Automatic Integration of Counter-Measures Against Fault Injection Attacks

Mehdi-Laurent Akkar[1]

Texas Instruments
821, Avenue Jack Kilby
BP 5
06271 Villeneuve-Loubet Cedex
France

ml.akkar@free.fr

Louis Goubin

Axalto
36-38, rue de la Princesse
BP 45
78430 Louveciennes Cedex
France

LGoubin@axalto.com

Olivier Ly

LaBRI
Université Bordeaux I
351, cours de la Libération
33405 Talence Cedex
France

ly@labri.fr

Abstract. *This paper describes a technology aiming at enforcing semi-automatically counter-measures against fault injection attacks of smart cards. This technology addresses in a generic way the whole software embedded on the card. In particular, it addresses threats going beyond cryptography-related parts of the embedded software, like threats against the firewall of the Java Card embedded virtual machine, the PIN code verification, etc. Counter-measures are automatically integrated to the source code at the pre-compilation step, according to a guideline defined by the programmer under the form of a set of directives included into the source code.*

Keywords. *Security, Hardware, Smart Cards, Fault Injection Attacks.*

Contents

[1] This work was done when the first author was at Schlumberger Smart Cards & Terminals (now Axalto).

1. Introduction

One of the motivations of this work is to be found in the well-known discovery of three researchers of Bell Core (Boneh, DeMillo and Lipton) in September 1996. They proposed a new attack model against smart cards, which they called "Cryptanalysis in the Presence of Hardware Faults" (cf. [3] or [4]). This attack model initially focused on several public-key cryptographic algorithms: the RSA signature scheme and the Fiat-Shamir and Schnorr authentication schemes. In [2], Biham and Shamir showed that DES is also potentially vulnerable to these kinds of attack.

The impact of these kinds of attack – *the fault injection attacks* - goes beyond the implementation of cryptographic algorithms. They actually concern the whole software embedded on the card and can be focused on any point on which the security of the card relies. For instance, they can be used to bypass the verification of the PIN code, or to bypass the run-time checking enforced the firewall of the Java Card virtual machine. They constitute at the moment a serious threat against smart card security, especially considering the growing complexity of the security models of modern smart cards like *e.g.* Java Card based smart cards (see [5]).

In this paper, we describe a technology which aims at *integrating counter-measures* against fault injection attacks in any software in a *semi-automatic way*.

Fault injection attacks consist in perturbing the smart card working at run-time by some physical means. This can be achieved by punctual modification of the physical environment of the card, for instance current glitches on the VCC of the chip, electromagnetic variations, Eddy current (see [4]), or laser emission. Such perturbations introduce errors in the working of the chip. They perturb the processing of core data used by the chip to execute its code (program counter, registers…). In particular, such an attack modifies the control flow of the execution, and in this way can make the smart card bypass some piece of code.

The technology presented here consists in securing the execution by integrating run-time checking aiming at detecting control flow inconsistencies. The principle is to dynamically maintain a history of the execution, and to verify its consistency at some points indicated by the developer.

The counter-measure enforcement is semi-automatic. It is integrated in the compilation process as a pre-processor. The source code is tagged by the developer to indicate the points not to be bypassed, typically the crucial checking related to security (PIN code, firewall checking, etc), and also the control point where he want some consistency checking to be enforced, for

instance just before delivering some asset (a certificate, field access across contexts for Java Card, or more generally any APDU[2] response). Moreover, it allows the developer to parameterise the run-time checking in order to specify some points of the program which must be passed or must not be passed depending on the state of the smart card (personalization, post-issuance, etc).

It must be emphasized that the counter-measures integrated by our system must satisfy some strong constraints regarding dynamic space and size of code overhead. Depending on the chip, they have indeed at their disposal only a few number of bytes of RAM[3], and the code overhead must be within a few hundreds of bytes.

The rest of the paper is organized as follows: first, we describe the general working of the technology, second we address the architecture of the system, then we discuss some experiment result, finally we describe two optimisation features of the system.

2. Principles

The technology consists of a pre-processor which takes a tagged source code as entry and integrates to it an *attack detection system*.

2.1. The Detection System

The principle of the detection system is to maintain an history of execution at run-time and to check the consistency of this history regarding the control flow of the program at some points – *the check-points* - determined by the developer.

As history of execution, we mean the list of the successive program points which have been passed from the beginning. Due to performance and space constraints, the maintenance at run-time of the full list is not possible. So we consider only partial information about it:

First, the detection system does not consider all passed program points but only a subset of them: *the flags*. Indeed, considering all program points would need to insert some processing at each program point which is not realistic: this would give raise to a too important increase of the size of the code. Moreover, the performance in term of execution time would be too much affected.

Second, the detection system can only consider the *set* of flags

[2] *Application Protocol Data Unit*. A command sequence that can be send to the smart card or returned by it.
[3] usually less than 10 bytes.

appearing in the list instead of the *full list* it-self, and/or the length of the list, i.e., *the number* of passed flags. Obviously, this allows to bound the size of dynamic space used by the system: one bit per considered program point plus a counter.

Defining the set of flags and the information the detection system maintains about history of execution (i.e., the full list of passed flags, or just the set of passed flags and/or the number of passed flags) actually constitutes a *compromise* between space, performance and security. It does not make sense to make the system automatically determine this; the interaction of the developer comes here (see Section 2.2).

At each check-point *p*, the detection system checks the history against the control flow of the program. This means that the pre-processor does a static analysis which computes the set of all the lists of passed flags which are possible when arriving at *p*, according to the control flow of the program. And it uses this data to determine which controls must be enforced at *p*.

On top of that, the detection system is refined by making the controls of check-points depend on the state of the card. This means that depending on the state of the card (*e.g.* the value of some variable), a check-point can accept or reject some execution. The set of accepted/rejected executions is defined by the developer, who also defines the conditions on the state of the card under which the check-point accept or reject. This feature is needed because some critical operations like for instance file creation are allowed or forbidden depending on the state of the card (personalization, post-issuance…).

Let us note that this method does not avoid any attack possibility, but actually reduces the weakness of the software to a few and identified points: the check-points.

2.2. The Pre-Processor

The pre-processor takes as entry a source code tagged by some directives given by the developer and automatically inserts the pieces of code constituting the detection system into it, according to the locations and data provided by the directives. The developer defines a guideline for the pre-processor by including *directives* into the source code to be protected. Directives are of the following sorts:

- *Starting Points:* These directives indicate the program points where the detection system must start recording history.
- *Flags:* These directives indicate the locations in the code to be considered by the history.

- *Check-Points:* These directives indicate the locations in the code where the check points must be enforced.
- *Race Declarations:* Such a directive specifies a family of executions – *a race* - together with a dynamic condition which determines whether execution of the family in question must be accepted or not. The family of accepted execution is defined with the flags directives. For each flag, the developer can specifies that the executions of a given family *can / must / must not pass* this flag or that *only* executions of a given family are allowed to pass this flag.

A starting point is replaced by a piece of code resetting the data structure used by the detection system to store the execution history. A flag is replaced by a piece of code updating this data structure. A check-point is replaced by a piece of code enforcing the controls. And a race declaration is replaced by a piece of code which activates or deactivates the race in question.

3. Architecture of the Pre-Processor

This section presents the global architecture of the pre-processor. We present the main ingredients used to construct the detection system.

The pre-processor is made of four global components: a C language parser, a control flow analysis module, a path analysis module, a detection system module, and the kernel. All these components have been developed with Ocaml language (see [8]).

1. The C language parser is a classical component of compilers (see [1]).

2. The control flow analysis module is in charge of computation of the *control flow graph* of each function of the program. Control flow graphs are the main objects used by the static analysis on which is based the construction of the detection system.
This module uses a classical technique based on graph grammars (see e.g. [6]). This consists in generating the control flow graph by applying a vertex replacement grammar along a run on the term representing the considered function. Let us note that we used inductive data types to represent semantic of the program under the form of terms. And we used pattern matching to generate recursively control flow graphs, all of this in a pure functional style.
Let us note that the system cannot address *function pointers* in general. Indeed, the use of this feature makes the control flow graph not computable[4] in general. However, we can address some use of it: for instance, the use of

[4] In the sense of computability theory.

a constant function pointer referencing functions of a constant array. In such a case, all the control flow graphs of the functions of the array are included in parallel in the control flow graph of the function using the pointer. And the transition of the control in one of these functions, which must replace the function call encoded by the function pointer, is considered as non-deterministic. Therefore, we get a non-deterministic transition system for the control flow graph.

3. The path analysis module is central in the system. It is in charge of the computation, for each check-point *p*, of the set of the lists of passed flags which can occur when arriving at *p*. Let us note that it is not in charge of the computation of the real information needed by the detection system. Indeed, this information only is partial information about this set of flag lists and is managed by the detection system module.
The first task of the path analysis module is to extract the sub-graph of directives from the control flow graph of the program. This task implies in particular an exploration of successive function calls where some directives can be inserted. It also implies the analysis of loops where some directive may occur. The insertion of a directive in a non-constant loop is *forbidden*. Indeed, taking into account such directives would implies to take into account flag paths of non constant lengths. The verification of the consistency of such paths would involve some more complex mechanism like for instance an encoding of path by rational expressions. And the consistency checking would need too much resources. So, after extraction of the sub-graph of directives, the only loops occurring are constant, and must be declared as that by the developer, who must also give the number of turns[5]. These loops are developed, in order to finally get the directive graph to be a DAG.
From this graph, and for each check-point *p*, the system computes all the paths starting at some start directive and ending at *p*. Let us note that, if any execution of the function follows one of these paths, some of these paths could however not correspond to any execution. Indeed, we only consider the underlying graph structure of the control flow graph which actually is a transition system. Therefore, the computation of the paths does not consider the conditions of transitions based on data. However, let us note that the set of paths corresponding to real executions of the system is not computable in general.
This set of path is then parted according to execution families defined by race declarations. This consists in selecting for each race all the paths which are admitted according to additional conditions specified into flag

[5] Let us note that the problem of determining whether a loop has a constant number of turn is undecidable in general. However, it would be possible to develop an upper layer which would analyse the loops of the program and decide for some of them of some restricted forms whether it is constant or not.

directives.
Finally, one gets as output of this processing a hash table associating to each check-point a pair made of the set of flag paths which are admitted when no race is activated and a hash table associating to each race *r* the set of admitted flag paths which are admitted when *r* is activated.

4. The detection system module takes this data to construct the actual detection system. The output of this module is a collection of pieces of code to be inserted into the original source code. These pieces of code are of the following kinds:

(i) Declaration of the dynamic data structures to be used by the detection system. Depending on which detection option has been chosen, this may consists of a stack recording flags, or a counter and/or an array of bits.

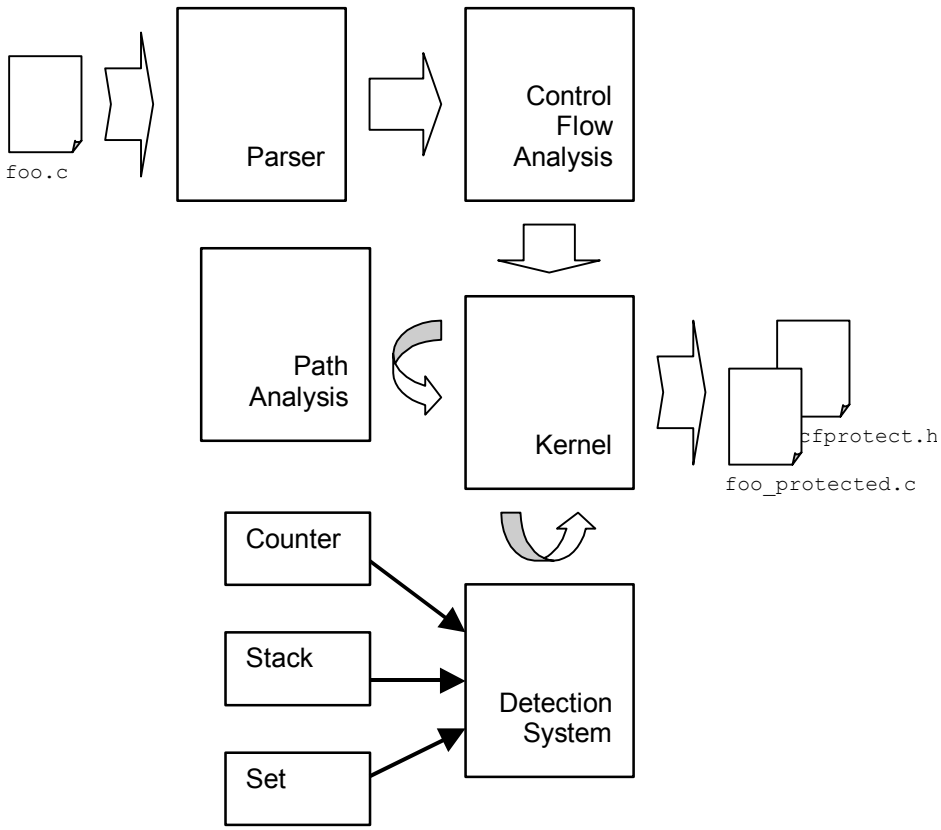
(ii) Definition of static data to be used by the detection system. This concerns the stack protection mode. It consists of all the arrays of flags used for checking the consistency of the current stack.

(iii) Resetting the dynamic data at starting points.

(iv) Updating the dynamic data for each flag. For instance, if the counter option has been chosen, this consists in incrementing the counter.

(v) Enforcing the control at check-points. From the data computed by the path analysis component, one constructs the controls to be enforced for each check-point. These controls may simply be verifications of the value of the counter. Or else they can be checking the stack against some flag static array.

5. The kernel manages the working of the pre-processor at a upper level. The pre-processor can enforce several distinct detection systems on the same project, each being associated to a particular function. Indeed, the different features of the system embedded on the card may not need the same levels of protection. Beside, some specific low level functions may need their own detection system, to be enforced in parallel with the detection systems of the upper level functions. For instance, this is the case of the DES function. The kernel is in charge of organizing the different detection systems.



4. Optimization

This section presents two optimisations aiming at increasing the level of confidence into the detection system, and also at adapting the detection system to the constraints of the smart card environment.

4.1. Balance Algorithm

This optimisation concerns the counter detection mode. Let us consider a check-point *p*. The different admitted paths ending at *p* may have different length. This implies that the control enforced by *p* has several distinct values. This fact presents a disadvantage. If the number of possible values for the counter is comparable to the values themselves, then probability of that some control flow error gives raise to a valid counter value at *p* increases. And thus the detection system may miss number of attack. This situation is not acceptable.

The solution consisted in considering a weight associated to each flag, and adding the value of this weight to the counter when a flag is passed, the weights being determined in order to make the counter value unique at each

check-point (see example page 9).

However, depending on the graph of directive, this balance is not always possible. And one actually may have to insert some additional flag directives into the source code. Again, there is a compromise to do between the number of possible counter values at check-points and the number of new inserted flags which increase the size of the code of the detection system.

4.2. Dynamic Memory Sharing

Here we consider an optimisation of the amount of dynamic memory used by the detection system.

The enforcement of several detection systems into the same embedded system implies that each of them has its own dynamic memory at its disposal. But some detection systems may never be involved at the same time, and therefore could share the same dynamic space.

Regarding that, the kernel computed a partition of the set of detection system into parts made of detection systems which never occur at the same time. Such a partition can be computed from the control flow graphs. And attributes a bloc of dynamic memory to each part, to be used by all the detection systems of the part.

5. Example

Here we give out an example of the processing. The entry of the pre-processor is the following piece of code, tagged by directives. Directives have the form:

`#pragma CFP` *`funid <directive>`*

Where *`funid`* is the identifier of the function to be protected, and *`<directive>`* is the actual directive, here it takes the following form:

- `counter|set|stack protection`: the detection mode (line 7).
- `start`: the beginning of the detection (line 8)
- `flag`: a flag (line 15,26 and 31)
- `verify`: a check-point (line 19).

```
/*********************************************************/
/* Control Flow Protection: Example                      */
/*********************************************************/

void process (int entry) {

#pragma CFP process counter protection
#pragma CFP process start

        if (cond1(entry)) {
                action1(entry);
        }

        if (cond2(entry)) {
#pragma CFP process flag
                action2(entry);
        }

#pragma CFP process verify
        return;
}

/*********************************************************/

int action1(int entry) {
#pragma CFP process flag

        if (cond11(entry))
                return action11(entry);

#pragma CFP process flag
        return action12(entry);
}

/*********************************************************/
/*********************************************************/
```

Here is the output of the processing:

```
#include "cfprotect.h"                                                  1
/*********************************************************/
/* Control Flow Protection: Example                      */
/*********************************************************/
                                                                        5
void process (int entry) {

#if CF_PROTECTION_PROCESS==1
__PROCESS_RESET(1);
#endif                                                                  10

        if (cond1(entry)) {
                action1(entry);
        }
#if CF_PROTECTION_PROCESS==1                                            15
else { __PROCESS_INCR(2); }
#endif

        if (cond2(entry)) {
#if CF_PROTECTION_PROCESS==1                                            20
__PROCESS_INCR(1);
#endif
                action2(entry);
        }
#if CF_PROTECTION_PROCESS==1                                            25
else { __PROCESS_INCR(1); }
#endif

#if CF_PROTECTION_PROCESS==1
if (!((__PROCESS_VERIFY(4))))                                           30
        { __PROCESS_ERROR; }
#endif
        return;
}
                                                                        35
/*********************************************************/

int action1(int entry) {
#if CF_PROTECTION_PROCESS==1
__PROCESS_INCR(1);                                                      40
#endif

        if (cond11(entry)) {
#if CF_PROTECTION_PROCESS==1
__PROCESS_INCR(1);                                                      45
#endif
                return action11(entry);
        }
#if CF_PROTECTION_PROCESS==1
__PROCESS_INCR(1);                                                      50
#endif
        return action12(entry);
}

/*********************************************************/             55
/*********************************************************/
```

Here the detection system uses a counter. Directives have been replaced by the corresponding pieces of code constituting the actual detection system. We do not give out the definitions of `__PROCESS_RESET`, `__PROCESS_INCR`, `__PROCESS_VERIFY` which are straightforward. They are included in a new file generated during the processing (`cfprotect.h`)

included at line 1.

Let us note that the optimisation described in Section 4.1 led to the insertion of new flags at lines 15, 25 and 44.

6. Experiment

The system has been experimented on a smart card chip. The embedded software has been designed especially for this experience. It consists in a basic arithmetical computation involving several successive constant loops. The computation is requested by an APDU, and the result is returned as an APDU.

We choose to enforce the detection system in the stack mode. This is the strongest detection mode, but also the most expensive regarding dynamic and static space and execution time. It is actually dedicated to the very crucial parts of the embedded software.

The experiment has consisted in perturbing the computation by laser emissions. For each execution, the following scenarios are possible:

1. The *result is correct* and the system *did not detect* any perturbation
2. The result *is not correct* and the system *detected* a perturbation.
3. The result *is correct* and the system *detected* a perturbation.
4. The result *is not correct* and the system *did not detected* a perturbation.

As a result of the experiment, it turned out that, among the executions leading to an erroneous result, approximately 80% correspond to a perturbation that was successfully detected by our protection system (this is scenario 2). Equivalently, approximately 20% correspond to a perturbation that was not detected (this is scenario 4).

Cases 1 and 2 correspond to what is expected of the detection system. The case 3 may occur if the laser emission perturbed the detection system itself without perturbing the computation. For instance, the laser emission may make the chip bypass a flag. There are two explanations for the case 4: first, the laser emission has been sufficiently precise to perturb a single operation without perturbing the detection mechanism. Second, the laser emission perturbed all the check-points after having introduced an error.

7. Conclusion

The technology presented in this paper contributes to improve the smart card development methodology regarding the state of the art of fault injection attacks.

It provides a solution which allows the developer to determine and to enforce compromises regarding the following constraints with very limited additional development costs:

- Maximize the resistance level of the embedded software.
- Minimize the overhead regarding the size of embedded code.
- Minimize the amount of dynamic memory allocated to the detection system.
- Minimize the overhead of execution time.

Furthermore, the use of directives to specify the detection system is well adapted to a modular development process. Indeed, the generic aspect of directives allows to specify the protection independently for each module of a software without taking into account protections of other modules and of the upper and the lower layers. The integration and the organization of the resulting detection systems is the charge of the pre-processor which does it automatically. So, this technology allows a modular and distributed process for the development of counter-measures.

8. References

[1]. A. Aho, R. Sethi, J. Ullman, *Compilers*. Addison-Wesley, 1986.

[2]. E. Biham, A. Shamir, *Differential Fault Analysis of Secret Key Cryptosystems*. In Proceedings of CRYPTO'97, Lecture Notes in Computer Science, Vol. 1294, Springer, pp. 513-528, 1997.

[3]. D. Boneh, R. DeMillo, R. Lipton, *New Threat Model Breaks Crypto Codes*. Bellcore Press Release, September 25th, 1996.

[4]. D. Boneh, R. DeMillo, R. Lipton, *On the Importance of Checking Cryptographic Protocols for Faults*. In Proceedings of EUROCRYPT'97, Lecture Notes in Computer Science 1233, Springer, pp. 37-51, 1997.

[5]. Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Addison Wesley 2000.

[6]. J. Engelfriet and G. Rozenberg. *Node Replacement Graph Grammars*. In *Handbook of Graph Grammars and Computing by Graph Transformation.* G. Rozenberg Ed. World Scientific 1997.

[7]. J.-J. Quisquater, D. Samyde, *Eddy Current for Magnetic Analysis with Active Sensor*. Proceedings of E-Smart 2002.

[8]. The OCAML Language. `http://www.ocaml.org`.