How to Make Smartcards Resistant to Hackers' Lightsabers?

Philippe Teuwen

NXP Semiconductors - Research, Leuven, Belgium,
philippe.teuwen@nxp.com

Abstract. Cracking smartcards has always been a prized hobby, for the academic glory, for fun (ha, breaking the self-claimed unbreakable...) and for profit (ask the organized crime). State-of-the-art techniques include laser blasts that inject various transient or permanent faults in a program execution, potentially making the smartcard do whatever the attacker wants. After a brief recap of the attack tools and their effects, this article describes how the programmer can protect his/her code using software techniques ranging from cookbook recipes to tool chain automation and how (s)he can evaluate the robustness of his/her code by means of fault injection simulators.

1 Introduction

Today powerful attacks can be mounted against chips protecting security-related secret data, e.g. set-top box chips or bank smartcards hiding cryptographic keys and PINs. In passive attacks, the attacker collects information leaking from the chip and can reconstruct the program structure by power analysis [1] or electromagnetic analysis. Even with a chip filtering its power consumption, power analysis can reveal main consuming operations such as EEPROM writing or the activity of a cryptographic coprocessor. Those passive attacks unveiling parts of the black box can then leverage active attacks where the attacker injects hardware faults e.g. by means of power glitches, active probing or light attacks [2]. Light attacks require the removal of the coating of the chip and consist in blasting photons globally with a white flash or locally with a laser beam on the chip. When a photon is absorbed, it creates a pair of electron-hole, so a local current interpreted e.g. as a logical 1 where there was a 0. Those bit-flips can lead to code or data corruption, ultimately breaking security barriers or introducing errors in cryptographic components. For example, introducing a single error in RSA computation can be enough for the attacker to retrieve the private key if the wrong signature leaks to the output [2]. A laser attack can have several consequences and can be disastrous [3], even if the induced modifications are not fully under attacker's control:

– Corrupting the data (one or several bytes) during its transfer between memory and registers can affect the program flow or the results; e.g. increasing a loop counter in a serial port output routine allows to see more of the memory

than the intended output and reducing a loop counter in an iterative cipher function degrades it into an easy to break single-round variant [4].
- Corrupting one or several opcodes as they're fetched from the memory can corrupt the data operations or prevent critical tests, jumps or calls to be properly executed.
- Corrupting permanently a memory cell is also feasible but harder [5].

Beside hardware protections such as metalization meshes, light detectors, duplication of hardware components or watchdogs, there are also software techniques to detect malicious hardware faults gathered under the term Secure Code Execution (SCE). A first layer of defense consists of a set of cookbook recipes providing a local hardening of the code such as [3]:

- Detection of corrupted data by checksums, variable redundancy or execution redundancy.
- Execution randomization to defeat differential power analysis and to impede the determination of the right moment to perform the injection.
- Ratification counters and baits: small extra operations are introduced and their result checked. After one or several attacks are detected the smartcard ceases to function.

The core cryptographic functions are usually well protected against fault injection and side-channel analysis by countermeasures very tight to the functions themselves such as replaying some specific rounds of DES or verifying a RSA signature after its generation. This is not sufficient; any security related code is to be protected against fault attacks. To achieve this, it is highly desirable to have a generic model encompassing those recipes such that it can be implemented in a compiler with some new pragmas for the developer to identify the secure elements to be protected. In order to evaluate the effectiveness of the recipes and ultimately of such a security-enhanced compiler, it is important to be able to test them as soon as possible in the development chain. This means developing a simulator capable of reproducing the logical effect of a hardware attack and analyzing its impact.
Note that there is also a wide category of fault-injection techniques described in the security domain but it's usually limited to the fuzzing of the environment of a software: its I/Os, environment variables, system calls, network connections etc. and it doesn't assume that the fault can appear in the software's own instructions flow.

The results presented in this paper are organized as follows. Section 2 tries to bring Software Fault Tolerance and Secure Code Execution together while highlighting their differences. Section 3 presents a brief overview of existing Software Fault Tolerance fault-injection techniques. Section 4 explains how a fault-injection tool currently in development and targeted for Secure Code Execution is designed. Section 5 gives more details on the models of faults to be injected. Section 6 sketches conclusions.

2 A Comparison With Software Fault Tolerance

Fighting hardware faults by software means is nowadays a very well studied field called Software Implemented Hardware Fault Tolerance (SIHFT) or just Software Fault Tolerance. But it is presented as a quality and safety problem against natural bit-flips, very rarely as a security problem against malicious fault injections [6]. Indeed SIHFT techniques are primarily developed for e.g. spacecrafts strafed by cosmic rays or equipments close to radioactive elements. Under those environments the hardware is also prone to transient errors and numerous hardware, software or hybrid techniques are studied to mitigate those errors. Given the numerous developments in SIHFT, it is very tempting to apply some fault-tolerance study techniques and frameworks to the field of malicious attacks against smartcards for developing hardening compilers as well as for developing fault-injection simulators.

But one must understand the differences and the limitations of such an attempt to bring those two disciplines together before going further:
The goals: the detection of errors is a common milestone but the finality is different.
On one side Software Fault Tolerance is focusing on the reliability of a system, either by detecting and discarding erroneous results (as e.g. with double-redundancy) or by recovering from errors (as e.g. with triple-redundancy).
On the other side when a smartcard is under attack, there is no need to recover gracefully from errors. But security items (such as the private key or the PIN) must be protected from an exploitation of those errors.
The faults: Software Fault Tolerance is almost always considering a very simple model of the physical transient faults caused by radioactivity or cosmic rays: single bit-flip errors, referred as the SEU (Single Event Upset) model. This is a perfectly suitable model given their working hypothesis that flipping a single bit is indeed the most probable logical effect of a physical interaction between the hardware and low-energy ions or electro-magnetic or nuclear radiation interferences.
But a voltage glitch or a light blast on a chip induces a much stronger effect as it usually affects the whole memory bus or other large parts of the circuit and can be applied or repeated during more than a single instruction cycle. Therefore modelling such attacks must perform more manipulations than a simple SEU.
Their occurrences: Software Fault Tolerance success is expressed in terms of statistics and probability, typically: given random SEU injections over the whole system, what are the percentage of SEU faults leading to errors and the percentage of errors being detected.
In malicious fault injection, the attacks are not randomly distributed but are educated guesses based on some side-channel information. As seen before, even without knowing the source code, the attacker is not in front of a completely black box. Some code structures can be recognized and specifically targeted such as DES rounds or EEPROM writing. As a consequence, the countermeasures will not be evenly distributed either but will concentrate on the protection of the security items.

Their processing: once an error is detected in Software Fault Tolerance, the system is designed to recover nicely by discarding the erroneous values, possibly by repeating the operation or in extreme cases by a reboot.
If the error is malicious, one must take care of not disclosing any information about the error that occurred, neither on the output nor by side-channels. Therefore the error must be detected soon enough. But detecting and reacting too soon can also leak information. For example if the code is comparing two buffers and stops abruptly as soon as there is a mismatch, it's a timing information available to the attacker and very useful to guess a data buffer byte per byte. So the comparison must always take the same time, no matter if and where there are mismatches. The branch in execution once an error is detected can also be a privileged target for a second injection if not designed properly.
The footprints: Software Fault Tolerance techniques easily doubles the memory, execution time and power consumption footprints.
In a smartcard, resources are quite scarce and doubling them is typically not affordable.
The evaluation: to evaluate the effectiveness of software-implemented techniques against hardware errors, both fields can benefit from a framework offering a software-based simulation of transient or permanent faults. Given the level of achievement of SIHFT, it is interesting to explore the different simulation frameworks developed for SEU and adapt them to the malicious injection cases. This subject is covered in the following chapter.

3 Related Work in Software-Implemented Hardware Fault Injection

Hopefully when it comes to simulating hardware faults by software means, the differences between Software Fault Tolerance and Secure Code Execution are more superficial and one can find inspiration in the first field for the second one. Indeed both approaches require the same kind of framework: a hardware simulator with hooks to inject faults and observe their effect, sometimes referred as Software-Implemented Hardware Fault Injection (SWIFI) tool [7].

Some SWIFI tools providing runtime injection (but limited to SEU) are quite appealing, among others:

- ReSP: a simulation platform accepting a SystemC description of a Multi-Processor System-On-Chip (MPSoC) [7].
  ReSP is written in Python and therefore allows all the imaginable hooks and runtime manipulations of internal objects. The analysis of error propagation is done by comparison with a golden model i.e. a non disturbed copy of the architecture. It is worth to note that ReSP is an open-source project [8].
- A JTAG-based fault injection system [9].
  To provide a very fast platform (2ms/fault at 4MHz), the authors were controlling a LPC2129 (an ARM7TDMI μP of NXP) by its JTAG interface; the control part being implemented in a dedicated FPGA. The counterpart

is the loss of flexibility: the faults list must be fixed in advance as well as the golden outputs.
– Tools controlling GDB, the GNU Debugger: initially FIESTA (Fault Injection for Embedded System Target Application) [10] but also others like FAUST (FAUlt-injection Script-based Tool) [11]. The advantage of those solutions is that they can deal with any language and target supported by GDB, which helps keeping focused on the design of the injection process itself.

4 A Tool for Fault Injection Attack Simulation

It was chosen to work on a setup similar to the FIESTA and FAUST frameworks: interacting directly with GDB to inject the faults. To have a more realistic setup of fault injection on small embedded μPs, GDB is combined with Skyeye [12], an opensource tool capable of simulating the ARM7TDMI μP. Skyeye supports the RDI (remote debug interface) protocol of GDB over TCP/IP so it can be controlled by a debugger instance supporting the target, in our case a gdb-arm-linux-gnueabi. Globally a SWIFI framework or pattern system [13] foreseen for Software Fault Tolerance can be applied to Secure Code Execution evaluation but there are still a number of minor differences to keep in mind.

– If building statistics out of a few hundreds random injections makes sense for Fault Tolerance, a more systematic approach is needed for a security evaluation in order to have a better coverage of the code.
– Injections must be repeatable to understand what happened. Therefore a time base trigger is not sufficient and is to be replaced by cycle accurate interrupts to guarantee that a successful injection can be replayed identically later.
– SEU model must be replaced by new classes of code or data manipulations mimicking the logical effects of power glitches or light blasts. This will be detailed in the next section.
– Characterization of the impact of an injection in Software Fault Tolerance is limited to observing the proper termination of the program and comparing the output with a golden output. But the goals are different in a smartcard attack so watchpoints will be placed on the secure elements in memory and breakpoints on the secure code areas to catch when an error gives access to a restricted data or code when it shouldn't and the output is compared with a golden reference when the cryptographic functions are justifiably used.

Because of all those differences, it is essential to work with a very flexible framework nevertheless still fast. The original FAUST approach to generate static scripts and to launch GDB for every single injection is quite slow and lacks flexibility. There are several promising approaches to bypass those limitations: the ability of GDB to load C functions aside the target code and execute them from GDB expressions, the Expect-like capabilities of Python to simulate an interactive session with GDB and a very recent project to integrate

Python scripting directly into GDB [14] where all GDB internals are accessible in Python. Ideally the injections could then be performed repetitively without leaving GDB.

5 Beyond the SEU Model

As said before, the SEU model is far from being sufficient for evaluating Secure Code Execution hardening techniques. On the other hand, performing a full coverage of SEU injections requires already $N \cdot (R + C + D) \cdot W$ simulations for a code executed in N cycles on a core with R registers and $C + D$ words of code and data memory of the supposedly same W-bit width. And any other more complex model will lead to an exponentially larger number of simulations: $1 \mapsto (2^Z - 1)$ to modify Z bits at once, $1 \mapsto Z$ to pursue the attack during Z cycles, $N \mapsto N^Z$ to perform Z attacks during the same execution, etc.

Therefore, it is preferable to reduce the space of possible injections to a set of carefully chosen ones, favoring those with greater chances of success to reveal a security problem. For example, it makes more sense to evaluate attacks on opcodes as the avoidance of the original operation rather than the injection of a new meaningful instruction. When an opcode is modified into another one, most of the time, it will generate the same logical error as if it was replaced by a NOP. Considering separately the mutation of an opcode into every single other value is a loss of time and will lead to unrealistic situations such as the injection of a jump to the secure assets, a situation that would be hard to defeat but which requires hopefully a level of control the attacker cannot reach.

The simplest attack on opcode can therefore be modelled by the replacement of the current opcode by one or several NOPs up to the opcode length. To model longer attacks, one can replace more than a single opcode. The changes must be transient to simulate an attack on the bus so if the changes are done in the memory they must be reverted at the next simulation cycle. Note that in a simulator incrementing the Program Counter is very easy and will have the same effect as substituting opcodes with NOPs. One can also consider arbitrary NOP substitution on a part of the opcode which will lead to a re-interpretation of the next bytes on-the-fly as new opcodes but, if very close to what could happen in a real attack, it will probably not lead to security issues.

Here are some other examples of opcode attacks chosen for their high probability in a real context and their high potential on security: interpret the current opcode and alter its data part (if any) with 0x00 or 0xFF while keeping the operative part; interpret the current opcode and alter its operative part while keeping the data (an INC for a DEC, a STORE for a LOAD, a PUSH for a POP, etc). Attacks on data memory can be limited to occurrences of LOAD and STORE opcodes. An attack will then replace (part of) the data by 0x00 or 0xFF. To make a transient error on a read operation the fault-injection tool will save the current data value, alter it, execute the LOAD instruction and restore the value in memory. For a permanent error on a write operation it will execute the STORE instruction then alter it in memory. Permanent errors on read are also possible to simulate attacks on RAM but don't make sense on EEPROM. Note that a permanent error in RAM is actually permanent up to the next power cycle.

6 Conclusions and Open Problems

The hardening of security sensible code in smartcards against malicious hardware fault injections is today still an artwork exercise left in the hands of developers. Their work would be greatly enhanced if this task was for a part automated in the toolchain. As a first step in this direction, the provisioning of a fault injection simulator tweaked for mimicking active attacks such as light attacks by laser would be of great help. This is the aim of our current project, using the capabilities of GDB to inject faults into an ARM7TDMI simulator. One limitation of the current approach is the absence of prediction of the side-channel leakages caused by a hardware injection.

7 Acknowledgments

I would like to thank my colleagues who offered me their help, especially Ventzislav Nikov, Mathias Wagner and Marc Vauclair for their thorough review and useful suggestions.

References

1. S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, 1st ed. Springer, Mar. 2007.
2. H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," Cryptology ePrint Archive, Report 2004/100, 2004.
3. K. Markantonakis, K. Mayes, M. Tunstall, D. Sauveron, and F. Piper, *Smart Card Security*, 2007, pp. 201–233.
4. H. Choukri and M. Tunstall, "Round reduction using faults," in *L. Breveglieri and I. Koren Eds., FDTC 2005*, 2005, pp. 13–24.
5. S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems.* Springer-Verlag, 2003, pp. 2–12.
6. P. Ammann and S. Jajodia, "Computer security, fault tolerance, and software assurance," *Concurrency, IEEE [see also IEEE Parallel & Distributed Technology]*, vol. 7, no. 1, pp. 4–6, 1999.
7. G. Beltrame, C. Bolchini, L. Fossati, A. Miele, and D. Sciuto, "A framework for reliability assessment and enhancement in multi-processor systems-on-chip," in *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT '07. 22nd IEEE International Symposium on*, 2007, pp. 132–142.
8. "Resp: a reflective simulation platform, open-source mpsoc design." [Online]. Available: http://www.resp-sim.org/

9. M. Portela-Garcia, C. Lopez-Ongil, M. Garcia-Valderas, and L. Entrena, “A rapid fault injection approach for measuring seu sensitivity in complex processors,” in *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International*, 2007, pp. 101–106.
10. N. Krishnamurthy, V. Jhaveri, and J. A. Abraham, “A design methodology for software fault injection in embedded systems,” in *Proc. IFIP Int'l Workshop Dependable Computing and Its Applications (DCIA-98)*, 1998, pp. 237–248.
11. A. Benso, S. D. Carlo, G. D. Natale, P. Prinetto, I. Solcia, and L. Tagliaferri, “Faust: fault-injection script-based tool,” in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, 2003, p. 160.
12. “Skyeye open source simulator.” [Online]. Available: http://www.skyeye.org/index.shtml
13. N. G. M. Leme and E. Martins, “A software fault injection pattern system,” in *Proceedings of the IX Brazilian Symposium on Fault-Tolerant Computing*, 2001.
14. T. Tromey, T. J. Bauermann, and V. Prus, “gdb-python: gdb including python scripting,” (Archived by WebCite at http://www.webcitation.org/5ZQUcBv6x), july 2008. [Online]. Available: http://gitorious.org/projects/gdb-python