universität
innsbruck

Defending against power analysis by balancing binary values

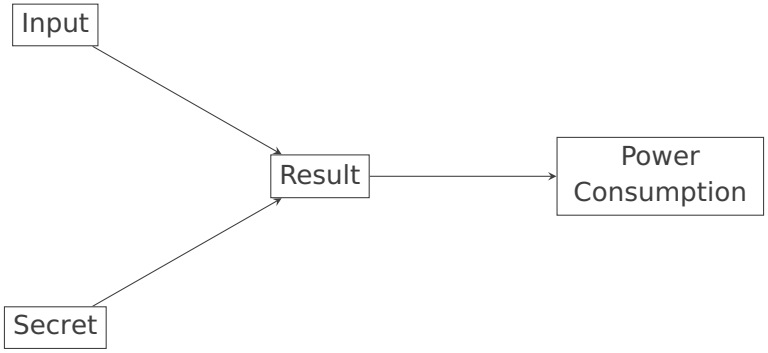
a compiler based approach

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

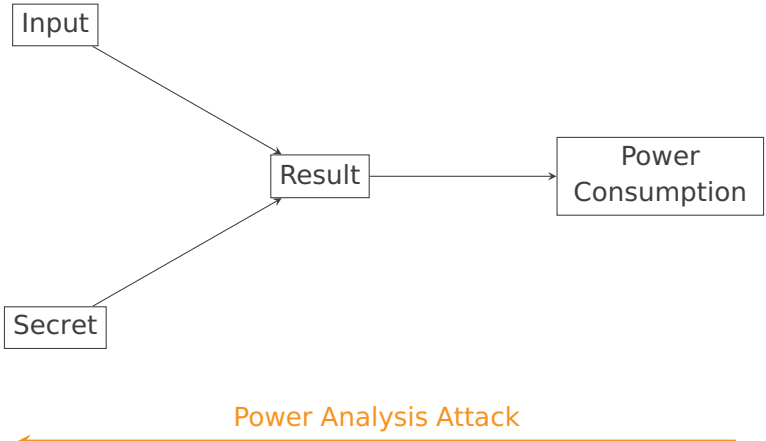
Motivation

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

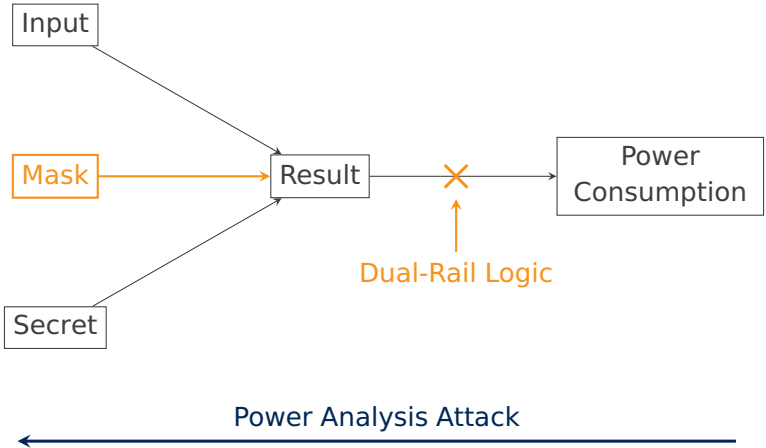
Motivation



Motivation


Input
Result
Power
Consumption
Secret
Power Analysis Attack

Motivation


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
- \- 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
- \- Specialized hardware

Best of both worlds

Apply balancing similar to Dual-Rail logic in software

Overview

Content

- Motivation
- Balancing
- Arithmetic
- Code Transformation
- Results
- Future Work & Conclusion

Overview

Content

- Motivation
- Balancing
- Arithmetic
- Code Transformation
- Results
- Future Work & Conclusion

Balancing

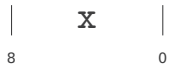
Working assumption:
Power consumption is directly proportional to Hamming weight
$\rightarrow$ constant Hamming weight = constant power consumption

Balancing

Working assumption:
Power consumption is directly proportional to Hamming weight
$\rightarrow$ constant Hamming weight = constant power consumption

Approach

Extend register size, and store inverse along with actual value



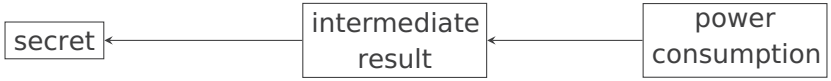
Balancing

Working assumption:
Power consumption is directly proportional to Hamming weight
$\rightarrow$ constant Hamming weight = constant power consumption

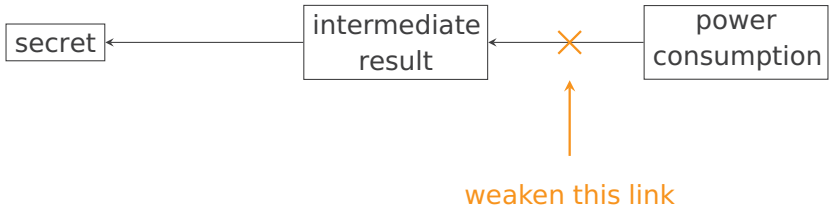
Approach

Extend register size, and store inverse along with actual value

Approach



Approach



Working assumption

Power consumption is proportional to Hamming weight

Approach cont.

constant Hamming weight $\rightarrow$ constant power consumption

char:

balanced char:

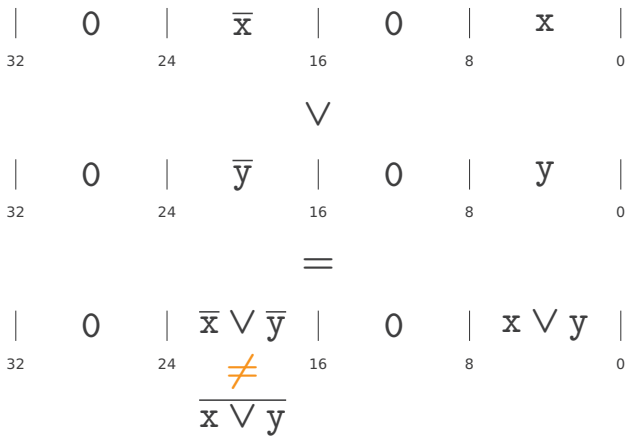
Overview

Content

- Motivation
- Balancing
- Arithmetic
- Code Transformation
- Results
- Future Work & Conclusion

Arithmetic

Regular operators will not work:



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

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

Verifying the arithmetic

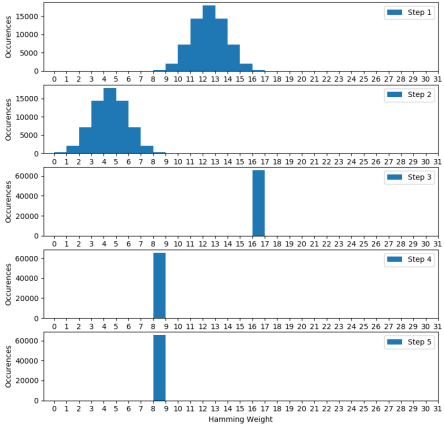
Perform exhaustive search of the input space:

```
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)
 ])
```



Overview

Content

- Motivation
- Balancing
- Arithmetic
- Code Transformation
- Results
- Future Work & Conclusion

Applying the changes

Possibilities for automatic balancing:

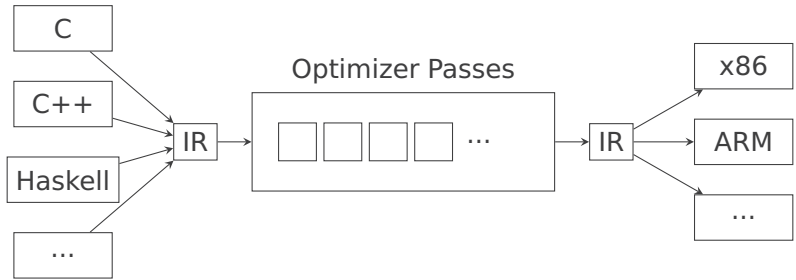
- Transform source
- Transform during compilation

Applying the changes

Possibilities for automatic balancing:

- Transform source
- Transform during compilation

LLVM:

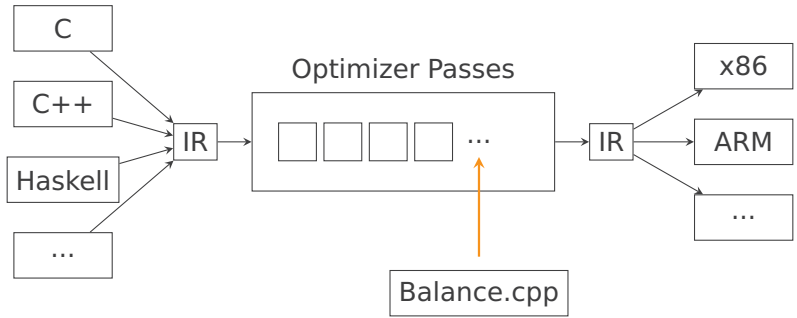


Applying the changes

Possibilities for automatic balancing:

- Transform source
- Transform 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
```

Overview

Content

- Motivation
- Balancing
- Arithmetic
- Code Transformation
- Results
- Future Work & Conclusion

Evaluation

How to generate “virtual” power traces?

Qemu alone

- \+ fast
- \- wrong resolution

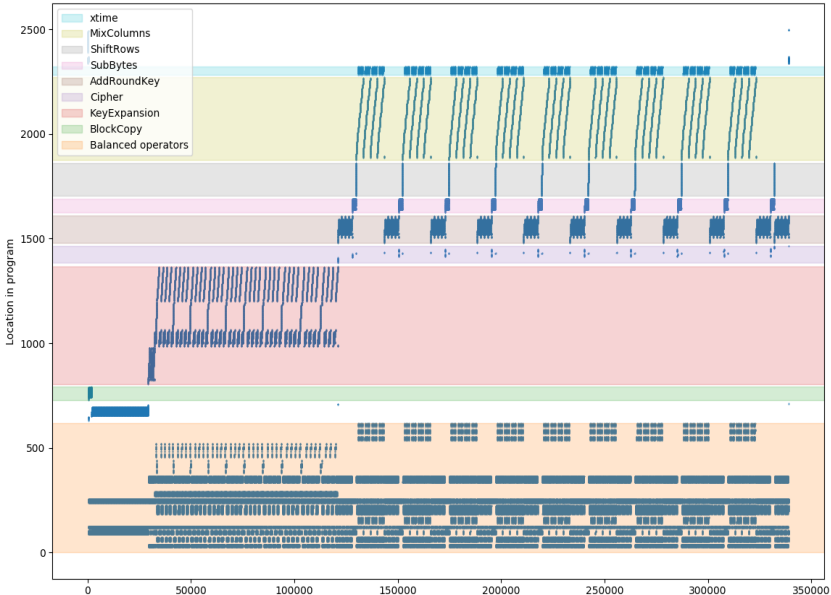
Qemu + gdb

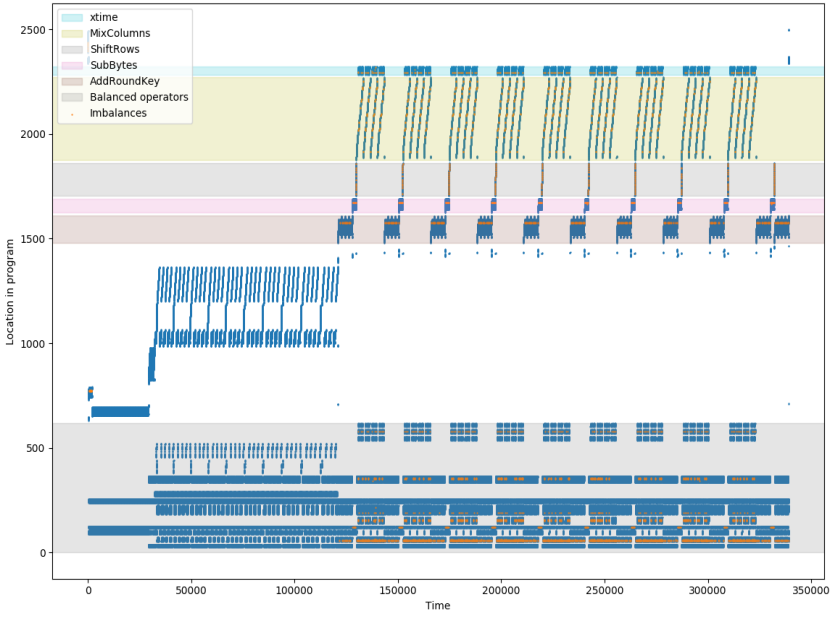
- \+ correct resolution
- \+ includes program location information
- \- **very** slow

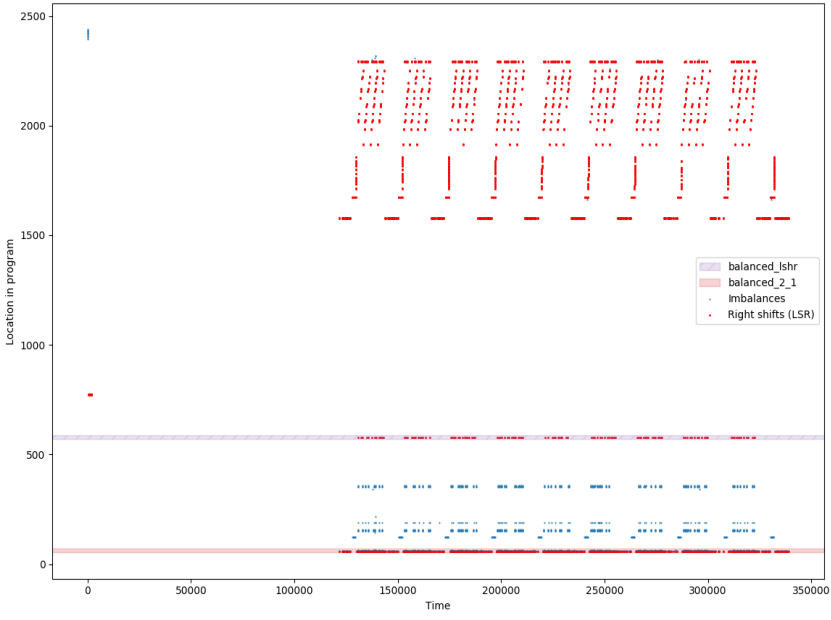
Execute instruction by instruction, dump registers every time

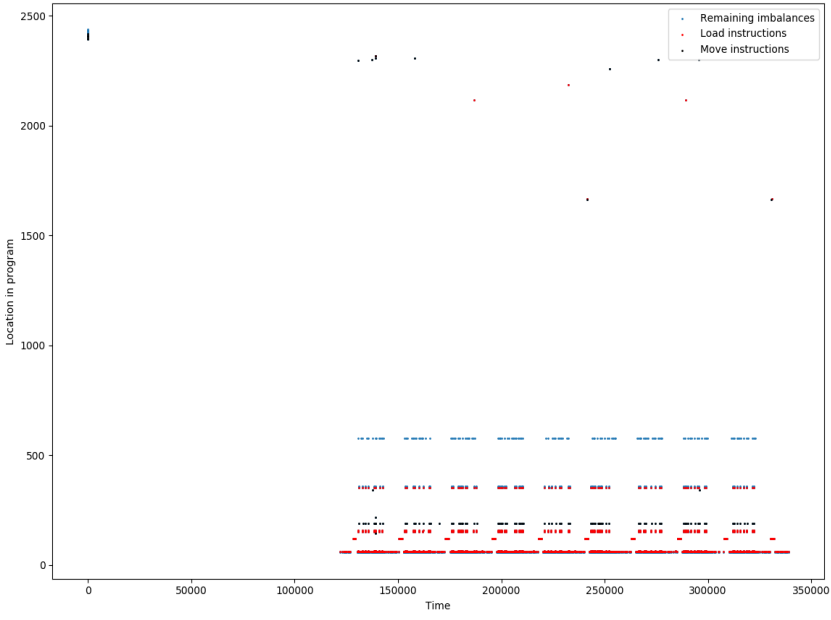
Results

	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
Location in program
Trace point
0
500
1000
1500
2000
2500
0
50000
100000
150000
200000
250000
300000
350000


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


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
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
Trace point
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

Overview

Content

- Motivation
- Balancing
- Arithmetic
- Code Transformation
- Results
- Future Work & Conclusion

Future work

Same idea with different methods:

- Test on actual hardware
- Balance globals
- Improve operators
- 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

- Increased robustness without program modifications
- Requires more powerful, but standard hardware
- 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.