Compilation of Software Countermeasures against Multiple Fault Injection Attacks on Embedded Systems

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

[1]Univ. Grenoble Alpes, F-38000 Grenoble, France CEA, LIST, MINATEC Campus, F-38054 Grenoble, France
[2]CEA/EMSE, Secure Architectures and Systems Laboratory CMP, 880 Route de Mimet, 13541 Gardanne, France – firstname.lastname@cea.fr

April 11, 2017

Fault injection attacks introduced by Bonet et al. [3] 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 [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. [5]. The downside of this approach is that the compiler provides no assurance that the countermeasure will be preserved after compilation. Except 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, 6, 4]. 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 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 protect sensitive parts of a program during compilation. We present our compiler-based approach that takes advantages of both the compiler code optimizers and code transformation opportunities provided by the compiler in order to generate a protected and optimized code.
The protections we considered are (1) tolerance scheme to protect against faults that lead to skip one or several instructions (2) and a control flow integrity scheme to protect against control flow hijacking. We will discuss how we efficiently combined these schemes within our compiler. 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.

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

[4] 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.

[5] 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.

[6] 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.