Countermeasures Optimization in Multiple Fault-Injection Context

Journée thématique sur les attaques par injection de fautes 2020

Etienne Boespflug, Cristian Ene
Laurent Mounier, Marie-Laure Potet

September 24, 2020

VERIMAG
`name.lastname@univ-grenoble-alpes.fr`

Supported by SECURIOT-2-AAP FUI 23 and by ANR-15-IDEX-02

Outline

An example: verifyPIN

```c
BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2, UBYTE size) {
    BOOL result = BOOL_TRUE;
    UBYTE i;
    for(i = 0; i < size; i++) {
        if(a1[i] != a2[i]) {
            result = BOOL_FALSE;
        }
    }

    if(i != size)
        killcard();

    return result;
}

BOOL verifyPIN() {
    if(g_ptc > 0)
        if(byteArrayCompare(g_userPin,
            g_cardPin, PIN_SIZE) == BOOL_TRUE) {
            // Authentication();
            g_authenticated = 1;
            g_ptc = 3;
            return BOOL_TRUE;
        } else {
            g_ptc--;
            return BOOL_FALSE;
        }
    return BOOL_FALSE;
}
```

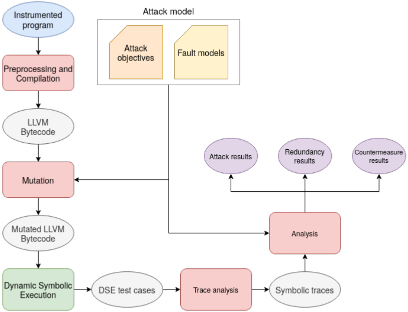
```c
/** Lazart analysis instrumentation main function. */

int main()
{
    verifyPIN();

    LAZART_ORACLE(g_killcard == 0 && g_authenticated == 1);

    return 0;
}
```

- **Functionality:** user authentication with secret PIN code
- **Attack objective:** authenticate with an incorrect user PIN

Lazart: source level analysis for multiple faults injection

⇒ **Lazart**[1] is a LLVM-level code robustness evaluation tool against multi-faults injection based on concolic execution (Klee)

- *Objectives*: Help developper/auditor to find attack paths and evaluate counter-measures.



[1] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: Asymbolic approach for evaluation the robustness of secured codes against control flow injections,"

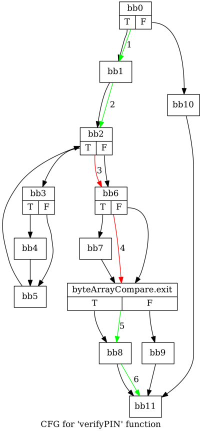
verifyPIN - Attack results

- Analysis parameters:
  - **Inputs**: Incorrect PIN
  - **Attack objective**: being authenticated with a false PIN
  - **Fault model**: up to N **test inversions**

Fault limit (N)	0	1	2	3	4
Attacks	0	1	1	0	1

- A successful 2-order attack (right) inverts the loop's condition i < size and the later check if(i != size) killcard();

Figure: The 2-faults attack (Test Inversion)



Definitions - Countermeasure

A **countermeasure** (in red) is a program transformation which:

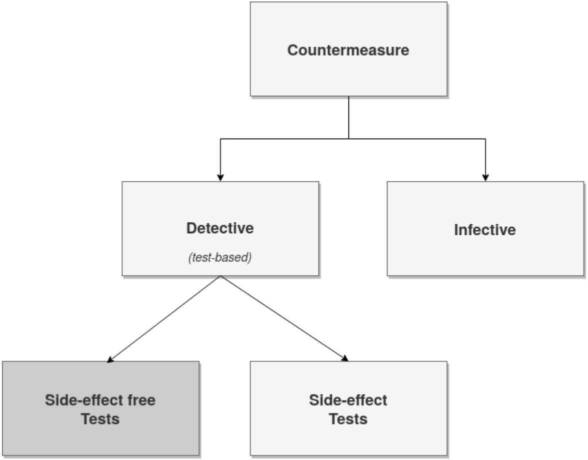
- preserves its observable behavior without faults
- increases security in presence of faults

```
BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2,
    UBYTE size, UBYTE size_dup) {
    int i;
    BOOL result = BOOL_TRUE;
    BOOL result_dup = BOOL_TRUE;

    for(i = 0; i < size; i++) {
        if(a1[i] != a2[i])
            result = BOOL_FALSE;
        if(a1[i] != a2[i])
            result_dup = BOOL_FALSE;

        if(result != result_dup)
            killcard();
    }

    if(i != size)
        killcard();
    if(i != size_dup)
        killcard();

    return result;
}
```

- Trade off between security and performance (speed, memory, size...)

Countermeasures

We consider **Detective** countermeasures, with side-effect free tests.



Definitions - CCPs and structures

We divide a **detective countermeasure** in two parts:

- Countermeasure Check Points (CCPs) are control point in the program corresponding to sanity checks about the current state
- The countermeasure's structures: shadow variables, parameters, additional computation etc.

```
BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2,
    UBYTE size, UBYTE size_dup) {
    int i;
    BOOL result = BOOL_TRUE;
    BOOL result_dup = BOOL_TRUE;

    for(i = 0; i < size; i++) {
        if(a1[i] != a2[i])
            result = BOOL_FALSE;
        if(a1[i] != a2[i])
            result_dup = BOOL_FALSE;

        if(result != result_dup)
            killcard();
    }

    if(i != size)
        killcard();
    if(i != size_dup)
        killcard();

    return result;
}
```

Objectives:

- determine if some CCPs could be removed
- remove related countermeasure's structures

Test Duplication - An example of Detective Countermeasure

- Designed for *Control Flow Integrity*
- Branching conditions are duplicated

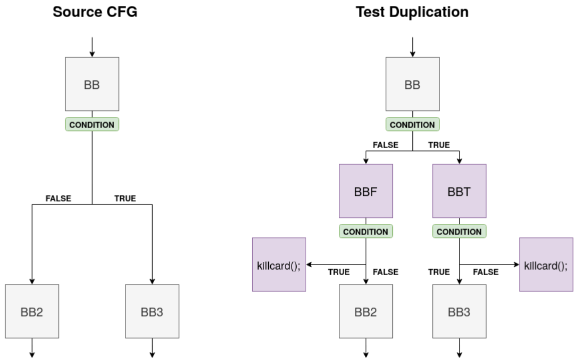


Table: Test duplication vs Non-protected version on `verifyPIN`

Program / Fault Count	0 faults	1 fault	2 faults	3 faults	4 faults
Non-protected	0	1	1	0	1
Test Duplication	0	0	1	0	1

Outline

1. Context
2. Countermeasure Optimization
3. Experimentation
4. Conclusion

Countermeasure Optimization Methodology

Goal: reduces the number of **CCPs** in a protected program without introducing new attacks

Methodology:

Input: a *program P* and *attack model* (fault model + attack objectives)
Output: a program P'

1. Generate the set of detected successful attack traces for P (successful regarding to attack objectives and blocked by at least a CCP)
2. Compute the **CCP Classification**
3. Choose a *removal strategy* and use **CCP selection algorithm** to find the optimal sets of CCP to be removed
4. Remove countermeasure's structures (variables, parameters...) related to the selected CCPs and generate the program P'

P' is the optimized protected version of P

Input Program - Test duplication on verifyPIN

Two **CCP** are generated for each conditional branching

```
BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2,
      UBYTE size) {
    BOOL result = BOOL_TRUE;
    int i;
    for(i = 0; i < size; i++) {  // CCP 2 & CCP 3
        if(a1[i] != a2[i]) { // CCP 4 & CCP 5
            result = BOOL_FALSE;
        }
    }

    if(i != size) // CCP 6 & CCP 7
        killcard(100); // CCP 100

    return result;
}
```

```
BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2,
      UBYTE size) {
    BOOL result = BOOL_TRUE;
    int i;
    BOOL c_1 = false;

    for(i = 0; BOOL c_1 = i < size; i++) {
        if(!c_1)
            killcard(); // CCP 2
        if(BOOL c_2 = a1[i] != a2[i]) {
            if(!c_2)
                killcard(); // CCP 4
            result = BOOL_FALSE;
        } else
            if(c_2)
                killcard(); // CCP 5
    }
    if(c_1)
        killcard(); // CCP 3

    if(BOOL c_3 = i != size) {
        if(!c_3)
            killcard(); // CCP 6
        killcard(); // CCP 100

    } else
        if(c_3)
          killcard(); // CCP 7

    return result;
}
```

Methodology - Step 1 & 2 - CCP classification on traces

The classification step considers the set A of attack traces (**step 1**) that are both:

- *successful*: satisfies the attack objectives
- *blocked*: at least one **CCP** is triggered

- We associate with each symbolic trace t a **repetition level** $L(t)$ as the number of **different** CCPs triggered

- Classify each **CCP** C_i according to its *Minimal Repetition Level*:
  $L_m(C_i) = min\{L(t) \mid t \in A \text{ and } C_i \text{ is triggered in } t\}$
  - Inactive: if $L_m(C_i) = \infty$ (never triggered)
  - Necessary: if $L_m(C_i) = 1$
  - Repetitive: otherwise, if $1 < L_m(C_i) < \infty$

- Inactive CCPs are removed

- If Repetitive CCPs are found, need to determine which of them should be removed

Methodology - Step 1 & 2 - Test Duplication results in 2 faults

VerifyPIN + Test Duplication:

- 86 traces in 2 faults with **Lazart**

Table: Test duplication CCP classification in 2 faults

CCP	0	1	2	3	4	5	6	7	8	9	100
Class	R	I	R	R	R	R	R	R	N	I	N

```
BOOL verifyPIN() {
    if(g_ptc > 0)       CCP 0 & CCP 1
        if(byteArrayCompare(g_userPin, g_cardPin
              , PIN_SIZE) == BOOL_TRUE) {   CCP
              8
            g_authenticated = 1;
            g_ptc = 3;
            return BOOL_TRUE;
        } else {  CCP 9
            g_ptc--;
            return BOOL_FALSE;
        }
    return BOOL_FALSE;
}
```

```
BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2,
     UBYTE size) {
    BOOL result = BOOL_TRUE;
    int i;
    for(i = 0; i < size; i++) {  CCP 2 & CCP 3
        if(a1[i] != a2[i]) {  CCP 4 & CCP 5
            result = BOOL_FALSE;
        }
    }

    if(i != size)  CCP 6 & CCP 7
        killcard(100);  CCP 100

    return result;
}
```

Methodology - Step 3 - CCP Selection algorithm

- **Objective**: compute the optimal sets of CCP to keep
- **Input**: The CCP classification, the attack traces and a weight function

Selection algorithm:

1. Let S a function associating a **weight** to a CCP (user-provided)
2. A set of CCP R_i is **valid** if for each trace t, at least one CCP in R_i is triggered
3. Lift the weight function S to sets of CCP R_i as $W_{R_i} = \sum_{CCP_i \in R_i} S(CCP_i)$
4. Find the sets with the **minimal weight**

Methodology - Step 4 - Structures Removal

- When the set of removed **CCPs** has been computed, unused *countermeasure's* structures can be removed
- Correspond to useless code elimination, can be done with a compiler or static analysis tools (Clang, GCC, Frama-C...)
- The program P' generated is equivalent to P if the initial set of traces is representative

Step 4 - Removed CCPs for verifyPIN (2 faults)

The removed and kept structures of *Test duplication* on verifyPIN for 2 faults

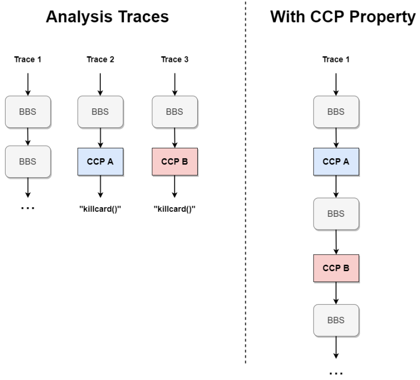
```
BOOL verifyPIN() {
    if(BOOL c_1 = g_ptc > 0) {
        if(!c_1)
            killcard();

        if(BOOL c_2 = byteArrayCompare(g_userPin,
              g_cardPin, PIN_SIZE) == BOOL_TRUE
              ) {
             if(!c_2)
                killcard();
            g_authenticated = 1;
            g_ptc = 3;
            return BOOL_TRUE;
        } else {
            if(c_2)
                killcard();
            g_ptc--;
            return BOOL_FALSE;
        }
    } else
        if(c_1)
            killcard();

    return BOOL_FALSE;
}
```

```
BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2,
     UBYTE size) {
    BOOL result = BOOL_TRUE;
    if i;
    BOOL c_1 = BOOL_FALSE;

    for(i = 0; BOOL c_1 = i < size; i++) {
        if(!c_1)
            killcard();
        if(BOOL c_2 = a1[i] != a2[i]) {
            if(!c_2)
                killcard();
            result = BOOL_FALSE;
        } else
            if(c_2)
                killcard();
    }
    if(c_1)
        killcard();

    if(BOOL c_3 = i != size) {
        if(!c_3)
            killcard();
        killcard();

    } else
        if(c_3)
           killcard();

    return result;
}
```

CCP's Properties

CCP's properties:

- The control flow must continue as if the CCP was not present
  $\rightarrow$ allows to don't stop path exploration when a CCP is triggered.
- Side-effect free (condition evaluation and killcard-like command)



Outline

Experimentations - Traces generation

Traces generation with **Lazart**:

- LLVM level
- Traces generated with *Dynamic Symbolic Execution*

Using *Test inversion* model

Experimentations - Test programs

Tested programs:

- *verifyPIN* (**VP**): smart-card PIN verification process
- *Firmware Updater* (**FU**): updates a firmware from remote source
- *Get Challenge* (**GC**): this program is an example of a nonce generation. The security property asserts that the nonce is updated with a randomly generated value
- *AES Cipher* (**AES**): implementation of AES encryption scheme. The isolated *AddRoundKey* (**AK**) step is also considered

Experimentations - Countermeasures

Three countermeasures experimented:

- *Test duplication* (**TD**): presented previously
- *SecSwift Control Flow* (**SSCF**)[2]: associates an unique identifier to each basic block and uses a xor-based mechanism to ensure that the correct branch has been taken
- **LBH**[3]: introduce step counters to protect against C-level instruction skips. Each counter verification is a `CCP`

[2] François de Ferrière. «A compiler approach to Cyber-Security». 2019.
[3] Lalande, J.F., Heydemann & al. 2014 «Software countermeasures for control flow integrity of smart card C codes». In European Symposium on Research in Computer Security

Experimentation results

Table: Percentage of removed CCP for each experimentation

Program	CCP	1 fault	2 faults	3 faults
VP + TD	11	72%	63%	18%
VP + SSCF	13	92%	76%	23%
VP + LBH	31	93%	93%	32%
FU + TD	14	0%	0%	0%
FU + SSCF	24	12%	12%	8%
GC1 + TD	39	37%	34%	34%
GC1 + SSCF	38	57%	28%	28%
AES RK + TD	2	50%	50%	0%
AES RK + SSCF	3	66%	33%	0%
AES C + TD	8	50%	50%	0%
AES C + SSCF	13	76%	61%	38%

Experimentation results - Playing with the Attack objectives

The **attack objective** strongly impacts the removed **CCPs**.

- ϕ_{auth}: being authenticated with a false PIN.
- ϕ_{ptc}: do not decrement the try counter with a false PIN.

Table: Removed CCP depending on attack objective (VP + TD)

Property	1 fault	2 faults	3 faults
ϕ_{auth}	83%	72%	18%
ϕ_{ptc}	72%	63%	9%
$\phi_{auth \wedge ptc}$	83%	72%	18%
$\phi_{auth \vee ptc}$	72%	63%	9%
ϕ_{true}	18%	9%	9%

Experimentation results - Time metrics

Table: Time metrics in 3-faults

Program	DSE (h)	Completed Paths	Traces	CCPO
VP + TD	0:00:03	7118	296	26ms
VP + SSCF	0:01:54	130 576	1005	89ms
VP + LL	0:38:24	1 173 312	37 347	371ms
FU + TD	0:39:16	935 409	43 328	736ms
FU + SSCF	1:04:39	1 490 767	91 713	4s
GC1 + TD	0:01:35	102 169	10 281	1s
GC1 + SSCF	0:31:45	1 048 354	58 367	2s
AES RK + TD	0:00:07	9 439	847	61ms
AES RK + SSCF	0:09:19	410 095	6 952	195ms
AES C + TD	1:17:25	1 064 007	38 810	575ms
AES C + SSCF	1:45:00	842 583	29 770	2s

Outline

Conclusion

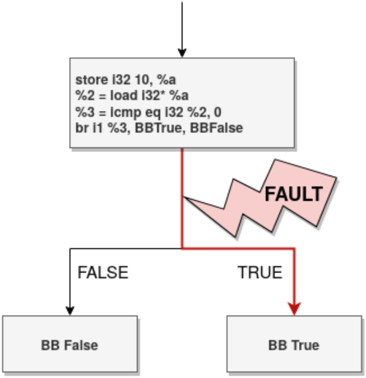
- A methodology to *optimize* program protected by `CCP-based` countermeasures
- Experimental results are very promising (up to 80% of `CCPs` removed)
- Only one DSE exploration $\rightarrow$ realistic analysis time for real world programs
- The methodology is generic regarding to the analysis level and trace generation method

Future work

- Assisted countermeasure placement
- Analysis of countermeasure outside of context
- Validate on further code examples (programs, libraries, fault models)

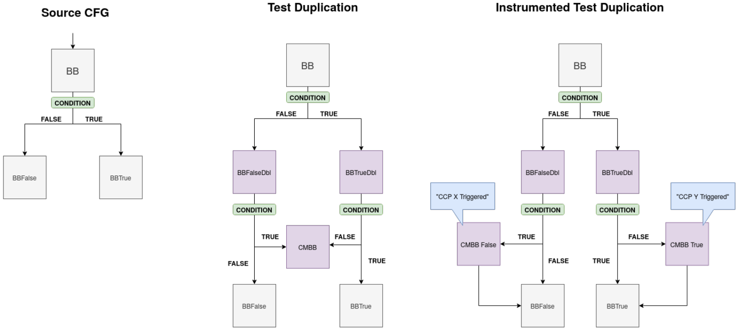
Test Inversion model

- Each conditional branching can be inverted (forcing the control flow to the wrong branch)
- Parameterized by a faults bound (here 3)
- Close to developer logic
- Maps to several low-level attacks:
  - NOP branching instruction
  - Faulting condition register
  - Jump to desired BB
  - ...

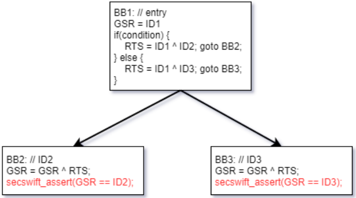


Test Duplication - CCP Instrumentation

The *Test Duplication* scheme is slightly instrumented to respect CCP properties.



SecSwift Control-Flow



SecSwift ControlFlow[4] is one of the 3 parts of SecSwift

- Designed for Control-Flow Integrity (CFI)
- Uses static signature for each basic block and propagate errors
- Each secswift_assert is a **CCP**

[4] François de Ferrière. «A compiler approach to Cyber-Security». 2019.

LBH's countermeasure[5]

```
#define INCR(cnt,val)  cnt = cnt + 1;
#define CHECK_INCR(cnt,val, cm_id) if(cnt != val) countermeasure(cm_id); \
    cnt = cnt + 1;
[...]


BOOL verifyPIN(unsigned short* CNT_0_VP_1)
{
    CHECK_INCR(*CNT_0_VP_1, CNT_INIT_VP + 0, 0LL)
    g_authenticated = 0;
    CHECK_INCR(*CNT_0_VP_1, CNT_INIT_VP + 1, 1LL)
    DECL_INIT(CNT_0_byteArrayCompare_CALLNB_1, CNT_INIT_BAC)
    CHECK_INCR(*CNT_0_VP_1, CNT_INIT_VP + 2, 2LL)
    BOOL res = byteArrayCompare(g_userPin, g_cardPin, PIN_SIZE, &CNT_0_byteArrayCompare_CALLNB_1);
[...]
```

- Insert *step-counters* for each C construct
- *Checking macros* (such as `CHECK_INCR`) are **CCPs**
- Analysis allows to know where the counter verification can be removed

[5] Lalande J.F. & al. 2014 «Software countermeasures for control flow integrity of smart card C codes». In European Symposium on Research in Computer Security