About an Automatic Fault Injection Protection System

Mehdi-Laurent Akkar[1]

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

ml.akkar@free.fr

Louis Goubin

Schlumberger Smart Cards
BP 45
36-38 rue de la Princesse
78431 Louveciennes Cedex
France

LGoubin@slb.com

Olivier Ly[2]

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

olivier.ly@labri.fr

Abstract

This paper describes a system aiming at enforcing semi-automatically countermesures against fault injection attacks of smart cards. This system consists of a preprocessor which processes C source code in order to make it resistant against fault injection attacks.

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 this kind of attack.

In the present paper, instead of considering the special case of smart card implementations of cryptographic algorithms, we are more generally interested in the whole operating system of a smart card, and more precisely in its global correct behaviour.

Fault injection attacks consist in perturbating the code execution within a smart card. This can be achieved by intentional modification of the physical environment of the card, for instance current glitches on the VCC, electromagnetic variations, Eddy current (see [4]), laser emission, ... This is a serious threat for smart card security and

[1] This work was done when the first author was at Schlumberger Smart Cards.
[2] This work was done when the third author was at Schlumberger Smart Cards.

can be used *e.g.* to bypass some crucial verification steps such as signature or PIN verification.

We present here the principle of a semi-automatic tool that secures any piece of software implemented on a smart card, by checking that the code is correctly executed, either concerning the intermediate steps of the different functions (code execution, loops, tests, ...) or concerning the calls between one function to another.

The protected code maintains dynamically a history of its execution; and at some points called *control points*, it checks the consistency of this history. The source code is tagged by special *flags* which indicate locations to be considered; the history is encoded by a stack which stores the list of flags which have been passed during the execution. Then, when a control point is reached, the consistency of the contents of the stack is *checked* according to some static information. If this checking fails, this means that the location which has been reached is not consistent regarding the history of the execution; this means that an error occured.

The preprocessor computes the achitecture of this runtime protection and includes it into the source code to be protected. This processing follows a guideline made of a set of directives given by the developer. These directives are of the following kinds:

- **Starting points**: such directives indicate the locations in the code where the protection must start.
- **Race conditions**: such directives specifies some dynamic conditions to be fulfilled by the execution of the program.
- **Simple flags**: such directives indicate the locations in the code to be considered by the history.
- **Control points**: such directives indicate the locations in the code where history consistency checks must by processed.

Let us note that race conditions goes beyond the scope of the protection. They actually allow the developer to specify a run-time checking of some execution properties. Precisely, such directives specify some families of executions, and a function defining, according to some dynamic conditions, the actual family to which the execution must belong. Each time a race condition is passed, a family of execution is chosen by this function. Then, the next control points perform their checking according to this choice.

The preprocessor processes a C source code tagged by such directives. The main ingredient of this processing is the computation of the *control flow graph* (see [1]) of the program. That is the graph whose vertices are the C expressions which have to be evaluated during execution; and arrows encode the ordering of these evaluations. This graph is computed by a static analysis of the code. Consistent executions of the program are encoded by paths in this graph. The preprocessor computes these paths, and produces, for each control point, the list of consistent flag list. Each time the program passes a control point, it verifies that the stack encoding the execution history is consistent regarding this list; moreover, if some race condition have been set up, this

verification is done regarding which family of execution have been authorized.

The solution proposed here thus provides a generic protection against fault injection attacks. Moreover, only a very limited additional work is required from the developer to prevent his code from such attacks.

2. Basic Tools

2.1 Flags and Flag Functions

Definition: A *flag* is a piece of information that defines the characteristics of an execution point of the program.

Definition: A *flag function* is a function that will be called by the program each time it passes the execution point corresponding to a flag, and will consist in storing in the shared memory some information about the flag.

Examples
A flag can be for example :

- An integer that allows to know the localisation of the flag, or a boolean that defines *e.g.* whether it is the first of the last flag.
- A more complex structure that describes a set of information concerning the current state of the electronic device that executes the code. For instance, a data structure that characterizes, depending on the value of a register, or a given variable, the set of flags we do NOT want to pass in the sequel of the program execution.

A flag function can for instance execute the following operations :

- Examine if the flag is the first flag (thanks to a specific data of the flag).
    - If yes, create an empty stack in the shared memory, put the flag identifier on the stack and continue the execution of the code.
    - If no, put the flag identifier on the stack and continue the execution of the code.

2.2 Control Points, History Verification

Definition: A *control point* is a data structure that contains the information that will be used in the history verification function.

Definition: A *history verification function* is a function that will be called at each control point, in order to verify the consistence of the information that was stored in

the shared memory during the successive flag function calls.

Examples

- A control point can be defined as the set of all the lists of flags that correspond to admissible execution paths that reach this control point.
- The verification function can consist in verifying that the contents of the stack (list of passed flags) corresponds to one of the precomputed lists stored in the control point. If this is not the case, an error is detected and signaled.

3. Principle of the Preprocessor

Here we describe the working of a preprocessor of C source code which enforce error detection in a semi-automatic way.

The preprocessor transforms a C source code fragment in order to make it detect errors at runtime.

The transformation is semi-automatic: it is driven by some special directives included in the source code fragment to be processed. This directives can take the following forms:

- **start**: this directive specifies that the flag stack must be emptied.
- **flag**: this directive specifies that a new flag must be pushed onto the stack.
- **verify**: this directive specifies that a control point intended to verify the flag stack consistency, i.e., that the stack well records a history of flag which corresponds to a correct execution of the program.
- **race *n* cond**: this directive specifies that if the boolean expression *cond* is true, then only the execution paths of the family *n* will be admitted by the control points occuring in the rest of the execution. Such a family is defined by the directives of the following sort:
- **flag !n1 ... !nk m1 ... mk**: this directive specifies that the execution paths of families *n1...nk* must not go across this point, and that the execution paths of families *m1...mk* must go across this point.
- **Loop n**: this directive specifies the start of a loop whose turn number is *n*.

Here we give a source code fragment annotated by some directives as above. The function int f(int x, int d) does action1(d) three times, then action2(d) if x is equal to 1. The function action2(int d) does action21(d) and then action22(d).
The directives are written under the form of #pragma C directives. Their effect are described in comments.

```
int f (int x,int d) {

  int i;
```

```
  /* starting point of the program part to be protected */
#pragma start

  /* definition of the possible scenarios (optional) */
#pragma race 0 x != 1
#pragma race 1 x == 1

  for(i=0; i<3; i++) {
    /* tells cfprotect that this loop has 3 turns */
#pragma loop 3
#pragma flag
    action1(d);
  }

  /* all the race must pass here, the flags are automatically numbered */
#pragma flag

  if (x == 1) {
    action2(d);
  }

  /* verification of the stack consistency, i.e. that the stack of flags
     is consistent regarding the control flow of the program */
#pragma verify
}

void action2(int d) {
  /* the race 1 must pass here and race 0 must not */
#pragma flag !0 1
#pragma verify
  action21(d);
  action22(d);
}
```

The source code fragment we obtain after processing is the following:

```
/*********************************************************************/
#ifdef CONTROL_FLOW_PROTECTION
#ifndef CONTROL_FLOW_PROTECTION_HEADER
#define CONTROL_FLOW_PROTECTION_HEADER
#include <string.h>
char __control_flow_stack[8];
char __cf_stack_index=0;
char __cf_race_flags=0;
char __fcPath0[] = {2, 3, 3, 3, 4, 0};
char __fcPath1[] = {2, 3, 3, 3, 4, 1, 0};
#define __cfSET_RACE(x) __cf_race_flags |= x
#define __cfRACE(x) __cf_race_flags & x
#define __cfNORACE !(__cf_race_flags)
#define __cfPUSH(x) __control_flow_stack[__cf_stack_index]=x, __cf_stack_index++
#define __cfRESET(x) __control_flow_stack[0]=x,__cf_stack_index=1,__cf_race_flags=0
#define __cfVERIFY(p) strcmp(__control_flow_stack,p)==0
#define __cfERROR printf("control flow error detected\n")
#endif
```

```
#endif
/*********************************************************************/

int f (int x, int d) {

  int i;

  /* starting point of the program part to be protected */
#ifdef CONTROL_FLOW_PROTECTION
__cfRESET(2);
#endif

  /* definition of the possible scenarios (optional) */
#ifdef CONTROL_FLOW_PROTECTION
if (x != 1) __cfSET_RACE(1);
#endif
#ifdef CONTROL_FLOW_PROTECTION
if (x == 1) __cfSET_RACE(2);
#endif

  for(i=0; i<3; i++) {
    /* tells cfprotect that this loop has 3 turns */
#pragma loop 3
#ifdef CONTROL_FLOW_PROTECTION
__cfPUSH(3);
#endif
    action1(d);
  }

  /* all the race must pass here, the flags are automatically numbered */
#ifdef CONTROL_FLOW_PROTECTION
__cfPUSH(4);
#endif

  if (x == 1) {
    action2(d);
  }

  /* verification of the stack consistency, i.e. that the stack of flags
     is consistent regarding the control flow of the program */
#ifdef CONTROL_FLOW_PROTECTION
  __control_flow_stack[__cf_stack_index]=0;
  if (!((__cfNORACE && (__cfVERIFY(__fcPath0) || __cfVERIFY(__fcPath1))) ||
            ((!(__cfRACE(1)) || (__cfVERIFY(__fcPath0))) &&
             (!(__cfRACE(2)) || (__cfVERIFY(__fcPath1))))))
    { __cfERROR; }
#endif
}

void action2(int d) {
  /* the race 1 must pass here and race 0 must not */
#ifdef CONTROL_FLOW_PROTECTION
__cfPUSH(1);
#endif
#ifdef CONTROL_FLOW_PROTECTION
```

```
 __control_flow_stack[__cf_stack_index]=0;
 if (!((__cfNORACE && (__cfVERIFY(__fcPath1))) ||
      ((!(__cfRACE(1)) &&
          (!(__cfRACE(2)) || (__cfVERIFY(__fcPath1)))))))
   { __cfERROR; }
#endif
  action21(d);
  action22(d);
}
```

The precompiler computes and introduces the data structures needed for the verification. These data structures are divided into two sorts : static and dynamic. The dynamic data structures are:

1. An array of byte intended to be use for the storage of the stack recording the flags successively passed during the execution;
2. A byte recording the length of the stack;
3. A byte recording the paths families which are active.

```
char __control_flow_stack[8];
char __cf_stack_index=0;
char __cf_race_flags=0;
```

The static data record the admissible forms of the stack, in our case:

```
char __fcPath0[] = {2, 3, 3, 3, 4, 0};
char __fcPath1[] = {2, 3, 3, 3, 4, 1, 0};
```

These data are computed by the preprocessor from the control flow graph of the program, also computed by the preprocessor (see Figure 1). The analysis of this graph shows which sequences of flags correspond to actual execution of the program from some starting directive to a control point.

The precompiler also defines some procedures, actually C macros, which are used when a directive is crossed.
These procedures are:

```
#define __cfSET_RACE(x) __cf_race_flags |= x
#define __cfRACE(x) __cf_race_flags & x
#define __cfNORACE !(__cf_race_flags)
#define __cfPUSH(x) __control_flow_stack[__cf_stack_index]=x, __cf_stack_index++
#define __cfRESET(x) __control_flow_stack[0]=x,__cf_stack_index=1,__cf_race_flags=0
#define __cfVERIFY(p) strcmp(__control_flow_stack,p)==0
#define __cfERROR printf("control flow error detected\n")
```

1. __cfSET_RACE(x) is used to make active the path family *x*.
2. __cfRACE(x) is a boolean which is true if and only if the family *x* is active.
3. __cfNORACE is a boolean which is true if and only if no path family is active.
4. __cfPUSH(x) push the flag *x* onto the stack. Let us note that the precompiler associate to each flag directive a unique token to be used in the stack.

5. __cfRESET(x) empties the stack.
6. __cfVERIFY(p) compare the current stack with an array *p* of flag representing a sequence of flag intended to be admissible. Such an array is a static data precomputed.
7. __cfERROR is the procedure to be invoked in the case of error discovery.

These functions are used to implement the error detection:

- A **start** directive is replaced by an initialization flag encoded by __cfRESET(x) where *x* is the token associated to the flag in question.
- A **flag** directive is replaced by a flag encoded by __cfPUSH(x) where *x* is the token associated to the flag in question.
- A **race n cond** is replaced by
  if (cond) __cfSET_RACE(x)
  where *x* encodes the path family *n*.
- A **verify** directive is replaced by a test, specific to the program point where the verify directive appears, which verifies that the current flag stack well corresponds to a correct execution, taking into account which path families have been activated. This verification relies on the static data precomputed by the precompiler.
- **loop n** directives are used by the precompiler only, they are deleted after the processing.

In the case of our example, the verification are done at the end of the function *f* and in the function *action2*. The array __fcPath0 and __fcPath1 define the execution paths which are admissible. There are two families, the first one is encoded by __fcPath0, the second one by __fcPath1. __fcPath0 corresponds to the execution which does not involve the call to *action2()*, __fcPath1 corresponds to the execution which involves *action2()* .

If the verification fails at some control point, this means that the current flag stack is not consistent regarding the control flow graph of the program. This means that some error occured in the execution.

4. Conclusion

The solution proposed here thus provides a generic protection against fault injection attacks. Moreover, only a very limited additional work is required from the developer to prevent his code from such attacks.

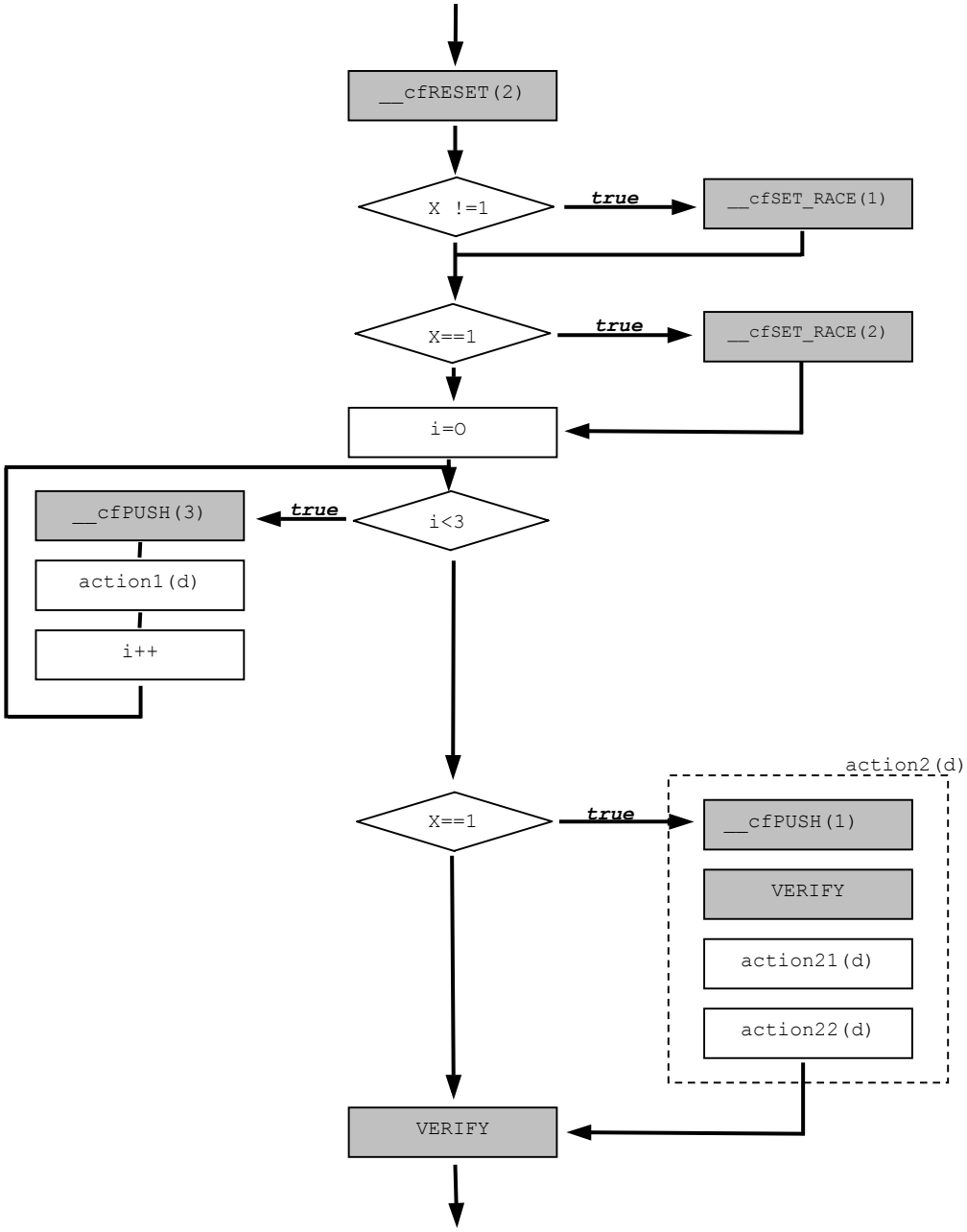

Figure 1. Control Flow Graph

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