Toward Balancing Arbitrary Code

June 25, 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 the 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. 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. Dual-Rail-Logic is especially notable because it is algorithm independent. It computes the inverse of every intermediate value, along with the values itself. By balancing the value with its inverse Dual-Rail-Logic achieves constant Hamming Weights, and thus a constant power consumption. 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[2], or variances in the production of transistors[4]. It also requires a significant increase in circuit size, doubling the required size or more[2].

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.

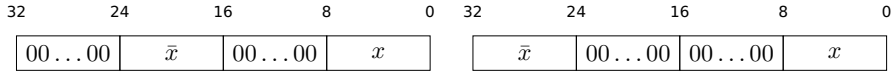
The rest of this thesis is split into these tree parts.

2 Related Work

aoeu

3 Theory

The first step in finding a balanced arithmetic was finding a balancing scheme for 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 operations. Figure 1 shows the two schemes that are used in my project.



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

Figure 1: 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 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.

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.

3.1 Finding Balanced Operations

After fixing the balancing scheme I started working on finding balanced variants for LLVM IR's binary operators. As stated above, 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 3.3.

The notation for the rest of this section is the following: A single line denotes an intermediate 32bit value, with the individual bytes split by $\|$. Notes to the right of $|$ explain how the value in the current line was derived.

Scheme 1 to Scheme 2

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

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

LSL here stands for logical shift left.

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.

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

ORR

Before finding a balanced variant of 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:

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

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: $\overline{x \oplus y} = (x \wedge y) \vee (\overline{x} \wedge \overline{y})$.

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

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:

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

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:

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

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:

$\%1 = 0$	$\| \overline{x}$	$\| 0$	$\| x$	
$\%2 = 0$	$\| \overline{y}$	$\| 0$	$\| y$	
$\%3 = \overline{y}$	$\| 0$	$\| 0$	$\| y$	$\|$ `transform_2_1`$(\%2)$
$\%4 = c$	$\| \overline{x} \cdot y$	$\| c'$	$\| x \cdot y$	$\| \%1 \cdot \%3$
$\%5 = c''$	$\| \overline{x \cdot y} + 1$	$\| c'$	$\| x \cdot y$	$\| \%4 + (\%2$ `LSL` $16)$
$\%6 = c'''$	$\| \overline{x \cdot y}$	$\| c'$	$\| x \cdot y$	$\| \%5 +$ `0x00ff0000`
$\%7 = 0$	$\| \overline{x \cdot y}$	$\| 0$	$\| x \cdot y$	$\| \%6$ `AND 0x00ff00ff`

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.

3.2 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 calculate all operations step by step while saving the intermediate results. 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
])
```

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 function to compare to ($x \cdot y$ in this case).

3.3 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 2. 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.

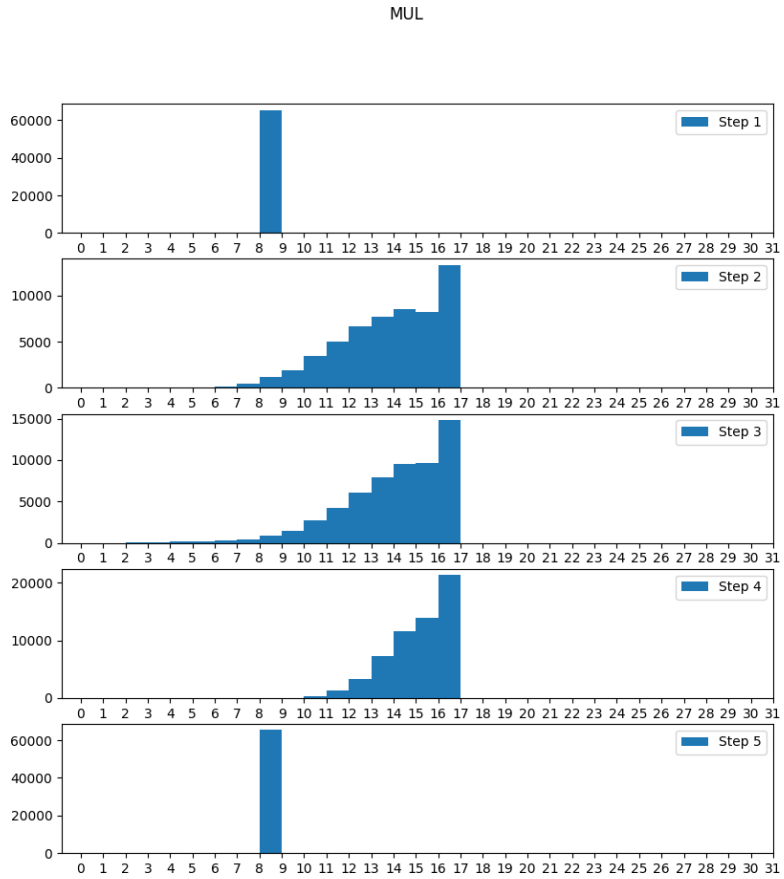
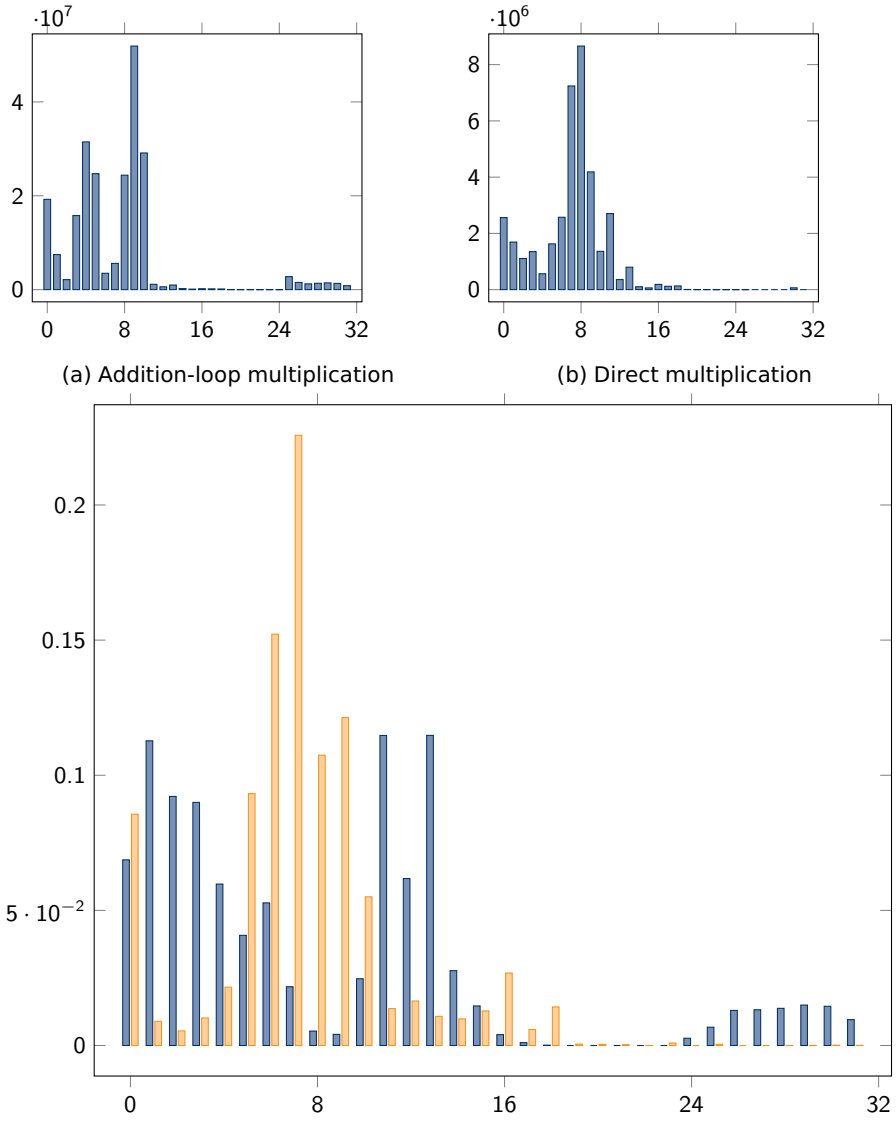


Figure 2: Histogram of Hamming Weights of direct balanced multiplication

While Figure 2 shows imbalanced values in the intermediate steps, it performed faster and better than multiplication via repeated addition. Figure 3 shows an evaluation of both variants, evaluated over the multiplications of all possible 8bit factors.



(a) Addition-loop multiplication

(b) Direct multiplication

(c) Scaled Hamming weight histograms for multiplication variants

Figure 3: Hamming weight histograms for direct and addition-loop multiplication

4 Implementation

The idea behind the balancing pass is very simple.

1. Change the type of all 8bit integers (*int8*) to 32bit integers (*int32*)
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, see if something needs to be done, and if so, performs the transformation. This is usually done by generating a new LLVM IR instruction and replacing references to the old instruction with it. LLVM provides some convenient helpers for such a process.

4.1 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 this 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 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, 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.

4.2 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 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.

4.3 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.

4.4 Balancing Loads

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

4.5 Balancing Binary Operations

I implemented the balanced operations described in Section 3.1 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 3. 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

```
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.

4.6 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.

4.7 Balancing Compares

In my main balancing scheme (Figure 1a) the inverse occupies more significant bits than the value itself. This changes the direction of comparison operations, meaning $<$ becomes $>$, $>=$ becomes $<=$ etc.

4.8 Implementation Status

With all of these instructions balanced, all operations on local variables are fully balanced. To relate this to my plan in **??**, please see Figure 4.

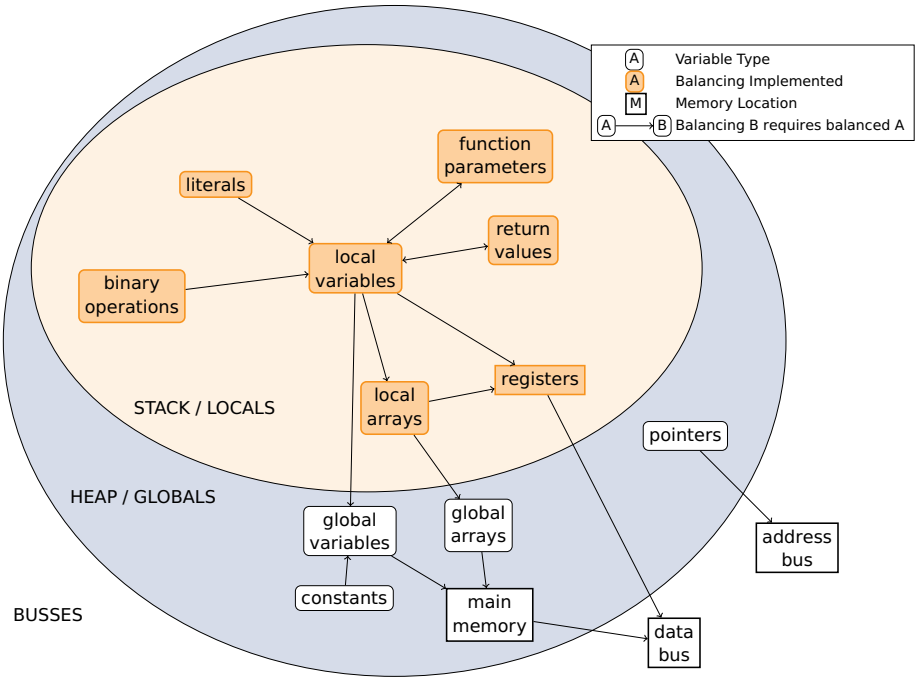


Figure 4: Implementation status of the balancing diagram

5 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 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.

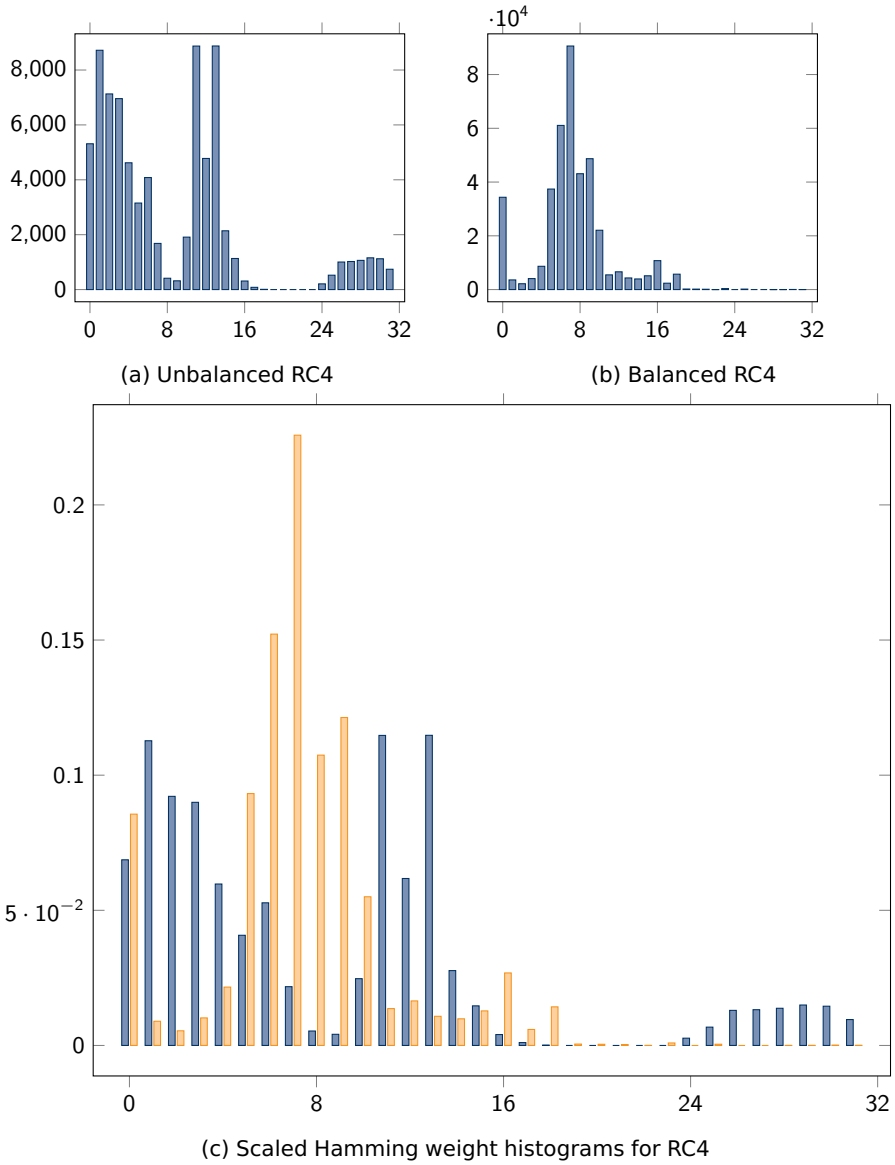
6 Results

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 5 and 6 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.

6.1 Robustness

For both algorithm the balancing 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 identical. Thus, the less evenly distributed the Hamming Weights of intermediate values are, the lower the confidence of her statistical attack.

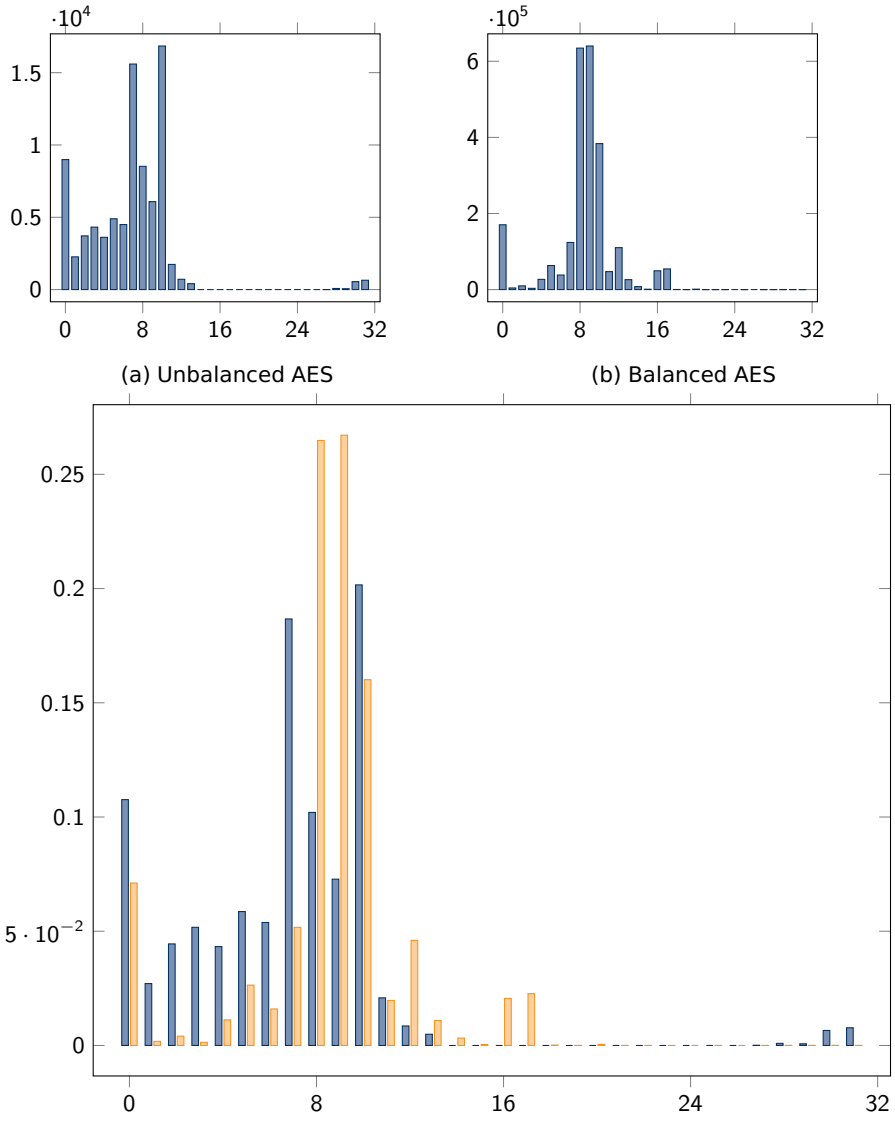


(a) Unbalanced RC4

(b) Balanced RC4

(c) Scaled Hamming weight histograms for RC4

Figure 5: Hamming weight histograms for balanced and unbalanced RC4



(a) Unbalanced AES

(b) Balanced AES

(c) Scaled Hamming weight histograms for AES

Figure 6: Hamming weight histograms for balanced and unbalanced AES

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.

6.2 Performance

The number of operations is $77\,349$ for unbalanced RC4, and $401\,287$ for balanced RC4. That is an increase in the number of operations by a factor of 5.19. For AES the unbalanced code has $83\,549$ operations, while the balanced code has $2\,396\,186$ operations. This is an increase by a factor of 28.68.

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 prevalent in cryptographic algorithms, as they mostly work on individual bytes.

However, performance was explicitly not a goal for my thesis, and I decided to focus solely on robustness.

7 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. 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 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 shift towards values around 8. 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.

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[3] 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] 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.

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

[4] 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.