Toward Balancing Arbitrary Code

July 2, 2019

Contents

1 Introduction

Unintended signal emissions are a major source of information leakage in modern processors. Especially cryptographic secrets are valuable targets for analyzing these so-called Side-Channels. While the physical access required for Side-Channel attacks is often a hurdle, embedded devices are usually shipped to consumers, making them vulnerable to this type of attack.

One such side-channel that is especially easy to measure is power consumption. 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 knows which cryptographic operation is being performed and can control the input (both reasonable assumptions for embedded devices), she can infer the value of the cryptographic secret via statistical anlysis of the power traces.[8] A comparatively low clock rate and power traces that are low in noise due to a lack of parallelism make embedded platforms especially susceptible to such a Power-Analysis attack.

As performing cryptographic operations is *exactly* the use case of many embedded devices (e.g. SmartCards, verifying OTA updates, etc.), defenses 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[26]. Dual-Rail-Logic is especially notable because it is algorithm independent. By computing the inverse, along with the actual value, Dual-Rail-Logic *balances* the number of 1s in intermediate values, and thus makes the power consumption constant. This makes it in theory impossible for an attacker to gain information via the power consumption.

Unfortunately this strategy suffers from multiple engineering problems, such as minute differences in clock timings between the regular and inverted path[5], or variances in the production of transistors[22]. It also requires a significant increase in circuit size, doubling the required size or more[5].

Even with these caveats, Dual-Rail-Logic still has the benefit that *any* code can be modified circuitry, without any modifications, while still experiencing increased robustness.

In my thesis I explore the possibilities of implementing a similar balancing in software. It works by only using part of the available word size for actual data, leaving the rest for balancing. Specifically, I store 8bit values, along with their balancing counterpart, in a 32bit register. This then means that the data has no influence on the power consumption anymore. I also propose an arithmetic on these balanced values, giving a balanced replacement for all integer operations required for a modern RISC instruction set. With this arithmetic and the balanced values, one can then execute *any* program, while benefiting from a massive increase in robustness against Power-Analysis attacks.

I also provide a plugin for the LLVM compiler that transforms code written for 8bit word-sizes into this balanced form. This proof-of-concept shows that it is possible to execute code using this balanced form. Additionally, it also shows that even such significant changes to the way code is executed can come at no extra cost to the programmer, and the job of generating secure code can at least in part be handed off to the compiler.

Finally I provide an evaluation of my balanced form. By running code compiled with my plugin in the QEMU emulator I can examine the Hamming Weight of values during the execution, and compare them to regular unbalanced code. Evaluating in such a manner simulates an extremely powerful attacker that can examine the Hamming Weight of the result of every single operation. An increased robustness in such a scenario then indicates increased robustness for every real-world attacker.

The rest of this thesis is split into these tree parts.

2 Power Analysis

In most cases the power consumption during execution is data-dependent. Setting a bit to 1 requires more power than setting it to 0. Power consumption is thus directly linked to the Hamming Weight of processed data. An attacker can then measure the power consumption and make inferences on the data being processed.

Performing Power-Analysis requires some setup: 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.

With these power traces an attacker then has the choice of multiple attack forms of varying complexity.[15][8] The simplest form is *Simple Power-Analysis* (SPA), and it involves directly examining the power consumption. As large control blocks can be identified, a data-dependent control flow can leak information this way. An example target would be RSA decryption being calculated via the square-and-multiply algorithm. The difference between the multiply and the square operation is directly observable from a single power trace. As the order of these operations is linked to the private key, identifying the control flow leaks the private key. Figure 1 shows a trace for square-and-multiply in RSA decryption, including the leaked private key bits.

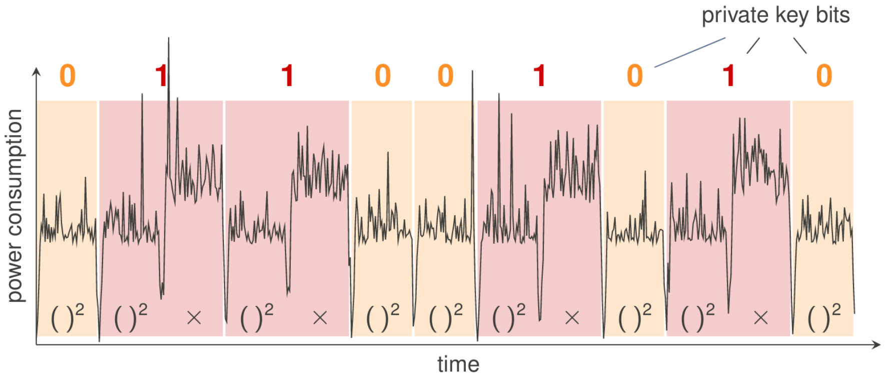


Figure 1: Simple Power Analysis on square-and-multiply RSA[7]

The control flow is often not enough to leak the entire secret, and it is very hard to gain information about the actual data from only SPA. For this a more complex variant of Power-Analysis can be used, namely *Differential Power-Analysis* (DPA). DPA requires a large number of traces, with one factor for the power consumption known. For cryptographic operations this equates a *chosen plaintext* attack scenario.

DPA then attacks the individual bits of the key. The attacker considers two cases, one for each value of the current key bit. First she assumes the value of the current key bit is 0. She then chooses a bitwise operation (e.g. XOR of the plaintext with the key in the first round of AES), and splits the power traces into two sets, based on the value of the target bit in the expected result of this operation. Next she calculates the mean power consumption of both sets. As the value of the other bits, as well as other factors for the power consumption, are randomly distributed, calculating the mean will neutralize them. If her assumption was correct, the difference of both means will exhibit a spike at the time of the chosen operation. Either way, the value of the current key bit is revealed to her.

Figure 2 shows a typical DPA result with the mean power consumptions of both sets, the difference between the two, and the difference with the Y axis magnified by a factor of 15. This analysis was performed on the output of the least significant bit after the first S-box substitution in AES.

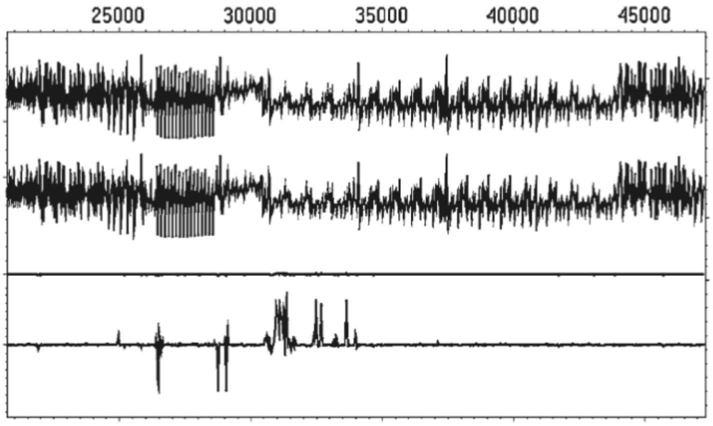


Figure 2: Difference in means during DPA[16]

Even more higher order information about the key can be found with *Correlational Power-Analysis* (CPA). CPA is the most complex of these attacks, but also offers the best results. The attacker starts by making a list of candidate values for every byte of the key. Attacking individual bytes at a time, instead of the whole key, still keeps the required effort feasible. As she knows which algorithm she is attacking, she knows which operations will take place, and can calculate expected intermediate results based on her chosen plaintexts and key candidates. She can then calculate the expected power consumptions for these intermediate results.

The attacker can now calculate the correlation between the expected power consumptions for every key byte candidate and the actual power consumption. The candidate with the highest correlation coefficient is then the most probable value for the current key byte.

Defenses against Power-Analysis attacks

Power-Analysis works because processed data directly influences the power consumption. Defenses against this class of attack usually work by either adding additional factors to the power consumption, thus increasing the computational effort required for analysis, or by reducing variances in power consumption altogether. This reduced variance decreases the information an attacker can gain from the same number of power traces, giving her reduced confidence in her result or requiring her to capture more traces.

Masking[12][10] for example is an algorithm specific defensive measure that adds a third factor to the power consumption by first performing adding a masking value to the plaintext via an invertible operation. The cryptographic algorithm then works on the masked value, and only in the end unmasks the result. As such the attacker has to calculate her correlation for each possible combination of key byte 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[26]. It works by calculating the inverse of every intermediate result along with the actual result, thus balancing the number of 1s. This results in a constant Hamming Weight and therefore a data-independent power consumption.

Unfortunately, Dual-Rail-Logic suffers from multiple engineering problems. The power required to set the value of a bit to 1 is dependent on properties of the underlying transistors, which are subject to variances in manufacturing.[22] Minimal differences in clock timings between both paths can also reduce the security of Dual-Rail-Logic[5]. Storing the inverse also requires significantly larger circuitry, doubling the circuit size or more[5].

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

3 Related Work

I considered related work from three major areas: Software based Power-Analysis defenses that can in theory be applied to any algorithm, approaches to Dual-Rail-Logic, and security related research utilizing LLVM.

First introduced by Messerges[21], masking is in theory applicable to any algorithm. In his paper he identified the core operations in the candidates algorithms for AES. He then devised a way to apply a masking to the input of these operations, in such a way that no intermediate value is stored in memory without being masked. For example, his mask for boolean operations was applied via XOR: $x_{masked} = x \oplus r$, where r is a randomly selected mask. Boolean operations can then be applied to x_{masked}, and intermediate results do not directly give an attacker information about any secrets in the algorithm.

An attacker can still try to extract both r and the secret key at the same time, however this exponentially increases her required effort.

As for many algorithms (including Rijndael, the now chosen algorithm for AES) also require arithmetic operations, Messerges also needed a masking for these. Additionally, a conversion method between both masking variants is required, and this transformation can at no point store x as an intermediate value. Messerge's approach stores either x or $\overline{x}$, depending on a random value.

While this provides adequate robustness against DPA attacking a single bit of the key, it is insufficient for attacks on multiple bits, as shown by Coron et. al.[10]. By attacking two bits at the same time, they could find whether both bits were identical or not, and thus gradually reduce the number of possible keys.

Other work has been done to avoid leaks such as this[4][23], however all of these approaches are currently algorithm specific, with most work being targeted at AES. The decomposition from [21] could be used to automate this masking process for algorithms utilizing these basic operators, and the masking could then be applied automatically during compilation.

Another software approach is inspired by secret sharing. Goubin et. al.[13] used their so-called "Duplication" method to defend DES against Power-Analysis. They did this by splitting the input v into multiple parts $v_1, ..., v_n$ using XOR, then performing all operations on the individual parts, and finally recombining them into the encrypted v'. This works because all operations of DES are invariant to the XOR split performed on v.

The approach was generalized by Chari et. al.[9] using the identity $v = f(v_1, v_2, ..., v_n)$. They state that this secret sharing method can then be applied to any algorithm, as long as no operation invalidates this identity. For example, if one were to use XOR for f as for DES, the Galois Field multiplication in AES would result in an invalid v'.

As such, while the general approach is applicable to all algorithms, the choice of f depends on the present operations, and thus prohibits automated application of this method. However, given multiple candidates for f and a set of operators invariant for each candidate, one could perform static analysis of a program and try to find an f that can tolerate all operations. Some algorithms could then automatically be made more robust using secret sharing.

Dual-Rail-Logic related work is mostly concerned with solving the engineering challenges mentioned in Section 2. Balancing the power consumption via $\overline{x}$ was first proposed by Saputra et. al.[25], where they loaded negations of memory values into a dummy register. This reduced the effect the Hamming Weight of data had on the power consumption, and thus increased the robustness. Sokolov et. al.[26] combined the security benefits of this method with Dual-Rail encoding in general, which had previously been used in circuits without an explicit clock signal.

In this encoding, only $(0, 1)$ and $(1, 0)$ are valid code words, and between every cycle the outputs are returned to a so-called *spacer* with value $(0, 0)$. The spacer value is not a valid code word and as such indicates that the current cycle is not finished.

Sokolov et. al. use the fact that both valid code words in Dual-Rail encoding have the same power consumption. By encoding every bit in Dual-Rail, they drastically reduce the effect of data on the power consumption.

The powerful analysis enabled by LLVM's intermediate representation (LLVM IR) has also sparked security research. One such paper is by Junod et. al.[14], which focuses on automatically obfuscating code during compilation. They insert bogus instructions and replace existing instructions with equivalent ones ($a + b = a - (-b)$), all in LLVM IR. This is very similar to what my balancing pass does, albeit with a different goal. They also change the control flow to increase the difficulty of reverse engineering, however in a constant fashion. Were the changes in control flow non-constant, their pass might also increase robustness against Power-Analysis.

Lyu et. al.[20] also use LLVM and QEMU to analyze binaries for different platforms. By using QEMU to translate machine code into QEMU's intermediate representation, and then translating that into LLVM's intermediate representation, they can utilize the plethora of analysis tools available for LLVM.

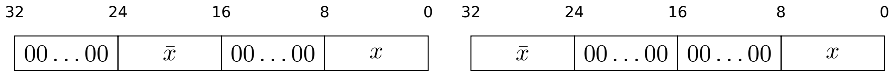
4 Theory

By performing Power-Analysis an attacker can gain information about the Hamming Weights during execution. This means that as long as their Hamming Weights are identical, values are indistinguishable by such an attacker. Having only perfectly balanced values then means that an attacker can gain no information via the power consumption, as all values look exactly the same.

Unfortunately, the design of the ALU does not permit such a scenario. The goal then is to come as close to this ideal as possible, i.e. having a minimal amount of unbalanced values. Balancing individual values can be done perfectly, only the balanced versions of binary operators *must* have imbalanced intermediate values. For my thesis I did not care too much about their optimality, focussing instead on their correctness. The derivation of my binary operators is already a proof of their correctness, and additionally I checked all possible combinations of 8bit values for incorrect results.

4.1 Balancing Individual Values

In resemblence of Dual-Rail-Logic itself, I balance the Hamming Weight by storing the inverse in the same register as the actual value. While this could be done while still keeping 16bit for actual data, using only 8bit gives me more space to store temporary values. This gives me more freedom to balance values during binary operations. With that fixed the only remaining decision is where in the register to put x and $\overline{x}$. I wanted to have room between x and $\overline{x}$ for shifts, so this left only 2 candidates (and their inverses) for what I call the *balancing schemes*. Figure 3 shows the two schemes I chose for my project.



(a) Balancing Scheme 1

(b) Balancing Scheme 2

Figure 3: Balancing Schemes

I found balanced operations for both schemes, but in the end decided to use Scheme 1 as a default because it exhibits nicer behavior 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 finding standardized transformations in both directions I could reuse them in the rest of my arithmetic.

4.2 Balancing Binary Operations

The biggest problem 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, there *must* be imbalanced temporary values during execution. My goal then was to limit the number of these imbalanced values.

For every operation I give the intermediate steps, with a single line denoting an intermediate value. The values are in the form

$$\begin{array}{lllll} \%i = x_1 & \| \, x_2 & \| \, x_3 & \| \, x_4 & | \, \texttt{operation} \end{array}$$

where $\%i$ denotes the "name" of the current intermediate, and x_1 through x_4 are the individual bytes of a register, with x_1 having the most significant, and x_4 the least significant bits. The `operation` denotes how the current value is obtained.

Transforming Scheme 1 to Scheme 2

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

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

LSL here stands for logical shift left.

Transforming Scheme 2 to Scheme 1

The other direction works very similar to the first, and is shown below. Note that ROR stands for rotational right shift, i.e. the values shifted out on the right are shifted back in on the left.

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

ORR

Before finding a balanced variant of bitwise 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:

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

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: $x \oplus y = (\overline{x} \wedge y) \vee (x \wedge \overline{y})$. It is better to create a balanced XOR from scratch, instead of compositioning it from ORR and AND, because both ORR and AND have the same imbalanced intermediate values.

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, and show only the result: $\overline{x \oplus y} = (x \wedge y) \vee (\overline{x} \wedge \overline{y})$.

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

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:

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

Both c and c' denote possible carry values 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:

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

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:

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

DIV and REM

I used repeated balanced subtraction for DIV and REM operations. The code was written in C and can be found in the git of my thesis[1].

Shifting

While performing logical shifts, I need to ensure that the correct bits are pushed in. When 0s are shifted in for x I have to shift in 1s for $\overline{x}$, and vice versa. This is done by ORRing the target value with 0xff000000 or 0x0000ff00, as needed. The shifting is performed normally and the result is then AND filtered with 0x00ff00ff to comply with Scheme 1 again.

4.3 Testing for Correctness

While the individual steps for each binary operator are themselves a theoretical proof for their correctness, I still wanted to validate them. As there are only 256 possible 8bit values, I could easily brute-force every combination of them, and verify the correctness of the result. For this purpose I wrote python code that allows execution of intermediate steps. By specifying the individual steps with lambdas, and then constructing the entire balanced operation from unary and binary operations, I can execute the operation step by step. The intermediate results are then stored in numpy arrays, allowing me to check if the results are correct, and for which values the results are incorrect, as well as where any errors happen. As an example, Listing 1 shows the intermediate steps for multiplication.

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 the AND is required due to python's arbitrary precision integers
    BinaryOperation(3,1, lambda x,y: x + (y << 16)), #4
    UnaryOperation(4, lambda x: x + 0x00ff0000), #5
    UnaryOperation(5, lambda x: x & 0x00ff00ff), #6
])
m.execute()
incorrectResults = m.testCorrectness(lambda x,y: (x*y)&0xff)
print(
      )
```

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. Correctness is then tested by checking if all final results are equal to the output of a reference operation ($x \cdot y$ in this case).

4.4 Evaluating the Balancedness

The 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 4. 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.

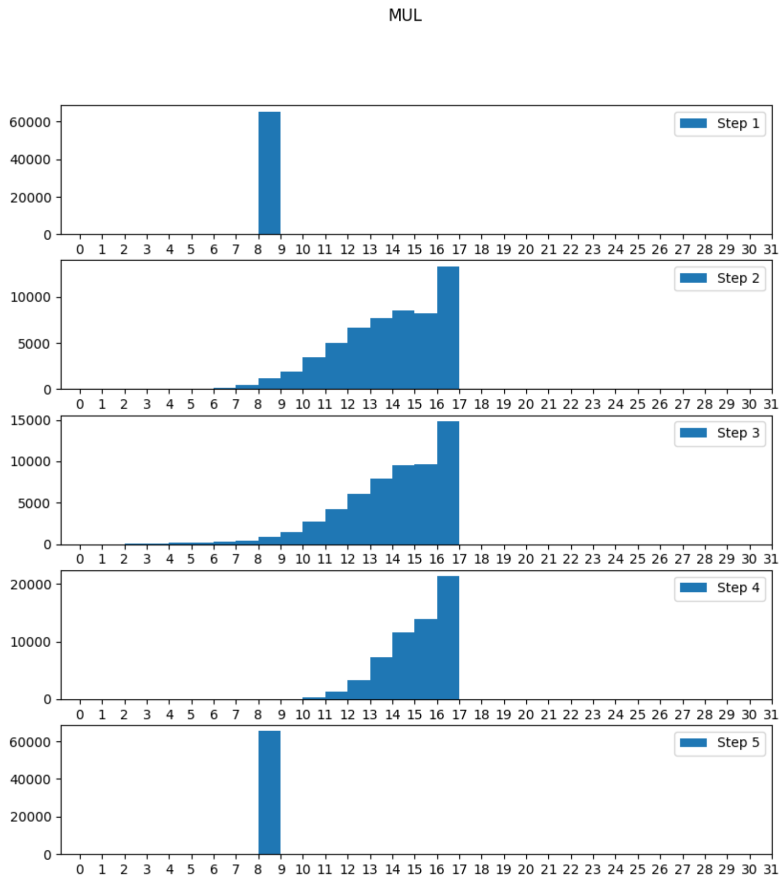


Figure 4: Histogram of Hamming Weights of direct balanced multiplication

While Figure 4 shows imbalanced values in the intermediate steps, it performed faster and more robust than multiplication via repeated addition. Figure 5 shows an evaluation of both variants, evaluated over the multiplications of all possible 8bit factors.

5 Implementation

The transformation of normal code into my balanced form happens automatically inside the compiler. This requires no additional considerations of the programmer, while still providing increased robustness. My thesis is intended as a proof of concept and as such does not balance all variables in the code. It balances only variables on the stack, including function parameters, return values, literals and local variables. I chose the stack as it provides a clean cut with little ambiguity for the programmer, while still balancing enough to decrease variance in Hamming Weights by a measurable amount. Figure 6 shows all candidates for balancing, as well as their relationship. If one candidate has their value set by another (e.g. result of binary operators being stored in variables), then this introduces a dependency for this balancing. These dependencies are shown as arrows in Figure 6. Memory locations (registers, heap memory, etc.) are balanced iff all values stored in them are balanced.

5.1 LLVM

The LLVM compiler infrastructure project[18] contains a number of subprojects, but it is mostly known for being an extremely versatile compiler. It works by using an intermediate representation (LLVM IR) specifically designed to be source and target independent, while allowing for easy automatic optimization. As most of the work in a compiler goes into the so-called optimization passes. This design makes the bulk of the work apllicable to all languages. A language can be added to LLVM by writing a frontend, translating source code to LLVM IR. The translation from LLVM IR to machine code for the target architecture is then done by backends, which can also easily be added. Figure 7 sketches this architecture.

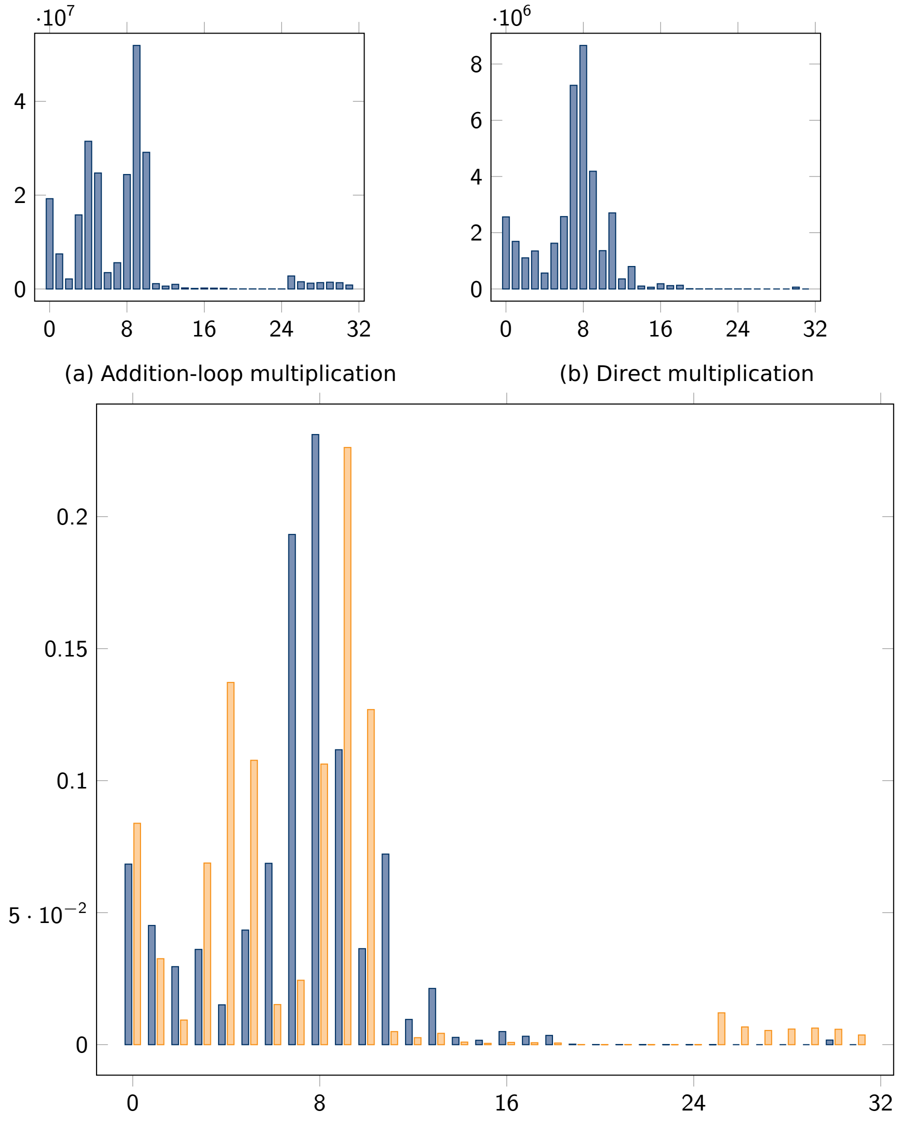
As the optimization passes take LLVM IR as input and provide it as output, they can easily be added. For this reason I implemented the transformation of code into my balanced form as an LLVM optimization pass.

5.2 Balancing Pass

The balancing pass stores balanced values 32bit integers instead of regular 8bit, and then uses my balanced arithmetic operators instead of the normal operators. While this alone is fairly simple, it causes type mismatches for interactions between balanced and unbalanced memory. These then need to be fixed.

Changing the arithmetic also has some implications on comparison operators. All transformations in the order they are applied are as follows:

1. Change the type of all 8bit integers (*int8*) to 32bit integers (*int3*2)



(a) Addition-loop multiplication

(b) Direct multiplication

(c) Scaled Hamming weight histograms for multiplication variants

Figure 5: Hamming weight histograms for direct and addition-loop multiplication

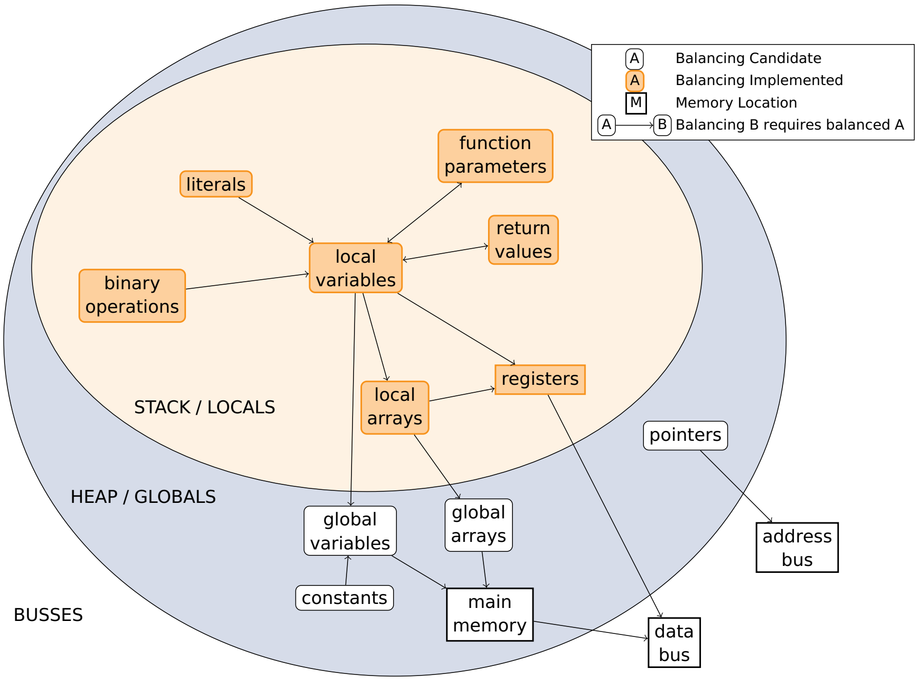


Figure 6: The balancing status of my thesis project

2. Balance constant initializers
3. Balance results of load operations if necessary
4. Unbalance values before store if necessary
5. Use balanced arithmetic operations instead of regular operators
6. Fix comparison directions
7. Fix type issues that arise in the instructions that have not been replaced

The LLVM C++ API provides an iterator over all instructions in a function. My pass uses this to go over all instructions, and transforms the current instruction if necessary. This is usually done by generating a new LLVM IR instruction and replacing references to the old instruction with it.

A special case are function parameters. As these cannot be changed for an existing function, the first step in my balancing pass is to clone the original functions with changed parameter and return types.

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 this creation the pass goes through all parameter

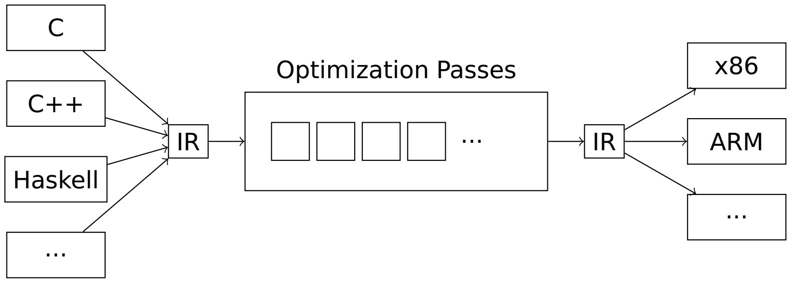


Figure 7: The general architecture of the LLVM compiler

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 inserted into the module, making it accessible in the future. All new functions are given named with the old name, prepended with a `balanced_`. This allows me to skip cloning already balanced functions and call the newly balanced functions.

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, will result in broken code 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.

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 similar 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 dimensions from the old allocation.

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.

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.

Balancing Binary Operations

I implemented the balanced operations described in Section 4.2 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. Division, and remainder however are implemented by repeated addition/subtraction. As an example, Listing 2 shows the balanced function for the `sdiv` (signed division) operation in LLVM IR.

Listing 2: Balanced sdiv

```
int balanced_sdiv(int lhs, int rhs) {
  uint32_t ret = 0x00ff0000;

  uint8_t negative = 0;
  if(rhs & 0x00000080){
    negative = 1;
    rhs = balanced_negative(rhs);
  }


  while (lhs <= rhs) { //~x <= ~y iff x >= y
    lhs = balanced_sub(lhs, rhs);
    ret = balanced_add(ret, 0x00fe0001);
  }

  if(negative)
    return balanced_negative(ret);
  else
    return ret;
}
```

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

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.

Balancing Compares

In both of my balancing schemes the inverse occupies more significant bits than the value itself. This changes the direction of comparison operations, meaning $<$ becomes $>$, $>=$ becomes $<=$ etc.

5.3 Building the Optimization 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 [24].

Listing 3: CMake configuration for my balancing pass

```cmake
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 are needed to build my pass. The pass itself is then built as a *MODULE* library, which tells CMake to build a shared library that can be dynamically loaded at runtime by the optimizer. As the pass is loaded by the optimizer, which is built without runtime type information (RTTI), the pass needs to be built without RTTI as well.

This configuration then builds a file called `libPasses.so`. During the compilation of my test code (Section 6.2) I can then load this library as a plugin for the LLVM optimizer.

6 Evaluation

To test the effectiveness of my balancing I compared balanced code with regular code. In order to decrease the turnaround time during development I decided to run the code in an emulator, as opposed to on actual hardware. This also reduces the amount of effort for evaluation.

Instead of running a full Power-Analysis attack on actual hardware I added code to the QEMU emulator that generates histograms of Hamming Weights during execution. This essentially simulates a scenario where an attacker has precise enough instruments to measure the Hamming Weight of every register during every clock cycle. Such a scenario would equate an extremely powerful attacker, and any robustness improvements for this scenario indicate improvements in real-world scenarios with reasonable confidence.

6.1 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, called *bare-metal emulation*, allows me to evaluate the performance of my thesis project on a simulated ARM processor running on my computer.

During the execution of emulated code, so-called guest code, it can also provide a GDB server, allowing for remote debugging of code running inside QEMU. Unfortunately this debugging has to be done in ARM assembly, as all C debugging information has been lost during the build process.

Memory Layout of QEMU Kernels

Even with bare-metal emulation, QEMU still takes its input as a kernel. Due to this, it starts execution at address `0x1000`, as everything before that address is usually reserved for interrupt handling. This requires some additional setup in my build process (see Section 6.2).

Extending QEMU

To perform any evaluation I first needed to add code to QEMU that computes Hamming Weight histograms during execution. Doing this proved harder than expected, due to optimizations that happen during execution of guest code.

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 while building QEMU). 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, i.e. 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.

6.2 Test Code

I tested my balancing with RC4[2] and AES[11]. RC4 used to be the industry standard for symmetric encryption, and has an extremely small codebase. AES is the current symmetric encryption standard. Both algorithms fit my target use case of encryption on an embedded device.

Building the Test Code

Because I need to load my plugin and my balanced arithmetic functions during the optimization step, and because of the memory layout in QEMU kernels, the build process is a lot more involved than in normal cross compilation.

As discussed in Section 5.1 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-as -ggdb  startup.s -o startup.o
2 clang -target arm-v7m-eabi -mcpu=arm926ej-s -O0 rtlib.c -S -emit-llvm -o rtlib.ll
3 clang -target arm-v7m-eabi -mcpu=arm926ej-s -O0 program.c -S -emit-llvm -o program.ll
4 llvm-link rtlib.ll program.ll -S -o linked.ll
5 opt -load="../../passes/build/libPasses.so" -insert linked.ll -S -o optimized.ll
6 Balancing module: linked.ll
7 llc optimized.ll -o optimized.S
8 arm-none-eabi-as -ggdb  optimized.S -o optimized.o
9 arm-none-eabi-ld -T startup.ld startup.o optimized.o -o program.elf
10 arm-none-eabi-objcopy -O binary program.elf program.bin
```

Lines 2 and 3 show the translation of the C code into LLVM code, using the Clang[17] 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 readable 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 7.

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 5 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 (*-insert* in this case). As discussed in Section 5.1 both the input and output of the optimizer are LLVM IR. Again the *-S* flag is used for human readable output. Line 6 shows output of the actual compiler pass.

In line 7 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 8) and then it is linked with a prewritten memory map and a fixed startup assembly file (line 9). The memory map is required due to QEMU's memory layout, as discussed in Section 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 [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*

```
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 pointer and loads the entry function *c_entry* in my test code, as shown in Listing 6.

Listing 6: Startup assembly code

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

7 Results

For the evaluation both the balanced and the unbalanced version were compiled using my Clang and GNU cross compiler toolchain, with the pass only being run for the balanced version. Both versions were compiled using the

-O0 option, to avoid the influence of any optimizations. Table 1 shows a summary of the properties of both versions. Figures 8 and 9 contain histograms of Hamming Weights during execution of the balanced and unbalanced code. A more detailed discussion of the results follows.

	RC4		AES	
	unbalanced	balanced	unbalanced	balanced
Balancedness	0.236	0.455	0.362	0.584
No. of Operations	77349	401287	83549	2396186
Relative Increase	1	5.19	1	26.68
Code Size	3.5 KB	3.1 KB	5.8 KB	14 KB

Table 1: Properties of balanced and unbalanced test code

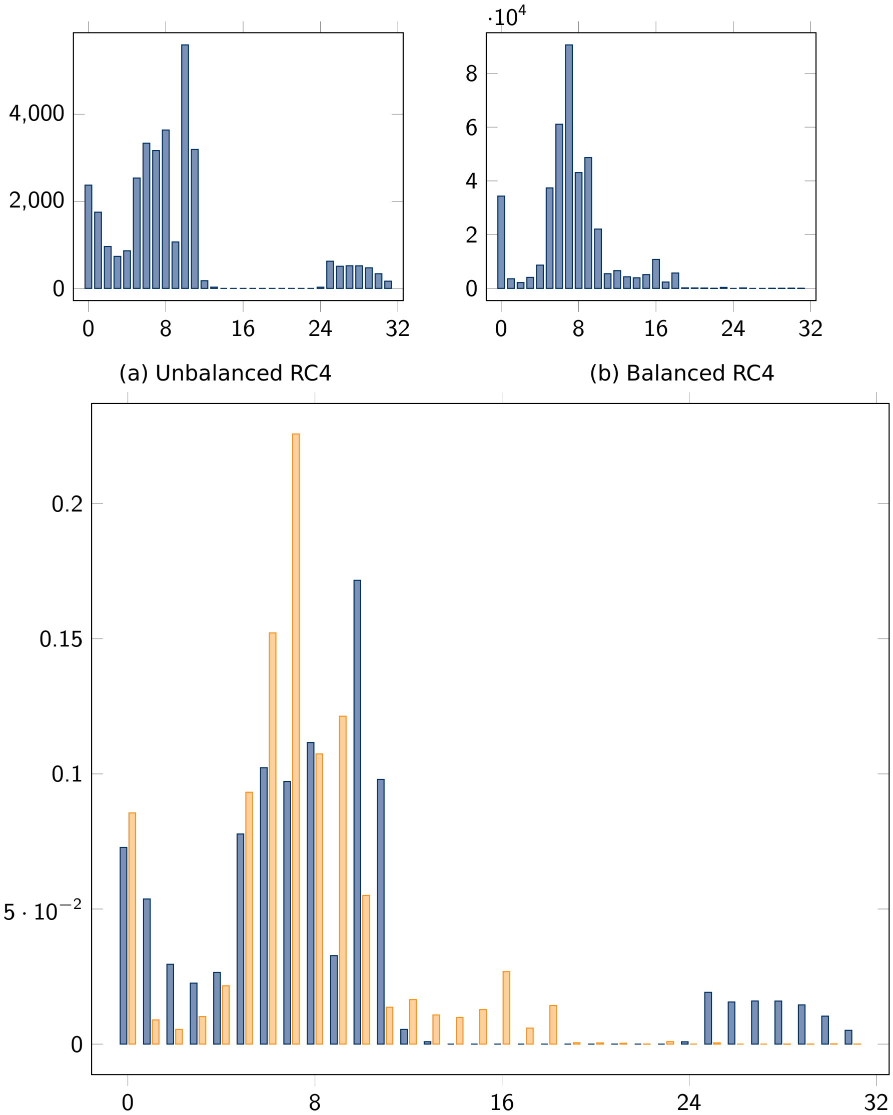
7.1 Robustness

As visible in both Figure 8 and Figure 9, the balanced version has a much narrower distribution of Hamming Weights. Although my balancing does not reach the perfect outcome of identical Hamming Weights for all operations, it shows a significant move towards it. Perfect balancing cannot be reached with current ALUs, as their design forces imbalances in intermediate values. For this reason the balancedness in Table 1 is the ratio of Hamming Weights in the range between 7 and 9, and not only the ratio of Hamming Weights of 8.

The balancedness metric in Table 1 is a very simplistic metric, however, and the histograms in Figures 8 and 9 paint a much clearer picture of the effectiveness of my balanced arithmetic. The balancing works especially well for AES, as shown in Figure 9b. A large number of values is perfectly balanced, as signified by the large value at 8. This is probably due to the large number of table lookups. As I moved the lookup tables for my implementation of AES onto the stack, these were completely balanced. The additional spike around 9 is likely caused by the large number of loops in AES, causing a many increment operations and therefore additions. However, I am not entirely certain this is the only reason, as RC4, which has a similarly high ratio of increments, does not exhibit the same behaviour.

For RC4 the balancing does not work as well. The largest spike in Hamming Weights is at 7 here, and the variance is higher. This value is only an intermediate Hamming Weight for balanced subtraction, which does not occur in RC4. Due to this I assume this is caused by array accesses, which require the value to be unbalanced before it can be used as index. Another possibility is that this is some artifact not directly linked to my arithmetic. The number of values with Hamming Weight 7 is roughly the same as in AES, which supports this theory.

For both AES and RC4 the histograms indicate increased robustness, and the reduced variance in histograms should reduce the amount of information retrievable by an attacker.

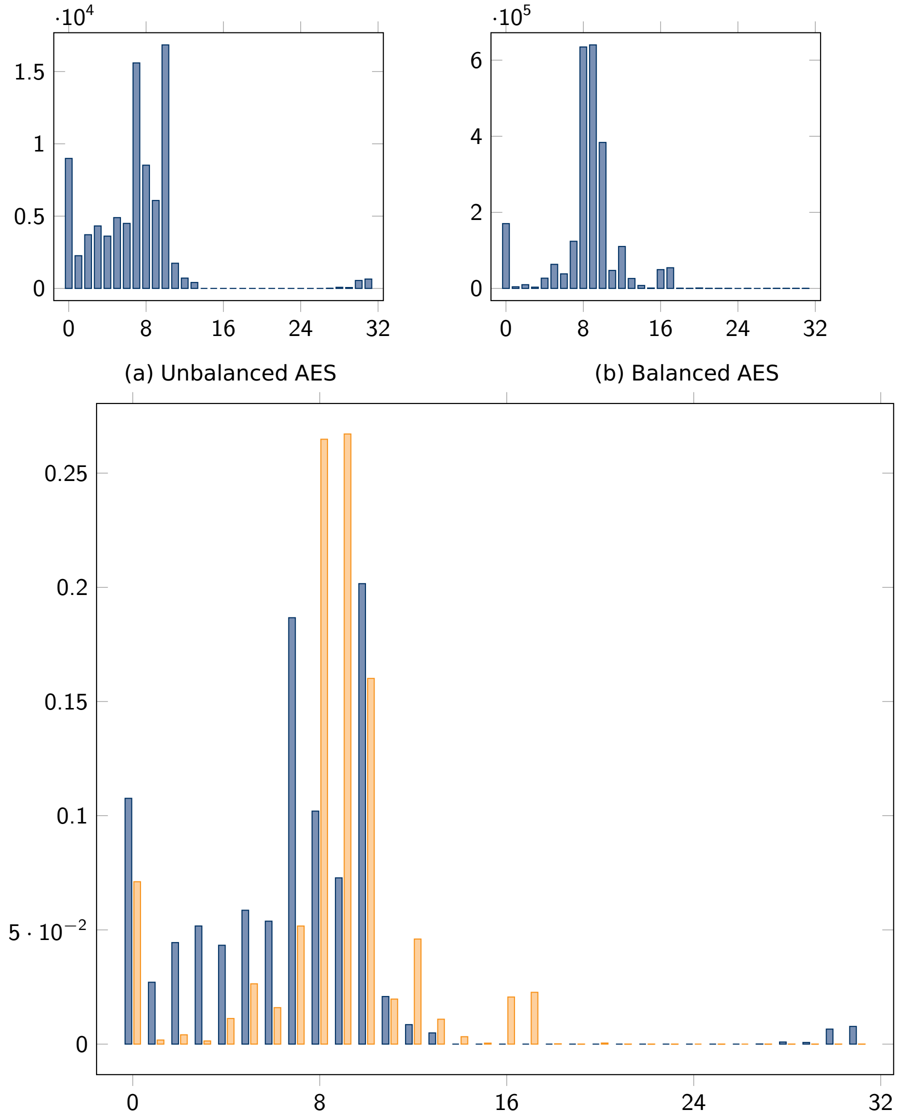


(a) Unbalanced RC4

(b) Balanced RC4

(c) Scaled Hamming weight histograms for RC4

Figure 8: Hamming weight histograms for balanced and unbalanced RC4



(a) Unbalanced AES

(b) Balanced AES

(c) Scaled Hamming weight histograms for AES

Figure 9: Hamming weight histograms for balanced and unbalanced AES

7.2 Performance

Unfortunately the performance impact of my balancing pass is rather large. As shown in Table 1 the number of operations increases by a factor of 5.19 for RC4, and by a factor of 26.68 for AES. A possible reason for this discrepancy is the larger number of array accesses in AES, which all require unbalancing.

Code size does not increase as drastically. For RC4, the balanced code actually requires *less* memory than the unbalanced version. This might be caused by a simple dead code removal routine I added to reduce the lines of code in the LLVM IR files during debugging. The code size for balanced AES should also be fine for most embedded platforms, as that already includes all lookup tables. The moderate increase in code size is caused by the balanced operators being implemented as functions, and not being inlined during compilation. This would allow for an additional memory-performance tradeoff.

As the code was not optimized (`-O0`), there was no loop unrolling. This would cause an additional increase in code size if the operators were inlined.

While the performance impact is (prohibitively) large, please note that performance was not a focus for my thesis.

8 Conclusion

In my thesis I evaluated the robustness of a pure software implementation of Dual-Rail-Logic. By writing a proof of concept implementation I explored a new perspective on hardening embedded platforms against power analysis attacks. 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 capabilities of ALUs. 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 8, as would be ideal. This causes a relative decrease of other values, reducing the information an attacker gains from power analysis attacks. She thus needs a larger number of traces to reach the same confidence in her attack.

While my evaluation was done using C code on the ARM platform, my balancing approach is applicable to code in *any* language for *any* platform. As it is implemented as a transformation from LLVM IR to LLVM IR it can be applied to anything that is compiled with LLVM. This generality is a major advantage over other current Power-Analysis defenses, as almost all of them are algorithm specific.

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 between 5 and 7 for RC4 and AES, respectively. When taking into account the reduction in word size this factor rises up to 28. 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[19] 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. However, 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 much harder by itself, even without additional defenses.

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 a programmer has to make, handing them off to the compiler. With this I hope to help taking a step towards compilers generating secure code automatically, thus allowing for more secure applications, even when neither the money nor the expertise is present for high-quality security auditing.

References

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

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

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

[4] Mehdi-Laurent Akkar and Christophe Giraud. “An implementation of DES and AES, secure against some attacks”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2001, pp. 309–318.

[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] Rainer Böhme. *Information Security II lecture notes*. 2017.

[8] Eric Brier, Christophe Clavier, and Francis Olivier. “Correlation power analysis with a leakage model”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2004, pp. 16–29.

[9] Suresh Chari et al. “Towards sound approaches to counteract power-analysis attacks”. In: *Annual International Cryptology Conference*. Springer. 1999, pp. 398–412.

[10] Jean-Sébastien Coron and Louis Goubin. “On boolean and arithmetic masking against differential power analysis”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2000, pp. 231–237.

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

[12] Jovan D Golić and Christophe Tymen. “Multiplicative masking and power analysis of AES”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2002, pp. 198–212.

[13] Louis Goubin and Jacques Patarin. “DES and differential power analysis the “Duplication” method”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 1999, pp. 158–172.

[14] Pascal Junod et al. “Obfuscator-LLVM - software protection for the masses”. In: *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*. IEEE. 2015, pp. 3–9.

[15] Paul Kocher, Joshua Jaffe, Benjamin Jun, et al. *Introduction to differential power analysis and related attacks*. 1998.

[16] Paul Kocher et al. “Introduction to differential power analysis”. In: *Journal of Cryptographic Engineering* 1.1 (2011), pp. 5–27.

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

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

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

[20] Yi-Hong Lyu et al. “DBILL: an efficient and retargetable dynamic binary instrumentation framework using LLVM backend”. In: *ACM Sigplan Notices*. Vol. 49. 7. ACM. 2014, pp. 141–152.

[21] Thomas S Messerges. “Securing the AES finalists against power analysis attacks”. In: *International Workshop on Fast Software Encryption*. Springer. 2000, pp. 150–164.

[22] 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.

[23] Matthieu Rivain and Emmanuel Prouff. “Provably secure higher-order masking of AES”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2010, pp. 413–427.

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

[25] Hendra Saputra et al. "Masking the energy behavior of DES encryption". In: *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*. IEEE Computer Society. 2003, p. 10084.

[26] 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.