Toward Balancing Arbitrary Code

June 21, 2019

Contents

1 Introduction

Shipping microcontrollers to consumers exposes them to a plethora of attacks on the actual hardware. One class of such attacks focuses on analyzing unintended information leakage from the microcontroller, the so-called Side-Channel attacks. If the processed data influences some observable metric, an attacker can measure that metric and then reason about the values of said data. The microcontroller can thus leak information, which is especially devastating if the information in question is a cryptographic secret.

One such metric that can leak information is the power consumption of the microcontroller. Setting a binary value in registers, main memory etc. consumes power directly related to the number of bits to be set to 1. By measuring the power consumption traces during execution an attacker can gain information about the Hamming Weight (number of 1s) of the processed data. If she also knows which cryptographic operation is being performed (a reasonable assumption under Kerckhoff's principle), and can control the input, she can infer the value of the cryptographic secret via statistical anlysis of the power traces. A comparatively low clock rate and power traces that are low in noise due to the lack of parallelism make embedded platforms especially susceptible to such an attack.

As performing cryptographic operations is *exactly* the use case of many embedded devices (SmartCards etc.) defensese against Power-Analysis have been amply explored. However, the most commonly used defenses are either algorithm specific, like masking, or require significant changes to the hardware, like Dual-Rail-Logic. Dual-Rail-Logic is the only defensive measure that is algorithm independent. It computes the inverse of every intermediate value, along with the values itself, to try and keep the power consumption constant and thus independent of processed data.

In my thesis I explore the possibilities of implementing Dual-Rail-Logic in software. I simulate the computation of the inverse by keeping the inverse of an 8bit value, along with the actual 8bit value, in the same 32bit register. By doing so I drastically reduce the variance of Hamming Weights during execution, and thus make the program more robust to power analysis. While this requires making significant changes to the code, all of these changes are made automatically in the compiler, and the hardening is thus applicable to *any* code that is written for an 8bit architecture.

By providing a proof of concept and an evaluation thereof I hope to open a new perspective on defending against side channel attacks, providing a solution for cases where execution time is not of the essence, but circuit space is limited. For these cases my thesis can provide a way of hardening the execution without any security knowledge of the programmer at all.

2 Background

Finishing my thesis project required knowledge from many different areas and projects. The contents of the Information Security I and II, as well as from the Compiler Construction lectures. This section will give a brief introduction to the relevant topics of those lectures.

2.1 Power Analysis Attacks

In most cases the power consumption during execution is data-dependent. Setting a bit to 1 requires more power than setting it to 0. While this difference is very hard to observe in real time, it is easy to detect using statistical analysis of the power consumptions of multiple executions. For easier analysis, the power consumption is quantized and stored in numerical format as a so-called power trace.

Embedded devices are often exposed to this kind of attack, as an attacker has physical access to the device. Their power consumption is also fairly low in noise, as they lack any kind of parallelism. Additionally, many use-cases for embedded devices include some cryptographic operation on sent input without much validation, giving the attacker an easy and valuable target for her attack.

An example attack would go like this: An attacker solders a resistor between the target processor and the ground of its power supply. She then measures the voltage difference between both ends with an oscilloscope (this voltage is directly proportional to the current flowing through the resistor). This gives her easy access to the power traces at a high resolution and for every clock cycle.

After the setup, she submits a large number of different plaintexts to the target processor (~1000 is a good starting point), collecting the power traces. She then starts attacking the secret key byte by byte (attacking the key byte by byte drastically reduces the search space, keeping this attack feasible). For this she calculates the expected power consumption of an operation she *knows* happens during the encryption, for each combination of plaintext and possible value of key byte. An example operation would be the first substitution box lookup in AES.

Now that she has the actual power traces and the expected power consumption per guessed key byte, she can calculate the correlation between the two. This will give her the most probable value for the current key byte, as well as a confidence measure for it.

2.2 Power Analysis Defenses

There is no absolute defense against power analysis attacks. All defensive measures can do is increase the amount of effort (required number of traces, computation time for analysis, etc.) required for an attacker to perform a successful attack.

Masking for example is an algorithm specific defensive measure that adds a third factor to the power consumption. The attacker then has to calculate

her correlation for each possible key byte value and mask value. This increases the number of traces she needs to capture (to still provide the same confidence in her analysis) and the computation time of her analysis.

Other defensive measures focus on creating a worse signal to noise ratio for the entire power consumption. One technique that has gained a lot of traction is Dual-Rail-Logic[14]. It works by calculating the inverse of every intermediate result along with the actual result. This, in theory, keeps the power consumption constant and thus independent of the data.

Unfortunately, Dual-Rail-Logic suffers from multiple engineering problems. The power required to set the value of a bit to 1 is dependent on the properties of the underlying transistor, which is subject to variances in manufacturing.[12] Minimal differences in clock timings between both paths can also reduce the security of Dual-Rail-Logic[5]. Dual-Rail-Logic also requires significantly larger cuitry, doubling the required size or more[5].

Even with these caveats, Dual-Rail-Logic has the major advantage that once it is applied to the circuitry, *any* code can be run on it without modifications, while still benefitting from the incresed robustness.

2.3 LLVM

The LLVM compiler infrastructure project[10] contains a number of subprojects. For my thesis the LLVM Core libraries are the only part that is relevant. They contain a source and target independent compiler, which can be extended using multiple front- and backends. This makes LLVM the most versatile compiler available.

At the heart of LLVM Core is a number of optimization passes. These passes take LLVM IR as input and provide LLVM IR as output. This allows easy addition and reordering of compiler passes, making it perfect for my thesis.

LLVM also has Clang as a frontend, making it an industry-grade C and C++ compiler, which keeps my project from being unusable due to some obscure toolchain.

2.4 Static Single Assignment Form

In order to understand LLVM IR we first need to understand the basics of static single assignment form (SSA). The basic premise of SSA is simple: every value assignment is stored in a new variable. Analysis of variable usage, register requirements (liveness), dead code, etc. is thus greatly simplified.

Some notations for SSA annotate the variable names with indices to make them unique. LLVM IR completely forgoes the names of variables, instead using just numbers, preceded by a %.

2.5 LLVM Intermediate Representation

LLVM IR is best described as typed assembly written in SSA. The instructions provided by LLVM IR are very similar to RISC assembly. LLVM IR uses SSA variables, which have types associated with them, based on their assignment. This allows typechecking during every step of the compiler, especially between optimization passes.

LLVM IR also explicitly defines functions with a prototype, complete with typed arguments and return type.

2.6 LLVM C++ API

LLVM provides a C++ API for extending the compiler. This API exposes all libraries that LLVM itself uses, giving the programmer full access to all capabilities. For my thesis I mainly used the code inspection and generation utilities, going through the generated LLVM IR code and balancing it, in an optimization pass.

The pass can then be compiled into a library (see Section 6.1), which is loaded as an LLVM plugin during the compilation process.

2.7 QEMU

QEMU is a generic and open source machine emulator and virtualizer.[6] While it can be used as a full fledged virtualization environment and sandbox, it can also emulate different processor architectures for programs without first emulating an OS. This process is called bare-metal emulation, and is used for my thesis.

QEMU is also open source, allowing for "easy" modification and addition of my evaluation code. Easy is a relative term here, as its size, the complexity of its build process, and its relative lack of documentation make this still a hard problem to tackle.

Memory Layout of QEMU Kernels

Even with bare-metal emulation, QEMU still takes its input as a kernel (same term as in Linux kernel). Due to this, it starts execution at addres `0x1000`, as everything before that address is usually reserved for interrupt handling. This requires some additional setup in my build process.

2.8 AES and RC4

AES[7] and RC4[1] are the two evaluation programs for my compiler pass. I chose RC4 because it is very simple and used to be the industry standard, and AES because it is the current industry standard for symmetric encryption. Both fit the main usecases of embedded devices, and are thus reasonable choices for evaluating the robustness of my thesis project.

3 Methodology

Before the implementation I wanted to specify what would and would not be part of my thesis. This allowed me to have a clear set of goals while also limiting the scope of my thesis to a feasible range.

3.1 Arithmetic

As the goal of my thesis was to work towards balancing arbitrary code, the first step to achieving this was finding an arithmetic capable of supporting this. It should include a scheme for balancing individual values, as well as a balanced way of performing operations on balanced values.

I needed to find balanced variants for all operations existing in LLVM IR:

- `add`, addition
- `sub`, subtraction
- `mul`, multiplication
- `div`, division
- `rem`, division remainder (Modulo)
- `shl`, shift left
- `ashr`, arithmetic shift right
- `lshr`, logical shift right
- `and`, bitwise AND
- `or`, bitwise OR
- `xor`, bitwise XOR

All operators should work on signed and unsigned 8bit numbers, and be semantically consistent with existing LLVM operators.

3.2 Balancing Pass

For the pass itself I first tried to identify all different types of values in LLVM IR. Then I split them into groups, depending on whether they are local or global values in the program. With this separation and based on the interaction between variable types I built the dependency graph shown in Figure 1.

The memory locations in Figure 1 are sources of information leakage that depend on code or data. A memory location is balanced if all data stored in that location is balanced. For my thesis I wanted to balance all local variables, which gives me balanced registers and a balanced stack. While loading from non-stack memory does cause imbalanced values to be stored in registers temporarily, this can be avoided by not using global variables.

3.3 Evaluation

At first my plan was to evaluate the performance of my pass using a full Power Analysis attack on an Arduino Due[4]. It is based on an ARM Cortex-M3 CPU with 32bit registers and a clock rate of 84 MHz. While I have performed power analysis attacks on a microcontroller before, my experience in tweaking the parameters for collecting the power traces is very limited. As such, with the amount of trial and error required and the time required for each trial the

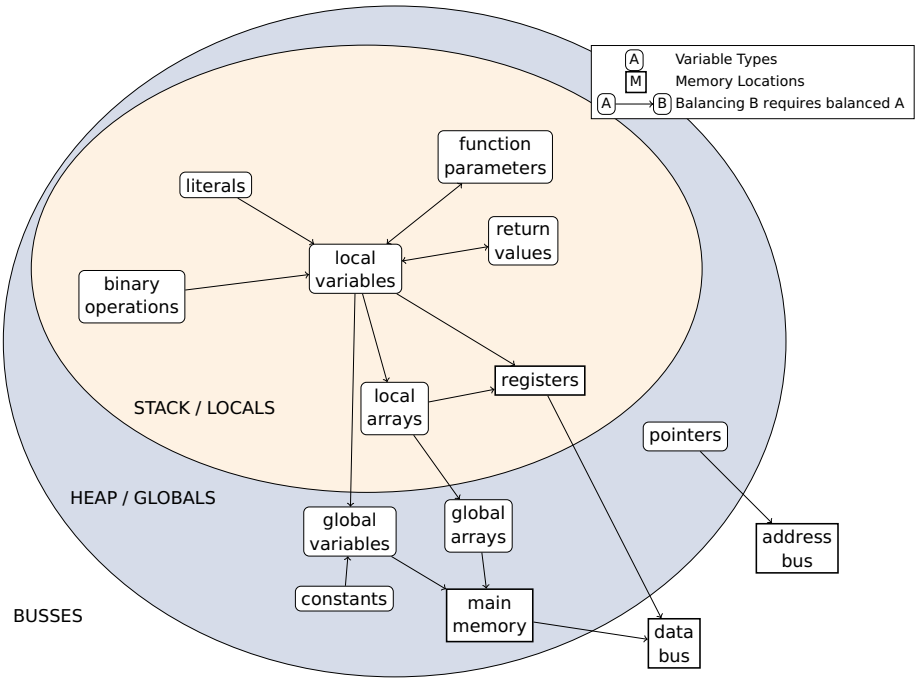


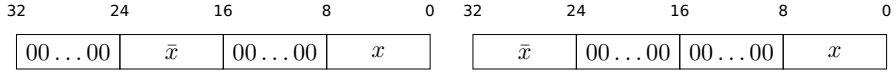
Figure 1: Balancing dependency graph in LLVM

prospects of getting meaningful results in a reasonable amount of time were very small.

My supervisor then had the idea to emulate the processor, completely removing the need for any external hardware. This drastically reduced the turnaround time between tries, and made the whole project feasible. As emulator we decided on QEMU, as it is very powerful and open-source, giving me “easy” access to its internals and allowing me to generate metrics. The primary evaluation metric we decided on was the distribution of Hamming Weights over the entire execution.

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.

The largest caveat of finding a balanced arithmetic was that $\overline{x \circ y}$ is not $\overline{x} \circ \overline{y}$ ($\circ$ here denotes any operator). As the ALU cannot execute two different operations on parts of the same register at the same time, this means that there *must* be imbalanced temporary values during execution. My goal then was to limit the number of these imbalanced values.

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:

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

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:

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

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:

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

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.

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

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:

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

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:

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

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:

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

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[2].

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

```
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, which performed a lot better in practice.

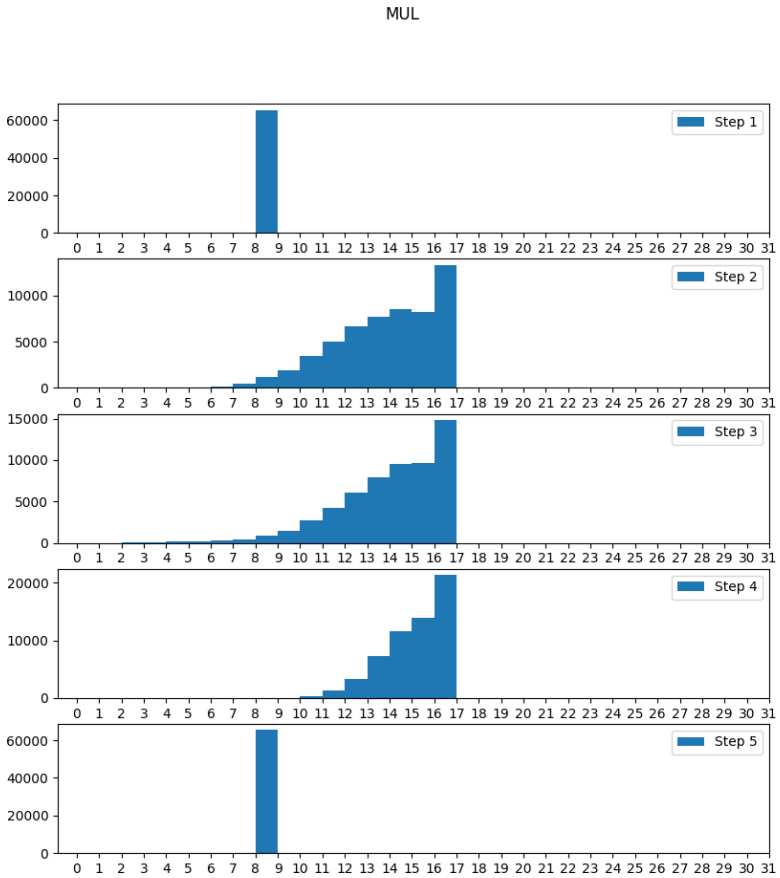


Figure 3: Histogram of Hamming Weights of direct balanced multiplication

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 *int32*. The same is done for the return type. This gives me a skeleton for the balanced function, which is then 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 *int32*. 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 (*int32*) 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 call to generate.

For most operations the balanced operation is a direct implementation of the respective steps in Section 4. Multiplication, division, and remainder however are implemented by repeated addition/subtraction. As an example, Listing 2 shows the balanced function for the `sdiv` operation in LLVM IR.

Listing 2: Balanced sdiv

```
1  int balanced_sdiv(int lhs, int rhs) {
2    uint32_t ret = 0x00ff0000;
3
4    uint8_t negative = 0;
5    if(rhs & 0x00000080){
6      negative = 1;
7      rhs = balanced_negative(rhs);
8    }
9
10
11   while (lhs <= rhs) { //~x <= ~y iff x >= y
12     lhs = balanced_sub(lhs, rhs);
13     ret = balanced_add(ret, 0x00fe0001);
14   }
15
16   if(negative)
17     return balanced_negative(ret);
18   else
19     return ret;
20 }
```

The semantics, especially the handling of negative values are made to be consistent with the semantics of LLVM.

5.7 Balancing Pointer Arithmetic

Balanced values cannot be used for array indexing directly. Therefore, whenever a balanced variable is used as index for an array access it is unbalanced before use. All array accesses use the *getelementptr* instruction in LLVM IR,

so this is easy to catch. This does not handle manual arithmetic operations 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.

6 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.

6.1 Building the Compiler Pass

The compiler pass is built using CMake as that makes loading the required parts of LLVM very easy. Listing 3 shows the *CMakeLists.txt* for my balancing pass. The code is based on the template repository provided in [13].

Listing 3: 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.

6.2 Building the Test Code

As discussed in Section 2.3 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 4.

Listing 4: 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[9] 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 2.3 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 2.7. 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 [3]), which causes the code defined in *startup.s* to be at memory address *0x1000*. The content of *startup.ld* is shown in Listing 5.

Listing 5: 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 Listing 6.

Listing 6: Startup code in *startup.s*

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

7 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.

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 benefits of my balancing pass. For the evaluation of both the performance and the robustness I use histograms of the Hamming Weights over the entire execution of the code. Figures 4 and 5 show a comparison of balanced and unbalanced histograms for RC4 and AES respectively.

The balanced version of both algorithms have been compiled with my balancing pass, while the unbalanced versions were compiled with GNU ARM Cross GCC.

8.1 Robustness

For both algorithm the balancig works very well. The Hamming Weights are concentrated around 8, with other values being much less frequent. Note that for an attacker performing Power Analysis, all values with the same Hamming Weight look the same. Thus, the less evenly distributed the Hamming Weights of intermediate values are, the lower the confidence of her statistical attack. As such, a perfect scenario for the defender would be all Hamming Weights having exactly the same value.

A significant number of operations also exhibit a Hamming Weight of 9 and 10, which is probably due to carry bits in arithmetic operations. This theory is supported by the fact that these Hamming Weights are more prevalent in AES, which utilizes a lot more loops and therefore additions.

The balancing is not perfect, as some intermediate steps of my balanced operators will *always* have unbalanced values. Value unbalancing for array indexing is also a factor for the distribution of Hamming Weights in the balanced code.

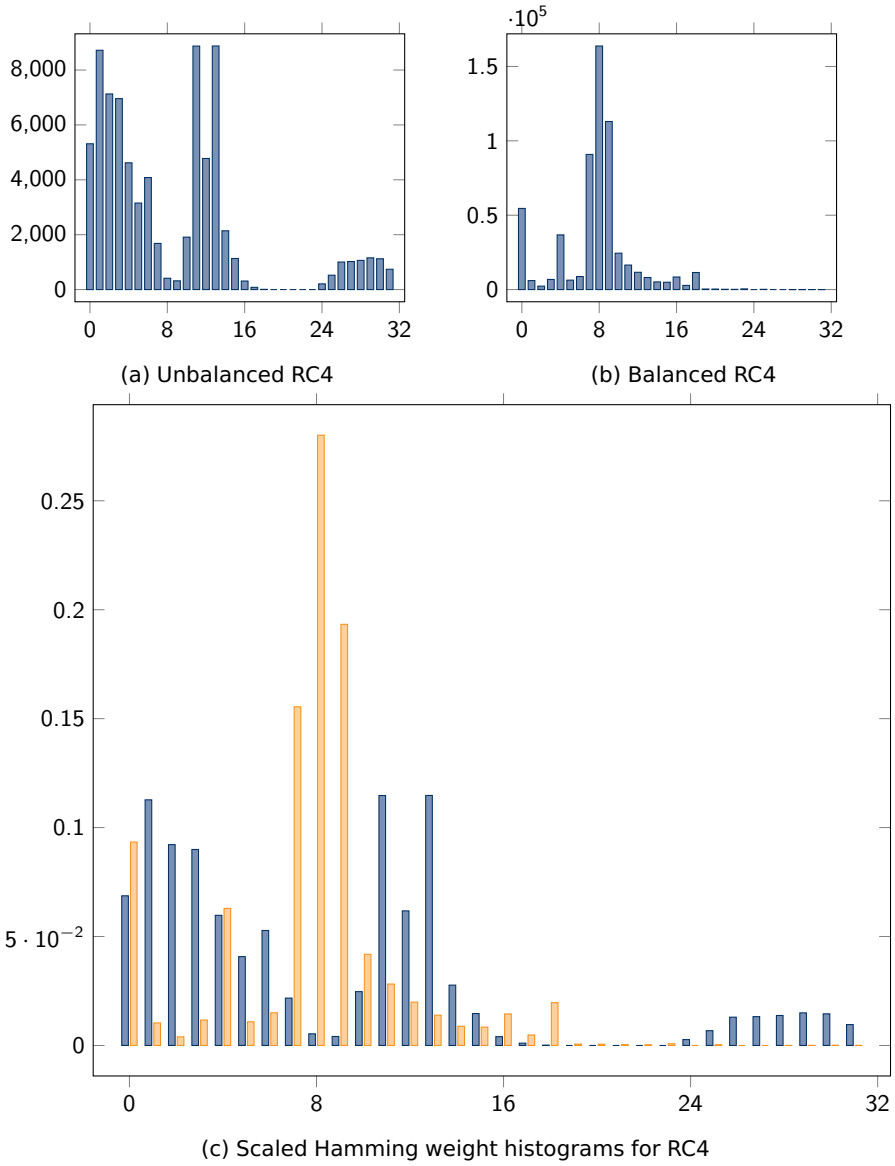
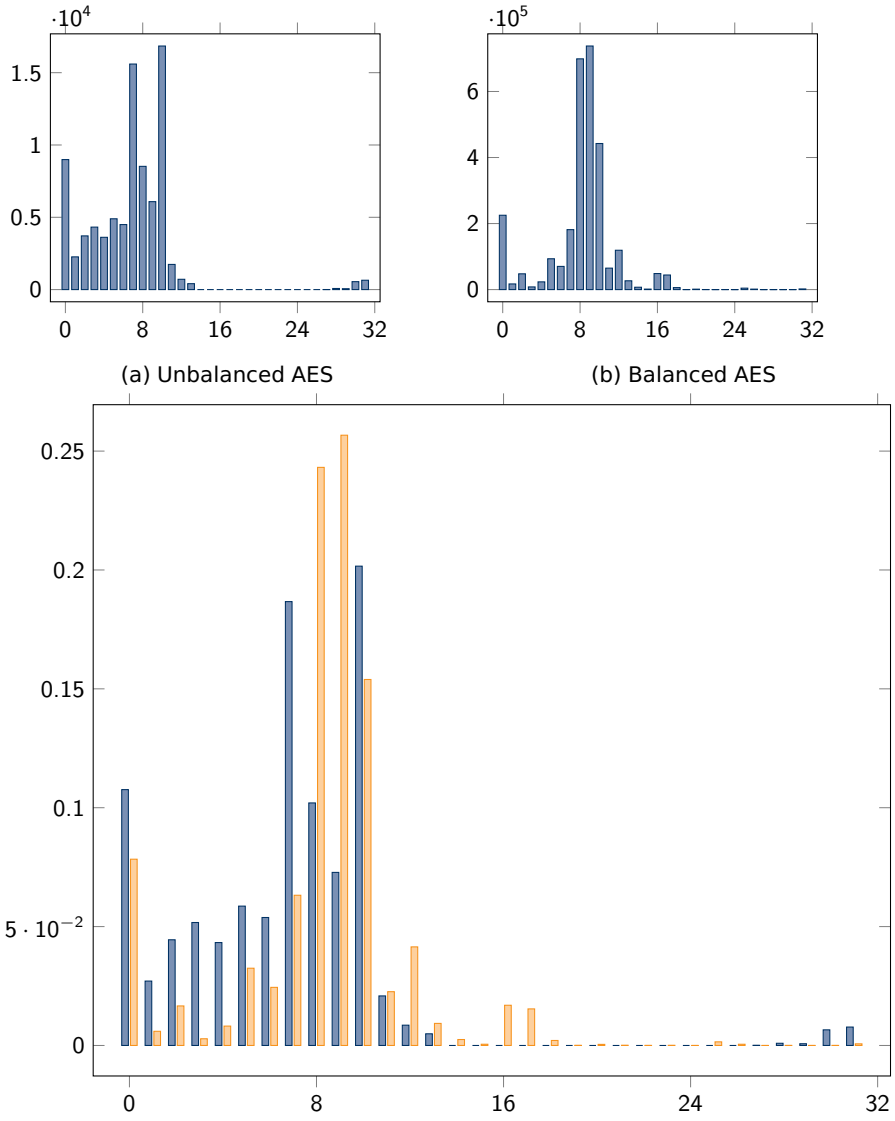


Figure 4: Hamming weight histograms for balanced and unbalanced RC4



(a) Unbalanced AES

(b) Balanced AES

(c) Scaled Hamming weight histograms for AES

Figure 5: Hamming weight histograms for balanced and unbalanced AES

8.2 Performance

The number of operations is 77349 for unbalanced RC4, and 584598 for balanced RC4. That is an increase in the number of operations by a factor of 7.56.

For AES the unbalanced code has 83549 operations, while the balanced code has 2873960 operations. This is an increase by a factor of 34.4. The performance impact for AES can be reduced to a factor of 28.51 when directly computing multiplication, which drops the number of operations to 2382048.

For both algorithms the largest part of the performance impact is probably due to MUL, DIV and REM operations being calculated via repeated addition/-subtraction.

In general it is also important to note that when the full 32bit range is required for the program the performance drops by an additional factor of 4, because then every operation needs to be performed on the individual bytes of a 32bit word. This is less true for cryptographic algorithms, as they mostly work on individual bytes.

9 Conclusion

In my thesis I explored the robustness of a pure software implementation of Dual-Rail-Logic. By writing a proof of concept implementation I explore a new perspective on hardening embedded platforms against power analysis attacks. Preliminary evaluation shows a drastic reduction in the signal to noise ratio of the power consumption, due to a decreased variance in Hamming Weights of intermediate values.

The security of balanced code is not perfect, as it is limited by the inability to perform different operations on parts of the same register at the same time. However, with this limitation in mind, I believe my balancing pass achieves quite good performance. Histograms of the Hamming Weights show a significant shift towards values around 8, as would be ideal. This causes a relative decrease of other values, reducing the information an attacker can gain from power analysis attacks.

A major disadvantage of the approach in my thesis is the increased number of operations taking place. The number of clock cycles increases by a factor of 8, and when taking into account the reduction in word size this leads to a factor of 32. Future work could reduce this performance impact by removing unnecessary transformations between balancing schemes, but the design of embedded platforms and RISC architectures in general sets a lower bound for the performance impact of my approach. While something like Intel's SIMD extensions[11] could drastically reduce the performance impact of software Dual-Rail-Logic, this is not possible for the intended target platforms.

As currently only stack values are balanced, balancing all types of variables is another avenue for future work. This would balance main memory, and thus the data bus at all times. The logical next step after this would be to balance the address bus as well, completely cutting an attacker of from getting any information via the power consumption. Howeve, this last approach

would require making major changes to the way memory is indexed, possibly changing paging controllers and (if present) cache controllers.

A third possibility for future work would be attacking actual hardware running balanced code, providing some real-world evaluation. The difficulty in this evaluation lies in the fact that my approach requires 32bit registers, which are typically only found in more powerful embedded processors running higher clock speeds, which makes power analysis harder by itself.

With the way it currently is, my proof of work provides a way for programmers without explicit security knowledge to harden their code against power analysis attacks, without making too large adjustments to their code. As my compiler pass balances all code, as long as it is on the stack, even substitution boxes can be used, they simply need to be passed as function parameters.

This reduces the number of considerations for the programmer to a minimum, handing them off to the compiler. With this I hope to provide a step towards compilers generating secure code automatically, akin to parallelization libraries like OpenMP[8], thus reducing the number of easily avoidable security flaws in practice.

References

[1] URL: `https://en.wikipedia.org/wiki/RC4` (visited on 07/21/2017).

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

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

[4] *Arduino Due Store Page and Technical Specs*. `https://store.arduino.cc/arduino-due`. Accessed: 2019-01-15.

[5] Karthik Baddam and Mark Zwolinski. “Path switching: a technique to tolerate dual rail routing imbalances”. In: *Design Automation for Embedded Systems* 12.3 (2008), pp. 207–220.

[6] Fabrice Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[7] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[8] Leonardo Dagum and Ramesh Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

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

[10] Chris Lattner et al. “The LLVM compiler infrastructure”. In: *URL http://llvm.org* (2010).

[11] Chris Lomont. “Introduction to intel advanced vector extensions”. In: *Intel White Paper* (2011), pp. 1–21.

[12] Alin Razafindraibe, Michel Robert, and Philippe Maurine. “Formal evaluation of the robustness of dual-rail logic against DPA attacks”. In: *International Workshop on Power and Timing Modeling, Optimization and Simulation*. Springer. 2006, pp. 634–644.

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

[14] Danil Sokolov et al. “Design and analysis of dual-rail circuits for security applications”. In: *IEEE Transactions on Computers* 54.4 (2005), pp. 449–460.