Toward Balancing Arbitrary Code

June 12, 2019

Contents

1 Introduction

aoeu

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 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.

4.2.1 Cloning Functions

aoeu

4.2.2 Balancing Allocates

aoeu

4.2.3 Balancing Stores

aoeu

4.2.4 Balancing Loads

aoeu

4.2.5 Balancing ZExts

aoeu

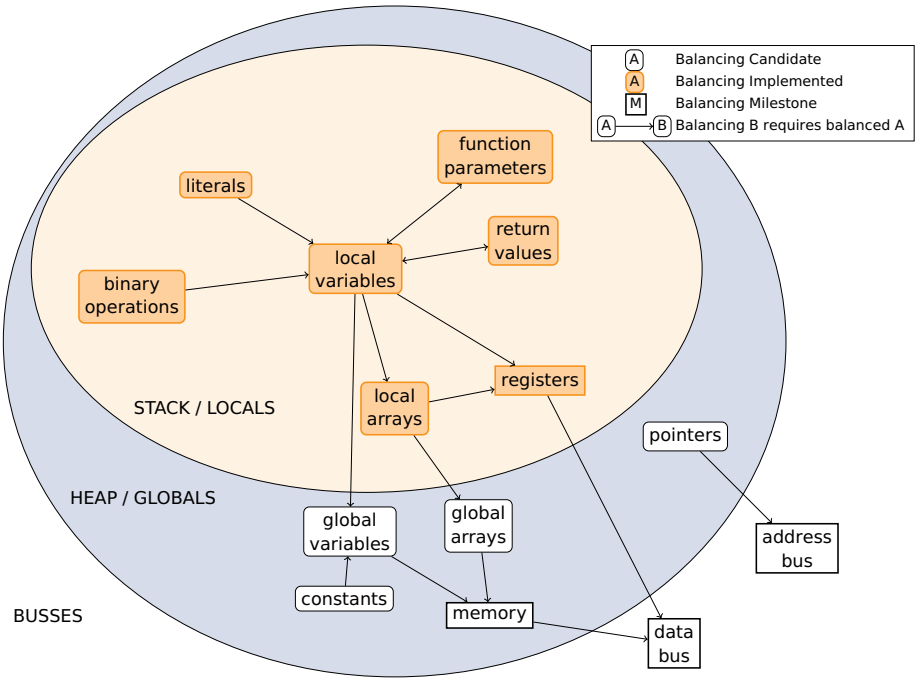


Figure 1: Balancing diagram for LLVM

4.2.6 Balancing Operators

aoeu

4.2.7 Balancing Pointer Arithmetic

aoeu

4.2.8 Balancing Compares

aoeu

4.3 Build Processes

Because my thesis project modifies the behaviour of the actual compiler and I thus need to control the individual steps of the compilation process, building the test code is a lot more involved than would be for simple cross-compilation. Building the pass itself also requires some additional configuration as it needs LLVM resources during compilation and it needs to be compatible to my build of the LLVM toolchain.

The following sections describe the build setup for the pass and the test code. They also explain why the additional steps and configurations are necessary, and include code where it benefits understanding.

4.3.1 Building the Compiler Pass

The compiler pass is built using CMake as that makes loading the required parts of LLVM very easy. **??** 1 shows the *CMakeLists.txt* for my balancing pass. The code is based on the template repository provided in [3].

Listing 1: CMake configuration for my balancing pass

```
1  cmake_minimum_required(VERSION 3.13)
2
3  find_package(LLVM REQUIRED CONFIG)
4  add_definitions(${LLVM_DEFINITIONS})
5  include_directories(${LLVM_INCLUDE_DIRS})
6  link_directories(${LLVM_LIBRARY_DIRS})
7
8  add_library(Passes MODULE
9      Insert.cpp
10 )
11
12 set(CMAKE_CXX_STANDARD 14)
13
14 # LLVM is (typically) built with no C++ RTTI. We need to match
       that;
15 # otherwise, we will get linker errors about missing RTTI data.
16 set_target_properties(PROPERTIES
17     COMPILE_FLAGS "-fno-rtti"
18 )
```

It uses the *find_package* function of CMake, which sets the locations for definitions, header files, and link directories. All these locations are needed to build my pass. The pass itself is then built as a *MODULE* library, which tells CMake to build a shared library (*.so* file) that can be dynamically loaded at runtime by the optimizer. As the pass is loaded by the optimizer, which is usually built without run-time type information (RTTI), the pass needs to be built without RTTI as well.

4.3.2 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 **??** 2.

Listing 2: 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[2] 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 [1]), which causes the code defined in *startup.s* to be at memory address *0x1000*. The content of *startup.ld* is shown in **??** 3.

Listing 3: 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 **??** 4.

Listing 4: Startup code in *startup.s*

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

4.4 Evaluation using QEMU

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

References

[1] URL: `https://balau82.wordpress.com/2010/02/28/hello-world-for-bare-metal-arm-using-qemu/` (visited on 06/12/2019).

[2] Chris Lattner. “LLVM and Clang: Next generation compiler technology”. In: *The BSD conference*. Vol. 5. 2008.

[3] Adrian Sampson. *LLVM for Grad Students*. 2015. URL: `http://www.cs.cornell.edu/~asampson/blog/llvm.html` (visited on 06/12/2019).

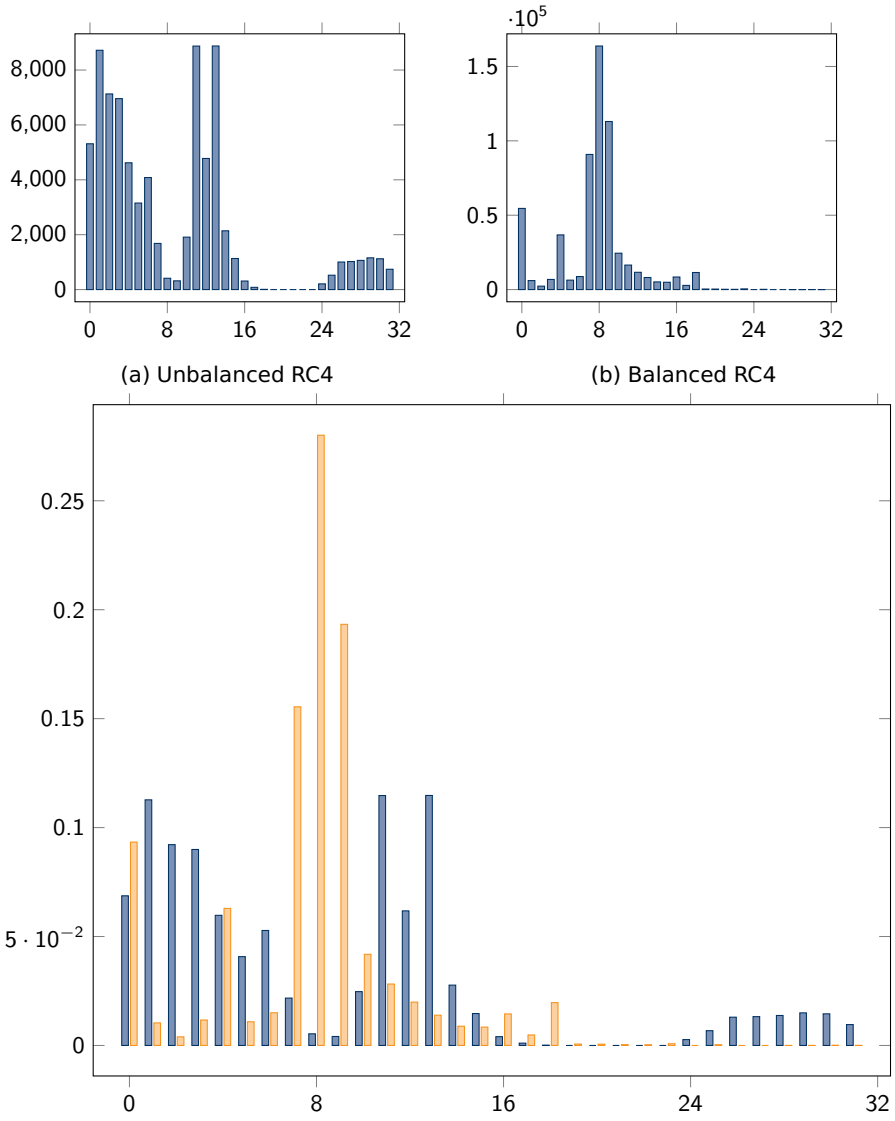
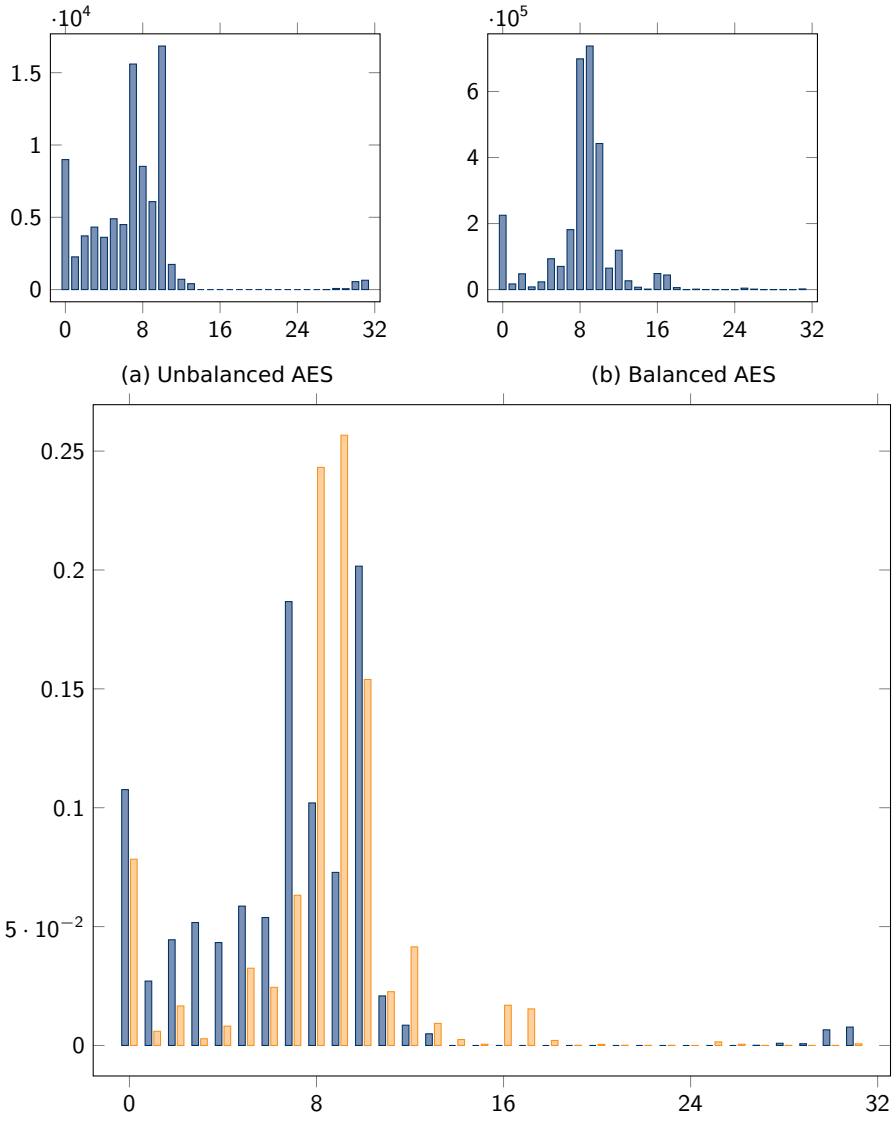


Figure 2: Hamming weight histograms for balanced and unbalanced RC4



(a) Unbalanced AES

(b) Balanced AES

(c) Scaled Hamming weight histograms for AES

Figure 3: Hamming weight histograms for balanced and unbalanced AES