Master Thesis Proposal

Dual rail logic in software as LLVM-IR transformation

Alexander Schlögl

January 14, 2019

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[6]. 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 alterations to the hardware, and almost doubles the required circuitry size. 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 Dual-Rail-Logic in software.

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 run-time - 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, and observe 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 proposal is organized as follows: Section 2 covers Dual-Rail-Logic as well as the LLVM and QEMU projects. In Section 3 I present my intended approach in full, and in Section 5 I discuss problems that might arise during implementation. Section 4 discusses previous work that has been done in similar directions.

2 Background

2.1 Dual-Rail-Logic

Usually, the arithmetic logic unit (ALU) of a processor has a single circuit for every operation it can handle. While there may be cases where there are multiple circuits to enable parallelism, I will restrict myself to the case of one circuit per operation. The main source of power consumption in the ALU is setting bits to 1, which makes it very susceptible to power analysis attacks. Dual-Rail-Logic avoids this fact by replicating a mirrored version of the ALU, which computes the inverse of the original result, i.e. for every intermediate result x the inverse $\bar{x}$ is also computed. This means that for every bit that is set to 1 in x, it is not set in $\bar{x}$. The power consumption due to 1 bits is therefore constant.

Replicating the ALU increases the circuit size for obvious reasons. It almost doubles, only mitigated by the fact that *NOT* operations in the original circuit can be avoided by "crossing the wires" i.e. using a bit from the inverted circuit and vice versa.

Beside the major drawback of increased circuit size, Dual-Rail-Logic is extremely powerful as it makes *every calculation* performed on the hardened ALU more robust against power analysis. This gives Dual-Rail-Logic an advantage over other power analysis countermeasures which operate on a logical level, such as masking[5], which only protect the algorithm they are incorporated into.

2.2 LLVM

The LLVM compiler infrastructure project[3] itself consists of a number of subprojects. The LLVM Core libraries provide source- and target-independent optimization and code-generation for many different CPUs.[4] It achieves this great versatility by working on IR code, which in turn is generated from source code by the compilers of the respective language. This architecture is shown in Figure 1. As visible, the optimization passes transform IR into IR, which makes all of them optional and the addition of new passes fairly easy.

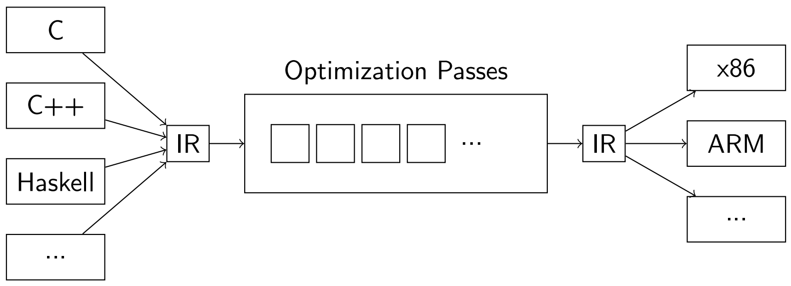

Figure 1: The general architecture of the LLVM compiler

2.3 QEMU

QEMU is a generic and open source machine emulator and visualizer.[1] While it can be used as a full fledged virtualization environment and sandbox, all I need is its ability to run machine code for different architectures. QEMU achieves high emulation speeds by changing the machine code of the emulated system (guest machine) into machine code that the running machine (host system) can understand, instead of simulating a guest machine CPU directly. This translation is done by the Tiny Code Generator (TCG). While this translation is exactly the opposite of what I would want, QEMU can be compiled with an optional filter called the TCG interpreter (TCI). This interpreter runs all operations in on a simulated CPU, and thus allows me to examine the results for "balancedness". Like using the LLVM compiler, using QEMU makes any evaluation mechanism I create for it applicable to desired target platform.

3 Intended Methodology

aoeu

4 Related Work

aoeu

5 Possible Difficulties

aoeu

References

[1] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

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

[3] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[4] Chris Lattner et al. The llvm compiler infrastructure. *URL http://llvm. org*, 2010.

[5] Thomas S Messerges. Securing the aes finalists against power analysis attacks. In *International Workshop on Fast Software Encryption*, pages 150–164. Springer, 2000.

[6] 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.