Master Thesis Proposal

Dual rail logic in software as LLVM-IR transformation

Alexander Schlögl

February 28, 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[4].

While there exist many different defenses against this, both in software and in hardware, the most versatile of them is Dual-Rail-Logic[9]. 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 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 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 4 I discuss problems that might arise during implementation. Section 5 discusses previous work that has been done in similar directions. Section 6 offers a final discussion and justification of my thesis project.

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 data-dependent 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 adding 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 set to 0 in $\bar{x}$. The power consumption due to 1 bits is therefore constant.

While this makes Dual-Rail-Logic extremely powerful in theory, there are multiple caveats to it. Dual-Rail-Logic only provides security if the capacity of the regular and inverted circuit matches closely[10]. As no perfect match is probable in a real-world scenario, all dual rail can provide is more robustness, increasing the number of traces required for a correctly guessed key. Asymmetries in routing between the regular and inverted circuit can also lead to leakage by causing one gate to fire before its inverted counterpart[8]. While methods have been found to mitigate these drawbacks (as discussed in [10] and [8]), power analysis is still an arms race of measurement precision versus leakage reduction.

Even with these caveats, Dual-Rail-Logic offers robustness for *any* code that is run on a hardened platform. This generability makes a reliable implementation of Dual-Rail-Logic very desirable.

2.2 LLVM

The LLVM compiler infrastructure project[5] itself consists of a number of subprojects. The LLVM Core libraries provide source- and target-independent optimization and code-generation for many different CPUs.[6] It achieves this great versatility by working on IR code, which in turn is generated from source code by compilers for 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.

2.3 QEMU

QEMU is a generic and open source machine emulator and virtualizer.[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 translating the machine code of the emulated system (guest machine) into machine code that the system running QEMU (host machine) 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 feature called the TCG interpreter (TCI). This interpreter runs all operations on a simulated CPU, and thus allows me to examine the results for "balancedness".

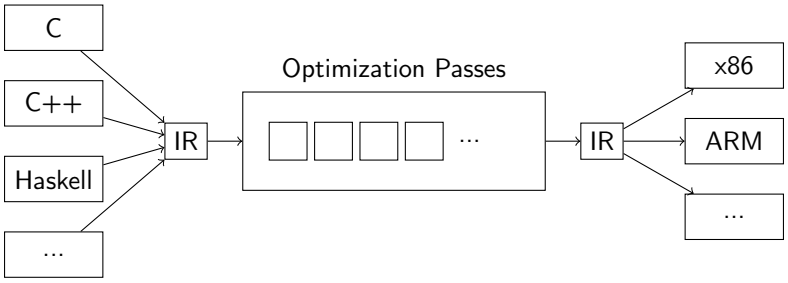


Figure 1: The general architecture of the LLVM compiler

Like using the LLVM compiler, using QEMU makes any evaluation mechanism I create for it applicable to any desired target platform.

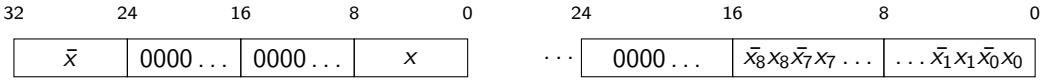
3 Intended Approach

In this section I will explain in more detail the main parts of my thesis:

- Finding a balanced arithmetic suitable for all arithmetic and logic operations present in LLVM IR.
- Implementing a transformation of regular IR operations into above representation.
- Evaluating the arithmetic for balancedness of all intermediate results using QEMU.

3.1 Finding a suitable arithmetic

There are many different ways to balance an 8-bit value in a 32 bit word. The most naive variant is probably balancing the bits directly, as in Figure 2b. However, this is also the variant that creates the most overhead for conversion and operations on it. Much nicer to work with is the variant in Figure 2a, as the balancing part is cleanly separated from the actual data.



(a) Balancing the byte

(b) Balancing the bits directly

Figure 2: Balancing Schemes

The largest problem of using balanced values is that no operation in the ALU will work on them anymore. This means that I have to find ways to execute all arithmetic and logic operations without generating unbalanced intermediate values. The simplest way to implement this would be lookup tables, but the size these would require makes this impossible, especially for embedded applications. A lookup table for 8 bits of data would contain $2^8 \cdot 2^8 = 65536$ entries, but this would have unbalanced indices, and require extraction of the unbalanced values before the lookup. To avoid this the table would need $2^{16} \cdot 2^{16}$ entries or 4.2GB of memory, making lookup tables unusable.

So I will have to try and find a chain of intermediate steps to calculate all operations of the LLVM IR without creating unbalanced intermediate values. While this may not entirely be possible, at least a decrease in unbalanced values can be achieved.

3.2 Implementing the optimization pass

Once I have found a (more) balanced chain of operations for each LLVM IR operation, I can create the optimization pass. Writing the pass should not be too large of a problem, as there are a number of resources for this, and the process is well documented. The naive version will simply have to traverse all original operations and exchange them for their balanced replacements.

Writing a pass that balances the operations without sacrificing all performance will be harder. I will have to see how well my custom pass works together with the performance optimization passes built into LLVM, both for security and performance. This will require testing the balancedness for different configurations and orders of the LLVM optimization process.

3.3 Evaluating using QEMU

This part already has a working first version. I added evaluation code to the optional TCI (see Section 2.3) feature of QEMU. This code logs the operation, the register, the value, and whether or not the value is balanced to a file. As an absolute count, this is completely sufficient. However, in order to debug my optimization pass and to have some more contextual information I might also be interested in the actual Assembly and the more high level call stack. This information can be extracted by moving the evaluation to a GDB (or the LLVM alternative LLDB) script, which can attach to the GDB-server stub that QEMU provides. By using such a script I could also output the C and Assembly code, as well as the actual addresses the guest system uses. By logging data from the TCI evaluating the address bus becomes harder, as the TCI works with host memory directly. However, running larger programs through a GDB script and evaluating registers at every Assembly instruction will probably be too slow, which is why I started with the TCI version.

For now, the current evaluation code is sufficient and provides enough mechanisms to check the data bus and registers for unbalanced values.

4 Possible Difficulties

Finding replacements for all operations will be the most difficult part of this thesis. As I want all intermediate results to be balanced, I might have to switch between different balancing schemes depending on the operation or location of the data. Correctness of the new operations will also be an issue. I will need to prove the correctness of the transformations to and from the balanced representation, as well as for all operations. However, if no single transformation compromises correctness, neither will the collection of all transformations.

The optimization pass also has potential for problems, as the performance impact of the transformations can prove to be too prohibitive. If that is the case I need to find a way to either make my balancing work with the existing optimizers of LLVM or implement some optimization strategies myself.

The most difficult part of creating the evaluation code will probably remain getting my bearings in the QEMU code base, but that has already happened. By now I have an understanding of the threading model and general architecture of the project, and I am confident I can add any required modifications in a reasonable amount of time.

5 Related Work

The only paper applying defensive techniques in the compiler itself is by Junod et al.[3]. They integrate a number of software obfuscation techniques, as well as software integrity checks into

LLVM as optimization passes. Their approach is very close to my master thesis, however the goal is entirely different. Junod et al. work on defending programs against reverse engineering and tampering at runtime. Some of their passes are available online, so that can be used as a tutorial for more complex passes with more interaction with high level data structures like the syntax-tree and the control-flow-graph.

Another paper that works very closely with both LLVM and QEMU is Lyu et al.[7]. They combine the LLVM optimizer with QEMU to extend static analysis tools utilizing the LLVM IR to different target architectures, enabling powerful desktop machines to run the analysis instead of small embedded devices.

6 Conclusion

If successful and without prohibitive performance impacts my thesis would allow generating binaries that are hardened against power analysis, and that for many different platforms. Even if not, future research on this topic can provide more security for any embedded developer, even without the knowledge needed to incorporate masking schemes etc. I believe that security needs more frameworks along the lines of OpenMP[2], to help programmers without security knowledge write secure programs, and so I believe that this thesis would be a good start.

References

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

[2] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[3] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM - software protection for the masses. In *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, pages 3–9. IEEE, 2015.

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

[5] 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.

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

[7] Yi-Hong Lyu, Ding-Yong Hong, Tai-Yi Wu, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. Dbill: an efficient and retargetable dynamic binary instrumentation framework using LLVM backend. In *ACM Sigplan Notices*, volume 49, pages 141–152. ACM, 2014.

[8] Rafael Soares, Ney Calazans, Victor Lomné, Philippe Maurine, Lionel Torres, and Michel Robert. Evaluating the robustness of secure triple track logic through prototyping. In *SBCCI'08: Symposium on Integrated Circuits and Systems Design*, pages 193–198. ACM, 2008.

[9] 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.

[10] Kris Tiri, David Hwang, Alireza Hodjat, Bo-Cheng Lai, Shenglin Yang, Patrick Schaumont, and Ingrid Verbauwhede. Prototype IC with WDDL and differential routing - DPA resistance assessment. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 354–365. Springer, 2005.