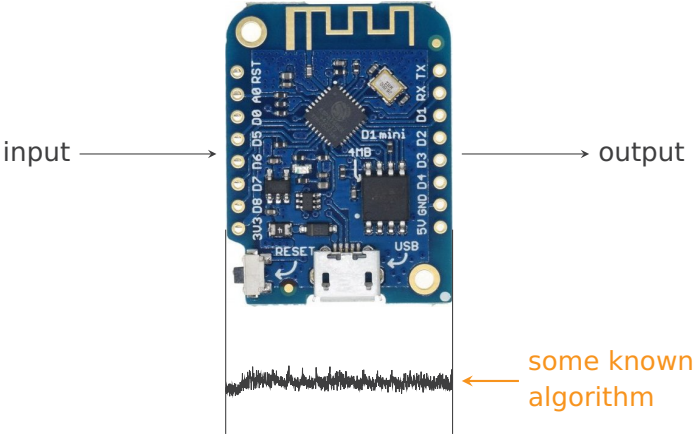
# Results

**No attack**

No attack was mounted, instead performed statistical analysis

| | AES | | |
|---|---|---|---|
| | unbalanced | balanced | filtered balanced |
| Executed instructions | 22 876 | 339 168 | 339 168 |
| Relative increase | 1 | 14.888 | 14.888 |
| Balanced operations | 20 571 | 334 521 | 337 852 |
| Unbalanced operations | 2211 | 4647 | 1316 |
| Balancedness | 0.903 | 0.986 | 0.996 |

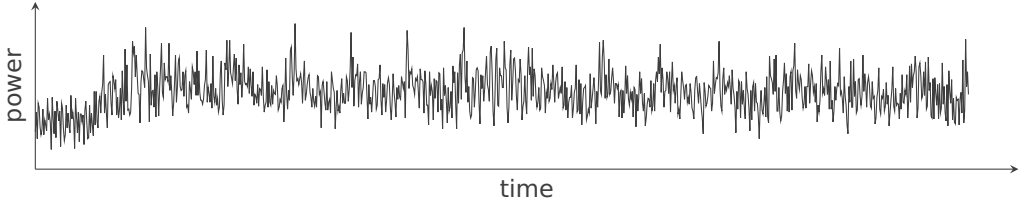
# Platform




https://www.tinytronics.nl/shop/en/communication/wemos-d1-mini-v3-esp8266-ch340

# Power analysis

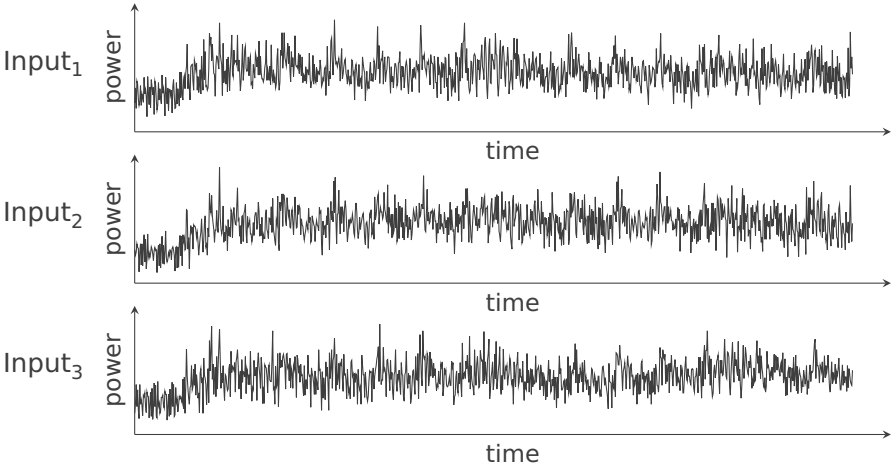
Power trace:



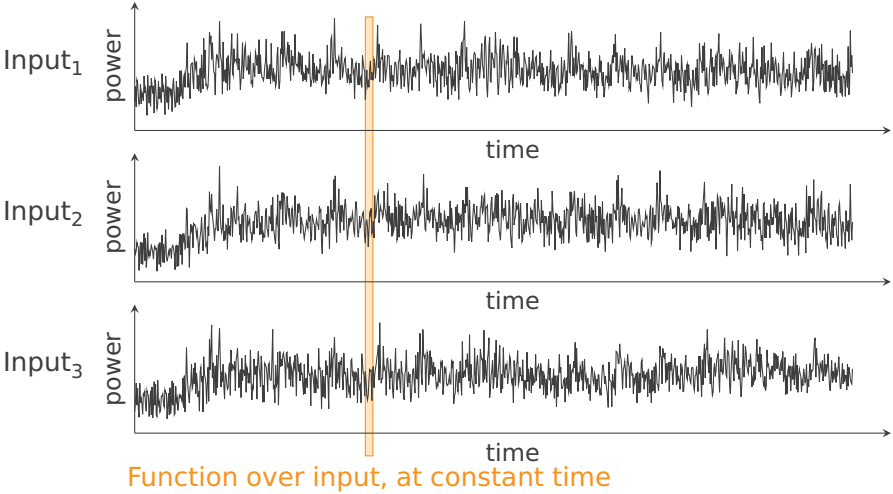

**Traces as functions**

Power traces are functions over time, with constant input

# Power analysis cont.



# Power analysis cont.



# Power analysis cont.

**Secret**

Power consumption depends on input **and secret**

```
for(i=0;i<4;++i)
  for(j = 0; j < 4; ++j)
    state[i][j] =
                input[i][j] ^
                secret[i][j];
```

Calculate “hypothetical” power consumption:

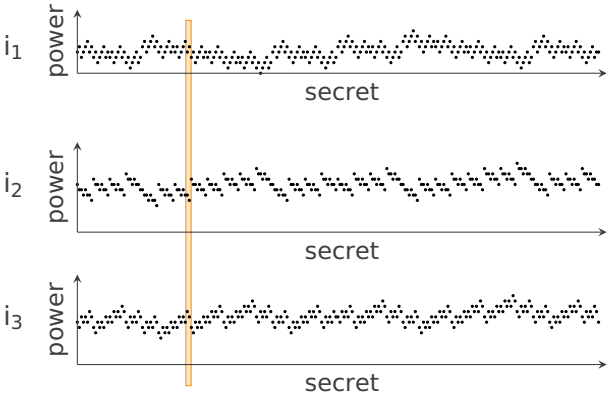
# Power analysis cont.

**Secret**

Power consumption depends on input **and secret**

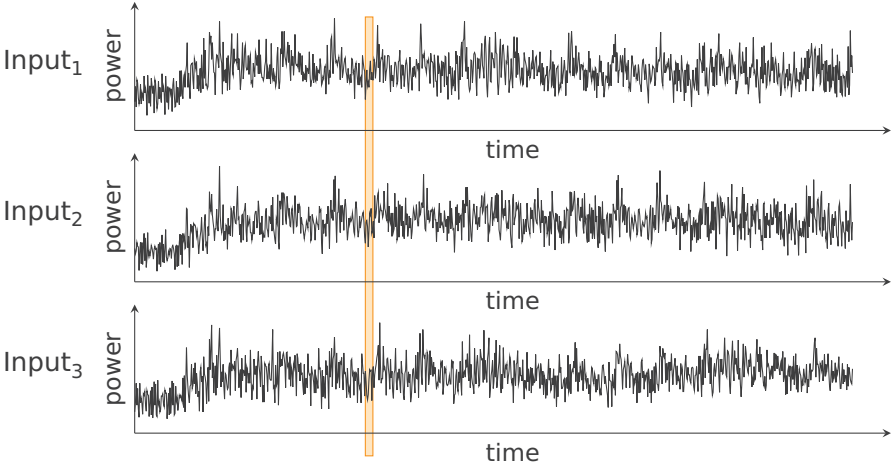
```
for(i=0;i<4;++i)
  for(j = 0; j < 4; ++j)
    state[i][j] =
                input[i][j] ^
                secret[i][j];
```

Calculate “hypothetical” power consumption:




Function over input, with constant secret

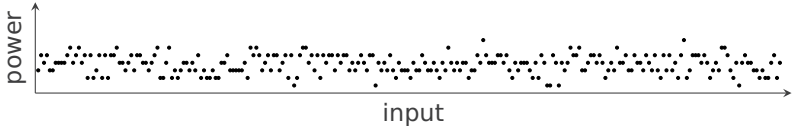
# Power analysis cont.



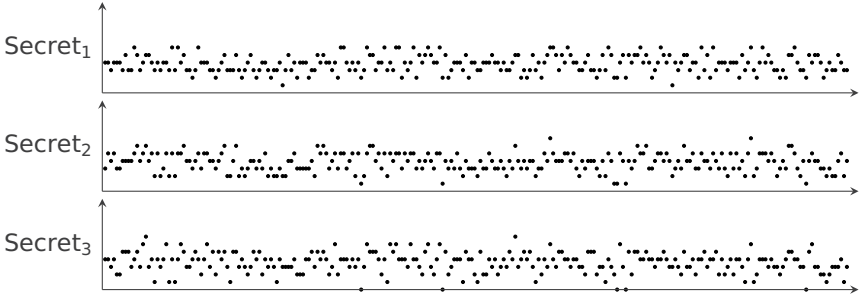

Function over input, at constant time

# Power analysis cont.

Actual consumption:

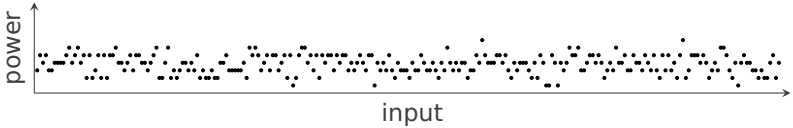



Hypothetical consumptions:



# Power analysis cont.

Actual consumption:




Hypothetical consumptions: