Toward Balancing Arbitrary Code

June 18, 2019

Contents

1 Introduction

aoeu

2 Background

2.1 Power Analysis Defenses

aoeu

2.2 LLVM

aoeu

2.3 Static Single Assignment Form

aoeu

2.4 LLVM Intermediate Representation

aoeu

2.5 LLVM C++ API

aoeu

2.6 QEMU

aoeu

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

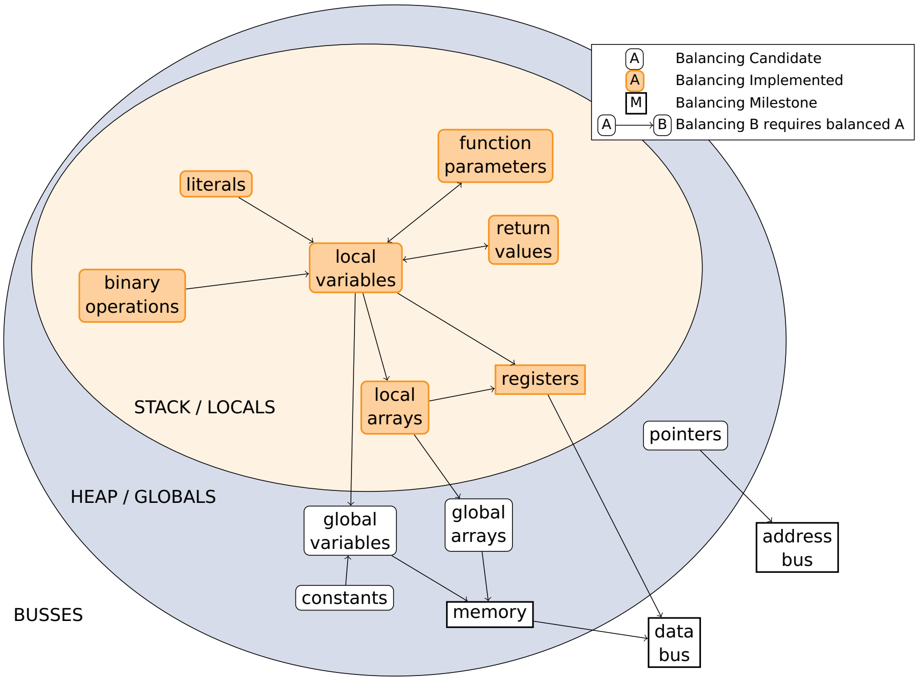


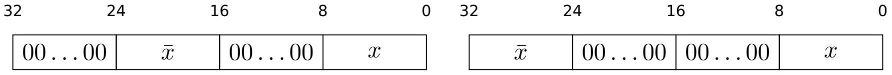
Figure 1: Balancing diagram for LLVM

3.3 Evaluation

aoeu

4 Arithmetic

The first step in finding a balanced arithmetic was finding a scheme for the balancing of the individual values. While the general shape of the scheme was pretty much clear from the start, the location of x and $\bar{x}$ emerged during my work on the balanced operation. Figure 2 shows the two schemes that are used in my project.



(a) Balancing Scheme 1

(b) Balancing Scheme 2

Figure 2: Balancing Schemes

In my theoretical work I found balanced operations for both schemes, but in the end decided to use Scheme 1 because it exhibits nicer behaviour for shifts, especially rotations. Both are worth mentioning however, because many of my operations will result in values formatted in Scheme 2 and require explicit transformation. By having explicit "names" for both schemes and finding standardized transformations in both directions I could simplify the process of finding the balanced arithmetic.

4.1 Finding Balanced Operations

After fixing the balancing scheme I started working on finding balanced variants for the binary operations in LLVM IR. Unfortunately most operations do not preserve balancedness over all intermediate steps. They do however decrease the signal-to-noise ration for an attacker. A more detailed analysis can be found in Section 4.3.

Scheme 1 to Scheme 2

For better reusability I wrote down the transformations between the schemes once, and then referenced this transformation. I thought this a better solution than implicitly including these transformations in multiple operations.

The transformation from Scheme 1 to Scheme 2 looks as follows:

$$\begin{array}{lllll}
\%1 = 0 & \| \overline{x} & \| 0 & \| x & \\
\%2 = \overline{x} & \| \overline{x} & \| x & \| x & | \%1 << 8 \\
\%3 = \overline{x} & \| 0 & \| 0 & \| x & | \%2 \texttt{ AND 0xff0000ff}
\end{array}$$

Scheme 2 to Scheme 1

The other direction works very similar to the first, it is shown in **??**. Note that ROR is the ARM assembly instruction for rotational right shift, i.e. the values shifted out on the right are shifted back in on the left. The transformation looks as follows:

$$\begin{array}{lllll}
\%1 = \overline{x} & \| 0 & \| 0 & \| x & \\
\%2 = \texttt{0xff} & \| \overline{x} & \| 0 & \| x & | \%1 \texttt{ ORR } (\%1 \texttt{ ROR } 24) \\
\%3 = 0 & \| \overline{x} & \| 0 & \| x & | \%2 \texttt{ AND 0x00ff00ff}
\end{array}$$

ORR

Before finding a balanced variant of binary or, I needed to find an expression for the inverse of the result. For this I utilized DeMorgan's law $\overline{x \vee y} = \overline{x} \wedge \overline{y}$. With this equality ORR looks as follows:

$$\begin{array}{lllll}
\%1 = 0 & \| \; \overline{x} & \| \; 0 & \| \; x & \\
\%2 = 0 & \| \; \overline{y} & \| \; 0 & \| \; y & \\
\%3 = 0 & \| \; \overline{x} \texttt{ ORR } \overline{y} & \| \; 0 & \| \; x \texttt{ ORR } y & | \; \%1 \texttt{ ORR } \%2 \\
\%4 = 0 & \| \; \overline{x} \texttt{ AND } \overline{y} & \| \; 0 & \| \; x \texttt{ AND } y & | \; \%1 \texttt{ AND } \%2 \\
\%5 = \overline{x} \texttt{ AND } \overline{y} & \| \; \overline{x} \texttt{ ORR } \overline{y} & \| \; x \texttt{ AND } y & \| \; x \texttt{ ORR } y & | \; \%3 \texttt{ ORR } (\%4 << 8) \\
\%6 = \overline{x \texttt{ ORR } y} & \| \; 0 & \| \; 0 & \| \; x \texttt{ ORR } y & | \; \%5 \texttt{ AND } \texttt{0xff0000ff} \\
\%7 = 0 & \| \; \overline{x \texttt{ ORR } y} & \| \; 0 & \| \; x \texttt{ ORR } y & | \; \texttt{transform\_2\_1}(\%6)
\end{array}$$

AND

As $\overline{x \wedge y} = \overline{x} \vee \overline{y}$ AND works almost the same as ORR, but uses different parts of the intermediate results.

XOR

XOR is at its base a combination of AND and ORR: $x \oplus y = (\overline{x} \wedge y) \vee (x \wedge \overline{y})$. As both balanced ORR and balanced AND have the same imbalanced intermediate values it is better to balance XOR from scratch instead of compositioning it. The inverse of the result can be found through repeated application of DeMorgan's law and simplification. I will skip the details of this simple transformation. The result is: $\overline{x \oplus y} = (x \wedge y) \vee (\overline{x} \wedge \overline{y})$.

My version of balanced XOR already includes some ARM specific optimizations. In ARM shift operations happen in a so-called barrel shifter, and can be applied to the right-hand argument of any other instruction. I utilize this property in my balanced version of XOR to save some unnecessary cycles.

$$\begin{array}{lllll}
\%1 = 0 & \| \; \overline{x} & \| \; 0 & \| \; x & \\
\%2 = 0 & \| \; \overline{y} & \| \; 0 & \| \; y & \\
\%3 = \overline{x} & \| \; \overline{x} & \| \; x & \| \; x & | \; \%1 \texttt{ ORR } (\%1 << 8) \\
\%4 = y & \| \; \overline{y} & \| \; \overline{y} & \| \; y & | \; \%2 \texttt{ ORR } (\%2 \texttt{ ROR } 24) \\
\%5 = \overline{x} \texttt{ AND } y & \| \; \overline{x} \texttt{ AND } \overline{y} & \| \; x \texttt{ AND } \overline{y} & \| \; x \texttt{ AND } y & | \; \%3 \texttt{ AND } \%4 \\
\%6 = x \texttt{ XOR } y & \| \; \overline{x \texttt{ XOR } y} & \| \; x \texttt{ XOR } y & \| \; \overline{x \texttt{ XOR } y} & | \; \%5 \texttt{ AND } (\%5 \texttt{ ROR } 16) \\
\%7 = \overline{x \texttt{ XOR } y} & \| \; x \texttt{ XOR } y & \| \; \overline{x \texttt{ XOR } y} & \| \; x \texttt{ XOR } y & | \; \%6 \texttt{ ROR } 8 \\
\%8 = \overline{x \texttt{ XOR } y} & \| \; 0 & \| \; 0 & \| \; x \texttt{ XOR } y & | \; \%7 \texttt{ AND } \texttt{0xff0000ff} \\
\%9 = 0 & \| \; \overline{x \texttt{ XOR } y} & \| \; 0 & \| \; x \texttt{ XOR } y & | \; \texttt{transform\_2\_1}(\%8)
\end{array}$$

ADD

For the inverse of arithmetic operations I utilized the definition of the negation in 2s complement: $-x = \overline{x}+1$. This also means that $\overline{x} = -x-1$ and therefore:

$$\overline{x+y} = -(x+y) - 1 = -x - y - 1 = \overline{x} + 1 + \overline{y} \cancel{+1} \cancel{-1} = \overline{x} + \overline{y} + 1$$

Using associativity of addition the balanced variant of ADD looks like the following:

$$\begin{array}{lllll}
\%1 = 0 & \parallel \overline{x} & \parallel 0 & \parallel x & \\
\%2 = 0 & \parallel \overline{y} & \parallel 0 & \parallel y & \\
\%3 = 0 & \parallel \overline{x} + 1 & \parallel 0 & \parallel x & \mid \%1 + \texttt{0x00010000} \\
\%4 = c & \parallel \overline{x + y} & \parallel c' & \parallel x + y & \mid \%3 + \%2 \\
\%5 = 0 & \parallel \overline{x + y} & \parallel 0 & \parallel x + y & \mid \%4 \wedge \texttt{0x00ff00ff}
\end{array}$$

Both c and c' denote possible carry bits that need to be filtered.

SUB

For subtraction I again use the definition of 2s complement, giving me the following for the inverse result:

$$\overline{x - y} = -(x - y) - 1 = y - x - 1 = y + (-x - 1) = y + \overline{x} = \overline{x} + y$$

Applying the same definition to the regular result yields

$$x - y = x + \overline{y} + 1$$

resulting in a quick and convenient balanced subtraction:

$$\begin{array}{lllll}
\%1 = 0 & \parallel \overline{x} & \parallel 0 & \parallel x & \\
\%2 = 0 & \parallel \overline{y} & \parallel 0 & \parallel y & \\
\%3 = 0 & \parallel y & \parallel 0 & \parallel \overline{y} & \mid \%2\ \texttt{ROR}\ 16 \\
\%4 = 0 & \parallel y & \parallel c & \parallel \overline{y} + 1 & \mid \%3 + \texttt{0x00000001} \\
\%5 = c' & \parallel \overline{x} + y & \parallel c'' & \parallel x + \overline{y} + 1 & \mid \%1 + \%4 \\
\%6 = 0 & \parallel \overline{x - y} & \parallel 0 & \parallel x - y & \mid \%5\ \texttt{AND}\ \texttt{0x00ff00ff}
\end{array}$$

MUL

The inverse result of multiplication can be calculated as follows:

$$\overline{x \cdot y} = -(x \cdot y) - 1 = (-x) \cdot y - 1 = (\overline{x} + 1) \cdot y = \overline{x} \cdot y + y - 1$$

Which gives us the following balanced multiplication:

$$\begin{array}{lllll}
\%1 = 0 & \parallel \overline{x} & \parallel 0 & \parallel x & \\
\%2 = 0 & \parallel \overline{y} & \parallel 0 & \parallel y & \\
\%3 = \overline{y} & \parallel 0 & \parallel 0 & \parallel y & \mid \texttt{transform\_2\_1}(\%2) \\
\%4 = c & \parallel \overline{x} \cdot y & \parallel c' & \parallel x \cdot y & \mid \%1 \cdot \%3 \\
\%5 = c'' & \parallel \overline{x \cdot y} + 1 & \parallel c' & \parallel x \cdot y & \mid \%4 + (\%2 << 16) \\
\%6 = c''' & \parallel \overline{x \cdot y} & \parallel c' & \parallel x \cdot y & \mid \%5 + \texttt{0x00ff0000} \\
\%7 = 0 & \parallel \overline{x \cdot y} & \parallel 0 & \parallel x \cdot y & \mid \%6\ \texttt{AND}\ \texttt{0x00ff00ff}
\end{array}$$

Practical evaluation shows that computing multiplication via repeated balanced addition shows better balancing properties (see Section 4.3) than the direct variant, so I used that for my thesis.

DIV and REM

Just like multiplication, I used repeated balanced subtraction for DIV (division) and REM (remainder) operations. The code was written in C and can be found in the git of my thesis[1].

Shifting

When performing logical shifts, I need to ensure that the correct bits are pushed in. As 0s are shifted in for x I have to shift in 1s for $\overline{x}$. This means that I have to ORR `0xff000000` for right shifts and `0x0000ff00` for left shifts. The shifting is performed normally and the result is then AND filtered with `0x00ff00ff` to comply with Scheme 1 again.

4.2 Testing for Correctness

Before I started implementing my balancing pass I wanted to verify the correctness of my arithmetic. For this purpose I wrote python code to calcuate all operations step by step while saving the intermediate results. Listing 1 shows the intermediate steps for addition.

Listing 1: Step-by-step execution of balanced multiplication

```python
m = MultiStepOperation([
    Convert_1_2(1), #2
    BinaryOperation(0,2, lambda x,y: (x*y) & 0xffffffff),#3
    BinaryOperation(3,1, lambda x,y: x + (y << 16)), #4
    UnaryOperation(4, lambda x: x + 0x00ff0000), #5
    UnaryOperation(5, lambda x: x & 0x00ff00ff), #6
])
```

The *Unary-* and *BinaryOperation* classes take the indices of the layers to operate on (0 and 1 are the inputs, all others are intermediate values), as well as the operation in form of a lambda. Executing the *MultiStepOperation* will then execute all lambdas in order and store the intermediate results in *numpy* arrays. After the execution there are $2^8 \cdot 2^8 = 2^{16}$ intermediate results for each operation (the inputs only have 2^8 values each). Correctness is then tested by checking if all final results are equal to the output of a function to compare to ($x \cdot y$ in this case).

4.3 Evaluating the Balancedness

Balancedness of my operations is evaluated using the same python code. As all intermediate results are stored during evaluation I can easily calculate the distribution of their Hamming Weights, as shown in Figure 3. I used these histograms to check if operations needed improvement, and if that was the case, I tried to find a different, more balanced way of performing them.

Figure 3 also shows that while directly computing multiplication is balanced for a lot of values, it still leaks information in almost every step. For this reason I implemented it as repeated addition in my pass.

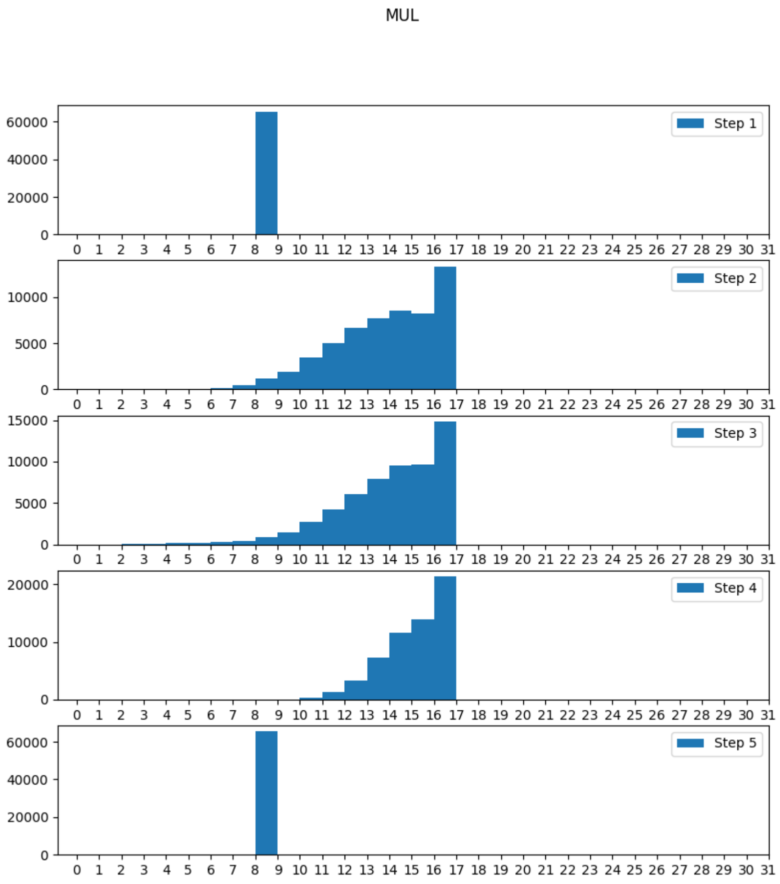


Figure 3: Histogram of Hamming Weights of direct balanced multiplication

5 Balancing Pass

The idea behind the balancing pass is very simple.

1. Change the type of all 8bit integers (*int8*) to 32bit integers (*int32*)
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 I decided to focus on stack memory only, the operations and lifecycle are specific to local variables.

5.1 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.

The content of the original function is then copied using a helper in the LLVM API called *CloneFunctionInto*. Without any additional parameters, the copied instructions will still reference function parameters of the original function, which are invalid in the new function. To avoid this I use a so-called *Value Mapper* to replace the old parameters with the new ones everywhere they are referenced. This change alone would cause type mismatches and generates code that does not compile, but the other steps of my pass fix these problems.

5.2 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.

5.3 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.

5.4 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.

5.5 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.

5.6 Balancing Binary Operations

I implemented the balanced operations described in Section 4.1 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.

5.7 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.

5.8 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.

5.9 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.

Building the Compiler Pass

The compiler pass is built using CMake as that makes loading the required parts of LLVM very easy. Listing 2 shows the *CMakeLists.txt* for my balancing pass. The code is based on the template repository provided in [4].

Listing 2: 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.

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 Listing 3.

Listing 3: 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[3] 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. 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 [2]), which causes the code defined in *startup.s* to be at memory address *0x1000*. The content of *startup.ld* is shown in Listing 4.

Listing 4: 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 Listing 5.

Listing 5: Startup code in *startup.s*

```
.global _Reset
```

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

6 Instrumenting QEMU for Evaluation

QEMU does not simply interpret the guest code in a simulated processor. Instead it translates the machine code for the guest platform into machine code for the host platform, and places that "patched" machine code in memory. A second executor thread then runs that code as it becomes available.

This translation backend is called the Tiny Code Generator (TCG), which not only performs the translation but also some optimizations. Instrumenting QEMU for analysis is hard due to the fact that the TCG works through multiple layers of indirection, utilizing both helper functions and preprocessor macros, some of which are defined in different files depending on the host architecture (the specific definition file is chosen during compilation). As documentation is also sparse, finding a good place to put my evaluation code required a lot of time and effort.

Even after understanding all the parts of QEMU's way of emulating code, I was left with a problem. The executor thread does not know what code it is executing, it only has a pointer (the simulated program counter) to the next instruction or the next basic block. The TCG on the other hand knows which operations are being executed, but it does not know the values of the operands. It also has no way of accessing these values as they might not even be computed yet. So short of either parsing the memory at the simulated program counter or writing a symbolic execution engine (essentially replacing QEMU) I did not know how to proceed.

Luckily, QEMU offers emulation via the TCG Interpreter (TCI). The TCI does exactly what I was looking for in the first place, ie. emulating the guest processor in C. I then placed my instrumentation code in the operator functions of the TCI, generating a histogram of Hamming Weights during the execution.

7 Nonfeatures and Limitations

aoeu

8 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.

8.1 Security

8.2 Performance

aoeu

9 Conclusion

aoeu

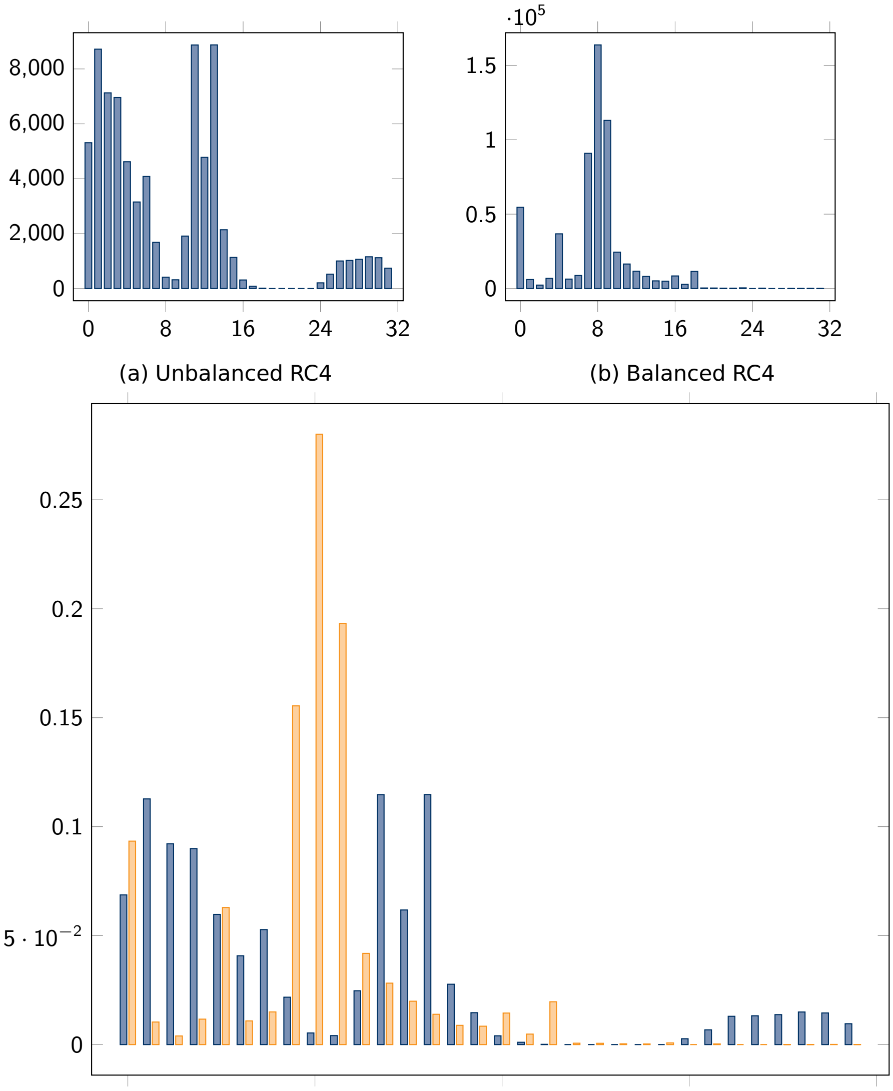
References

[1] URL: https://github.com/alxshine/dual-rail.

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

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

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

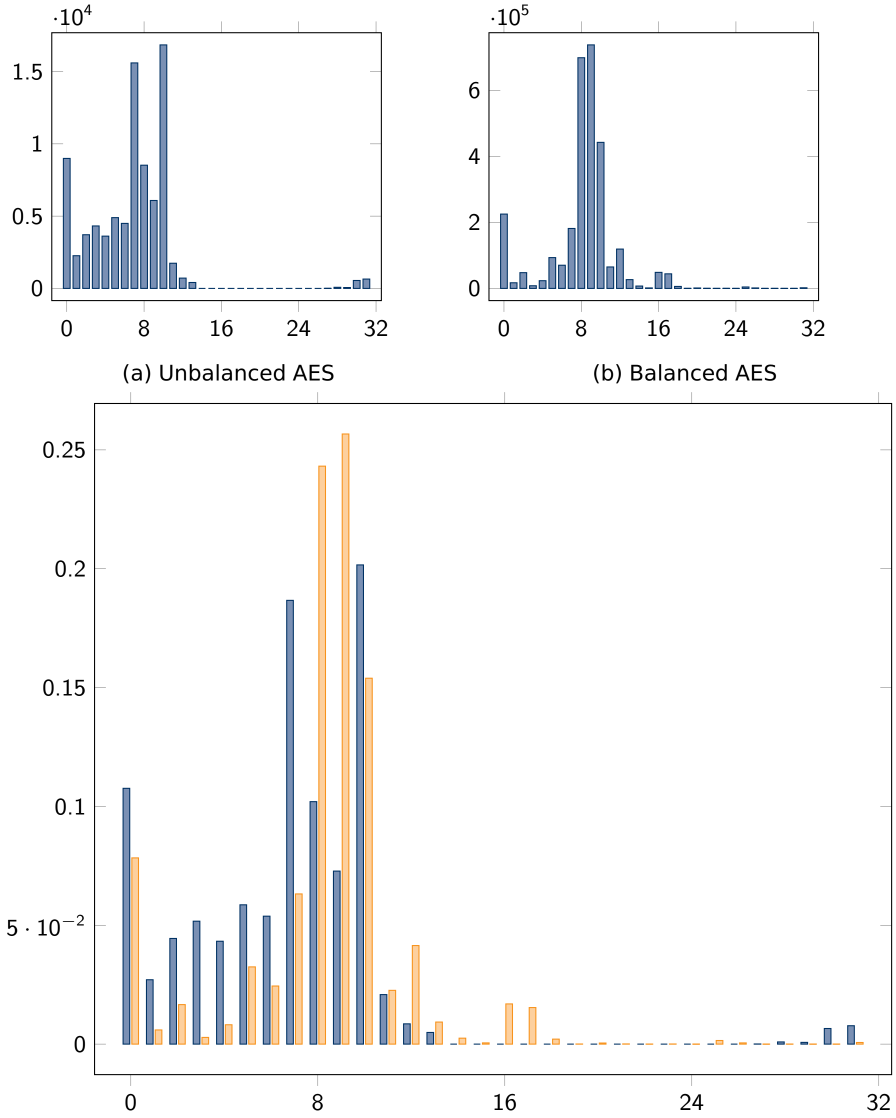


(a) Unbalanced RC4

(b) Balanced RC4

(c) Scaled Hamming weight histograms for RC4

Figure 4: Hamming weight histograms for balanced and unbalanced RC4



(a) Unbalanced AES

(b) Balanced AES

(c) Scaled Hamming weight histograms for AES

Figure 5: Hamming weight histograms for balanced and unbalanced AES