Code Analysis and Security

Which tools ?
For which assurances ?

Marie-Laure Potet
in collaboration with Laurent Mounier

VERIMAG, Grenoble, France

16 juin 2017
Mon parcours

- Méthode B (98-2012) :
  - composition et preuve (invariant et raffinement)
  - développements sûrs de programmes embarqués
  - modélisation et raffinement pour la certification CC (analyse de conformité, méthodologie pour les exigences contrôle d’accès)

- Sécurité et analyse de code (2010-) :
  - Les outils dans la pratique (limitations et complémentarités)
  - Analyse de code binaire, vulnérabilités et exploitabilité
  - Robustesse à l’injection de fautes

⇒ Domaines d’application :


Outline

Code analysis and Security

Combining Static and Concolic execution for UaF detection

Fault injection and Code Analysis
Evaluation of the security of codes

Vulnerability analysis ≠ conformity. In general with regard to the State of the Art:

Attack models:
- Attacker quantification (threats and profits, expertises)
- Robustness w.r.t. classical attack scenarii
- Formal models of attackers

Results:
- Vulnerability must be evaluated in its context
  - produce a PoC or an exploit
  - Qualification/quantification of the vulnerability plausibility

Example: **AVA_VAN.5.4E**: The evaluator shall conduct penetration testing based on the identified potential vulnerabilities to determine that the TOE is resistant to attacks performed by an attacker possessing **High attack potential**.
Attack Potential

Elapsed time:

- The value is weighted in accordance with the elapsed time, such as "less than one day" (value: 0), "between one day and one week" (value: 1), "between one week and two weeks" (value: 2), and "between two weeks and one month" (value: 4).

Specialist expertise:

- The value is weighted in accordance with the level of knowledge, such as "layman" (value: 0), "proficient person" (value: 3), and "expert" (value: 6).

Knowledge of evaluation target:

- The value is weighted in accordance with the difficulty in obtaining the product information, such as "public information" (value: 0), "restricted information" (value: 3), and "sensitive information" (value: 7).

Window of opportunity:

- The value is weighted in accordance with the difficulty involved in accessing the product without the attack being noticed until the success of the attack, such as "unnecessary/unlimited access" (value: 0), "easy access" (value: 1), "moderate access" (value: 4), and "difficult access" (value: 10).

Equipment:

- The value is weighted in accordance with the difficulty in obtaining the equipment, such as "standard equipment" (value: 0), "specialized equipment" (value: 4), and "bespoke equipment" (value: 7).
Several targets

- correct implementations of security functionality (AC, crypto, authentification, ...)
- no classical exploitable flaws (CVE)
- specific properties as execution time, information flow, memory reset, access control, ...
- programming guides (banned functions, CERT, ...)
- robustness against malicious active platforms (faut injection)

⇒ Methods dedicated to targeted properties, possibly based on non-standard execution models.
Example 1: Vulnerability and Exploitability

```c
int main (int argc, char *argv[])
{
    char x=0 ;
    char t1[8] ;
    int i;
    for (i =0; i<= atoi ( argv [2]) ;i++)
        t1[i]= atoi ( argv [1]) ;
    if (x != 0) printf ("You win !\n") ;
    else printf ("You loose ...\n") ;
    return 0 ; }
```

example1 2 7, exemple1 2 11, example1 2 17:  
You loose ..., *** stack smashing detected ***, *** stack smashing detected ***

example1-without-canary(-fno-stack-protector) :  
You loose ..., You win ..., non termination
Exemple 2 : Hardened VerifyPIN

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

BOOL verifyPIN_5()
{
    g_authenticated = C_FALSE;
    if(g_ptc >= 0) { g_ptc --;
        if(byteArrayCompare(g_userPin,g_cardPin,PIN_SIZE)==C_TRUE)
            if(byteArrayCompare(g_cardPin,g_userPin,PIN_SIZE)==C_TRUE)
                {g_ptc = 3; g_authenticated = C_TRUE; return C_TRUE; }
            else countermeasure(); }
    return C_FALSE;
}
```

⇒ hardened boolean, fixed-time loop, first decremented ptc, double call
Level of analysis

After the compiler:
- optimisation, protection can disappear
- execution effect of undefined behaviours
- properties on the executed code (execution time, reset, taint)
- no available source code: library, obfuscated code, ...

Combined analyses:
- from high-level to binary level, including the compiling process
- countermeasures can be introduced and combined at each level
Symbolic reasoning (static analysis/symbolic execution)

Pros:
- correctness
- exhibit potential vulnerable statements

Cons:
- code not written to be analyzed (modularity, data structures, bitwise operations, ...)
- how to take into account libraries, interaction with the OS, ...
- difficulty to build reproducible PoC or exploit

Difficulties inherent to low level code:
- data structure recovery (variable, data, frame)
- CFG recovery (code/data, dynamic jump, call return, ...)  
- adapted memory models, uninitialized values (esp, ebp ...)

### Dynamic or Concolic Execution

#### Pros:
- black-box and white-box fuzzing
- adapted for all types of codes
- possibility of instrumentation (collect of information)

#### Cons:
- input generation (guiding strategy)
- completeness criteria (or quantitative criteria)
- overhead

#### Security application:
- security practice: fuzzing + crash + produce exploits
- combinaison with other analyses/instrumentations (taint, dependency, AddressSanitizer)
- AFL (fuzzer), SAGE, Klee, S2E, AngR, Triton (DSE)…
Code analysis for security at Vérimag

Software vulnerability detection and analysis

- Combination of static and dynamic analyses
- Combination of high-level and low level analyses
- Counter-measures analysis and attack models

2 main applications:

- Software vulnerability detection and analysis (ANR Binsec 2013-2017, Josselin Feist’s thesis)
- Robustness evaluation against fault injections (ASTRID Sertif 2014-2017, Louis Dureuil’s thesis)
Outline

Code analysis and Security

Combining Static and Concolic execution for UaF detection

Fault injection and Code Analysis
Use after Free example

```c
p = malloc(sizeof(int));
p_alias = p;                  // p and p_alias points
                                 // to the same addr
read(f, buf, 255);            // buf is tainted

if (strncmp(buf, "BAD\n", 4) == 0)
    { free(p);                // exit() is missing
    }
else{ ...
    }                            // some computation

if (strncmp(&buf[4], "is a uaf\n", 9) == 0)
    { p = malloc(sizeof(int));  }
else{ p = malloc(sizeof(int));
      p_alias = p;            }

*p = 42;                      // not a uaf
*p_alias = 43;                // uaf if 6 and 14 = true
```
Use after Free example

```
1  p = malloc(sizeof(int));
2  p_alias = p;               // p and p_alias points
3          // to the same addr
4  read(f,buf,255);          // buf is tainted
5
6  if (strncmp(buf,"BAD\n",4)==0)
7      { free(p);             // exit() is missing
8          }
9      else { ..
10         // some computation
11     }
12
13  if (strncmp(&buf[4],"is a uaf\n",9)==0)
14      { p = malloc(sizeof(int));  }
15  else{ p = malloc(sizeof(int));
16       p_alias = p;  }
17
18  *p = 42;                    // not a uaf
19  *p_alias = 43;             // uaf if 6 and 14 = true
```

- Difficult to detect (distant events, reasoning with heap, ..)
- No easy "pattern" (like for buffer overflow / string format)
- Lots of *Use-After-Free* in browsers and in other apps (proftpd CVE-2011-4130, privoxy CVE-2015-1031, openssh...)
Josselin Feist’s thesis Approach

Combining adequately static and DSE analyses

- **Static analysis** to extract potential vulnerable paths
- **Dynamic Symbolic Execution** to confirm Use-after-Free
- Application to real codes (binary pbs + scalability)
Static analyzer: GUEB

Static analysis features

- dangerous path discovery: pointer and aliases, inter-procedural
- Use-After-Free characterization: 2 heap models

Scalability features

- some (unsound) heuristics: loop unrolling and inlining, ...
- a very separated memory model taking into account uninitialized memory (ebp, esp, ...)
p = malloc(sizeof(int));

p_alias = p; // p and p_alias points to the same addr

read(f,buf,255); // buf is tainted

if(strncmp(buf,"BAD\n",4)==0){
    free(p);
    // exit() is missing
}
else{
    .. // some computation
}

if(strncmp(&buf[4],"is a uaf\n",9)==0){
    p = malloc(sizeof(int));
}
else{
    p = malloc(sizeof(int));
    p_alias = p;
}

*p = 42; // not a uaf

*p_alias = 43; // uaf if 6 and 14 = true
GUEB : Experimentations

Results

- Several experiments: UaF detection accuracy (no real existing benchmark), applicability to real applications (below), scalability (400 binaries).
- Several new *Use-After-Free* and referenced CVE found (CVE-2015-5221, CVE-2015-8871, CVE-2016-3177)

<table>
<thead>
<tr>
<th>name</th>
<th>#REIL ins</th>
<th>time</th>
<th>#UAF</th>
<th>#EP</th>
<th>max size reached</th>
</tr>
</thead>
<tbody>
<tr>
<td>alsabat</td>
<td>99 933</td>
<td>7s</td>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>gnome-nettool (-OO)</td>
<td>226 514</td>
<td>16s</td>
<td>4</td>
<td>56</td>
<td>0</td>
</tr>
<tr>
<td>gnome-nettool</td>
<td>260 882</td>
<td>17s</td>
<td>7</td>
<td>76</td>
<td>0</td>
</tr>
<tr>
<td>gifcolor *</td>
<td>233 303</td>
<td>21s</td>
<td>15</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>jasper *</td>
<td>2 154 927</td>
<td>4m23s</td>
<td>255</td>
<td>205</td>
<td>5</td>
</tr>
<tr>
<td>accel-ppd</td>
<td>3 907 862</td>
<td>5m5s</td>
<td>35</td>
<td>299</td>
<td>0</td>
</tr>
<tr>
<td>openjpeg *</td>
<td>2 170 081</td>
<td>6m10s</td>
<td>329</td>
<td>305</td>
<td>12</td>
</tr>
</tbody>
</table>
Dynamic Symbolic Execution

Binary → Static analysis → Interesting part of the binary → Symbolic execution → PoC

Inputs generation

```
(declare-const x int)
(declare-const y int)
(declare-const z int)
(push)
(cassert (= (* y 10)))
(cassert (= (* x (* 2 y)) 20))
(check-sat)
(pop); remove the two assertions
(pop)
(cassert (= (* (+ 3 x) y) 10))
(cassert (= (* x (* 2 y)) (* 2 y)) 21))
(check-sat)
```
DSE Features

Exploration strategy

- Guided by slices and distance metrics
- C/S policies (ISSTA 2016)

A condition to determine real UaF

- reinforcing the path predicate $\Pi$ with the set of constraints:
  \[ a_f = a_m \land a_u \in [a_m, a_m + \text{size}_{\text{alloc}} - 1] \]  
  (1)

- Data-dependency between $a_f$ and $a_m$ and between $a_u$ and $a_f$
  (no symbolic value for $a_m$ violating (1)).

An iterative process to discover a reachable initial state

- Obtain a model $m = (i, s)$ from $\Pi$, extract constraints $C$ on $s$
  from $P(i)$ and resolve $\Pi \land C$ and so on ...
Implementation in the BinSec platform + XP

**BINSEC/SE**

- based on the BinSec open source platform offering semantic binary level analyses: disassembly, simulation, symbolic execution, static analysis
- Our DSE: selection strategies, guiding modules and heuristics
- [http://binsec.gforge.inria.fr/tools](http://binsec.gforge.inria.fr/tools)

**Jasper (Jasper-JPEG-200 CVE-2015-5221)**

- 20 mins
- 9 test cases generated, one triggering the *Use-After-Free*
- PoC:
  - MIF component
Experimental validation of our approach

<table>
<thead>
<tr>
<th>Name</th>
<th>Time</th>
<th>MIF line</th>
<th>UAF found</th>
<th># Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DSE (in BINSEC/SE)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WS-Guided + LDH</td>
<td>20m</td>
<td>3min</td>
<td>Yes</td>
<td>9</td>
</tr>
<tr>
<td>WS-Guided</td>
<td>6h</td>
<td>3min</td>
<td>No</td>
<td>44</td>
</tr>
<tr>
<td>DFS (slice)</td>
<td>6h</td>
<td>3min</td>
<td>No</td>
<td>68</td>
</tr>
<tr>
<td>DFS</td>
<td>6h</td>
<td>3min</td>
<td>No</td>
<td>354</td>
</tr>
<tr>
<td>standard fuzzers (arbitrary seed)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AFL</strong></td>
<td>7h</td>
<td>&lt; 1min</td>
<td>No</td>
<td>174†</td>
</tr>
<tr>
<td>Radamsa</td>
<td>7h</td>
<td>&gt; 1h</td>
<td>No</td>
<td>∼ 1000000‡</td>
</tr>
<tr>
<td>standard fuzzers (MIF seed)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AFL</strong> (MIF input)</td>
<td>&lt; 1min</td>
<td>&lt; 1min</td>
<td>Yes</td>
<td>&lt; 10</td>
</tr>
<tr>
<td>Radamsa (MIF input)</td>
<td>&lt; 1min</td>
<td>&lt; 1min</td>
<td>Yes</td>
<td>&lt; 10</td>
</tr>
</tbody>
</table>

† AFL generates more input, 174 is the number of unique paths.
‡ For radamsa it is not trivial to count the number of unique path.

Table – JasPer evaluation
Combination is fruitful

⇒ A end-to-end approach for Use-After-Free detection, with scalability and binary concerns.

**Binary level static analysis**

- dangerous path discovery concerns
- subgraphs to explore with possible incompleteness
- scalability but some incorrect heuristics

**Dynamic Symbolic Execution**

- guided trace exploration: combining the initial CFG with slices
- C/S policies and reachable initial states
- libraries and scalability
Outline

Code analysis and Security

Combining Static and Concolic execution for UaF detection

Fault injection and Code Analysis
Context

⇒ Secure components (Hardware and Software) providing security services (authentication, cryptography) and secure storage of information.

■ Attractive targets for attackers
■ Can be physically attacked

⇒ Must be protected against high level attack potential (AVA-VAN.5)
Fault injection

- Perturbation attacks (EM or laser) \(\Rightarrow\) fault injection.
- Fault injection could modify the control and data flows.

```c
int verify(char buffer[4]) {
    int i;
    int authenticated = 1;
    // comparison loop
    for(i = 0; i < 4; i++) {
        if(buffer[i] != pin[i]) {
            authenticated = 0;
        }
    }
    // CM: redundant check
    if (i != 4) { // CM
        muteCard();
    }
    return authenticated;
}
```

```asm
MOV R0, #00h ; i = 0
MOV R3, #01h ; authenticated = 1
JMP WHILE
DO:
    MOV R2, [buffer+i]
    MOV A, [pin+i]
    CMP A, R2
    JE ITER ; buffer[i] == pin[i]?
    MOV R3, #00h ; authenticated = 0
ITER:
    INC R0 ; i++
    WHILE:
        MOV A, R0
        CMP A, #04h
        JB DO ; i < 4?
        MOV A, R0
        CMP A, #04h
        JNE muteCard ; i != 4?
        MOV A, R3
        RET
```
Fault injection

- Perturbation attacks (EM or laser) \(\implies\) fault injection.
- Fault injection could modify the control and data flows.

```c
int verify(char buffer[4]) {
    int i;
    int authenticated = 1;
    goto ATTACK;
    for (i = 0; i < 4; i++) {
        if (buffer[i] != pin[i]) {
            authenticated = 0;
        }
    }
ATTACK:
    if (i != 4) {
        // CM
        muteCard();
    }
    return authenticated;
}
```

```assembly
MOV R0, #00h ; i = 0
MOV R3, #01h ; authenticated = 1
JMP WHILE
DO:
MOV R2, [buffer+i]
MOV A, [pin+i]
CMP A, R2
JE ITER ; buffer[i] == pin[i]?
MOV R3, #00h ; authenticated = 0
ITER:
INC R0 ; i++
WHILE:
MOV A, R0
CMP A, #04h
NOP
MOV A, R0
CMP A, #04h
JNE muteCard ; i != 4?
MOV A, R3
RET
```
Fault injection

- Perturbation attacks (EM or laser) ⇒ fault injection.
- Fault injection could modify the control and data flows.

```c
int verify(char buffer[4]) {
    int i;
    int authenticated = 1;
    // comparison loop
    for(i = 4; i < 4; i++) {
        if(buffer[i] != pin[i]) {
            authenticated = 0;
        }
    }
    // CM: redundant check
    if (i != 4) { // CM
        muteCard();
    }
    return authenticated;
}
```

```assembly
| MOV R0, #04h ; i = 0 |
| MOV R3, #01h ; authenticated = 1 |
| JMP WHILE |
| DO: |
| MOV R2, [buffer+i] |
| MOV A, [pin+i] |
| CMP A, R2 |
| JE ITER ; buffer[i] == pin[i]?
| MOV R3, #00h ; authenticated = 0 |
| ITER: |
| INC R0 ; i++ |
| WHILE: |
| MOV A, R0 |
| CMP A, #04h |
| JB DO ; i < 4? |
| MOV A, R0 |
| CMP A, #04h |
| JNE muteCard ; i != 4? |
| MOV A, R3 |
| RET |
```
Attacker cannot choose the fault in code with precision

\[ f \approx (i = 124, \text{store}([0x540d], 0)) \]

Only chooses the parameters of the equipment

\[ p \approx (x = 12 \, \mu m, y = 24 \, \mu m, d = 3800 \, \text{ns}, w = 850 \, \text{ns}) \]
Assessing Robustness Against Fault Injection

Is an embedded application robust against fault injection?

- **Penetration Testing**: Physical perturbation attacks on the application under test to **inject faults**.
  - Look for successful attacks (compromising security).
  - Factors for Attack Potential Calculation

- **Code Analysis**: Detect vulnerabilities in the application with a code review.
  - Look for attack paths using a given fault model.
  - Originally manual process, now with automatic tools
  - Success rate $T = \frac{F_s}{F}$.

<table>
<thead>
<tr>
<th>Elapsed time</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; one hour</td>
<td>0</td>
</tr>
<tr>
<td>&lt; one day</td>
<td>3</td>
</tr>
<tr>
<td>&lt; one week</td>
<td>4</td>
</tr>
<tr>
<td>&lt; one month</td>
<td>6</td>
</tr>
<tr>
<td>&gt; one month</td>
<td>8</td>
</tr>
<tr>
<td>Not practical</td>
<td></td>
</tr>
</tbody>
</table>

**Figure – The 2 processes**
The Louis Dureuil’s end-to-end Approach

- Device
- Attacker Equipment
- Fault Model
- Fault Model Inference
- Fault Model
- Application
- Fault Injection Simulator
- Successful Attacks
- Attacker Model
- Rating
- Metrics for Robustness

Device Level
Applicative Level
Evaluation Level
Fault Detection Programs

- Purpose: Directly outputs the fault injected by perturbation attack during an execution.
- An example: EEPROM-RAM buffer copy
  - Executed from RAM
  - Sentinel RAM-RAM buffer copy

```assembly
; main_loop:
58:   ldrb    r5, [r0, #0] ; r5 <- @EEPROM
5a:   strb    r5, [r2, #0] ; r5 -> @IO_EEPROM
5c:   ldrb    r5, [r1, #0] ; r5 <- @RAM
5e:   strb    r5, [r3, #0] ; r5 -> @IO_RAM
60:   add.w   r0, r0, #1 ; @EEPROM += 1
64:   add.w   r1, r1, #1 ; @RAM += 1
68:   add.w   r2, r2, #1 ; @IO_EEPROM += 1
6c:   add.w   r3, r3, #1 ; @IO_RAM += 1
```
Fault Model Inference

⇒ Successively test several fault detection programs in sequence under perturbation attack

- Extract a PFM: $\mathcal{P}r(F = f | p)$
- An example:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Effect</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(d = d_0 + k\delta)$</td>
<td>16 bytes : $(a_d \rightarrow 0x00)$</td>
<td>16%</td>
</tr>
<tr>
<td>$(d = d_0 + k\delta)$</td>
<td>16 bytes : $(a_d \rightarrow 0xFF)$</td>
<td>0.3%</td>
</tr>
<tr>
<td>$(d = d_0 + k\delta)$</td>
<td>No fault</td>
<td>83.7%</td>
</tr>
</tbody>
</table>

- 4 models inferred on 3 cards
- Fault models at the device level
Metrics for code analysis?

The experimental metrics $\varphi$

$$\varphi = \frac{\text{# of experimentally successful attacks}}{\text{# of experimentally injected faults}}$$

Number of successful attacks $N$

$$N = \text{# of successful attacks in simulation}$$

$\Rightarrow$ sensibility to the surface paradox

Success rate $\tau$

$$\tau = \frac{\text{# of successful attacks in simulation}}{\text{# of injected faults in simulation}}$$

$\Rightarrow$ sensibility to the dilution paradox
Our proposal: Vulnerability rate $\mathcal{V}$

- Solves attack surface paradox: probability of success
- Solves dilution paradox: model attacker’s behavior
- Uses Probabilistic Fault Models from fault model inference

$$\mathcal{V} = \sum_{p \in \mathcal{P}} \sum_{f \in \mathcal{F}_S} \Pr(F = f \mid p) \cdot \Pr(p)$$

**Equipment CELTIC**

Attacker model:

$$\mathcal{V} = \frac{\sum_{p \in \mathcal{P}'} \sum_{f \in \mathcal{F}_S} \Pr(F = f \mid p)}{|\mathcal{P}'|}, \text{ with } \mathcal{P}' \subseteq \mathcal{P}$$

$\mathcal{P}'$: some knowledge of inefficient parameters to exclude from the attacks (zero-knowledge: $\mathcal{P}' = \mathcal{P}$, All-knowing $\mathcal{P} = \{p_{\text{max}}\}$).
Comparison with $\tau$ and $\varphi$

- $\varphi$ obtained from three-day physical experiments on cards A and B.
- $\tau$ computed with CELTIC from traditional fault model.
- $V_{real}$ computed with CELTIC from PFM

<table>
<thead>
<tr>
<th>Card</th>
<th>Command</th>
<th>$V_{real}$</th>
<th>$\tau$</th>
<th>$\varphi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>VerifyPIN</td>
<td>2.35 × 10^{-5}</td>
<td>3.2 × 10^{-2}</td>
<td>3.40 × 10^{-5}</td>
</tr>
<tr>
<td>A</td>
<td>SecureVerifyPIN</td>
<td>2.08 × 10^{-6}</td>
<td>8.5 × 10^{-5}</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>GetChallenge</td>
<td>2.01 × 10^{-5}</td>
<td>1.75 × 10^{-3}</td>
<td>2.94 × 10^{-5}</td>
</tr>
<tr>
<td>A</td>
<td>SecureGetChallenge</td>
<td>7.1 × 10^{-7}</td>
<td>2.74 × 10^{-6}</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>GetChallenge</td>
<td>1.1 × 10^{-3}</td>
<td>1.2 × 10^{-3}</td>
<td>1.4 × 10^{-3}</td>
</tr>
<tr>
<td>B</td>
<td>SecureGetChallenge</td>
<td>0</td>
<td>2.14 × 10^{-4}</td>
<td>0</td>
</tr>
</tbody>
</table>

- $V$ closer from $\varphi$ than $\tau$
- $V$ allows to predict $\varphi$ **without** physical experiments
  - Lower cost
  - Can be faster
  - “Theoretical” results
**Elapsed time comparison**

- \( t_V = (s_V)^{-1} \)
- Card A: \( s = 1.27 \text{ attack\cdot s}^{-1} \).
- Card B: \( s = 3.30 \text{ attack\cdot s}^{-1} \).

<table>
<thead>
<tr>
<th>Card</th>
<th>Command</th>
<th>Our approach ((s_V)^{-1}) (ET)</th>
<th>Traditional ((s_T)^{-1}) (ET)</th>
<th>Reference ((s_\varphi)^{-1}) (ET)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>VerifyPIN</td>
<td>8h (3)</td>
<td>24s (0)</td>
<td>6h (3)</td>
</tr>
<tr>
<td>A</td>
<td>SecureVerifyPIN</td>
<td>1w (4)</td>
<td>2.5h (3)</td>
<td>&gt; 3d ((\geq 4))</td>
</tr>
<tr>
<td>A</td>
<td>GetChallenge</td>
<td>10h (3)</td>
<td>7min (0)</td>
<td>7.4h (3)</td>
</tr>
<tr>
<td>A</td>
<td>SecureGetChallenge</td>
<td>2w (6)</td>
<td>3.5d (4)</td>
<td>&gt; 3d ((\geq 4))</td>
</tr>
<tr>
<td>B</td>
<td>GetChallenge</td>
<td>5min (0)</td>
<td>5min (0)</td>
<td>5min (0)</td>
</tr>
<tr>
<td>B</td>
<td>SecureGetChallenge</td>
<td>not practical (_)</td>
<td>20min (3)</td>
<td>&gt; 3d ((\geq 4))</td>
</tr>
</tbody>
</table>

**Elapsed time**

<table>
<thead>
<tr>
<th>Elapsed time</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; one hour</td>
<td>0</td>
</tr>
<tr>
<td>&lt; one day</td>
<td>3</td>
</tr>
<tr>
<td>&lt; one week</td>
<td>4</td>
</tr>
<tr>
<td>&lt; one month</td>
<td>6</td>
</tr>
<tr>
<td>&gt; one month</td>
<td>8</td>
</tr>
<tr>
<td>Not practical</td>
<td>_</td>
</tr>
</tbody>
</table>
FISSC : our secure collection

Content :

- A collection of (extensible) examples with success oracles
- Result of the Sertif project: sertif-projet.forge.imag.fr

<table>
<thead>
<tr>
<th>Example</th>
<th>Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>VerifyPIN</td>
<td>( g_authenticated == 1 )</td>
</tr>
<tr>
<td>VerifyPIN</td>
<td>( g_ptc &gt;= 3 )</td>
</tr>
<tr>
<td>KeyCopy</td>
<td>(! ) \text{equal}(key, keyCpy)</td>
</tr>
<tr>
<td>GetChallenge</td>
<td>( ) \text{equal}(challenge, prevChallenge)</td>
</tr>
<tr>
<td>CRT-RSA</td>
<td>( (g_cp == \text{pow}(m,dp) % p &amp;&amp; g_cq != \text{pow}(m,dq) % q))</td>
</tr>
<tr>
<td></td>
<td>(</td>
</tr>
</tbody>
</table>

Countermeasures: hardened booleans, virtual stack, double arguments, step counter, loop counter, data redundancy, double calls, double tests, control flow integrity

Programming Features: Explicit call, Fixed Time Loops, inlining
Content

- Normalized and modular examples
- C sources and Thumb-2 assembly listings
- High-level attack scenarios on CFG produced by Lazart

Example | 1-fault | 2-fault
---|---|---
VerifyPIN | 2 | 0
+fixed time loops | 2 | 1
+FTL +inlining | 2 | 1
+FTL +INL +loop counter | 2 | 0
+FTL +double calls | 0 | 4
+FTL +INL +double tests | 0 | 3
+FTL +INL +DT +step counter | 0 | 2
+control flow integrity | 0 | 2
+FTL +INL +DT +SC +CFI | 0 | 1
Next steps

Code analysis
- reasoning with attacker model
- counter-measure analysis (accuracy and efficiency)

Adaptable tools suit
- from high level to low level analyses
- adaptable certification process (TEE, IoT, ...)

Vérimag People involved

**Permanent staff** : Cristian Ene, Laurent Mounier, Marie-Laure Potet, Jean-Louis Roch

**PhD students** : Louis Dureuil, Josselin Feist, Franck de Goer, Maxime Puys, Benjamin Farinier, Jonathan Salwan

**Postdoc and visitors** : Thanh-Dinh Ta, Guillaume Petiot, Gustavo Grieco (Univ. of Rosario, Argentina), Sanjay Rawat (Univ. of Amsterdam, The Netherlands)

Some Security activities at Grenoble