universität
innsbruck

Defending against power analysis by balancing binary values

a compiler based approach

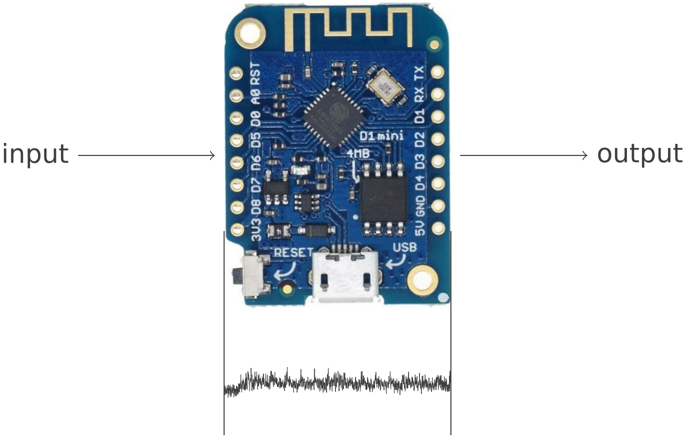
Alexander Schlögl, supervised by Univ.-Prof. Dr. Rainer Böhme

Overview

Content

- Power analysis
- Approach
- Arithmetic
- Compiler Pass
- Results
- Future Work

Platform



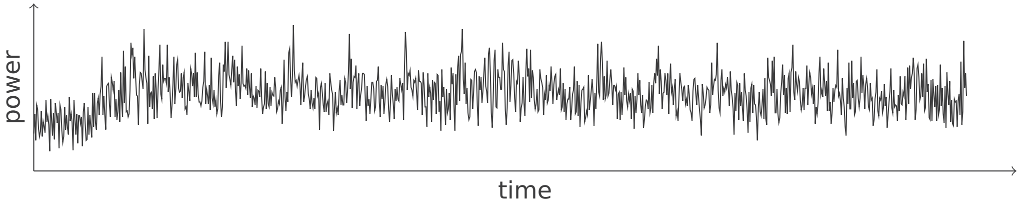
input

output

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

Power analysis

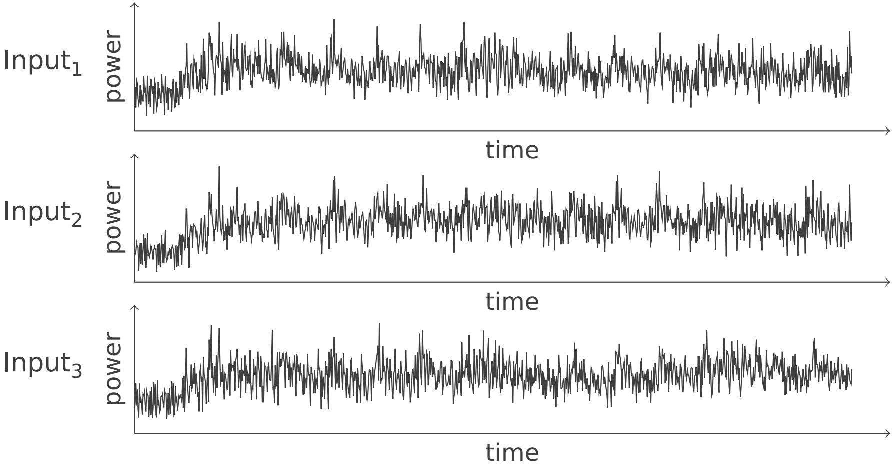
Power trace:



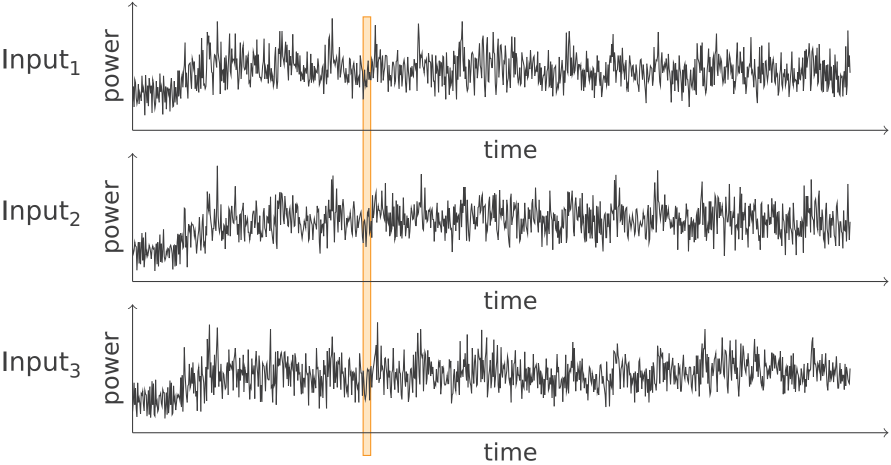
Traces as functions

Power traces are functions over time, with constant input

Power analysis cont.



Power analysis cont.



Function over input, at constant time

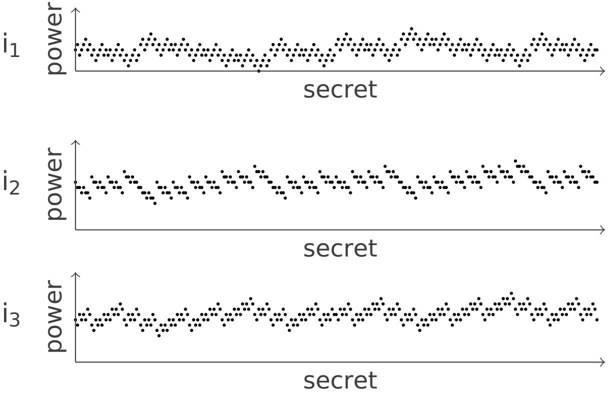
Power analysis cont.

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];
```

Generate “hypothetical” power traces:



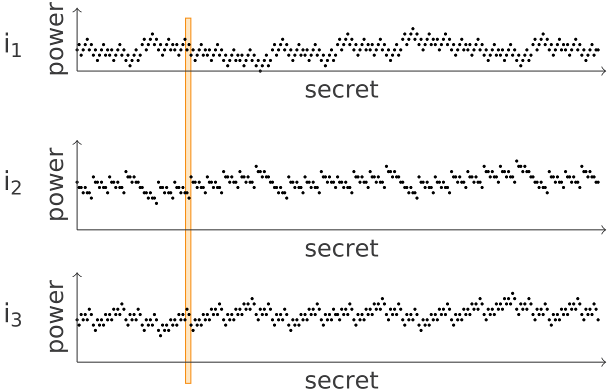
Power analysis cont.

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];
```

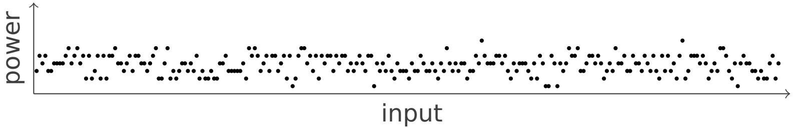
Generate “hypothetical” power traces:



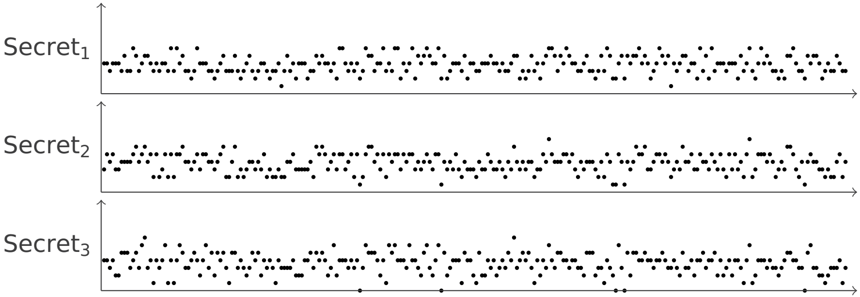
Function over input, with constant secret

Power analysis cont.

Actual consumption:

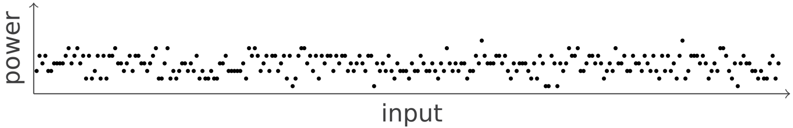


Hypothetical consumptions:

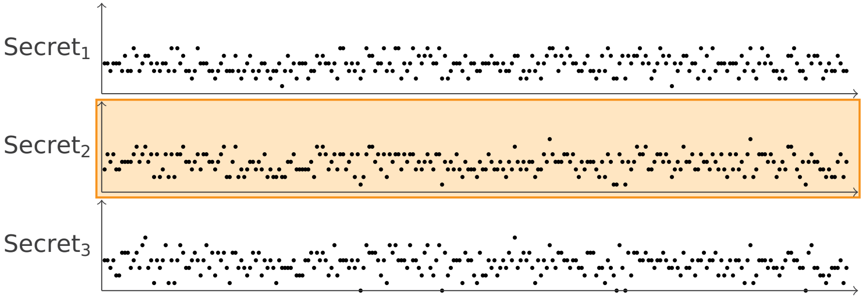


Power analysis cont.

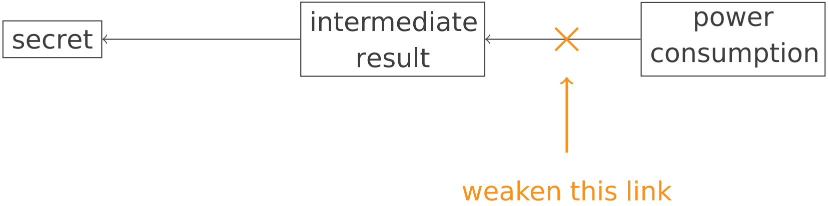
Actual consumption:



Hypothetical consumptions:



Approach



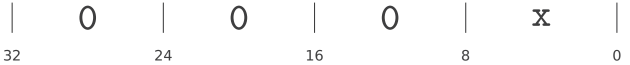
Working assumption

Power consumption is proportional to Hamming weight

Approach cont.

constant Hamming weight → constant power consumption

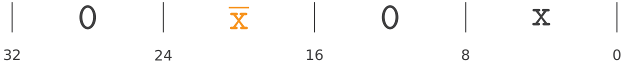
char:



| 0 | 0 | 0 | x |

32 24 16 8 0

balanced char:

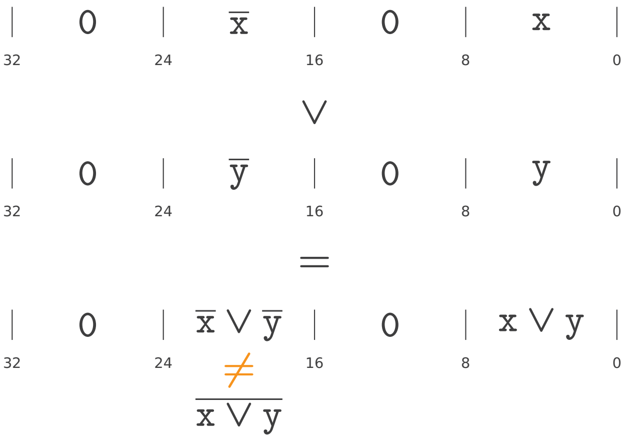


| 0 | $\overline{x}$ | 0 | x |

32 24 16 8 0

Arithmetic

Regular operators will not work:


| 0 | $\overline{x}$ | 0 | x |
32 24 16 8 0
$\vee$
| 0 | $\overline{y}$ | 0 | y |
32 24 16 8 0
$=$
| 0 | $\overline{x} \vee \overline{y}$ | 0 | $x \vee y$ |
32 24 16 8 0
$\neq$
$\overline{x \vee y}$

Arithmetic cont.

Find replacements for:

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

Arithmetic cont.

Find replacements for:

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

$$
\begin{array}{llll}
\%1 = 0 & \| \; \bar{x} & \| \; 0 & \| \; x \\
\%2 = 0 & \| \; \bar{y} & \| \; 0 & \| \; y \\
\%3 = 0 & \| \; \bar{x} \texttt{ ORR } \bar{y} & \| \; 0 & \| \; x \texttt{ ORR } y \\
\%4 = 0 & \| \; \bar{x} \texttt{ AND } \bar{y} & \| \; 0 & \| \; x \texttt{ AND } y \\
\%5 = \bar{x} \texttt{ AND } \bar{y} & \| \; \bar{x} \texttt{ ORR } \bar{y} & \| \; x \texttt{ AND } y & \| \; x \texttt{ ORR } y \\
\%6 = \overline{x \texttt{ ORR } y} & \| \; 0 & \| \; 0 & \| \; x \texttt{ ORR } y \\
\%7 = \texttt{0xFF} & \| \; \overline{x \texttt{ ORR } y} & \| \; 0 & \| \; x \texttt{ ORR } y \\
\%8 = 0 & \| \; \overline{x \texttt{ ORR } y} & \| \; 0 & \| \; x \texttt{ ORR } y
\end{array}
$$

Verifying the arithmetic

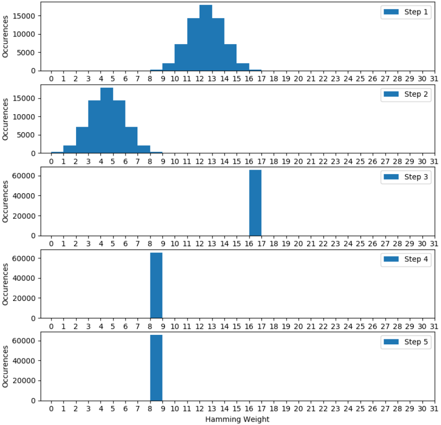
Perform exhaustive search of the input space:

```
m = MultiStepOperation([
    BinaryOperation(0, 1,
          lambda x, y: x | y),
    BinaryOperation(0, 1,
          lambda x, y: x & y),
    BinaryOperation(2, 3,
          lambda x, y: x | (y << wordsize)),
    UnaryOperation(4,
          lambda x: x & scheme2_filter),
    Convert_2_1(5)
 ])
```

Verifying the arithmetic

Perform exhaustive search of the input space:

```
m = MultiStepOperation([
  BinaryOperation(0, 1,
      lambda x, y: x | y),
  BinaryOperation(0, 1,
      lambda x, y: x & y),
  BinaryOperation(2, 3,
      lambda x, y: x | (y << wordsize)),
  UnaryOperation(4,
      lambda x: x & scheme2_filter),
  Convert_2_1(5)
 ])
```



Applying the changes

Possibilities for automatic balancing:

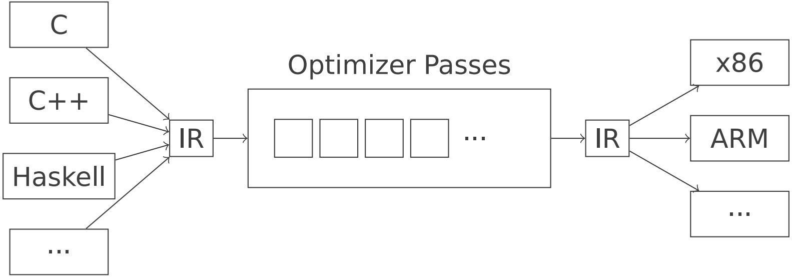
- Transform source
- During compilation

Applying the changes

Possibilities for automatic balancing:

- Transform source
- During compilation

LLVM:

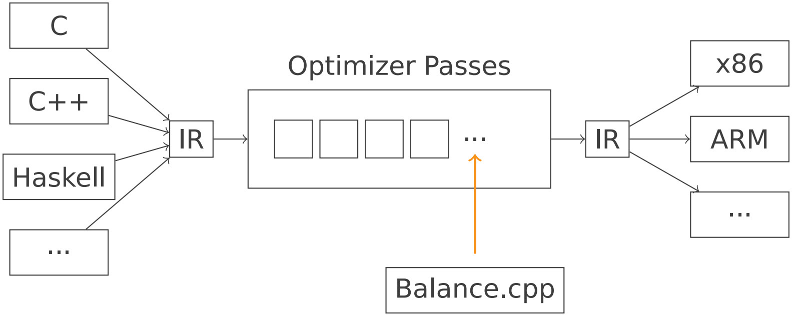


Applying the changes

Possibilities for automatic balancing:

- Transform source
- During compilation

LLVM:



Optimizer Pass

Transforms:

- function arguments
- allocations
- stores
- loads
- casts
- binary operators
- getElementPtr
- compares
- returns
- function calls

Optimizer Pass

Transforms:

- function arguments
- allocations
- stores
- loads →
- casts
- binary operators
- getElementPtr
- compares
- returns
- function calls

```
void balanceLoad(LoadInst *load,
    IRBuilder<> builder,
    vector<Instruction *> &to_remove,
    unordered_set<Value *> &balanced_values) {
  if (balanced_values
      .count(load->getPointerOperand())) {
    auto *new_load = builder
        .CreateLoad(load->getPointerOperand());
    load->replaceAllUsesWith(new_load);
    balanced_values.insert(new_load);
    to_remove.push_back(load);
    return;
  }
}
```

Binary operators

- written as C functions
- linked into same module
- llvm operators changed to calls

Tradeoff

+ simplicity
+ modularity
+ small binaries
- (currently) on inlining
- overhead

Binary operators

written as C functions

linked into same module

llvm operators changed to calls

Tradeoff

+ simplicity
+ modularity
+ small binaries

- (currently) on inlining
- overhead

```
uint32_t balanced_or(uint32_t lhs,
        uint32_t rhs) {
  uint32_t temp_or = lhs | rhs;
  uint32_t temp_and = lhs & rhs;
  uint32_t combined = (temp_and << 8)
          | temp_or;
  combined &= 0xff0000ff;
  return balanced_2_1(combined);
}
```

Optimizer Pass cont.

```
%2 = alloca i8, align 1
store i8 %0, i8* %2, align 1
%3 = load i8, i8* %2, align 1
%4 = zext i8 %3 to i32
%5 = shl i32 %4, 1
%6 = load i8, i8* %2, align 1
%7 = zext i8 %6 to i32
%8 = ashr i32 %7, 7
%9 = and i32 %8, 1
%10 = mul nsw i32 %9, 27
%11 = xor i32 %5, %10
%12 = trunc i32 %11 to i8
ret i8 %12
```

```
%2 = alloca i32
store i32 %0, i32* %2, align 1
%3 = load i32, i32* %2
%4 = call i32
   @balanced_shl(i32 %3, i32 0xfe0001)
%5 = load i32, i32* %2
%6 = call i32
   @balanced_ashr(i32 %5, i32 0xf80007)
%7 = call i32
   @balanced_and(i32 %6, i32 0xfe0001)
%8 = call i32
   @balanced_mul(i32 %7, i32 0xe4001b)
%9 = call i32
   @balanced_xor(i32 %4, i32 %8)
ret i32 %9
```

Evaluation

How to generate “virtual” power traces?

Qemu alone

- + fast
- - wrong resolution

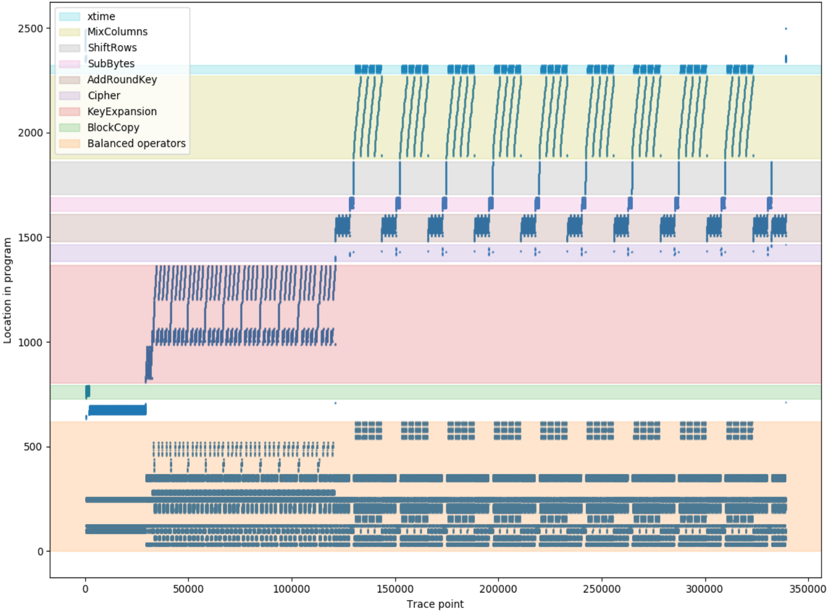
Qemu + gdb

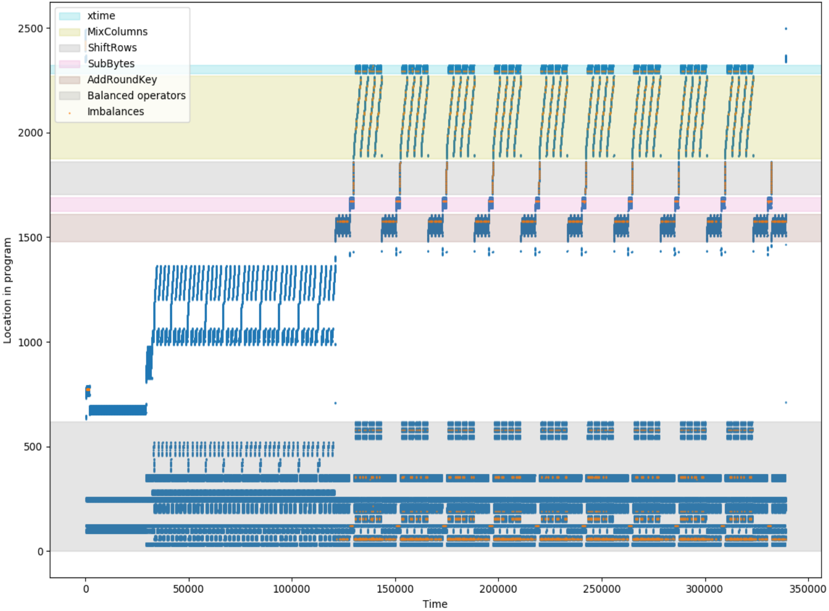
- + correct resolution
- + includes program location information
- - **very** slow

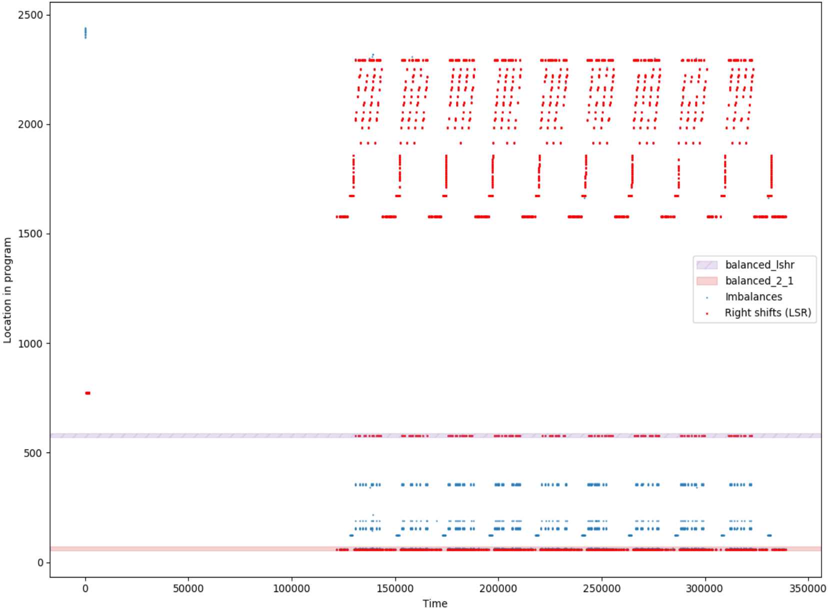
Execute instruction by instruction, dump registers every time

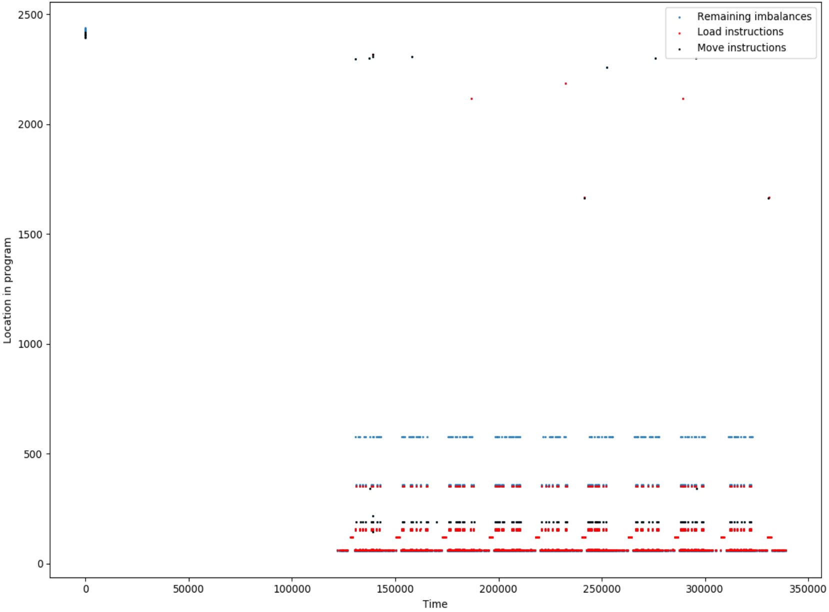
Results

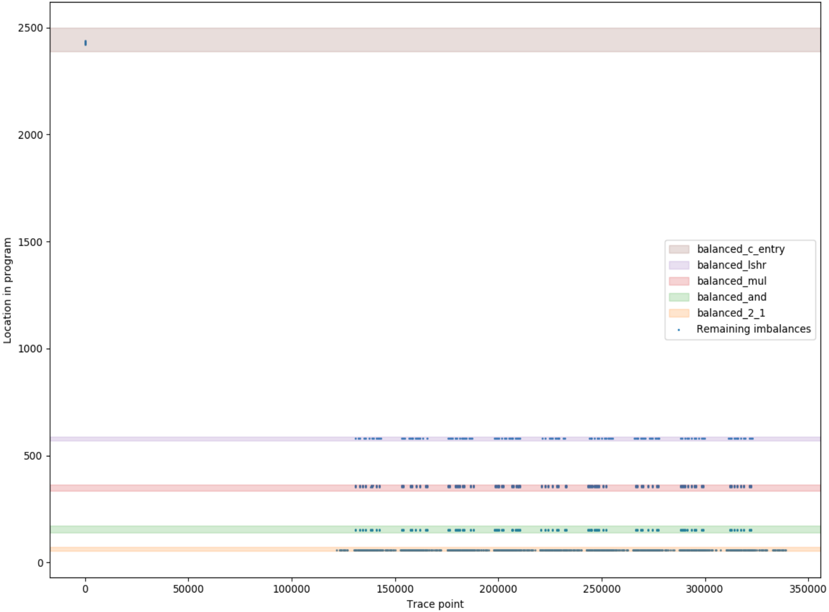
	AES	
	unbalanced	balanced
No. of instructions	22 876	339 168
Relative increase	1	14.888
Balanced operations	20 571	334 521
Unbalanced operations	2211	4647
Balancedness	0.903	0.986
Code size	76 KB	78 KB


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


xtime
MixColumns
ShiftRows
SubBytes
AddRoundKey
Balanced operators
Imbalances
Location in program
2500
2000
1500
1000
500
0
Time
0
50000
100000
150000
200000
250000
300000
350000


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


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


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

Results

	AES	
	unbalanced	balanced
No. of instructions	22 876	339 168
Relative increase	1	14.888
Balanced operations	20 571	334 521
Unbalanced operations	2211	4647
Balancedness	0.903	0.986
Code size	76 KB	78 KB

Filtered Results

	AES	
	unbalanced	balanced
No. of instructions	22 876	339 168
Relative increase	1	14.888
Balanced operations	20 571	**337 852**
Unbalanced operations	2211	**1316**
Balancedness	0.903	**0.996**
Code size	76 KB	78 KB

Note: no filtering applied to unbalanced variant

Future work

Same idea with different methods:

- Test on actual hardware
- Balance globals
- Mark balancing targets
- Move balancing to type system

Different ideas with same method:

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

Conclusion