Compilation of Tolerance and Control Flow Integrity Countermeasures against Multiple Fault Injection Attacks

Thierno Barry[1], Damien Couroussé[1], Karine heydemann[2], and Bruno Robisson[3]

[1]Univ. Grenoble Alpes, F-38000 Grenoble, France CEA, LIST, MINATEC Campus, F-38054 Grenoble, France
[2]Sorbonne Universités, UPMC, Univ. Paris 06, CNRS,LIP6,UMR 7606 75005 Paris, France
[3]CEA/EMSE, Secure Architectures and Systems Laboratory CMP, 880 Route de Mimet, 13541 Gardanne, France, thierno.barry@cea.fr

January 27, 2017

1 Introduction

Over the last decade, embedded systems have increasingly become a critical part of our daily life, as they represent the largest consumer electronics market segment. Due to the sensitivity and the importance of the data they manipulate, the security of these systems reveals itself as a major concern both for industrial companies as well as for state organizations. Physical attacks, especially fault injection attacks introduced by Bonet et al. [4] aim to exploit the effect of a deliberate disturbance of a system during its operation. They have been shown to require inexpensive equipment and a short amount of time to extract secret information such as a cryptographic key, or to bypass security checks such as PIN verifier. Fault injection can be carried out by means of different techniques, the most common of them are: Variation of the supply voltage, variations in the clock signal, extreme variation of the temperature, focused white light, electromagnetic injection, X-rays and ion beams [1].

Commonly used approaches for software-based countermeasures against fault attacks are: (1) *Source code approach*, which consists of inserting the countermeasure at the source code level, e.g. [6]. The downside of this approach is that the compiler provides no assurance that the countermeasure will be preserved after compilation. Except either disabling the compiler code optimizers, which significantly impacts the code size and its execution speed, or inlining assembly code, which makes the code difficult to maintain, or reinvesting a manual effort to review and rewrite the generated assembly code. (2) The *Assembly code approach* consists in putting the countermeasure at the assembly code level, e.g. [2, 7, 5]. At this level, the code lacks semantic information, such as symbols, type information and number of available registers, making any code transformation difficult to achieve and not without considerable additional costs. Moreover, when it comes to protect a program against

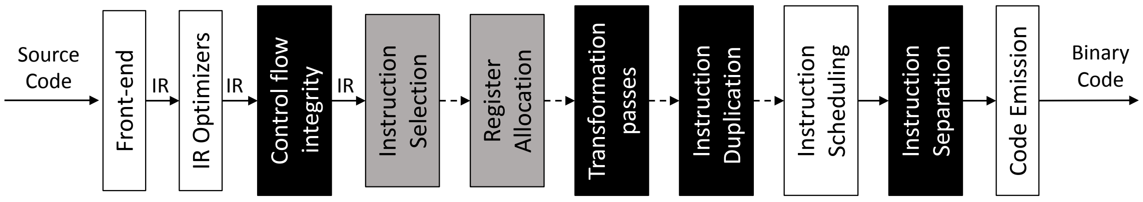

Figure 1: Internal structure of the modified compiler. Gray boxes represent modified passes, black boxes the implemented passes

various models of fault attacks, different countermeasures are incrementally applied regardless of the impact one can have on another. The aforementioned points explain why security experts still manually harden the sensitive parts of an application that need to be secured, and why the application of software countermeasures is still a challenging and costly task in the industry area. In order to reduce design costs and increase the confidence in secure designs, because manual insertion is error prone, industries are strongly in demand of automated tools able to combine various protections while taking advantage of code optimization.
In this work, we investigate the use of a general purpose compiler to automatically harden sensitive parts of a program during compilation. The question raised by this approach is : How to implement hardening passes inside a general purpose compiler primarily designed to produce the most optimized code possible ?

2 Compilation for security

We propose to present the changes we have made to existing LLVM passes and the new passes we implemented in order to enable LLVM to efficiently apply two existing protection schemes, while taking advantage of optimization passes. The protections we considered are designed to protect against fault injection attacks that lead (1) to skip one or several instructions (2) to divert the normal execution flow of the program.
The first one is a tolerance scheme that extends and improve the scheme presented by Barry et al. [3], and relies on that formally verified by Moro et al. [7]. It consists of duplicating the instructions to tolerate instruction-skip fault attacks. Before the duplication, all the instructions must be transformed into idempotent[1] forms.
The second one implement a control-flow integrity (CFI) scheme that seeks to ensure that the execution only passes through approved paths of the program CFG. It is achieved using the step counter principle that consists of initializing a counter at the beginning of each basic block, incrementing the counter after each instruction, and checking the validity of the counter at basic block boundaries to detect a possible control-flow corruption.

We will discuss in a general terms the difficulties that can be encountered when attempting to plug code hardening passes among code optimization passes, for which the underlying questions are : which protection must be applied first ? Before or after which optimization pass ?
In the context of our implemented protections, we will explain why it was profitable in terms of code size and execution speed, to modify some existing LLVM passes.

[1]An idempotent instruction is an instruction that can be freely re-executed, producing always the same result.

The internal structure of our modified compiler is illustrated by Figure 1, where gray boxes depict modified LLVM passes and black boxes depict new implemented passes.
The supplied source code can be annotated to describe the sections of the program that need to be protected. The CFI protection composed of CFG analysis and transformation passes is applied first after IR optimizers. The instruction selection and Register allocation passes are modified to have the generated instructions in an idempotent form. A set of instruction transformation passes are applied afterward to convert the rest of non idempotent instructions into equivalent sequences of idempotent instructions. Then instruction are safely duplicated before instruction scheduling to take benefit of having duplicated instructions scheduled. A separation pass is then run to leave a certain distance between original instructions and its duplicates, the goal is add an additional level of resistance against fault injections. The duplication is applied late in the compilation flow, right before the code emission for the following reasons: (1) to be sure that the duplicates will not be removed, (2) to harden the sections where the CFI is applied, because The CFI alone is vulnerable to instruction-skip.

We will report our experimental results that illustrates the effectiveness, in terms of code performance, of the compiler approach for security in general, and for combining different protections in particular, compared to the source to source approach and the assembly approach. These evaluations have been conducted on an ARM Cortex-M3 microcontroller.

Moreover, depending on the amount of time that will be assigned to us if our submission is accepted, we envisage a short demonstration to illustrate the effectiveness of the implemented countermeasures regarding to the fault injection models they are supposed to protect against.

References

[1] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[2] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni. Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In *Proceedings of the 5th WESS*, page 7. ACM, 2010.

[3] T. Barry, D. Couroussé, and B. Robisson. Compilation of a countermeasure against instruction-skip fault attacks. In *CS2'16*, pages 1–6. ACM, 2016.

[4] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In *EUROCRYPT'97*. Springer, 1997.

[5] R. De Keulenaer, J. Maebe, K. De Bosschere, and B. De Sutter. Link-time smart card code hardening. *International Journal of Information Security*, pages 1–20, 2015.

[6] J.-F. Lalande, K. Heydemann, and P. Berthomé. Software countermeasures for control flow integrity of smart card C codes. In *ESORICS'14*, pages 200–218. Springer, 2014.

[7] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson. Formal verification of a software countermeasure against instruction skip attacks. *JCE*, 4(3):145–156, 2014.