Results

Aude Crohen Louis Dureuil Guillaume Petiot
Marie-Laure Potet

July 8, 2016

Abstract

This document explains and compares the results found with three of the tools of the partners, LAZART, CELTIC and EFS, on an example from the Secure Collection. [1]

1 Attacks found on VerifyPIN (version 2_HB+FTL_auth)

The considered `VerifyPIN` example is presented in listing 3. It implements the following countermeasures:

- boolean encoding (`0xAA` is true,`0x55` is false);
- loop iteration counter in the function `byteArrayCompare`.

The `g_authenticated` variable is initialized to `BOOL_FALSE` and the attack objective is for the variable to have the value `BOOL_TRUE` at the end of the execution, although the PIN is incorrect. A PTC modification up to `0x03` may be also considered as a successful attack as it may allow a brute-force attack on the PIN value.

The `byteArrayCompare` function is implemented as described in listing 2. This function uses the function `countermeasure` (described in listing 1) as a detection attack method. For our code test needs, this function only set a flag which can be checked on purpose.

```
#define BOOL_TRUE 0xAA
#define BOOL_FALSE 0x55
typedef unsigned char UBYTE;
typedef unsigned char BOOL;

void countermeasure()
{
  g_countermeasure = 1;
}
```

Listing 1: Defines and Implementation of `countermeasure`

We will now detail the vulnerabilities detected by some security tools on this example. We consider LAZART (Laser Attack Robustness), a framework

[1] http://sertif-projet.forge.imag.fr/pages/benchmark.html

```c
BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2)
{
  int i = 0;
  BOOL status = BOOL_FALSE;
  BOOL diff = BOOL_FALSE;
  for(i = 0; i < PIN_SIZE; i++) {
    if(a1[i] != a2[i]) {
      diff = BOOL_TRUE;
    }
  }
  if(i != PIN_SIZE) {
    countermeasure();
  }
  if (diff == BOOL_FALSE) {
    status = BOOL_TRUE;
  }
  else {
    status = BOOL_FALSE;
  }
  return status;
}
```

Listing 2: Implementation of `byteArrayCompare`

```c
void VerifyPIN()
{
  g_authenticated = BOOL_FALSE;

  if(g_ptc > 0) {
    if(byteArrayCompare(g_userPin, g_cardPin) == BOOL_TRUE) {
      g_ptc = 3;
      g_authenticated = BOOL_TRUE; // Authentication();
    }
    else {
      g_ptc--;
    }
  }
}
```

Listing 3: Implementation of `VerifyPIN`

for the evaluation of the robustness of software against multiple fault injections. Lazart relies on the symbolic test generator Klee and focuses on faults disrupting the control flow graph. We also consider the generic smartcard and fault injection simulator Celtic. Celtic generates each possible mutant of the code under analysis according to a fault model, simulates the execution of each mutant, and evaluates the vulnerability of the application. Finally, we consider Efs (Embedded Fault Simulator). Efs is a software framework allowing smart card developers to perform on-target fault injection simulations on a running application.

2 Results for Lazart

The following table indicates for each LLVM block what is the first correspond-ingline number in the source code, and the corresponding function ("bAC" for `byteArrayCompare`, "VPIN" for `VerifyPIN`).

entry	if.then	for.cond.i	for.body.i	if.then.i	if.end.i	for.end.i	if.then8.i
VPIN l.3	VPIN l.6	bAC l.6	bAC l.7	bAC l.8	bAC l.9	bAC l.11	bAC l.12

if.end9.i	if.then13.i	if.else.i	byteArrayCompare.exit	if.then5	if.else	if.end	if.end6
bAC l.14	bAC l.15	bAC l.18	VPIN l.6	VPIN l.7	VPIN l.12	VPIN l.14	VPIN l.15

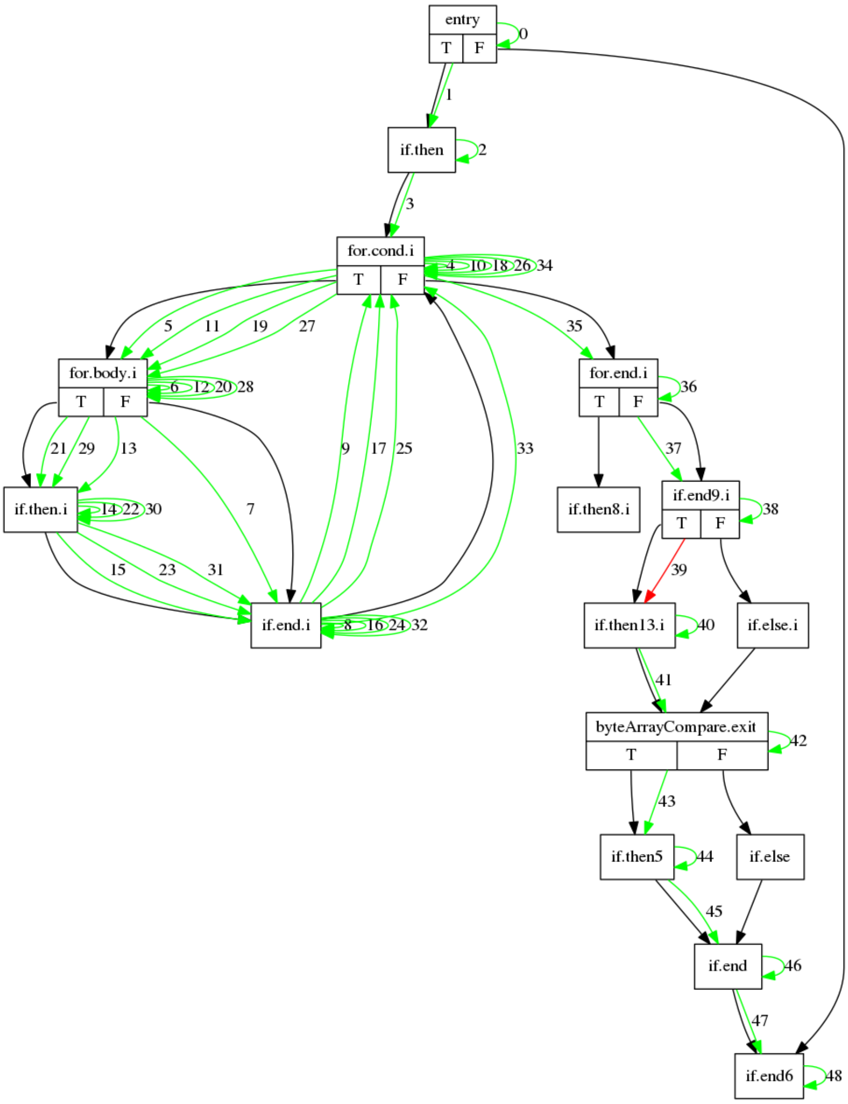
On this example, Lazart detected two 1-fault attacks, one 2-faults attacks, zero 3-faults attacks and one 4-faults attacks.

Example	#attacks for i faults			
	1	2	3	4
VerifyPIN_2_HB+FTL_auth	2	1	0	1

2.1 1-fault attacks

2.1.1 First attack: changing the returned value of byteArrayCompare

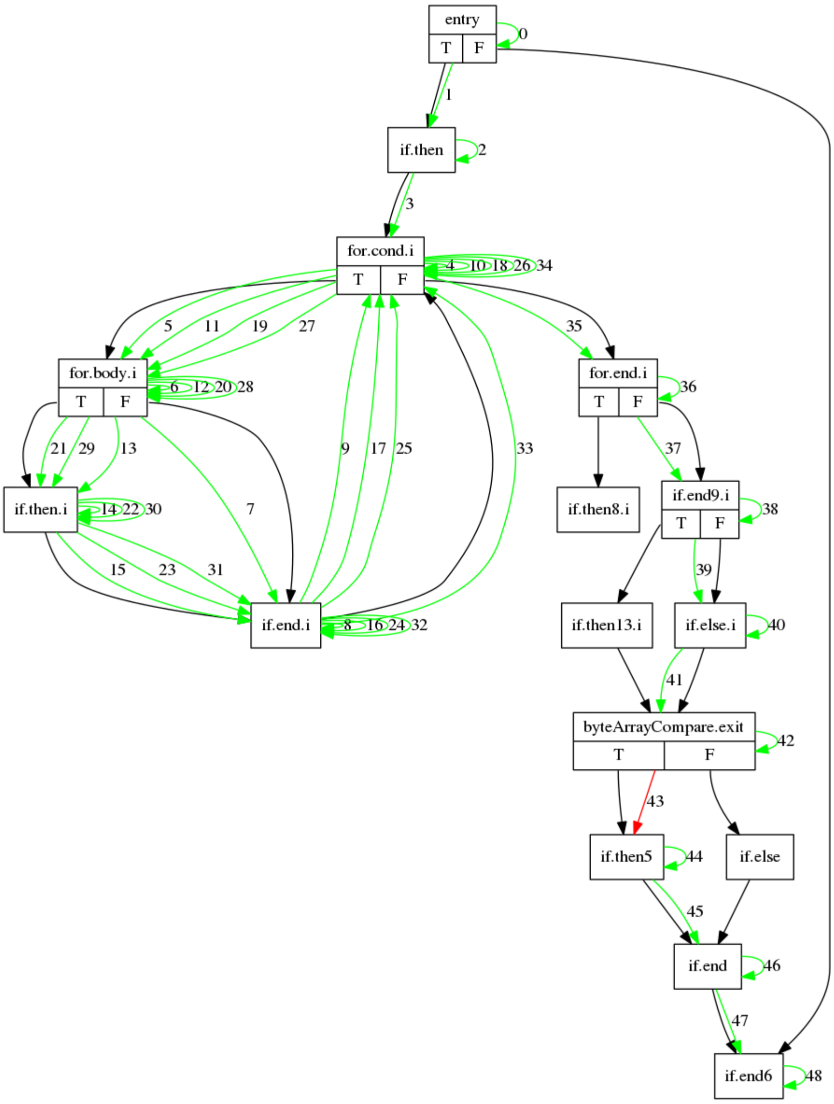
The first attack (the LLVM CFG is displayed bellow) consists in inverting the condition of the block if.end9.i to reach the block if.then13.i (branch then) instead of the block if.else.i (branch else). The block if.end9.i corresponds to the line 14 of the function `byteArrayCompare`: `if(diff == BOOL_FALSE)`. This fault leads to the execution of the block if.then13.i, corresponding to the line 15 of `byteArrayCompare`: `status = BOOL_TRUE;`. The program's execution then proceeds normally. This fault changes the value returned by `byteArrayCompare`: `BOOL_TRUE` is returned when the PIN comparison fails, so we can authenticate with an incorrect PIN.



CFG for 'verifyPIN_2' function

2.1.2 Second attack: ignore the returned value of byteArrayCompare

This second attack (the LLVM CFG is displayed bellow) consists in inverting the condition of the block byteArrayCompare.exit to reach the block if.then5 instead of the block if.else. The block byteArrayCompare.exit corresponds to the line 6 of `VerifyPIN`: `if(byteArrayCompare(g_userPin, g_cardPin)== BOOL_TRUE)`. This faults leads to the execution of the block if.then5, corresponding to the line 7 of `VerifyPIN` (`g_ptc = 3;`), instead of the block starting at line 12 of `VerifyPIN` (`g_ptc--`). The program's execution then proceeds normally. This fault allows to ignore the value returned by the function `byteArrayCompare` so we can authenticate with an incorrect PIN.



CFG for 'verifyPIN_2' function

2.2 2-faults attack

The 2-faults attack consists in inverting the loop condition before the first loop iteration, so that the loop is not executed, then inverting the test `i != PIN_SIZE` to not trigger the countermeasure. This attack succeeds because `diff = BOOL_FALSE` is executed before the loop.

2.3 4-faults attack

The 4-fault attack consists in injecting a fault during the PIN comparison: a fault for each invalid character (for a PIN of size 4).

3 Results for Celtic

Celtic is a dynamic simulator of binary smartcards, able to inject faults during the simulation. Celtic is being developed at the CEA-LETI, and currently supports a wide range of fault models and is able to simulate several machines.

3.1 Assembly listings

We provide the assembly listings for both `byteArrayCompare` and `VerifyPIN`:

```
080041a8 <byteArrayCompare>:
 80041a8 b570      push {r4, r5, r6, lr}
 80041aa 2200      movs r2, #0
 80041ac 2455      movs r4, #85 ; 0x55
 80041ae 5c83      ldrb r3, [r0, r2]
 80041b0 5c8d      ldrb r5, [r1, r2]
 80041b2 42ab      cmp r3, r5
 80041b4 d000      beq.n 80041b8 <byteArrayCompare+0x10>
 80041b6 24aa      movs r4, #170 ; 0xaa
 80041b8 1c52      adds r2, r2, #1
 80041ba 2a04      cmp r2, #4
 80041bc dbf7      blt.n 80041ae <byteArrayCompare+0x6>
 80041be d001      beq.n 80041c4 <byteArrayCompare+0x1c>
 80041c0 f000 f826 bl 8004210 <countermeasure>
 80041c4 2c55      cmp r4, #85 ; 0x55
 80041c6 d001      beq.n 80041cc <byteArrayCompare+0x24>
 80041c8 2055      movs r0, #85 ; 0x55
 80041ca bd70      pop {r4, r5, r6, pc}
 80041cc 20aa      movs r0, #170 ; 0xaa
 80041ce bd70      pop {r4, r5, r6, pc}
```

Listing 4: Assembly listing of `byteArrayCompare` for Celtic tests

```
080041d0 <verifyPIN_A>:
 80041d0 b570       push {r4, r5, r6, lr}
 80041d2 4d0b       ldr r5, [pc, #44] ; (8004200 <verifyPIN_A+0x30>)
 80041d4 4c0b       ldr r4, [pc, #44] ; (8004204 <verifyPIN_A+0x34>)
 80041d6 2055       movs r0, #85 ; 0x55
 80041d8 7028       strb r0, [r5, #0]
 80041da f994 0000  ldrsb.w r0, [r4]
 80041de 2800       cmp r0, #0
 80041e0 dd08       ble.n 80041f4 <verifyPIN_A+0x24>
 80041e2 4909       ldr r1, [pc, #36] ; (8004208 <verifyPIN_A+0x38>)
 80041e4 4809       ldr r0, [pc, #36] ; (800420c <verifyPIN_A+0x3c>)
 80041e6 f7ff ffdf  bl 80041a8 <byteArrayCompare>
 80041ea 28aa       cmp r0, #170 ; 0xaa
 80041ec d003       beq.n 80041f6 <verifyPIN_A+0x26>
 80041ee 7820       ldrb r0, [r4, #0]
 80041f0 1e40       subs r0, r0, #1
 80041f2 7020       strb r0, [r4, #0]
 80041f4 bd70       pop {r4, r5, r6, pc}
 80041f6 2003       movs r0, #3
 80041f8 7020       strb r0, [r4, #0]
 80041fa 20aa       movs r0, #170 ; 0xaa
 80041fc 7028       strb r0, [r5, #0]
 80041fe bd70       pop {r4, r5, r6, pc}
 8004200 20008014  .word 0x20008014
 8004204 20008015  .word 0x20008015
 8004208 2000801b  .word 0x2000801b
 800420c 20008017  .word 0x20008017
```

Listing 5: Assembly listing of `VerifyPIN` for CELTIC tests

3.2 Results

On the considered example, CELTIC detected 432 attacks, using the “‘exhaustive byte replacement” fault model, where 1 byte of the code is replaced with another value during the execution of the code, and 3 successful attacks using the NOP fault model, where 1 instruction of the code is replaced with a NOP instruction. The table below details the number of attacks found at each memory address in the exhaustive byte replacement fault model. We give a name to some selected attacks for later reference and proceed to explain them in the following paragraphs.

Address	Number of attacks	Name given
0x80041d6	1.0	
0x80041d9	1.0	
0x80041db	1.0	
0x80041e2	1.0	
0x80041e3	8.0	jump_in_auth
0x8004208	2.0	
0x80041e4	1.0	
0x80041e5	8.0	
0x800420c	1.0	
0x80041b4	5.0	
0x80041b8	5.0	
0x80041b9	6.0	
0x80041ba	1.0	
0x80041bc	4.0	
0x80041bd	29.0	skip_loop
0x80041b7	1.0	
0x80041b6	1.0	
0x80041be	2.0	
0x80041c4	1.0	
0x80041c5	63.0	skip_compare
0x80041c7	8.0	skip_branch
0x80041c8	1.0	
0x80041c9	6.0	
0x80041cb	119.0	skip_return
0x80041ea	1.0	
0x80041eb	4.0	
0x80041ed	8.0	
0x80041ef	1.0	
0x80041f3	6.0	
0x80041f5	136.0	skip_return

Table 1: Number of attacks per address

3.3 Attack `skip_compare`

This attack results in us forcing our way in the if part of the conditional statement line 14 of `byteArrayCompare`. Indeed, this if is implemented in the following way:

```
80041c4 2c55       cmp r4, #85 ; 0x55
80041c6 d001       beq.n 80041cc <byteArrayCompare+0x24>
80041c8 2055       movs r0, #85 ; 0x55
80041ca bd70       pop {r4, r5, r6, pc}
80041cc 20aa       movs r0, #170 ; 0xaa
80041ce bd70       pop {r4, r5, r6, pc}
```

By replacing the cmp instruction at address 0x41c4 with an instruction that sets the Z flag, the branch beq instruction at address 0x41c6 is taken, and the execution goes in the if branch of the conditional statement. For instance, this instruction can be replaced with an instruction to assign a register with another register, whose value is 0, as this sets the Z flag.

3.4 Attack `skip_branch`

This attack is similar to the attack `skip_compare` and targets the same conditional statement. However, instead of replacing a cmp instruction, it replaces the beq instruction with another branch instruction, which is taken (either because it is unconditional or because its condition is met), for instance bne. There are 8 such branch instructions.

3.5 Attack `skip_return`

This attack results in us forcing our way in the if part of the conditional statement at line 6 of `VerifyPIN` or at line 19 of `byteArrayCompare`. Indeed, the conditional statement at line 6 of `VerifyPIN` is implemented in the following way:

```
80041ea 28aa        cmp r0, #170 ; 0xaa
 80041ec d003        beq.n 80041f6 <verifyPIN_A+0x26>
 80041ee 7820        ldrb r0, [r4, #0]
 80041f0 1e40        subs r0, r0, #1
 80041f2 7020        strb r0, [r4, #0]
 80041f4 bd70        pop {r4, r5, r6, pc}
 80041f6 2003        movs r0, #3
 80041f8 7020        strb r0, [r4, #0]
 80041fa 20aa        movs r0, #170 ; 0xaa
 80041fc 7028        strb r0, [r5, #0]
 80041fe bd70        pop {r4, r5, r6, pc}
```

In this snippet, the beq instruction at address 0x41ec is normally not taken (because the result of the call to `byteArrayCompare` is `BOOL_FALSE`), therefore the else branch of the conditional is executed (from address 0x41ee to 41f4) and returns from the function with the pop instruction at 41f4. By replacing the pop instruction at address 0x41f4 with any instruction that is not a (taken) branch, we remove the implied return at the end of the else branch of the conditional statement, and we continue the execution in the if branch of the conditional, therefore executing both branches of the conditional, with the if branch overwriting the effect of the else branch.

Similarly, the conditional statement at line 19 of `byteArrayCompare` is implemented as follows:

```
 80041c4 2c55       cmp r4, #85 ; 0x55
 80041c6 d001       beq.n 80041cc <byteArrayCompare+0x24>
 80041c8 2055       movs r0, #85 ; 0x55
 80041ca bd70       pop {r4, r5, r6, pc}
 80041cc 20aa       movs r0, #170 ; 0xaa
 80041ce bd70       pop {r4, r5, r6, pc}
```

Again, the beq instruction at address 0x41c6 is not taken (because the `diff` variable equals `BOOL_TRUE`), and the else branch is executed. By replacing the pop instruction at address 0x41ca with any instruction that is not a (taken) branch, we remove the implied return at the end of the else branch of the conditional statement, and we continue in sequence with the if branch of the conditional, therefore executing both branches of the conditional, with the if branch overwriting the effect of the else branch.

3.6 Attack `skip_loop`

The loop iteration counter at line 11 of `byteArrayCompare` has been compiled in the following way:

1. Comparison of i wih the value 4:

```
80041ba 2a04      cmp r2, #4
80041bc dbf7      blt.n 80041ae <byteArrayCompare+0x6>
80041be d001      beq.n 80041c4 <byteArrayCompare+0x1c>
```

2. Conditional jump in the loop body if the comparison returns <

```
80041bc dbf7      blt.n 80041ae <byteArrayCompare+0x6>
```

3. Conditional jump to `countermeasure()` if the comparison does not return ==

```
80041be d001      beq.n 80041c4 <byteArrayCompare+0x1c>
80041c0 f000 f826 bl 8004210 <countermeasure> ; goto countermeasure
80041c4 2c55      ... ; continue executing
```

The attack replaces the first conditional jump with an arithmetic instruction that sets the Z flag (used in equality comparison). For instance:

```
41be 0af7 lsr r7, r6, #11
```

which shifts the content of the register r6 by 11 bits. If r6 $< 2^{11}$, then the result is 0 and the Z flag is set.

3.7 Attack `jump_in_auth`

During this attack, we replace the following instruction:

```
80041e2 4909        ldr r1, [pc, #36] ; (8004208 <verifyPIN_A+0x38>)
```

By stomping the upper byte of this instruction at address 0x41e3, it is possible to replace it with a branching instruction, e.g.:

```
80041e2 e009        b  80041f8 ;
```

(by replacing 0x48 with 0xe0)

Due to the layout of the code and the encoding of the original instruction, the offset happens to be #22, which jumps to address 0x41f8, i.e., the following instruction:

```
80041f8 7020        strb r0, [r4, #0]
```

This instruction is the middle of the assignment at line 6 of the `VerifyPIN` function, and will set `g_ptc` at whatever value is contained in r0. Then, the authentication code will be executed in sequence.

4 Results for EFS

4.1 Assembly listings

In order to perform the result analysis, we provide the generated assembly code of the targeted functions for the attack.

```
<byteArrayCompare>:
0x08080040 B570       PUSH {r4-r6,lr}
0x08080042 2200       MOVS r2,#0x00     ; i <- 0;
0x08080044 2455       MOVS r4,#0x55     ; r4 <- diff = BOOL_FALSE;
                                        ; 'for' loop beginning
0x08080046 5C83       LDRB r3,[r0,r2]   ; r3 <- a1[i]
0x08080048 5C8D       LDRB r5,[r1,r2]   ; r5 <- a2[i]
0x0808004A 42AB       CMP  r3,r5        ; a1[i] a2[i] comparison
0x0808004C D000       BEQ  0x0800245A   ; branch if a1[i] and a2[i] equal
0x0808004E 24AA       MOVS r4,#0xAA     ; diff = BOOL_TRUE
0x08080050 1C52       ADDS r2,r2,#1     ; i++ (loop)
0x08080052 2A04       CMP  r2,#0x04     ; i and PIN_SIZE comparison
0x08080054 DBF7       BLT  0x08002450   ; branch if i < PIN_SIZE (loop)
0x08080056 D001       BEQ  0x08002466   ; branch if i == PIN_SIZE (CM)
0x08080058 F000F806 BL.W countermeasure (0x08002474)
0x0808005C 2C55       CMP  r4,#0x55     ; diff = BOOL_FALSE ?
0x0808005E D001       BEQ  0x0800246E   ; branch if diff == BOOL_FALSE
                                        ; if diff != BOOL_FALSE
0x08080060 2055       MOVS r0,#0x55     ; return BOOL_FALSE
0x08080062 BD70       POP  {r4-r6,pc}   ; POP r0
                                        ; if diff == BOOL_FALSE
0x08080064 20AA       MOVS r0,#0xAA     ; return BOOL_TRUE
0x08080066 BD70       POP  {r4-r6,pc}   ; POP r0
```

Listing 6: Assembly listing of `byteArrayCompare` for EFS tests

```
<verifyPIN>:
0x08080000 B570       PUSH {r4-r6,lr}
0x08080002 4D0B       LDR  r5,[pc,#44]    ; r5 <- g_authenticated
0x08080004 4C0B       LDR  r4,[pc,#44]    ; r4 <- g_ptc
0x08080006 2055       MOVS r0,#0x55       ; r0 <- BOOL_FALSE
0x08080008 7028       STRB r0,[r5,#0x00] ; g_authenticated <- r0
0x0808000A 7820       LDRB r0,[r4,#0x00] ; r0 <- g_ptc
0x0808000C 2800       CMP  r0,#0x00       ; ptc and 0 comparison
0x0808000E D008       BEQ  0x08002342     ; branch if g_ptc == 0
0x08080010 4909       LDR  r1,[pc,#36]    ; r1 = g_userPin
0x08080012 480A       LDR  r0,[pc,#40]    ; r0 = g_cardPin
0x08080014 F000F814   BL.W byteArrayCompare (0x0800244A) ; return in r0
0x08080018 28AA       CMP  r0,#0xAA       ; result & BOOL_TRUE comparison
0x0808001A D003       BEQ  0x08002344     ; branch if result == BOOL_TRUE
                                          ; if r0 != BOOL_TRUE
0x0808001C 7820       LDRB r0,[r4,#0x00] ; r0 <- g_ptc
0x0808001E 1E40       SUBS r0,r0,#1       ; r0 <- r0 - 1
0x08080020 7020       STRB r0,[r4,#0x00] ; g_ptc <- r0 ( = g_ptc-1)
0x08080022 BD70       POP  {r4-r6,pc}     ; POP verifyPIN result
                                          ; if r0 == BOOL_TRUE
0x08080024 2003       MOVS r0,#0x03       ; r0 <- 3
0x08080026 7020       STRB r0,[r4,#0x00] ; g_ptc <- r0 (g_ptc <- 3)
0x08080028 20AA       MOVS r0,#0xAA       ; r0 <- 0xAA = BOOL_TRUE
0x0808002A 7028       STRB r0,[r5,#0x00] ; g_authenticated <- r0 = 0xAA
0x0808002C BD70       POP  {r4-r6,pc}     ; POP verifyPIN result
```

Listing 7: Assembly listing of `VerifyPIN` for EFS tests

4.2 Fault model: simple fault with PC modification

The fault model considered here consists in adding n bytes to the PC value (Program Counter). This modification skips n/m instructions, where m is the size of a single instruction. On the STM32 board, instructions are 2 or 4 bytes wide.

The fault model applied in this study is the $PC \leftarrow PC + 2$, so that instructions on two bytes are bypassed, but not the 4 bytes instructions.

The verifyPIN code requires 116 clock cycles to run. We apply the fault model to each of these cycles, so we have a total of 116 attack time slots. Depending on the number of cycles required to execute an instruction, some instructions have been attacked sereval times. This impacts the success rate. Depending on the pipeline's state at the moment of the attack, the fault may have different effects.

On this example, EFS detected the following behaviors, the details are presented in table 2. The lines highlighted in blue are considered as successful attacks.

Result type	**Occurs**	**Success rate**	**Description**
No attack detected (normal behaviour)	55 times	47.41 %	
Reboot of the STM32 board	22 times	18.96 %	
Counter-measure triggered	9 times	7.76 %	4.2.1
Successful authentication and PTC set to 3	5 times	4.31 %	4.2.2
Authentication flag set to 0x58	6 times	5.17 %	4.2.3
PTC set to `0x03`	17 times	14.65 %	4.2.4
PTC set to `0x54`	2 times	1.72 %	4.2.5

Table 2: Obtained result types

4.2.1 Counter-measure triggered

We have observed an activated counter-measure flag in several cases, as described in table 3.

Address	g_ptc	g_authenticated	g_countermeasure	**Description**
0x08080042	0x02	0x55	0x01	4.2.1 Case 1
0x08080052	0x03	0xAA	0x01	4.2.1 Case 2
0x08080052	0x02	0x55	0x01	4.2.1 Case 3
0x08080054	0x03	0xAA	0x01	4.2.1 Case 4
0x08080054	0x02	0x55	0x01	4.2.1 Case 5
0x08080056	0x02	0x55	0x01	4.2.1 Case 6

Table 3: Perturbations leading to activate the counter-measure

Case 1 The faults is performed on `MOVS r2,#0x00` in `byteArrayCompare`. It is the initialisation of `i` before the beginning of the for loop. As this initialization is skipped, `r2` remains to its last set value.

This case is an attack if `r2` was set to a value greater than `PIN_SIZE` before the call to `byteArrayCompare`.

Case 2 The fault is performed on `CMP r2,#0x04` in `byteArrayCompare`, just after the first byte comparison of the PIN. So this instruction is not executed. In this case, the loop is reduced to a single round and diff is not set to `BOOL_TRUE` because the first byte of both PINs are the same. The `byteArrayCompare` function returns `BOOL_TRUE`, so the PTC is set to 3 and `g_authenticated` is set to `BOOL_TRUE`.

As the `CMP` instruction is skipped, flags `N`, `Z`, `C` and `V` are not updated. They keep their previous value which has been set by the previous `ADD` instruction. `ADD` clears the `N`, `Z`, `C` and `V` flags if it performs a simple increment.

Thus, the next `BLT` at `0x08080054` is not taken (branch if N != V), as well as the next `BEQ` (branch if `Z` = 1) at $0x08080056$. The code falls through `BL.W countermeasure` at `0x08080058`, which sets the `g_countermeasure` flag.

Case 3 The fault is performed on `CMP r2,#0x04` in `byteArrayCompare`, just after the first byte comparison of the PIN. In this case, the loop is reduced to 2, 3 or 4 rounds depending on the timing of the attack. `diff` is set to `BOOL_TRUE` because the 2nd, 3rd and 4th bytes of the PIN are different from those of the reference PIN. The `byteArrayCompare` function returns `BOOL_FALSE`, so the PTC is set to 2 and `g_authenticated` to `BOOL_FALSE`.

As the `CMP` instruction is skipped, flags `N`, `Z`, `C` and `V` are not updated. They keep their previous value which has been set by the previous `ADD` instruction. `ADD` clears the `N`, `Z`, `C` and `V` flags if it performs a simple increment. Thus, the next `BLT` at $0x08080054$ is not taken (branch if `N` != `V`), as well as the next `BEQ` (branch if `Z` $= 1$) at `0x8002460`. The code falls through `BL.W countermeasure` at `0x08080058`, which sets the `g_countermeasure` flag.

Case 4 The fault is performed on `BLT 0x08080046` in `byteArrayCompare`, just after the first byte comparison of the PIN. In this case, the loop is reduced to a single round because the code doesn't jump back to `0x08080046`, even if `i` is different than `PIN_SIZE`. `diff` is not set to `BOOL_TRUE` because the first byte of both PINs are the same. The `byteArrayCompare` function returns `BOOL_TRUE`, so the PTC is set to 3 and `g_authenticated` is set to `BOOL_TRUE`.

The flags set by the `CMP` instruction are used to evaluate `BEQ 0x0808005C`, and as `i` is different than `PIN_SIZE`, the code jumps right to `0x08080058` and calls the `countermeasure` function.

Case 5 The fault is performed on `BLT 0x08080046` in `byteArrayCompare`, after the 2nd or 3rd byte comparison of the PIN, depending on the timing of the attack. In this case, the loop is reduced to 2 or 3 rounds because the code doesn't jump back to `0x08080046`, even if `i` is different than `PIN_SIZE`. `diff` is not set to `BOOL_TRUE` because the 2nd and 3rd bytes of the PIN are different from those of the reference PIN. The `byteArrayCompare` function returns `BOOL_FALSE`, so the PTC is set to 2 and `g_authenticated` to `BOOL_FALSE`.

The flags set by the `CMP` instruction are used to evaluate `BEQ 0x0808005C`, and as `i` is different than `PIN_SIZE`, the code jumps right to `0x08080058` and calls the `countermeasure` function.

Case 6 The fault is performed on `BEQ 0x0808005C` in `byteArrayCompare`, after the end of the loop. In this case, the conditional branch is skipped and the code continues at `0x08080058` and calls the `countermeasure` function.

4.2.2 Successful authentication and PTC set to `0x03`

This result has been observed in several cases, as described in table 4.

Address	g_ptc	g_authenticated	g_countermeasure	Description
0x0808005C	0x03	0xAA	00	4.2.2 Case 1
0x08080062	0x03	0xAA	00	4.2.2 Case 2
0x08080022	0x03	0xAA	00	4.2.2 Case 3

Table 4: Perturbations leading to a successful authentication and PTC=3

Case 1 The fault is performed on `CMP r4,#0x55` in `byteArrayCompare`, so this instruction is not executed.

So the flags N, Z, C and V are not updated, they keep their previous value which has been set by CMP r2,#0x04. As the condition if (i != PIN_SIZE) evaluates to true, the conditional branch BEQ 0x08080064 is taken and the execution continues at 0x08080064.

The next instructions executed are MOVS r0,#0xAA and POP. The byteArrayCompare function returns BOOL_TRUE.

Case 2 The fault is performed on POP {r4-r6,pc} at the end of the byteArrayCompare function.

The execution continues with the instructions MOVS r0,#0xAA and POP of the other branch of the byteArrayCompare function, which returns BOOL_TRUE.

Case 3 The fault is performed on POP {r4-r6,pc} at the end of the verifyPIN function.

The execution continues at address 0x08080024. In this case, the code takes the same branch as if byteArrayCompare returned BOOL_TRUE.

The PTC is consequently set to 3 and g_authenticated to BOOL_TRUE.

4.2.3 Authentication flag set to 0x58

The result is observed in a single case, as shown by table 5.

Address	g_ptc	g_authenticated	g_countermeasure
0x08080006	0x02	0x58	00

Table 5: Perturbations leading to authentication flag set to 0x58

The fault is performed on MOVS r0,#0x55 within verifyPIN function, so this instruction is not executed.

As a consequence, the r0 register is not updated and keeps its previous value, 0x58, which comes from the caller of verifyPIN). Then r0 is copied to g_authenticated which is then never modified.

4.2.4 PTC set to 0x03

We get this result in several cases, as shown by table 6.

Address	g_ptc	g_authenticated	g_countermeasure	Description
0x08080004	0x03	0x55	00	4.2.4 Case 1
0x08080040	0x03	0x55	00	4.2.4 Case 2
0x0808001E	0x03	0x55	00	4.2.4 Case 3
0x08080020	0x03	0x55	00	4.2.4 Case 4

Table 6: Perturbations leading to PTC set to 0x03

Case 1 The fault is performed on LDR r4,[pc,#44] inside the verifyPIN function, so this instruction is not executed.

Consequently, the global value `g_ptc` is not loaded into the `r4` register which keeps its previous value coming from the caller. During the processing of instruction `LDRB r0,[r4,#0x00]` at address `0x080232A`, the processor doesn't read the global value containing the PTC but another RAM element.

Given the obtained result, we can conclude that this RAM element is equal to 0, as it seems that the execution jumped directly to the end of the function (PTC is equal to 3 and `g_authenticated` to `BOOL_FALSE`)).

Case 2 The fault is performed on `PUSH {r4-r6,lr}` in the `byteArrayCompare` function, so this instruction is not executed.

As a consequence, the processor enters the `byteArrayCompare` function without pushing the `r4` to `r6` registers onto the stack, and without updating the `lr` register (Link Register, which is equal to `r14` on the STM32). At the end of `byteArrayCompare`, the final `POP` loads the `lr` with the `lr` value that was pushed at the beginning of `verifyPIN`. The global effect is that the code exits the `byteArrayCompare` and `verifyPIN` functions without executing the end of the `verifyPIN` function, so without decrementing the PTC.

Case 3 The fault is performed on `SUBS r0,r0,#1` in the `verifyPIN` function, so this instruction is not executed.

The PTC is not decremented.

Case 4 The fault is performed on `STRB r0,[r4,#0x00]` in the `verifyPIN` function, so this instruction is not executed.

Consequently, the program skips the instruction responsible for storing the local value of the PTC inside the global `g_ptc` value.

4.2.5 PTC set to `0x54`

This result is observed in a single case as described in table 7.

Address	g_ptc	g_authenticated	g_countermeasure
0x0808001C	0x54	0x55	00

Table 7: Perturbations leading to PTC set to `0x54`

The fault is performed on `LDRB r0,[r4,#0x00]` inside the `verifyPIN` function, so this instruction is not executed.

`g_ptc` is not loaded into the `r0` register which keeps its previous value. This value is `BOOL_FALSE = 0x55`, so `g_ptc` is updated with `r0 -1 = 0x54`.

4.3 Results comparison with other tools

4.3.1 Comparison with Lazart

Fault Model Matching

First attack of Lazart The first attack of LAZART described in 2.1.1 corresponds to the attack of the EFS described in section 4.2.2 Case 1. Indeed, skipping the `CMP r4,#0x55` instruction implies to invert the result of the comparison in the specific case of this implementation.

Second attack of Lazart The second attack of LAZART described in 2.1.2 fits with two attacks of the EFS described in section 4.2.2 Case 2 and 4.2.2 Case 3. These two EFS attacks are two ways to implement this LAZART attack.

Skipping the execution of `POP {r4-r6,pc}` implies to execute the other branch of the `byteArrayCompare` function end, and therefore force the change of the return value.

Avoiding the execution of the `POP {r4-r6,pc}` at the end of the `verifyPIN` function results in forcing the execution of the conditional statement in `verifyPIN` function which allows the authentication.

Other attacks of Lazart The multiple fault attacks of LAZART are not matched by single fault attacks of EFS with the fault model of $PC \leftarrow PC + 2$.

Matching by address In order to perform a matching by address with the HL tool LAZART, it is possible to perform a match between the CFG produced by LLVM and CFGs produced by IDA Pro.

We display below the flow chart produced by IDA Pro of the functions `VerifyPIN` in figure 1 and `byteArrayCompare` in figure 2, reinforced with the corresponding LLVM block name (highlighted in grey).

With this information, we are able to determine which assembly code line corresponds to which LLVM block.

CFGs of LLVM and the flow chart of IDA Pro don't perfectly match here: as `PIN_SIZE` is a fixed value in `byteArrayCompare` implementation, the compiler consider the for loop as a do while loop (as it will necessarily be done once). So the for.body.i is performed prior to for.cond.i block in figure 2 while the for.cond.i is performed prior to for.body.i block in LLVM CFG.

First attack of Lazart The first attack of LAZART described in 2.1.1 corresponds to attack the LLVM block if.then9.i, which corresponds to the addresses `0x0808005C` and `0x0808005E` of the assembly code attacked by the EFS (see listing 8).

```
0x0808005C 2C55      CMP  r4,#0x55     ; diff = BOOL_FALSE ?
0x0808005E D001      BEQ  0x0800246E   ; branch if diff == BOOL_FALSE
```

Listing 8: EFS assembly listing of `byteArrayCompare` corresponding to LLVM block if.then9.i

This attack founds one implementation with the EFS as described in 4.2.2 Case 1.

Second attack of Lazart The second attack of LAZART described in 2.1.2 corresponds to attack the LLVM block byteArrayCompare.exit, which corresponds to the addresses `0x08080018` and `0x0808001A` of the assembly code attacked by the EFS (see listing 9).

```
0x08080014 F000F814   BL.W byteArrayCompare (0x0800244A) ; return in r0
0x08080018 28AA       CMP  r0,#0xAA       ; result & BOOL_TRUE comparison
0x0808001A D003       BEQ  0x08002344     ; branch if result == BOOL_TRUE
```

Listing 9: Assembly listing of `VerifyPIN` for EFS tests

This attack founds no implementation with the EFS with the fault model of $PC \leftarrow PC + 2$.

Other attacks of Lazart The multiple fault attacks of LAZART are not matched by single fault attacks of EFS with the simple fault model of $PC \leftarrow PC + 2$.

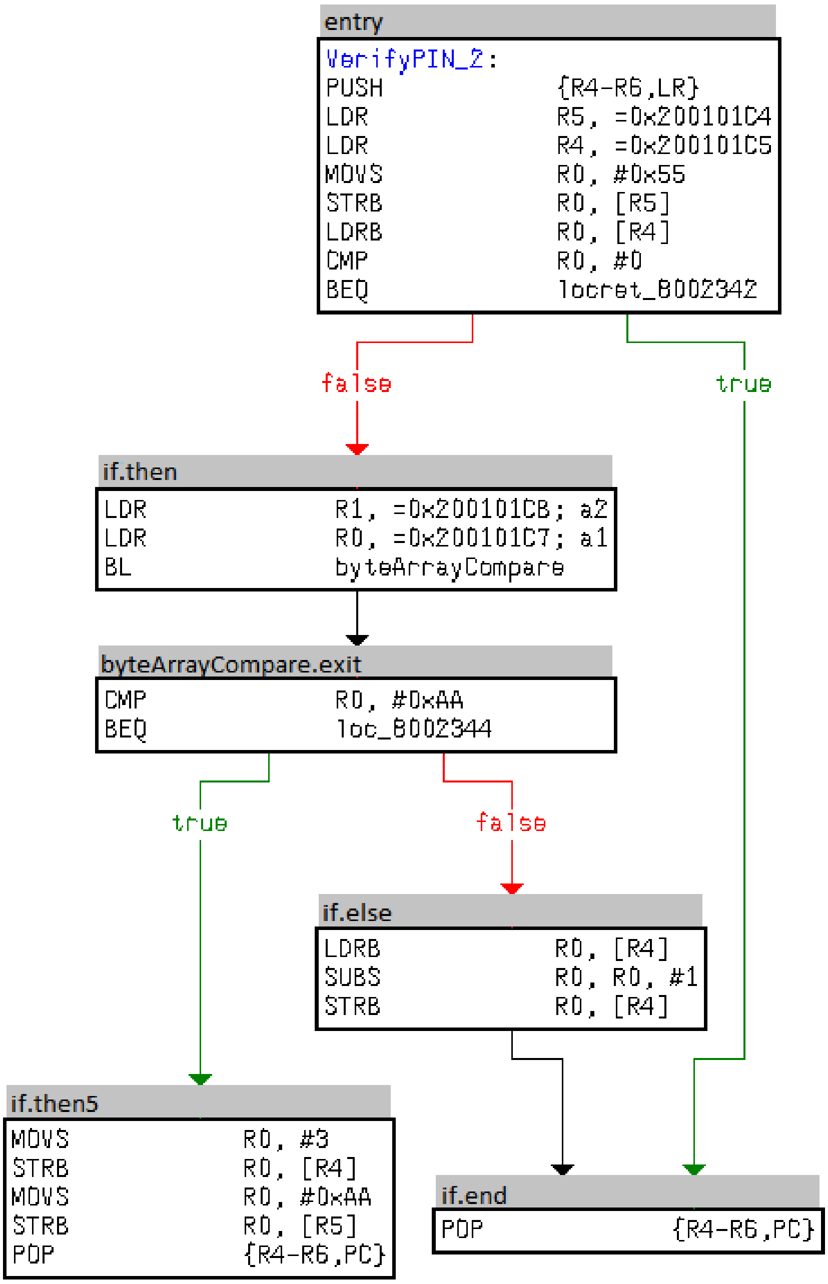


Figure 1: IDA flow chart of `VerifyPIN`

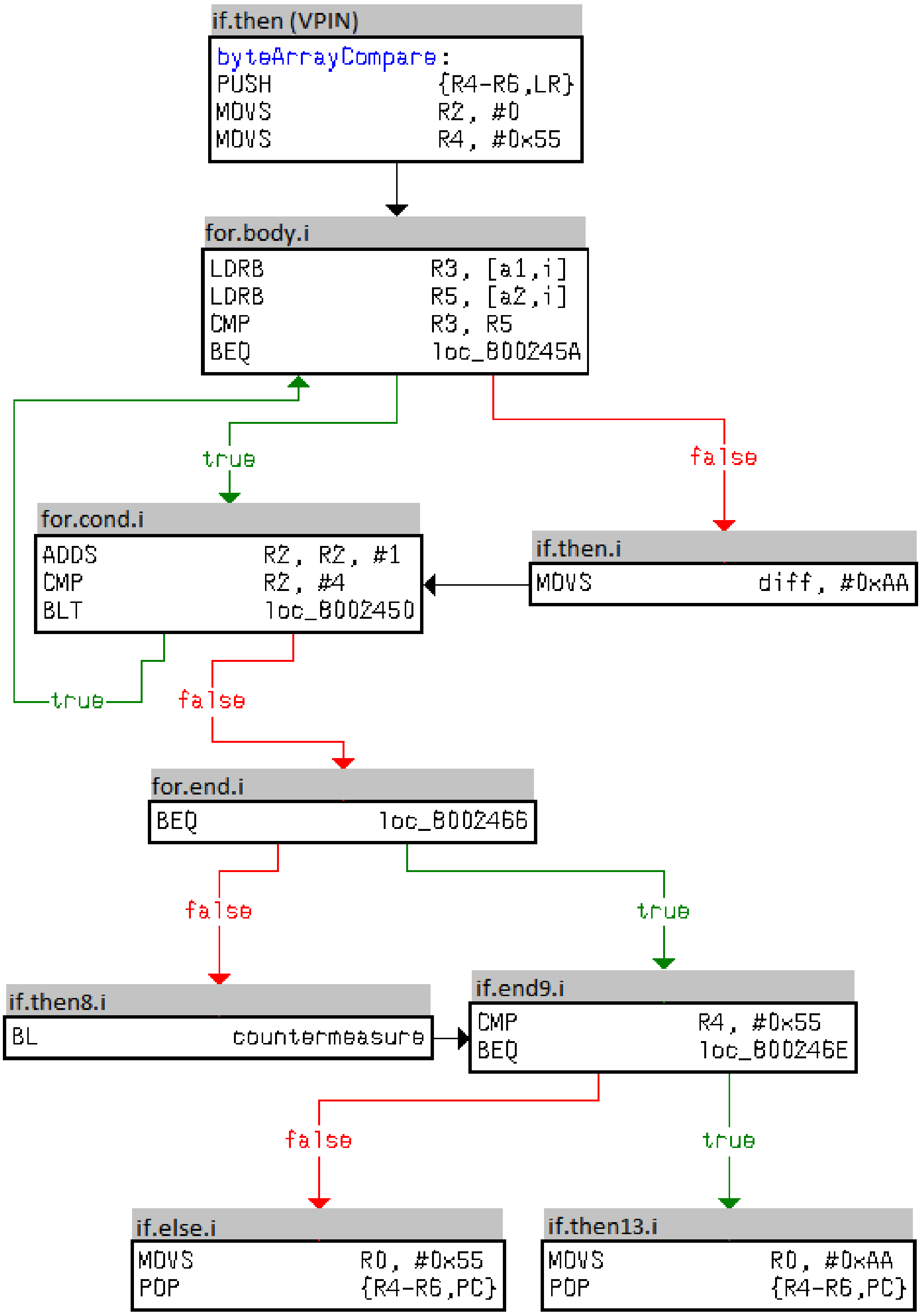


Figure 2: IDA flow chart of `byteArrayCompare`

4.3.2 Comparison with Celtic

We only discuss here the address matching as the 2 tools are LL-tools and the fault models are based on instruction modification.

Matching by address

skip_compare The attack `skip_compare` of CELTIC described in section 3.3 corresponds to the attack described in 4.2.2 case 1 of EFS.

Both attacks consequence is that the `Z` flag is in a state that makes the next `beq` instruction branch in the unexpected conditional statement part.

skip_return The attack `skip_return` of CELTIC described in section 3.5 corresponds to the attacks described in 4.2.2 case 2 and in 4.2.2 case 3 of EFS.

Both attacks avoid the execution of the `POP {r4-r6,pc}` at the end of the `verifyPIN` function. This attack results in forcing the execution of the conditional statement in `verifyPIN` function which allows the authentication.

The attack is the same on `byteArrayCompare` function.

Other results The other faults founds by CELTIC are not matched by single fault attacks of EFS with the simple fault model of $PC \leftarrow PC + 2$.