Toward Balancing Arbitrary Code

June 13, 2019

Contents

1 Introduction

aoeu

2 Background

2.1 Power Analysis Defenses

aoeu

2.2 LLVM

aoeu

2.3 Static Single Assignment Form

2.4 LLVM Intermediate Representation

2.5 LLVM C++ API

2.6 QEMU

aoeu

2.6.1 Memory Layout of QEMU Kernels

aoeu

2.7 GNU Cross Tools

aoeu

2.8 AES

aoeu

2.9 RC4

aoeu

3 Methodology

3.1 Arithmetic

aoeu

3.2 Balancing Pass

aoeu

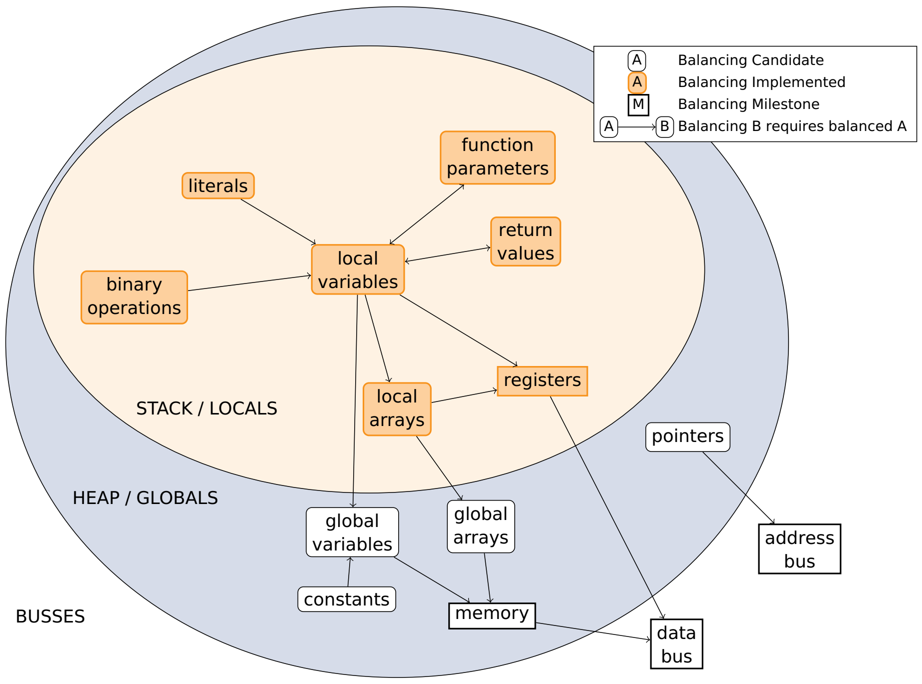


Figure 1: Balancing diagram for LLVM

3.3 Evaluation

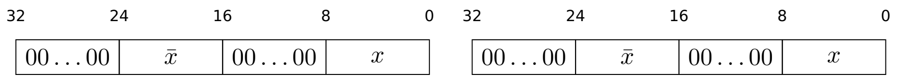
aoeu

4 Implementation

aoeu

4.1 Arithmetic

aoeu



(a) Balancing Scheme 1 (b) Balancing Scheme 2

Figure 2: Balancing Schemes

4.1.1 Finding Equivalent Operations

aoeu

4.1.2 Testing For Correctness

aoeu

4.2 Balancing Pass

The idea behind the balancing pass is very simple.

1. Change the type of all 8bit integers (*int8*) to 32bit integers (*int3*2)
2. Use balanced arithmetic operations instead of regular operators
3. Fix comparison directions
4. Fix type issues that arise in the instructions that have not been replaced

In the following subsections I will describe the changes that my pass makes, ordered by the lifecycle of a variable in LLVM IR. As we decided to focus on stack memory only, the operations and lifecycle are specific to local variables.

4.2.1 Cloning Functions

The first types used in a function are its return type and the types of its function parameters. As these types cannot be changed for an existing function in LLVM I need to clone the functions with updated types.

Cloning functions is done in two parts. First the prototype for the new function is created. During creation the pass goes through all parameter and changes their types from *int8* to *int3*2. The same is done for the return type. This gives me a skeleton for the balanced function, which can be inserted into the module, making it accessible in the future.

Even though copying the instructions from the original function to the balanced function should be more complex in theory, LLVM provides a nice way of doing this. The main work of copying the instructions is done by the LLVM function *CloneFunctionInto*. This function takes a *ValueToValueMapper* as parameter, which can be used to replace SSA references in instructions. My pass uses this mapper to update references of the function parameters. This change alone results in type mismatches and keeps the code from compiling successfully, but it gives me a nice entry point to start balancing my functions from.

4.2.2 Balancing Allocations

In order to declare and use local variables in LLVM IR the memory for them first has to be allocated using the *alloca* instruction. Even function parameters are not used directly but first copied into memory explicitly allocated for this function. Note that even though the naming is similare to C's *malloc* call, the memory for *alloca* is on the stack in this case.

The *alloca* instruction takes the type to be allocated as parameter, and returns a pointer to that type. This means that for balancing all the pass has to do is replace the *alloca* for *int8* with one for *int3*2. Allocations for local arrays work the same way, the pass just needs to extract the number of elements from the old allocation.

4.2.3 Balancing Stores

It can happen that the target code tries to store a balanced variable (*int3*2) into an unbalanced pointer (*int8*). In this case the pass unbalances the variable in a temporary before storing it.

While this does cause information leakage and a reduction in robustness, such a case can be avoided fairly easily. As only global memory is unbalanced, this does not happen when the program stores all values on the stack.

4.2.4 Balancing Loads

Balancing loads is a mirror case of balancing stores. When loading from an unbalanced pointer into a balanced variable, the pass first loads into an unbalanced temporary and then balances the value before storing it in the local variable.

4.2.5 Balancing ZExts

ZExt stands for zero extend, and it is an instruction used to promote integer types to larger bit sizes. While my pass is meant to balance code utilizes *only 8bit integers*, I needed to balance *ZExts* for compatibilty reasons during development and have left the balancing procedure in the code.

When zero extending from 8 to 32 bit, the pass replaces the instruction with a call to my balance function. When extending from 32 to 64 bit it unbalances the value first and then zero extends to the target type.

4.2.6 Balancing Binary Operations

I implemented the balanced operations described in Section 4.1.1 in C, each as an individual function. In order to balance binary operations they need to be replaced by calls to these new functions. As all binary operations are represented by the same instruction in the LLVM API, the pass needs to examine the *opcode* of the instruction. Based on that it decides which function to call.

4.2.7 Balancing Pointer Arithmetic

Balanced values cannot be used as indices directly. Therefore, whenever a balanced variable is used as index for an array access it is unbalanced before usage. All array accesses use the *getelementptr* instruction in LLVM IR, so this is easy to catch. What is not handled is manual arithmetic with pointers, but that is by design.

4.2.8 Balancing Compares

In my main balancing scheme (Figure 2a) the inverse occupies more significant bits than the value itself. This changes the direction of comparison operations, meaning $<$ becomes $>$, $>=$ becomes $<=$ etc. For $==$ nothing changes and the other comparisons are simply replaced.

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

```
cmake_minimum_required(VERSION 3.13)

find_package(LLVM REQUIRED CONFIG)
add_definitions(${LLVM_DEFINITIONS})
include_directories(${LLVM_INCLUDE_DIRS})
link_directories(${LLVM_LIBRARY_DIRS})

add_library(Passes MODULE
    Insert.cpp
)

set(CMAKE_CXX_STANDARD 14)

# LLVM is (typically) built with no C++ RTTI. We need to match
    that;
# otherwise, we will get linker errors about missing RTTI data.
set_target_properties(PROPERTIES
    COMPILE_FLAGS "-fno-rtti"
)
```

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 2.6.1. 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*

```
ENTRY(_Reset)
SECTIONS
{
 . = 0x10000;
 .startup . : { startup.o(.text) }
 .text : { *(.text) }
 .data : { *(.data) }
 .bss : { *(.bss COMMON) }
 . = ALIGN(8);
 . = . + 0x1000; /* 4kB of stack memory */
 stack_top = .;
}
```

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*

```
.global _Reset
_Reset:
 LDR sp, =stack_top
 BL c_entry
 B .
```

4.4 Evaluation using QEMU

aoeu

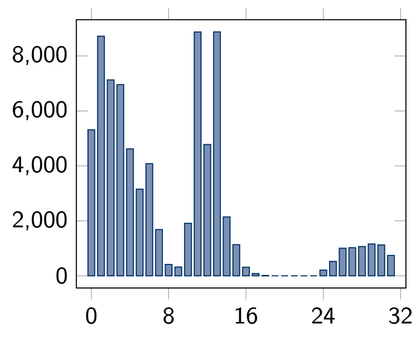
5 Nonfeatures and Limitations

aoeu

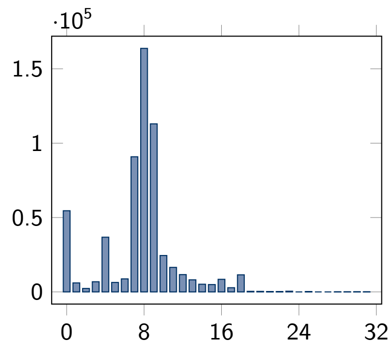
6 Results

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.

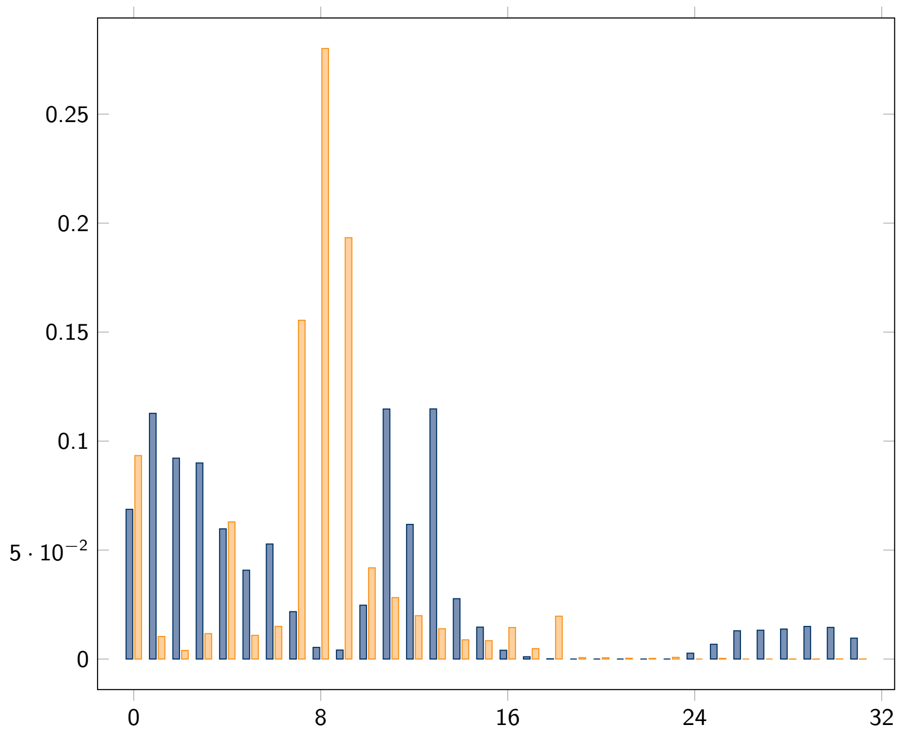
6.1 Security



(a) Unbalanced RC4



(b) Balanced RC4



(c) Scaled Hamming weight histograms for RC4

Figure 3: Hamming weight histograms for balanced and unbalanced RC4

6.2 Performance

aoeu

7 Conclusion

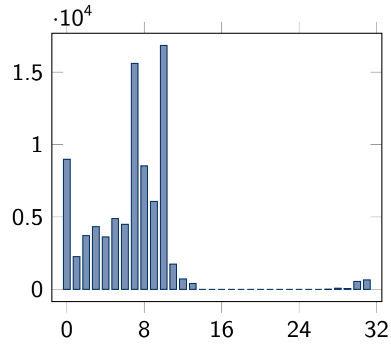
aoeu

References

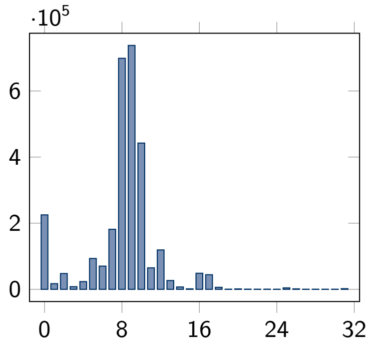
[1] URL: `https://balau82.wordpress.com/2010/02/28/hello-world-for-bare-metal-arm-using-qemu/` (visited on 06/12/2019).

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

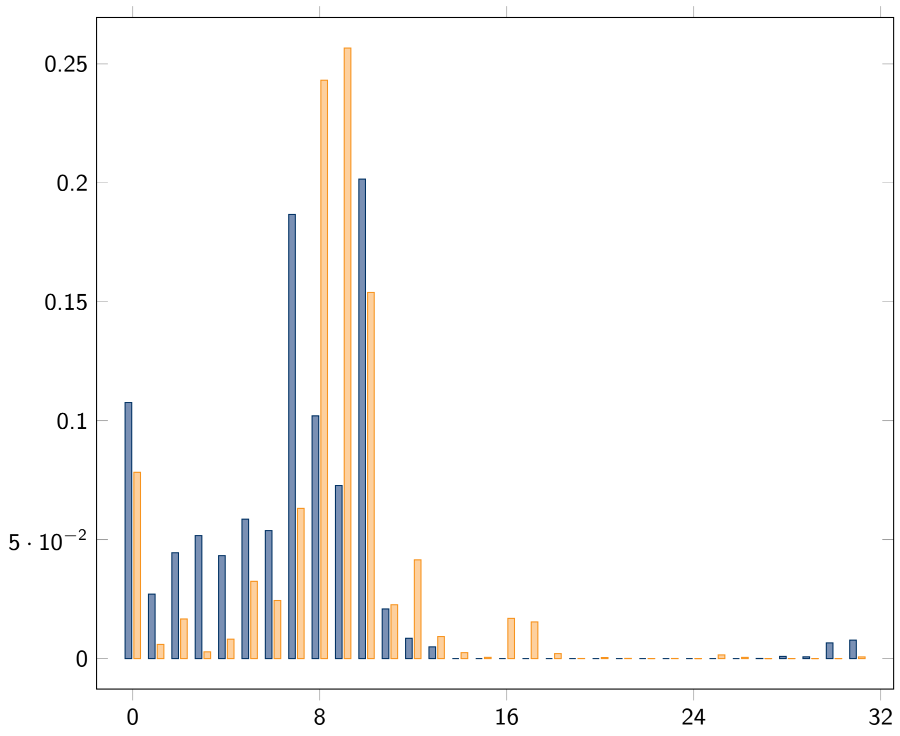
[3] Adrian Sampson. *LLVM for Grad Students*. 2015. URL: `http://www.cs.cornell.edu/~asampson/blog/llvm.html` (visited on 06/12/2019).



(a) Unbalanced AES



(b) Balanced AES



(c) Scaled Hamming weight histograms for AES

Figure 4: Hamming weight histograms for balanced and unbalanced AES