Toward Balancing Arbitrary Code

June 11, 2019

Contents

1 Introduction

Embedded devices very rarely utilize instruction level parallelism. Thus, as the power consumption is directly related to the bits in intermediate results that are set to 1, their power consumption directly reflects their computation results without much noise. If the device is running a cryptographic operation, this can result in a leakage of keys. This is known as a power analysis side channel attack[2].

While there exist many different defenses against this, both in software and in hardware, the most versatile of them is Dual-Rail-Logic[3]. Unlike most other defense mechanisms, Dual-Rail-Logic can be applied to any program, and works by calculating the inverse result $\bar{x}$ for each intermediate result x. This way, the power consumption (which is directly linked to the number of 1s in the result) is always the same, and the program is thus more robust against power analysis. Unfortunately, using Dual-Rail-Logic requires a significant overhead, doubling the circuit size or more[1]. This requirement makes it unsuitable for small embedded applications like e.g. SmartCards. In order to create a way of hardening *any* application against power analysis attacks, even when there are tight constraints on space, I would like to implement something similar to Dual-Rail-Logic in software. By balancing the values on the data bus, the registers, and the address bus, in this order of priority, I can harden execution against power analysis.
To do this, I want to find a way to represent a balanced 8-bit arithmetic in a 32-bit architecture. While representing $\bar{x}$ and x should in theory only halve the word size, I will need additional space to represent carry bits and (new) intermediate steps in the registers as well, so the word size will probably be reduced to a quarter. The idea is to find a balancing scheme that allows me to perform all arithmetic and logic operations present in the intermediate representation (IR) of the LLVM compiler. Ideally, this scheme has no unbalanced intermediate results at all and utilizes no table lookups.

After finding such a balancing scheme and arithmetic, I want to transform the original code into balanced code in a custom LLVM optimization pass. This pass will transform the IR code of the original program into my balanced arithmetic operation by operation. Keeping the performance impact of this transformation as low as possible - both during compile- and runtime - will be a major concern.

Finally I need a way of evaluating my work. For this I assume a perfect attacker capable of observing the power signature of every intermediate value. The robustness against such an attacker is then represented by the number of unbalanced values during the execution, as well as the ratio of balanced vs. unbalanced values. To find this number I run the resulting code in the QEMU emulator, observing the result of every operation. This allows me to easily test my work in a controlled environment and without any additional hardware.

The rest of this thesis is organized as follows: Section 2 gives an introduction to the tools used, as well as a brief refresher of the algorithms used for testing. Section 3 describes my approach, and Section 4 the implementation details of my thesis. Section 5 shows the evaluation results of my PoC. Finally, in Section 6 I offer a discussion of the results as well as an outlook to possible future work.

2 Background

2.1 Power Analysis Defenses

aoeu

2.2 LLVM

aoeu

2.3 QEMU

aoeu

2.4 AES

aoeu

2.5 RC4

aoeu

3 Methodology

3.1 Arithmetic

aoeu

3.2 Balancing Pass

aoeu

3.3 Evaluation

aoeu

4 Implementation

4.1 Build Process

aoeu

4.2 Cloning Functions

aoeu

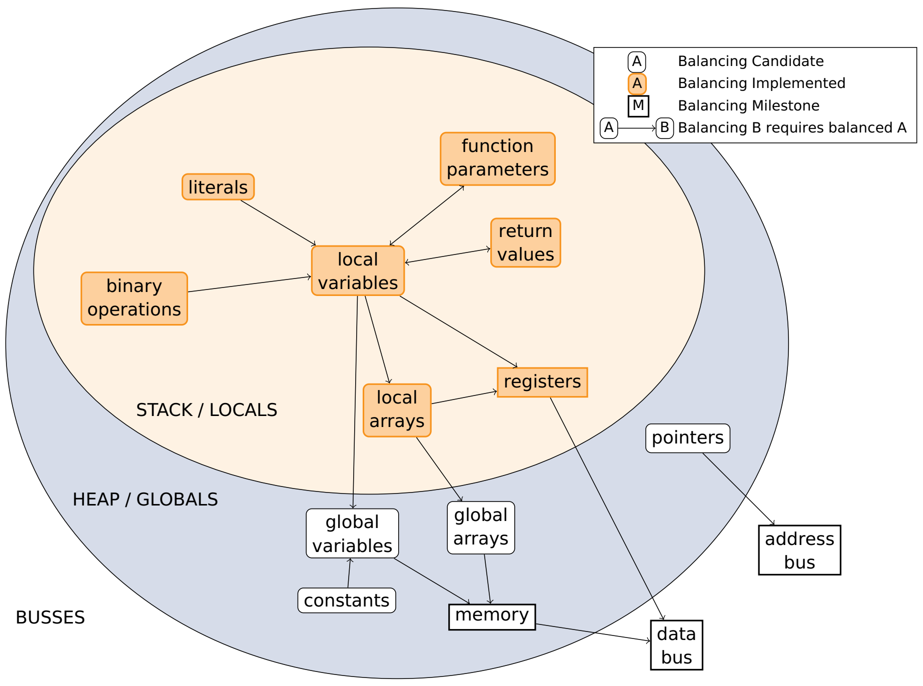


Figure 1: Balancing diagram for LLVM

4.3 Balanced Allocates

aoeu

4.4 Balanced Stores

aoeu

4.5 Balanced Loads

aoeu

4.6 Balanced ZExts

aoeu

4.7 Balanced Operators

aoeu

4.8 Balanced Pointer Arithmetic

aoeu

4.9 Balanced Compares

aoeu

5 Results

In this section I will discuss the balancing results for the two main algorithms I tested the pass on: RC4 and AES. Both algorithms have been written/adapted so that they utilize the stack as much as possible, maximizing the benefit of my balancing pass.

5.1 Security

5.2 Performance

aoeu

6 Conclusion

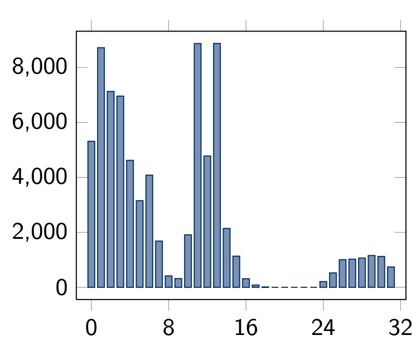
aoeu

References

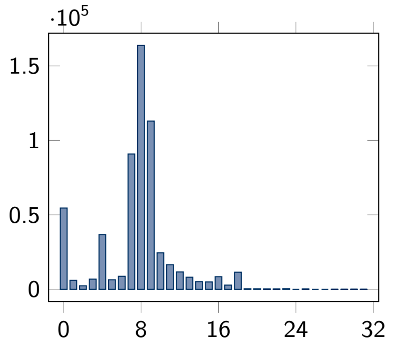
[1] Karthik Baddam and Mark Zwolinski. Path switching: a technique to tolerate dual rail routing imbalances. *Design Automation for Embedded Systems*, 12(3):207–220, 2008.

[2] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

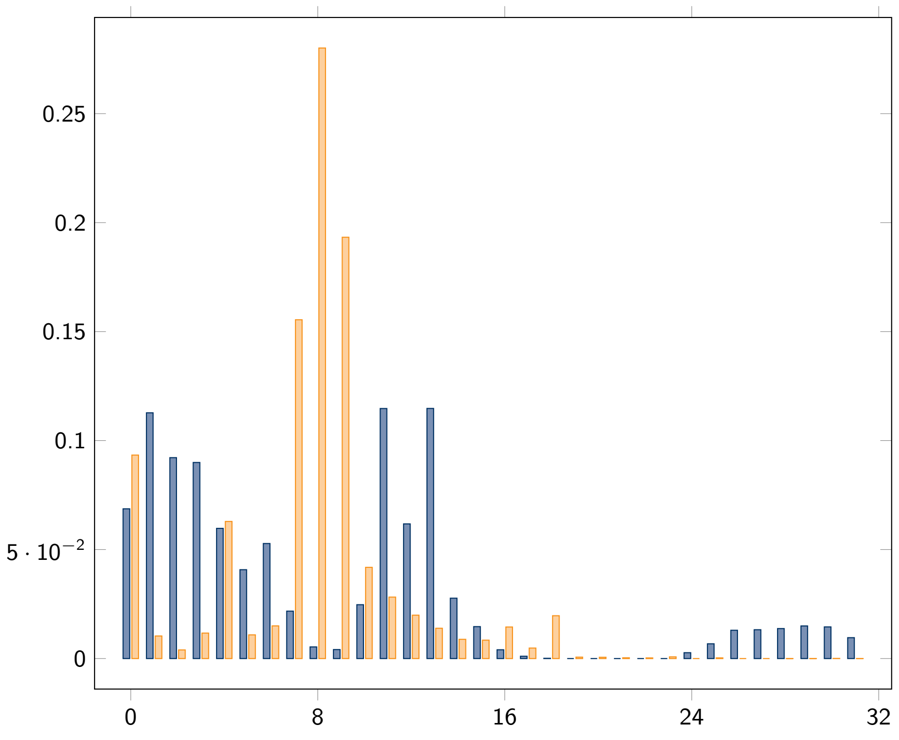
[3] Danil Sokolov, Julian Murphy, Alexander Bystrov, and Alexandre Yakovlev. Design and analysis of dual-rail circuits for security applications. *IEEE Transactions on Computers*, 54(4):449–460, 2005.



(a) Unbalanced RC4

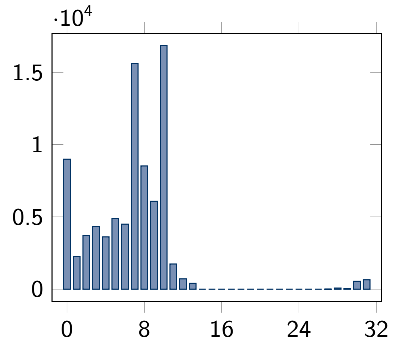


(b) Balanced RC4

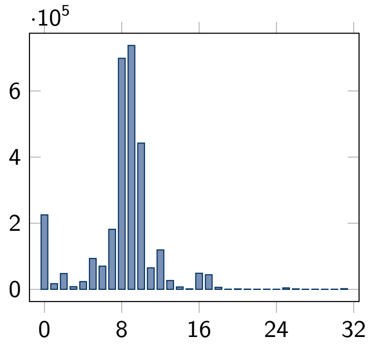


(c) Scaled Hamming weight histograms for RC4

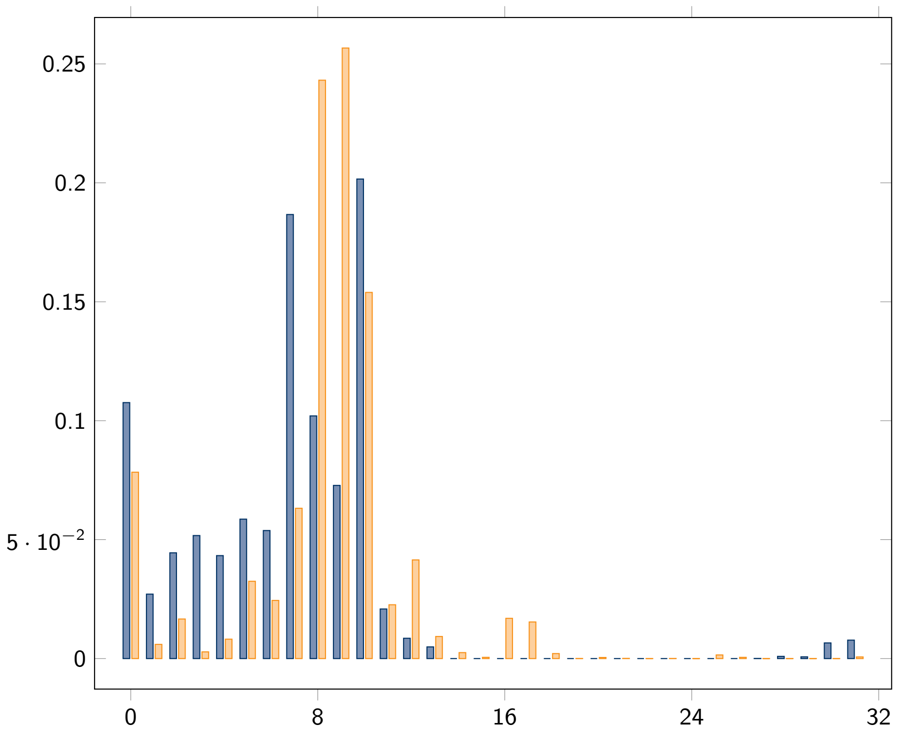
Figure 2: Hamming weight histograms for balanced and unbalanced RC4



(a) Unbalanced AES



(b) Balanced AES



(c) Scaled Hamming weight histograms for AES

Figure 3: Hamming weight histograms for balanced and unbalanced AES