Countermeasures Optimization in Multiple Fault-Injection Context

FDTC 2020

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

September 12, 2020

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

Outline

1 Context

2 Countermeasure Optimization

3 Experimentation

4 Conclusion

An example: verifyPIN

```
1   BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2,
        UBYTE size) {
2       BOOL result = BOOL_TRUE;
3       UBYTE i;
4       for(i = 0; i < size; i++) {
5           if(a1[i] != a2[i]) {
6               result = BOOL_FALSE;
7           }
8       }
9
10      if(i != size)
11          killcard();
12
13      return result;
14  }
15
16  BOOL verifyPIN() {
17      if(g_ptc > 0)
18          if(byteArrayCompare(g_userPin,
19              g_cardPin, PIN_SIZE) == BOOL_TRUE) {
20              // Authentication();
21              g_authenticated = 1;
22              g_ptc = 3;
23              return BOOL_TRUE;
24          } else {
25              g_ptc--;
26              return BOOL_FALSE;
27          }
28      return BOOL_FALSE;
29  }
```

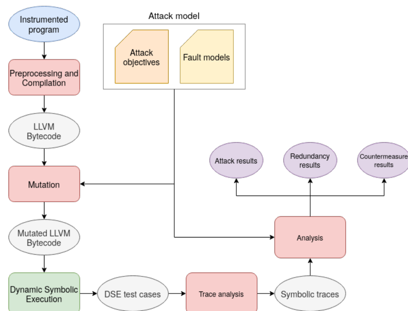
```
1   /** Lazart analysis instrumentation main
        function. */
2
3   int main()
4   {
5       verifyPIN();
6
7       LAZART_ORACLE(g_killcard == 0 && g_authenticated ==
            1);
8
9       return 0;
10  }
```

- **Functionality:** user authentication with secret PIN code
- **Attack goal:** 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,”

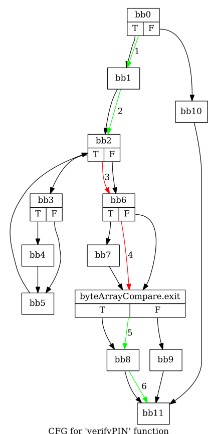
verifyPIN - Attack results

- Analysis parameters:
  - **Inputs**: Incorrect PIN, security property
  - **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

```
1   BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2,
2       UBYTE size, UBYTE size_dup) {
3       int i;
4       BOOL result = BOOL_TRUE;
5       BOOL result_dup = BOOL_TRUE;
6
7       for(i = 0; i < size; i++) {
8           if(a1[i] != a2[i])
9               result = BOOL_FALSE;
10          if(a1[i] != a2[i])
11              result_dup = BOOL_FALSE;
12
13          if(result != result_dup)
14              killcard();
15      }
16
17      if(i != size)
18          killcard();
19      if(i != size_dup)
20          killcard();
21
22      return result;
23  }
```

- Trade off between security and performance (speed, memory, size...)
- Multiple faults $\rightarrow$ the countermeasure itself can be attacked

Definitions - CCPs and structures

We divide a **CCP-based 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.

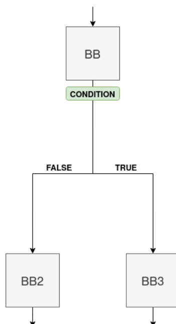
```
1  BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2,
2      UBYTE size, UBYTE size_dup) {
3      int i;
4      BOOL result = BOOL_TRUE;
5      BOOL result_dup = BOOL_TRUE;
6
7      for(i = 0; i < size; i++) {
8          if(a1[i] != a2[i])
9              result = BOOL_FALSE;
10         if(a1[i] != a2[i])
11             result_dup = BOOL_FALSE;
12
13         if(result != result_dup)
14             killcard();
15     }
16
17     if(i != size)
18         killcard();
19     if(i != size_dup)
20         killcard();
21
22     return result;
23 }
```

Objectives:

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

Test Duplication - An example of Countermeasure

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



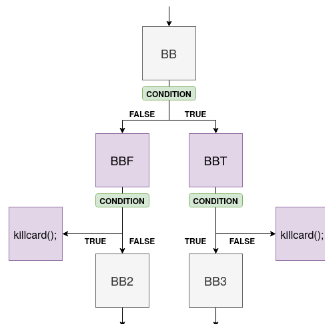


Table: Test duplication vs Non-protected version on `verifyPIN`

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 (+ attacker model)
Output: a program P'

1. Generate the set of (symbolic) detected attack traces for P
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. Use **structure removal rules** to 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

```
1    BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2,
         UBYTE size) {
2        BOOL result = BOOL_TRUE;
3        int i;
4        for(i = 0; i < size; i++) {  // CCP 2 & CCP
             3
5            if(a1[i] != a2[i]) { // CCP 4 & CCP 5
6                result = BOOL_FALSE;
7            }
8        }
9
10       if(i != size) // CCP 6 & CCP 7
11           killcard(100); // CCP 100
12
13       return result;
14   }
```

```
1    BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2,
         UBYTE size) {
2        BOOL result = BOOL_TRUE;
3        int i;
4        BOOL c_1 = false;
5
6        for(i = 0; BOOL c_1 = i < size; i++) {
7            if(!c_1)
8                killcard(); // CCP 2
9            if(BOOL c_2 = a1[i] != a2[i]) {
10               if(!c_2)
11                   killcard(); // CCP 4
12               result = BOOL_FALSE;
13           } else
14               if(c_2)
15                   killcard(); // CCP 5
16       }
17       if(c_1)
18           killcard(); // CCP 3
19
20       if(BOOL c_3 = i != size) {
21           if(!c_3)
22               killcard(); // CCP 6
23           killcard(); // CCP 100
24
25       } else
26           if(c_3)
27              killcard(); // CCP 7
28
29       return result;
30   }
```

Methodology - Step 1 & 2 - CCP classification on traces

The classification step considers the set A of attack traces (**step 2**) that are both:

- *successful*: violate the security property
- *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* (**MRL**):
  $L_m(C_i) = min\{L(t) \mid t \in A \textit{ and } C_i \textit{ is triggered in } t\}$
  - 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 symbolic 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

```
1   BOOL verifyPIN() {
2       if(g_ptc > 0)       CCP 0 & CCP 1
3           if(byteArrayCompare(g_userPin, g_cardPin
                , PIN_SIZE) == BOOL_TRUE) {   CCP
                8
4               g_authenticated = 1;
5               g_ptc = 3;
6               return BOOL_TRUE;
7           } else {  CCP 9
8               g_ptc--;
9               return BOOL_FALSE;
10          }
11      return BOOL_FALSE;
12  }
```

```
1   BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2,
           UBYTE size) {
2       BOOL result = BOOL_TRUE;
3       int i;
4       for(i = 0; i < size; i++) {  CCP 2 & CCP 3
5           if(a1[i] != a2[i]) {  CCP 4 & CCP 5
6               result = BOOL_FALSE;
7           }
8       }
9
10      if(i != size)  CCP 6 & CCP 7
11          killcard(100);  CCP 100
12
13      return result;
14  }
```

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 dead code elimination, can be done with a compiler or static analysis tools (Clang, GCC, Frama-C...)

Step 4 - Removed CCPs for verifyPIN (2 faults)

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

```
1   BOOL verifyPIN() {
2       if(BOOL c_1 = g_ptc > 0) {
3           if(!c_1)
4               killcard();
5
6           if(BOOL c_2 = byteArrayCompare(g_userPin,
                    g_cardPin, PIN_SIZE) == BOOL_TRUE
                    ) {
7               if(!c_2)
8                   killcard();
9               g_authenticated = 1;
10              g_ptc = 3;
11              return BOOL_TRUE;
12          } else {
13              if(c_2)
14                  killcard();
15              g_ptc--;
16              return BOOL_FALSE;
17          }
18      } else
19          if(c_1)
20              killcard();
21
22      return BOOL_FALSE;
23  }
```

```
1   BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2,
        UBYTE size) {
2       BOOL result = BOOL_TRUE;
3       if i;
4       BOOL c_1 = BOOL_FALSE;
5
6       for(i = 0; BOOL c_1 = i < size; i++) {
7           if(!c_1)
8               killcard();
9           if(BOOL c_2 = a1[i] != a2[i]) {
10              if(!c_2)
11                  killcard();
12              result = BOOL_FALSE;
13          } else
14              if(c_2)
15                  killcard();
16      }
17      if(c_1)
18          killcard();
19
20      if(BOOL c_3 = i != size) {
21          if(!c_3)
22              killcard();
23          killcard();
24
25      } else
26          if(c_3)
27              killcard();
28
29      return result;
30  }
```

Outline

1 Context

2 Countermeasure Optimization

3 Experimentation

4 Conclusion

Experimentations - Test programs

- Traces generation with **Lazart** (symbolic traces and LLVM level)
- 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 Security property

The **security property** 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 property (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

1 Context

2 Countermeasure Optimization

3 Experimentation

4 Conclusion

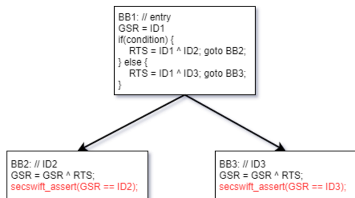
Conclusion

- 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

Future work

- Application for assembly code analysis (based on a low-level DSE-engine)
- Extends our tools for other kinds of fault models and countermeasures
- Validate on further code examples (programs, libraries)

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]

```
1   #define INCR(cnt,val)  cnt = cnt + 1;
2   #define CHECK_INCR(cnt,val, cm_id) if(cnt != val) countermeasure(cm_id); \
3       cnt = cnt + 1;
4   [...]
5
6
7   BOOL verifyPIN(unsigned short* CNT_0_VP_1)
8   {
9       CHECK_INCR(*CNT_0_VP_1, CNT_INIT_VP + 0, 0LL)
10      g_authenticated = 0;
11      CHECK_INCR(*CNT_0_VP_1, CNT_INIT_VP + 1, 1LL)
12      DECL_INIT(CNT_0_byteArrayCompare_CALLNB_1, CNT_INIT_BAC)
13      CHECK_INCR(*CNT_0_VP_1, CNT_INIT_VP + 2, 2LL)
14      BOOL res = byteArrayCompare(g_userPin, g_cardPin, PIN_SIZE, &CNT_0_byteArrayCompare_CALLNB_1);
15  [...]
```

- 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