Toward Balancing Arbitrary Code

June 12, 2019

Contents

1 Introduction

Embedded devices very rarely utilize instruction level parallelism. Thus, as the power consumption is directly related to the bits in intermediate results that are set to 1, their power consumption directly reflects their computation results without much noise. If the device is running a cryptographic operation, this can result in a leakage of keys. This is known as a power analysis side channel attack[**kocher1999differential**].

While there exist many different defenses against this, both in software and in hardware, the most versatile of them is Dual-Rail-Logic[**sokolov2005design**]. Unlike most other defense mechanisms, Dual-Rail-Logic can be applied to any program, and works by calculating the inverse result $\bar{x}$ for each intermediate result x. This way, the power consumption (which is directly linked to the number of 1s in the result) is always the same, and the program is thus more robust against power analysis. Unfortunately, using Dual-Rail-Logic requires a significant overhead, doubling the circuit size or more[**baddam2008path**]. This requirement makes it unsuitable for small embedded applications like e.g. SmartCards. In order to create a way of hardening *any* application against power analysis attacks, even when there are tight constraints on space, I would like to implement something similar to Dual-Rail-Logic in software. By balancing the values on the data bus, the registers, and the address bus, in this order of priority, I can harden execution against power analysis.
To do this, I want to find a way to represent a balanced 8-bit arithmetic in a 32-bit architecture. While representing $\bar{x}$ and x should in theory only halve the word size, I will need additional space to represent carry bits and (new) intermediate steps in the registers as well, so the word size will probably be reduced to a quarter. The idea is to find a balancing scheme that allows me to perform all arithmetic and logic operations present in the intermediate representation (IR) of the LLVM compiler. Ideally, this scheme has no unbalanced intermediate results at all and utilizes no table lookups.

After finding such a balancing scheme and arithmetic, I want to transform the original code into balanced code in a custom LLVM optimization pass. This pass will transform the IR code of the original program into my balanced arithmetic operation by operation. Keeping the performance impact of this transformation as low as possible - both during compile- and runtime - will be a major concern.

Finally I need a way of evaluating my work. For this I assume a perfect attacker capable of observing the power signature of every intermediate value. The robustness against such an attacker is then represented by the number of unbalanced values during the execution, as well as the ratio of balanced vs. unbalanced values. To find this number I run the resulting code in the QEMU emulator, observing the result of every operation. This allows me to easily test my work in a controlled environment and without any additional hardware.

The rest of this thesis is organized as follows: Section 2 gives an introduction to the tools used, as well as a brief refresher of the algorithms used for testing. Section 3 describes my approach, and Section 4 the implementation details of my thesis. Section 5 shows the evaluation results of my PoC. Finally, in Section 6 I offer a discussion of the results as well as an outlook to possible future work.

2 Background

2.1 Power Analysis Defenses

aoeu

2.2 LLVM

aoeu

2.3 QEMU

aoeu

2.3.1 Memory Layout of QEMU Kernels

aoeu

2.4 GNU Cross Tools

aoeu

2.5 AES

aoeu

2.6 RC4

aoeu

3 Methodology

3.1 Arithmetic

aoeu

3.2 Balancing Pass

aoeu

3.3 Evaluation

aoeu

4 Implementation

aoeu

4.1 Arithmetic

aoeu

4.1.1 Finding Equivalent Operations

aoeu

4.1.2 Testing For Correctness

aoeu

4.2 Build Processes

aoeu

4.2.1 Building the Compiler Pass

4.2.2 Building the Test Code

As discussed in Section 2.2 the LLVM compilation process can be split into multiple steps. I use this feature multiple times in the build process of my test code. The output of the build process for the RC4 code is shown in **??** 1.

Listing 1: Output of the Makefile

```
1  arm-none-eabi-gcc --specs=nosys.specs program.c -o
       program_unbalanced.bin
2  arm-none-eabi-as -ggdb  startup.s -o startup.o
3  clang -target arm-v7m-eabi -mcpu=arm926ej-s -O0 rtlib.c -S -emit
       -llvm -o rtlib.ll
4  clang -target arm-v7m-eabi -mcpu=arm926ej-s -O0 program.c -S -
       emit-llvm -o program.ll
5  llvm-link rtlib.ll program.ll -S -o linked.ll
6  opt -load="../../passes/build/libPasses.so" -insert linked.ll -S
        -o optimized.ll
7  Balancing module: linked.ll
8  llc optimized.ll -o optimized.S
9  arm-none-eabi-as -ggdb  optimized.S -o optimized.o
10 arm-none-eabi-ld -T startup.ld startup.o optimized.o -o program.
       elf
11 arm-none-eabi-objcopy -O binary program.elf program.bin
```

Line 1 shows the compilation of the unbalanced version that I use for comparison. This version is compiled using only the GNU ARM Cross GCC compiler. Lines 3 and 4 show the translation of the C code into LLVM code, using the Clang[**lattner2008llvm**] C frontend for LLVM. *Program.c* is the file containing the RC4 code and *rtlib.c* contains the balanced binary operations. The *-S* flag specifies output to be in human readabale LLVM IR instead of bytecode, which allows for easier debugging. The specified *-target* platform and CPU (*-mcpu*) are written into the preamble of the LLVM IR, and carried on through the entire toolchain until the compilation into target code on line 8.

Then both LLVM files are merged using *llvm-link*, which is simply a concatenation of both files and some reordering. This merger puts the functions

declared in *rtlib.c* in the same module as the target code, and makes them accessible to the compilation pass running on that module.

Line 6 runs the LLVM optimizer on the module, loading my balancing pass, which is contained in *libPasses.so*. The pass is run by issuing the flag assigned to it during registering (*-insert* in this case). As discussed in Section 2.2 both the input and output of the optimizer are LLVM IR. Again the *-S* flag is used for human readable output. Line 7 shows output of the actual compiler pass.

In line 8 the LLVM IR code is compiled into target code, in this case ARM assembly. The specification of the target platform is taken from the preamble of the LLVM IR file, as specified in the frontend call.

The final three steps are handled by the GNU Cross Tools. First the target code is assembled (line 9) and then it is linked with a prewritten memory map and a fixed startup assembly file (line 10). The memory map is required due to QEMU specifics, as described in Section 2.3.1. QEMU starts execution with the program counter set to address *0x1000* Unfortunately, I cannot control the memory layout of the code during and after the compilation process, so I have no guarantee that the *main* main function will land at the desired address. For this I use a memory map *startup.ld* (as described in [**armbare**]), which causes the code defined in *startup.s* to be at memory address *0x1000*. The content of *startup.ld* is shown in **??** 2.

Listing 2: Memory map in *startup.ld*

```
1  ENTRY(_Reset)
2  SECTIONS
3  {
4   . = 0x10000;
5   .startup . : { startup.o(.text) }
6   .text : { *(.text) }
7   .data : { *(.data) }
8   .bss : { *(.bss COMMON) }
9   . = ALIGN(8);
10  . = . + 0x1000; /* 4kB of stack memory */
11  stack_top = .;
12 }
```

The code in *startup.s* then fixes the stack location and loads the entry function *c_entry* in my test code. Its contents are shown in **??** 3.

Listing 3: Startup code in *startup.s*

```
1 .global _Reset
2 _Reset:
3  LDR sp, =stack_top
4  BL balanced_c_entry
5  B .
```

4.3 Balancing Pass

Here I will describe the individual parts of the balancing process. First is an outline of the build process (i.e. the *Makefile*) because the binary operators

are not generated in the compiler pass itself, but in a different C file. This file has to be translated into LLVM IR and then linked together with the rest of the code in order to be accessible for the compiler pass.

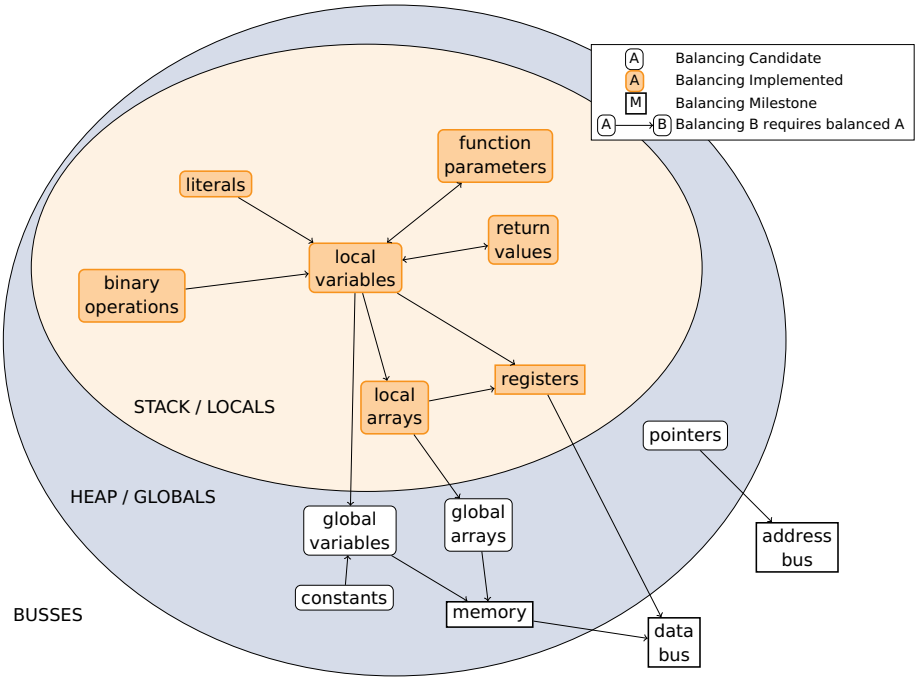


Figure 1: Balancing diagram for LLVM

4.3.1 Cloning Functions

aoeu

4.3.2 Balanced Allocates

aoeu

4.3.3 Balanced Stores

aoeu

4.3.4 Balanced Loads

aoeu

4.3.5 Balanced ZExts

aoeu

4.3.6 Balanced Operators

aoeu

4.3.7 Balanced Pointer Arithmetic

aoeu

4.3.8 Balanced Compares

aoeu

5 Results

In this section I will discuss the balancing results for the two main algorithms I tested the pass on: RC4 and AES. Both algorithms have been written/adapted so that they utilize the stack as much as possible, maximizing the benefit of my balancing pass.

5.1 Security

5.2 Performance

aoeu

6 Conclusion

aoeu

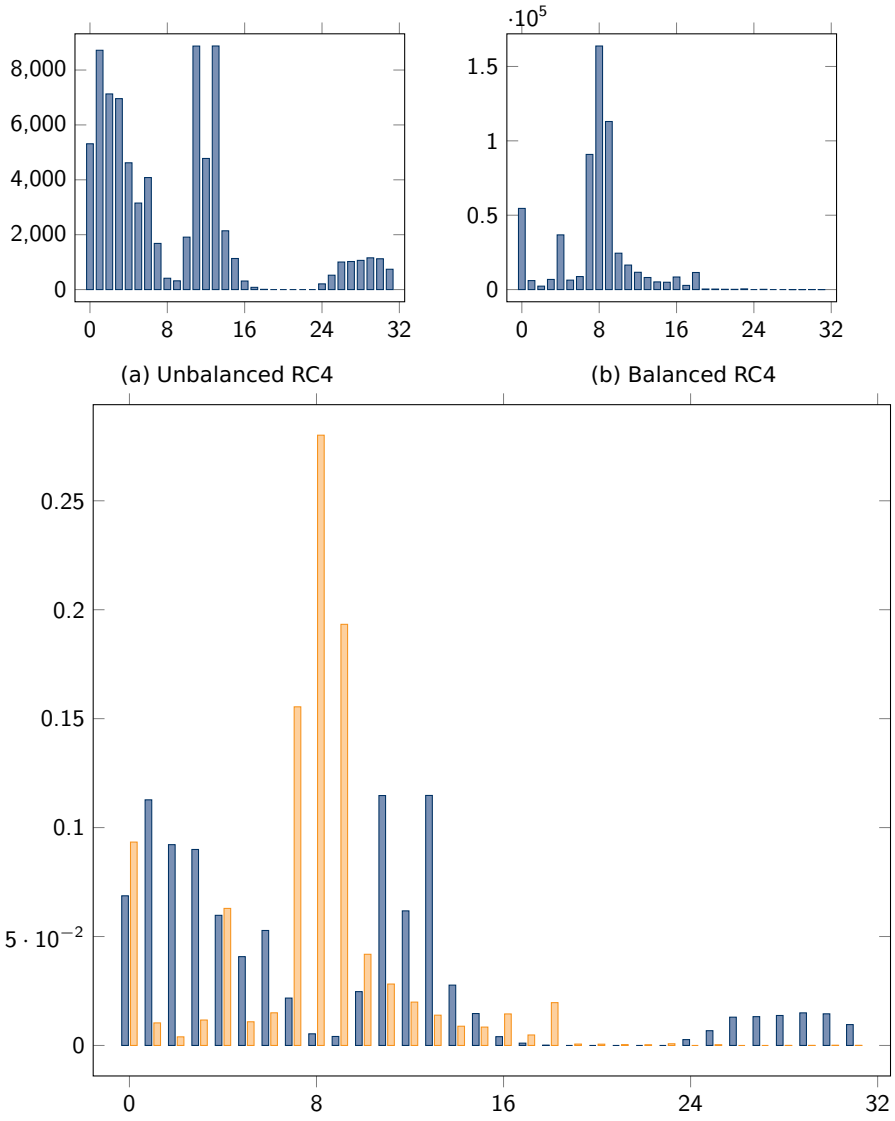
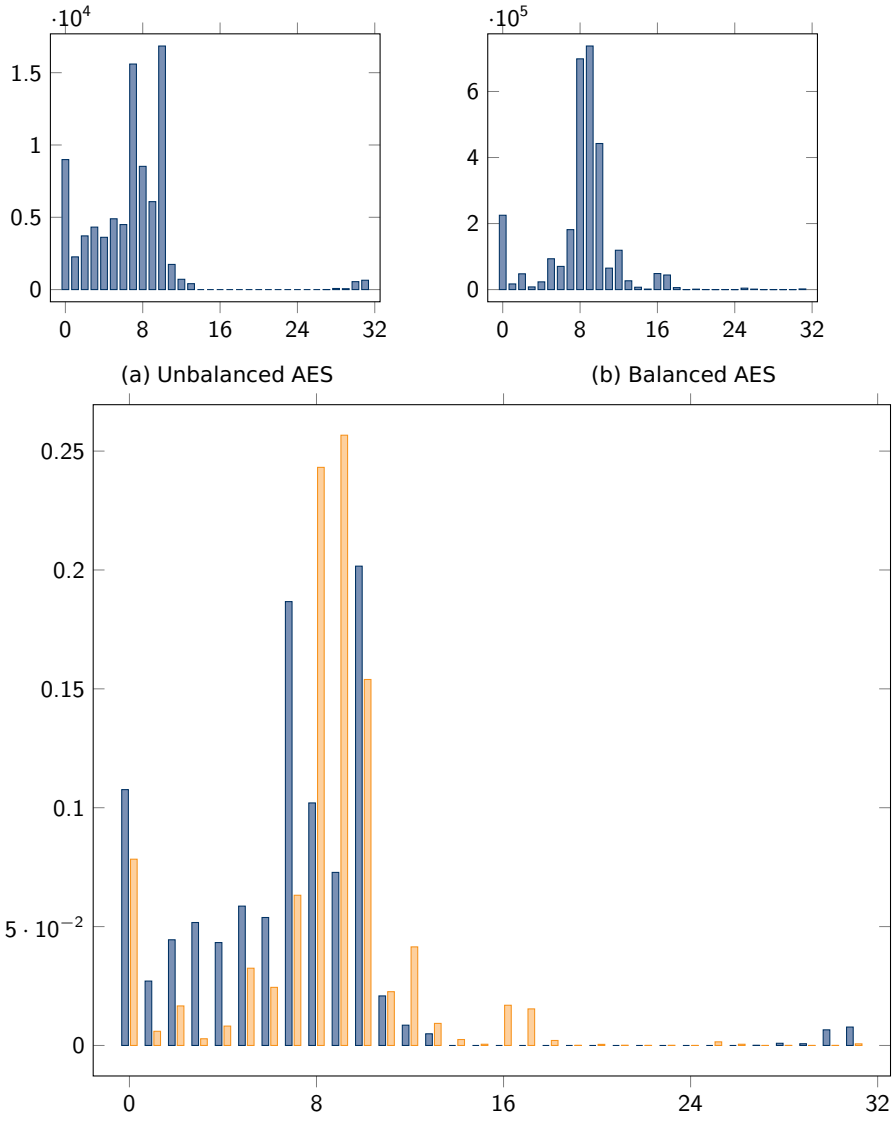


Figure 2: Hamming weight histograms for balanced and unbalanced RC4



(a) Unbalanced AES

(b) Balanced AES

(c) Scaled Hamming weight histograms for AES

Figure 3: Hamming weight histograms for balanced and unbalanced AES