



# Defending against power analysis by balancing binary values a compiler based approach

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

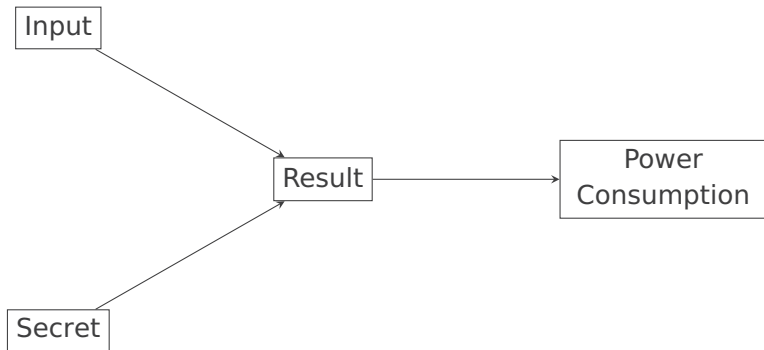
# Motivation



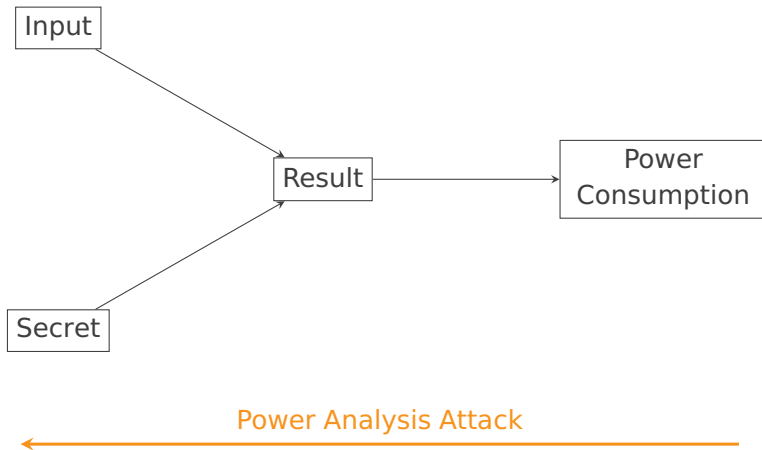
---

[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://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)

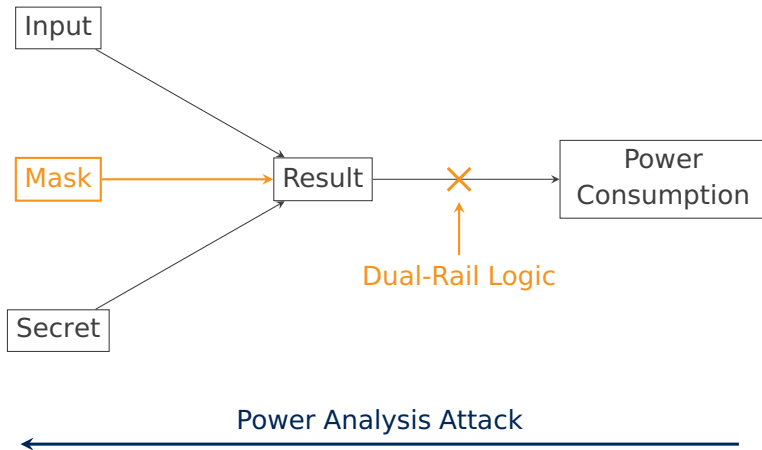
# Motivation



# Motivation



# Motivation



# 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
- Future Work & Conclusion



# Overview

## Content

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

# Balancing

## **Working assumption:**

Power consumption is proportional to Hamming weight

→ constant Hamming weight = constant power consumption

# Balancing

## **Working assumption:**

Power consumption is proportional to Hamming weight

→ 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

→ constant Hamming weight = constant power consumption

## Approach

Extend register size, and store inverse along with actual value



# Overview

## Content

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

# Arithmetic

Regular operators will not work:

$$\begin{array}{c} \begin{array}{c|c|c|c|c|c} & 0 & & \bar{x} & & 0 \\ \hline 32 & & 24 & & 16 & \\ \hline \end{array} & \begin{array}{c|c|c|c|c|c} & 0 & & \bar{y} & & 0 \\ \hline 32 & & 24 & & 16 & \\ \hline \end{array} \\ \vee \\ \begin{array}{c|c|c|c|c|c} & 0 & & \bar{x} \vee \bar{y} & & 0 \\ \hline 32 & & 24 & & 16 & \\ \hline \end{array} & \begin{array}{c|c|c|c|c|c} & 0 & & x & & 0 \\ \hline 32 & & 24 & & 16 & \\ \hline \end{array} & \begin{array}{c|c|c|c|c|c} & 0 & & y & & 0 \\ \hline 32 & & 24 & & 16 & \\ \hline \end{array} \\ = \\ \begin{array}{c|c|c|c|c|c} & 0 & & \bar{x} \vee \bar{y} & & 0 \\ \hline 32 & & 24 & & 16 & \\ \hline \end{array} & \begin{array}{c|c|c|c|c|c} & 0 & & x \vee y & & 0 \\ \hline 32 & & 24 & & 16 & \\ \hline \end{array} \\ \neq \\ \hline \bar{x} \vee \bar{y} & \hline x \vee y \end{array}$$


# 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

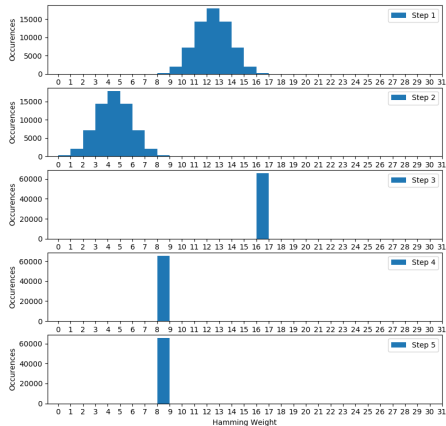
- Takes individual steps as lambdas
- Executes over all inputs
- Stores intermediate values
- Checks correctness
- Plots Hamming weight histograms

# 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



# Overview

## Content

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

# Applying the changes

## **Automatic balancing**

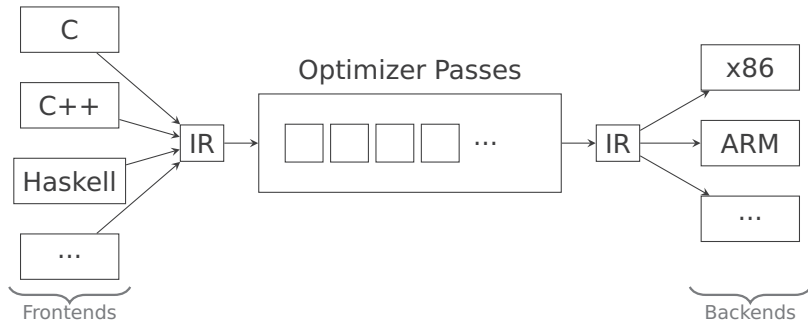
Rewrite code during compilation

# Applying the changes

## Automatic balancing

Rewrite code during compilation

LLVM:

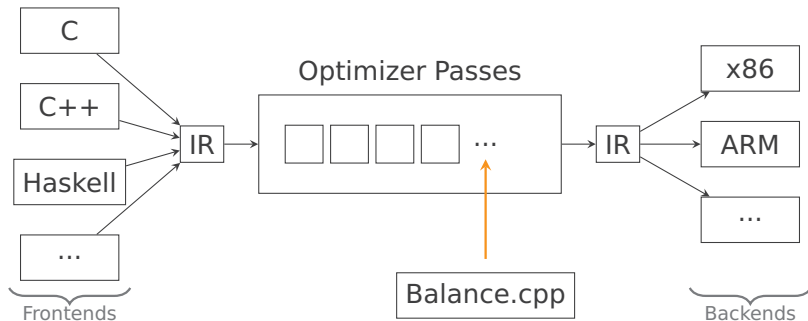


# Applying the changes

## Automatic balancing

Rewrite code during compilation

LLVM:



# Optimizer Pass


## 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

```
int 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

```
%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

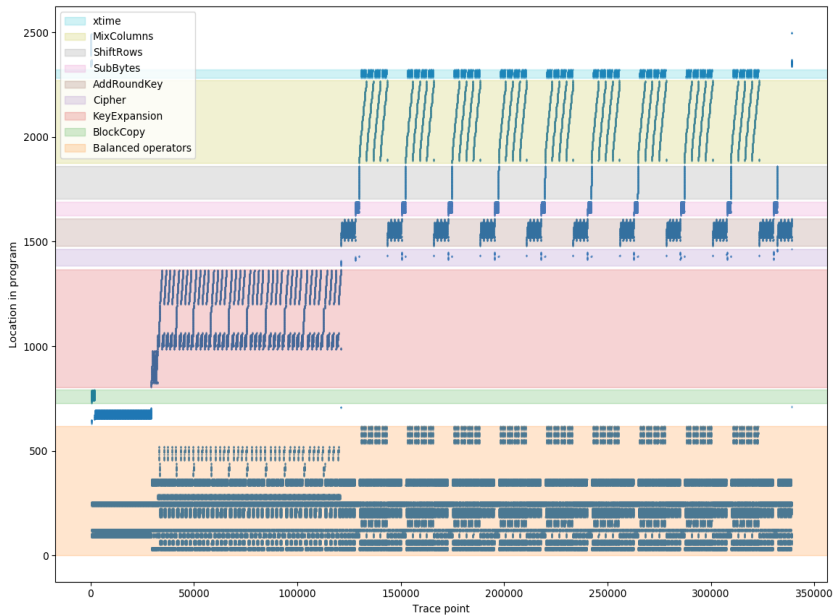
## Qemu + gdb

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

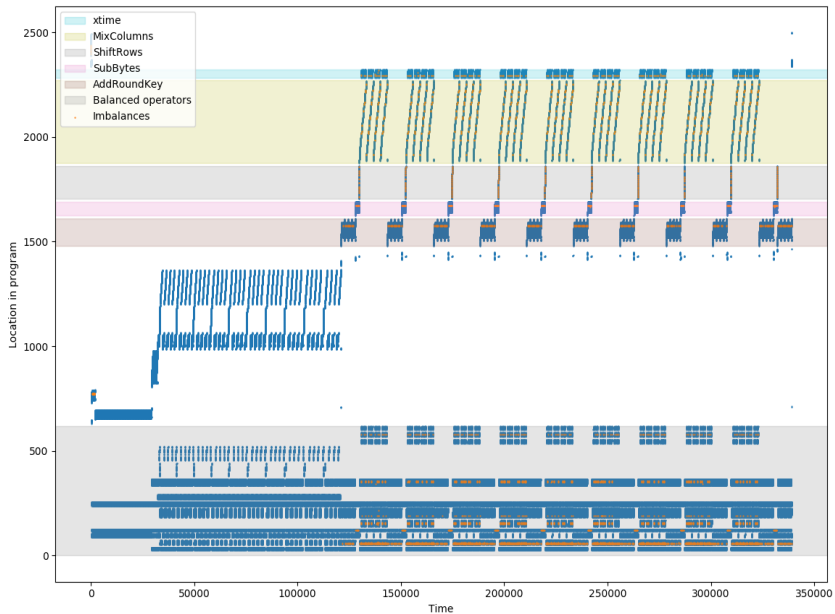
Execute instruction by instruction, dump registers every time

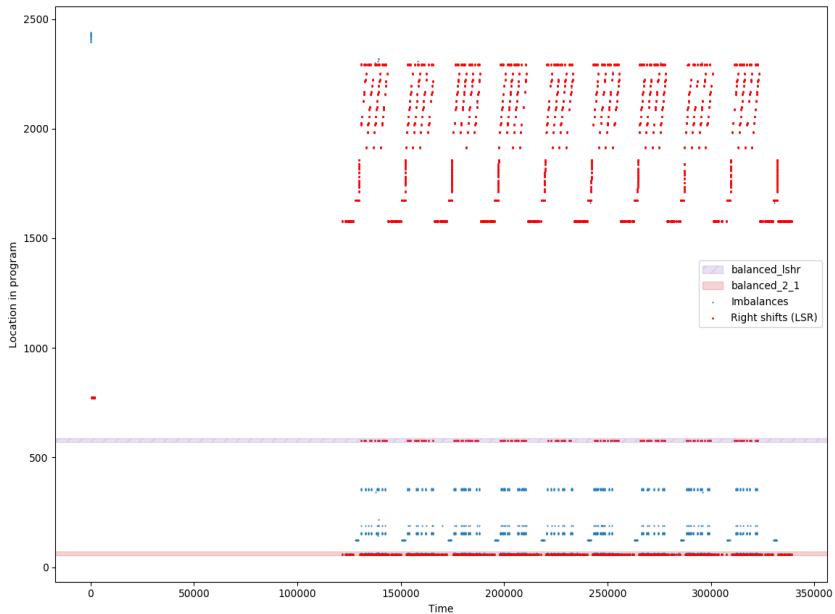
# Results

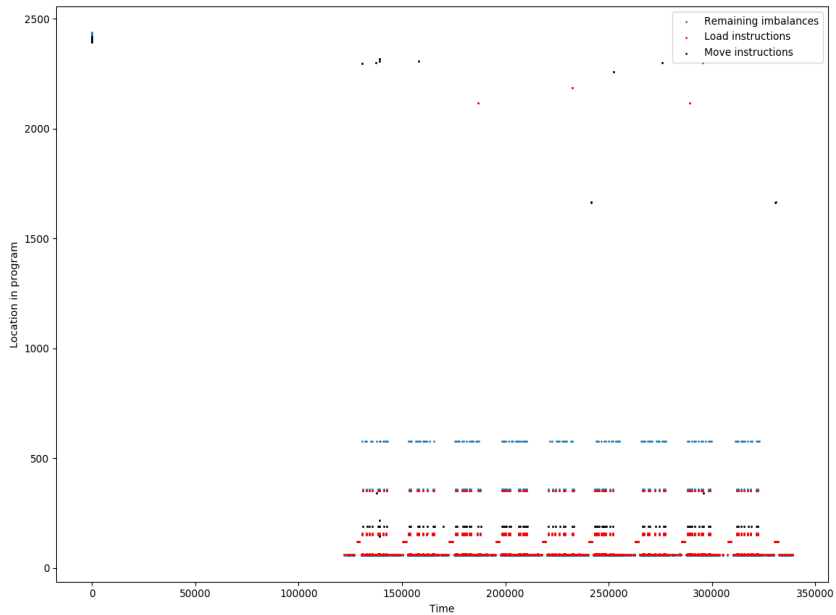
	AES	
	unbalanced	balanced
No. of instructions	22 876	339 168
Relative increase	1	14.888
Balanced operations	20 571	334 521
Unbalanced operations	2211	4647
Balancedness	0.903	0.986
Code size	76 KB	78 KB

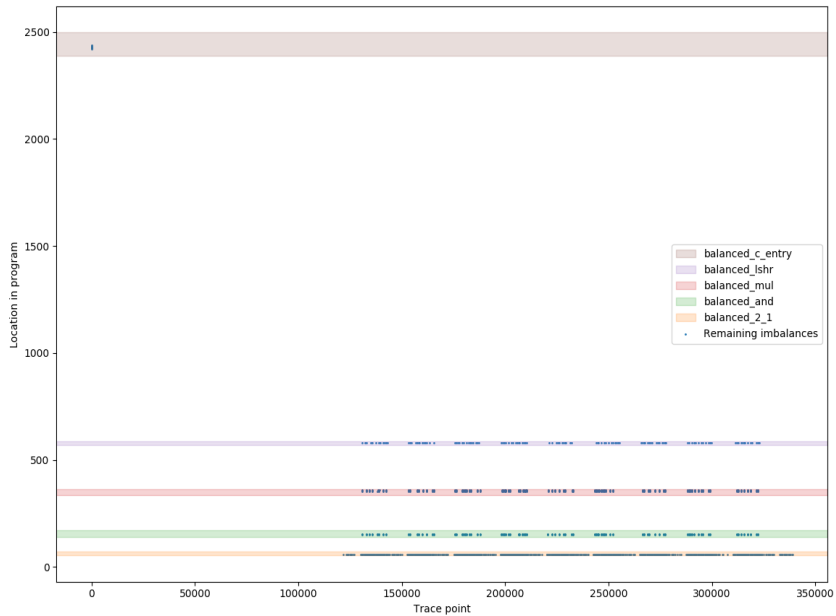












# Results

	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	<b>337 852</b>
Unbalanced operations	2211	<b>1316</b>
Balancedness	0.903	<b>0.996</b>
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 optimization, but also for security.