Cybersecurity Institute
Univ. Grenoble Alpes

UGA Université Grenoble Alpes

KLEE-Assisted Code Robustness Evaluation Against Fault Injections

Klee Workshop 2021

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

June 11, 2021

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

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

Outline

1 Context

2 Lazart: interacting with Klee

3 Conclusion

Physical attacks

Side-channel attacks

- Power Analysis
- Electromagnetic Analysis
- Timing attacks
- Cache attacks

Goal: find secret information about the program

Fault-injection attacks

- Laser
- Electromagnetic pulses
- Power & clock glitches
- Expensive tooling

Goal: modify program behavior/state

Fault injection attacks effects

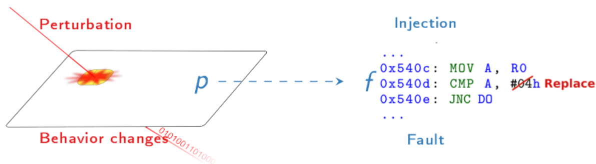
Local Fault effects:

- Instruction replacement
- Register/RAM mutation
- Out of ISA effects
- ...

Program effects:

- Control Flow Graph / Call Graph modification
- Program execution modification
- Alteration of program's invariants
- ...

$\Rightarrow$ Standard analysis cannot be trivially applied



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) // Countermeasure
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       klee_assume(_not_equal(g_userPin, g_cardPin));
6       verifyPIN();
7
8       klee_assume(g_killcard == 0 && g_authenticated == 1);
9
10      return 0;
11  }
```

Functionality: user authentication with secret PIN code

Attack objective: authenticate with an incorrect user PIN

verifyPIN - Attack results

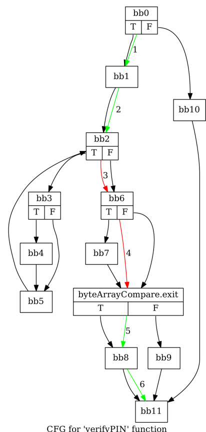
Figure: The 2-faults attack (Test Inversion)

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	5	10	11

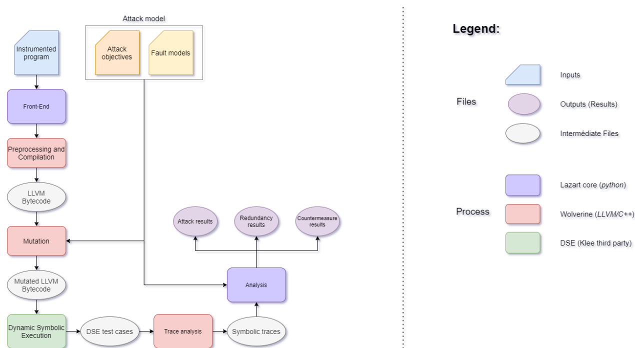
A successful 2-order attack (right) inverts the loop's condition i < size and the later check if(i != size) killcard();



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 countermeasures.



[1] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections", ICST14

Generation of the High-order mutant

- **symbolic boolean variables** determine if a fault should be injected or not (with a specified maximal fault limit)
- a fault injection is simulated by a **mutation function** (the injected value corresponds to a symbolic variable)
- can manage different **fault models**: test inversion, data fault injection
- each fault model corresponds to the transformation of LLVM instructions of some specific type
  - **Test inversion**: *branch instructions*
  - **Data mutation**: *load instructions*

$\hookrightarrow$ High-order mutant, including all possible **faults** *injection points*, which is provided to Klee

Injecting a fault - mutation function

```
1   int lz_mutation_data_i32(int original, int fault_limit, int (*predicate)(int),
         const char* ip_name)
2   {
3       int inject, value;
4       klee_make_symbolic(&inject, sizeof(inject), "inject");
5       if (inject && fault < fault_limit){
6           klee_make_symbolic(&value, sizeof(value), "value");
7           klee_assume(predicate(value));
8           fault++;
9           printf("[FAULT] at %s from %d to %d\n", ip_name, original,
                 klee_get_value_i32(value));
10          return value;
11      }
12      return original;
13  }
```

↪ check that a new fault injection should be done

↪ in the positive case, the injected value should be used instead of the original one

↪ possibly, the new injected value should satisfy some constraints

↪ each injected fault is logged for later processing

LLVM mutation example

Original bytecode

```
1    [...]
2    %6 = load i32* %tmp, align 4
3    %7 = sext i32 %6 to i64
4    %8 = call i32 @memcmp(i8* %4, i8* %5, i64 %7) #5
5    %9 = icmp ne i32 %8, 0
6    br i1 %9, label %bb26, label %bb25
```

- a fault injection is simulated by a call to the mutation function
- data injection on variable %tmp
- test inversion : injection on first operand of the instruction br

Mutated bytecode

```
1    [...]
2    %6 = alloca i32
3    %7 = load i32* %tmp, align 4
4    %funCall4 = call i32 @lz_mutation_data_i32(i32
          %7, i32 4, i32 (i32)* @P_tmp, i8*
          getelementptr inbounds ([9 x i8]* @"bb24:
          tmp", i32 0, i32 0)) #4
5    store i32 %funCall4, i32* %6, align 4
6    %8 = load i32* %6, align 4
7    %9 = sext i32 %8 to i64
8    %10 = call i32 @memcmp(i8* %4, i8* %5, i64 %9)
          #5
9    %11 = icmp ne i32 %10, 0
10   %12 = sext i1 %11 to i32
11   %funCall5 = call i32 @lz_mutation_test_inversion
          (i32 %12, i32 4, i8* getelementptr
          inbounds ([5 x i8]* @memcmps_s2, i32 0,
          i32 0), i8* getelementptr inbounds ([5 x
          i8]* @memcmps_s3, i32 0, i32 0), i8*
          getelementptr inbounds ([5 x i8]*
          @memcmps_s4, i32 0, i32 0)) #4
12   %13 = icmp ne i32 %funCall5, 0
13   br i1 %13, label %bb26, label %bb25
```

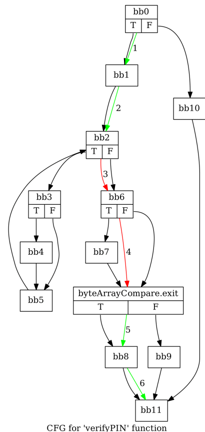
Processing Klee's results

- LLVM bytecode provided to Klee is instrumented with printf to log:
  - basic blocks traversed
  - faults injected
  - Others (countermeasures triggered, custom events...)
- traces are obtained from ktests files using Klee's replay tools and parsing stdout logs
- traces are then used in subsequent analysis of Lazart

Figure: VerifyPIN 2-fault attack trace (Python API)

```
>>> t22 = attacks_results(analysis)[22]
>>> print(t22)
(<2> t22: [BB(bb0), BB(bb1), BB(bb2), FAULT(bb2 -> bb6), BB(bb6), FAULT(bb6 -> byteArrayCompare.exit), BB(byteArrayCompare.exit), BB(bb8), BB(bb11), ]: Correct)
>>>
```

Figure: VerifyPIN 2-fault attack graph



erimag

Lazart's interface

User's input:

- attack objectives are expressed with `klee_assume`
- fault models and fault injection scopes are defined in a strategy file
- finer granularity with source instrumentation

```
1  fault-models:
2    - &ti
3      type: test-inversion
4    - &dl
5      type: data
6      var:
7        - tmp: 0
8        - x: symbolic
9        - y: symbolic
10 fault-scope:
11   functions:
12     - __all__:
13       - *ti
14     - foo:
15       - type: data
16         all: symbolic
17     - bar:
18       - *dl
```

Lazart's other features:

- python API for
  - manipulation of analysis and traces
  - generation of reports and attacks graphs
- automated countermeasure application (test duplication, SecSwift)
- countermeasure optimization[1]

[1] E. Boespflug, L. Mounier, C. Ene, M.-L. Potet, "Countermeasures Optimization in Multiple Fault-Injection Context", FDTC20

Conclusion and Future Works

Conclusion:

- benefits of using mutation functions
  - minimize impact of mutation on code structure
  - extensibility of fault models
- using DSE gives realistic analysis time for multiple faults injection
- already used in a certification context (Wookey bootloader [2])

Future Works:

- "Quantitative" analysis (weighting the attacks)
- Countermeasures analysis

[2] https://www.amossys.fr/upload/SSTIC2020-Article-inter-cesti_methodological_and_technical_feedbacks_on_hardware_devices_evaluations-benadjila.pdf

The End

Thanks for watching

Analysis script example

```
1  #!/usr/bin/python3
2
3  from lazart.main import *
4  import glob
5
6  # Init script
7  params = install_script()
8
9  # Create analysis
10 a = Analysis(params.order, glob.glob("../src/*.c"),
11     fault_model=FaultModel.TI, functions=["verifyPIN", "byteArrayCompare"],
12     flags=AnalysisFlag.AttackAnalysis, compiler_args="-Wall")
13
14 execute(a)
```

Experimentation results - Time metrics

Experimentation 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* (**RK**) step is also considered

Countermeasures tested: test duplication (TD), source counters[3] (LBH), swift

Table: Time metrics with 3-faults limit

Program	DSE (h)	Completed Paths	Traces	Countermeasures analysis
VP + TD	0:00:03	7118	296	26ms
VP + Swift	0:01:54	130 576	1005	89ms
VP + LBH	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

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