The Lazart tool - multiple faults attacks and DSE

Master Advanced Security 2024

Etienne Boespflug (*etienne.boespflug@gmail.com*)

2024-2025

Université de Grenoble Alpes

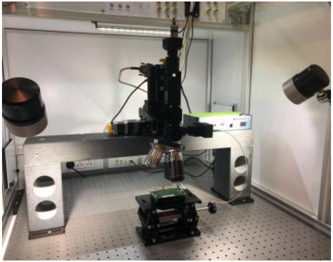
Outline

Fault Injection

Fault-injection attacks

- Lasers
- Electromagnetic pulses
- Temperature
- Power & clock glitches
- Software induced

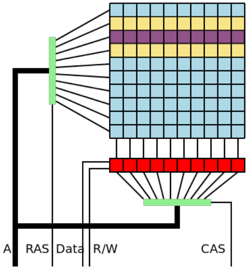
Figure: Laser fault injection bench [1]



Goal: modify device behavior/state to break security property and gain advantage

Fault Injection

Figure: Rowhammer principle [2]



Fault-injection attacks

- Lasers
- Electromagnetic pulses
- Temperature
- Power & clock glitches
- Software induced

Goal: modify device behavior/state to break security property and gain advantage

verify_pin program

PIN verification program from FISSC [1] collection

```
bool compare(uchar* a1, uchar* a2, size_t size)
{
    bool ret = true;
    size_t i = 0;
    for(; i < size; i++)
        if(a1[i] != a2[i])
            ret = false;

    return ret;
}

bool verify_pin(uchar* user_pin) {
    if(try_counter > 0)
        if(compare(user_pin, card_pin, PIN_SIZE)) {
            // Authentication
            try_counter = 3;
            return true;
        } else {
            try_counter--;
            return false;
        }
    return false;
}
```

- Compare user PIN against the card's one in constant time
- *Attack objective:* being authenticated with a false PIN

Faults injection - Example on `verify_pin`

PIN verification program from FISSC [1] collection

```c
bool compare(uchar* a1, uchar* a2, size_t size)
{
    bool ret = true;
    size_t i = 0;
    for(; i < size; i++) // Fault: avoid the loop
        if(a1[i] != a2[i])
            ret = false;

    return ret;
}

bool verify_pin(uchar* user_pin) {
    if(try_counter > 0)
        if(compare(user_pin, card_pin, PIN_SIZE)) {
            // Authentication
            try_counter = 3;
            return true;
        } else {
            try_counter--;
            return false;
        }
    return false;
}
```

- Fault model: modelisation of the faults to be injected

  → ex: **Test inversion**: inverse the branch taken during conditional branching

Faults injection - Example on `verify_pin`

PIN verification program from FISSC [1] collection

```
bool compare(uchar* a1, uchar* a2, size_t size)
{
    bool ret = true;
    size_t i = 0;
    for(; i < size; i++) // Fault
        if(a1[i] != a2[i])
            ret = false;

    if(i != size) // Countermeasure
        killcard();

    return ret;
}

bool verify_pin(uchar* user_pin) {
    if(try_counter > 0) {
        if(compare(user_pin, card_pin, PIN_SIZE)) {
            // Authentication
            try_counter = 3;
            return true;
        } else {
            try_counter--;
            return false;
        }
    return false;
}
```

- Fault model: modelisation of the faults to be injected

  → ex: Test inversion: inverse the branch taken during conditional branching

- **Software countermeasures (program transformations) can be placed to protect against faults**

Faults injection - Example on `verify_pin`

PIN verification program from FISSC [1] collection

```
bool compare(uchar* a1, uchar* a2, size_t size)
{
    bool ret = true;
    size_t i = 0;
    for(; i < size; i++) // Fault 1
        if(a1[i] != a2[i])
            ret = false;

    if(i != size) // Fault 2 => countermeasure attack
        killcard();

    return ret;
}

bool verify_pin(uchar* user_pin) {
    if(try_counter > 0)
        if(compare(user_pin, card_pin, PIN_SIZE)) {
            // Authentication
            try_counter = 3;
            return true;
        } else {
            try_counter--;
            return false;
        }
    return false;
}
```

- Fault model: modelisation of the faults to be injected

  → ex: Test inversion: inverse the branch taken during conditional branching

- Software countermeasures (program transformations) can be placed to protect against faults

  multiples faults → countermeasures themselves can be attacked

Robustness evaluation in multiple faults

State of the art attacks combine several faults to achieve their goal. [2, 3, 4]

Robustness evaluation in multiple faults

State of the art attacks combine several faults to achieve their goal. [2, 3, 4]

- Comparing the robustness of different protected versions of a program is not trivial

  ⇒ *attack surface paradox [Dureuil 2016]*: countermeasure can add attack surface to the code

Robustness evaluation in multiple faults

State of the art attacks combine several faults to achieve their goal. [2, 3, 4]

- Comparing the robustness of different protected versions of a program is not trivial
  $\Rightarrow$ *attack surface paradox [Dureuil 2016]*: countermeasure can add attack surface to the code
- How to count attacks in case of multiple faults ?
  $\Rightarrow$ Which program is the most secure ?

`verify_pin` version (from FISSC [1])	countermeasures	0-faults	1-fault	2-faults	3-faults	4-faults
vp_0	∅	0	3	0	0	1
vp_1	HB	0	2	0	0	1
vp_2	HB+FTL	0	2	1	0	1
vp_3	HB+FTL+INL	0	2	1	0	1
vp_4	FTL+INL+DPTC+PTCBK+LC	0	2	0	1	1
vp_5	HB+FTL+DPTC+DC	0	0	4	4	1
vp_6	HB+FTL+INL+DPTC+DT	0	0	3	0	1
vp_7	HB+FTL+INL+DPTC+DT+SC	0	0	2	0	1

Legend:

- HB: hardened booleans
- FTL: fixed time loops
- INL: inlined function
- PTC: try counter decremented first
- PTCBK: try counter backup
- DC: double call
- LC: loop counter verification
- SC: step counter
- DT: double test
- CFI: control flow integrity [5]

Robustness evaluation in multiple faults

State of the art attacks combine several faults to achieve their goal. [2, 3, 4]

- Comparing the robustness of different protected versions of a program is not trivial
  $\Rightarrow$ *attack surface paradox [Dureuil 2016]*: countermeasure can add attack surface to the code

- How to count attacks in case of multiple faults ?
  $\Rightarrow$ Which program is the most secure ?

`verify_pin` version (from FISSC [1])	countermeasures	0-faults	1-fault	2-faults	3-faults	4-faults
vp_0	∅	0	3	0	0	1
vp_1	HB	0	2	0	0	1
vp_2	HB+FTL	0	2	1	0	1
vp_3	HB+FTL+INL	0	2	1	0	1
vp_4	FTL+INL+DPTC+PTCBK+LC	**0**	**2**	**0**	1	1
vp_5	HB+FTL+DPTC+DC	**0**	**0**	**4**	4	1
vp_6	HB+FTL+INL+DPTC+DT	0	0	3	0	1
vp_7	HB+FTL+INL+DPTC+DT+SC	0	0	2	0	1

Legend:

- HB: hardened booleans
- FTL: fixed time loops
- INL: inlined function
- PTC: try counter decremented first
- PTCBK: try counter backup
- DC: double call
- LC: loop counter verification
- SC: step counter
- DT: double test
- CFI: control flow integrity [5]

Low-Level Virtual Machine (LLVM)

LLVM [6] is an intermediate representation commonly used for compilers (clang, rustc, Swift, Julia...), analysis tools (KLEE, AdressSanitizer...) and other projects (Unity with Burst).

Properties

- Infinite number of register (called *temporaries*
- Generic and typed assembly language
- Single Static Assignment (SSA) form [**?**]
- LLVM-IR used in binary and textual form between optimisation / analysis pass

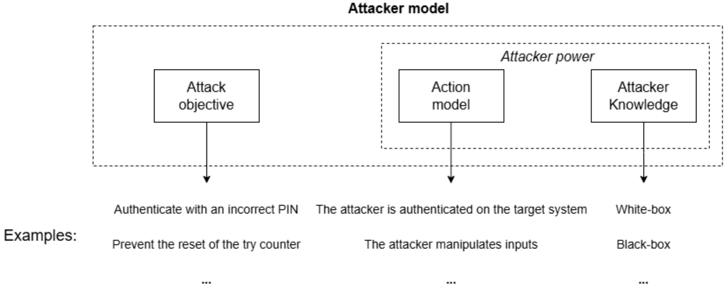
Listing: *Hello World!* program in IR LLVM (LLVM-9)

```
@.str = private unnamed_addr constant [14 x i8] c"Hello world !\00", align 1

define dso_local i32 @main(i32 %0, i8** %1) {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  %6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i64 0, i64 0))
  ret i32 0
}

declare dso_local i32 @printf(i8*, ...)
```

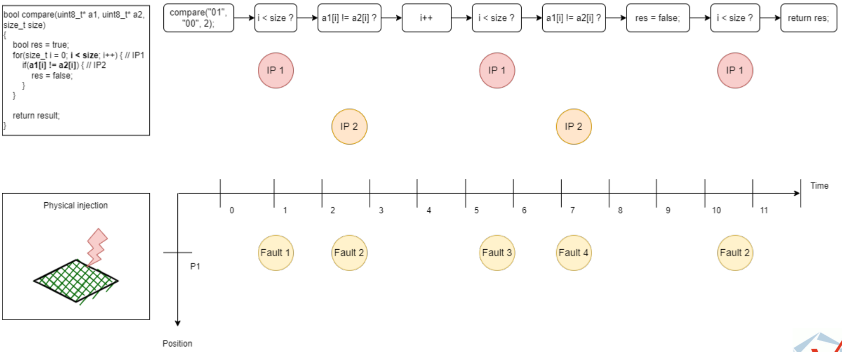
Attack model and representation level



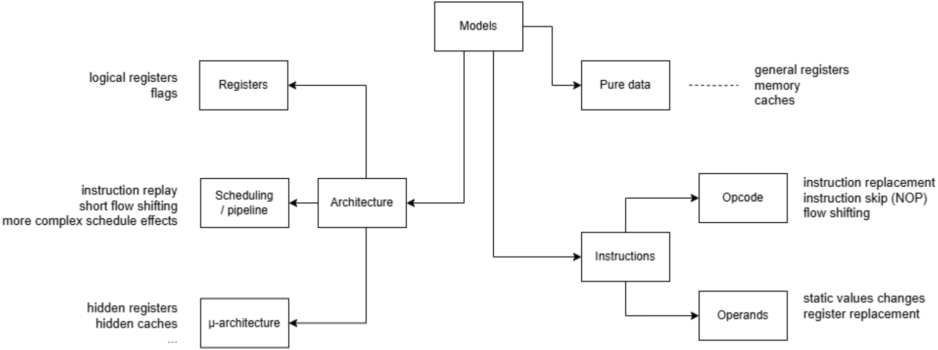
Attack model and representation level

Attacker model strongly depends on representation level.

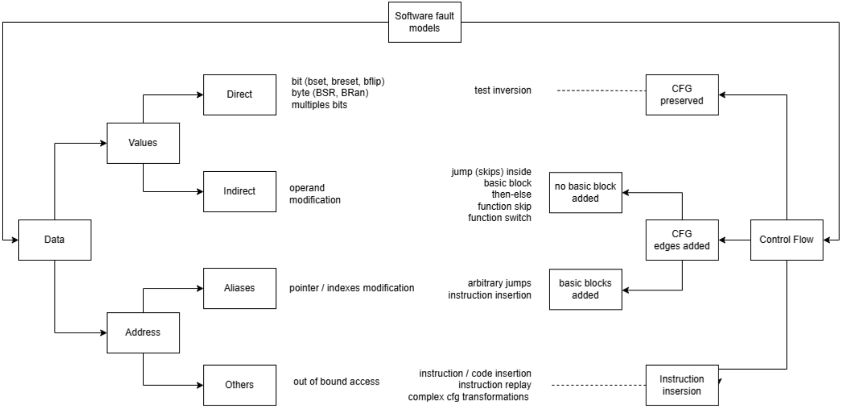
- Comparison of a faulted execution on software-level and physical-level:



Binary/architectural fault models



Software fault models



Faults Models and exploitation

Binary level effects:

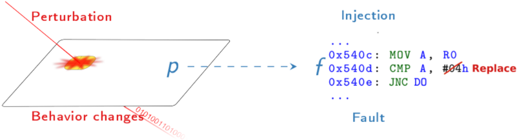
- Opcode / Operand replacement [7]
- Data modification: register or memory mutation [8, 9]
- Instruction replay [10]
- Out of ISA effects [11]

Software level effects:

- Control Flow modification [12, 4]
- Call graph modification [4]
- Variables / address alteration [5]

Exploitation:

- Side-channel [13]
- Bypassing secure boots [14]
- Privilege escalation [15]
- Buffer overflow [4]



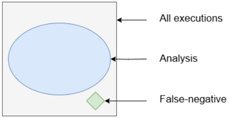
⇒ Combining several fault from different fault level allows to create more complex fault models.

Outline

Code analysis approaches

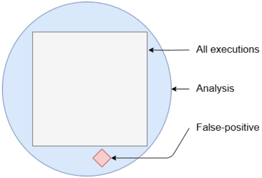
Under-approximation methods

examples: fuzzing, symbolic execution, unit tests...



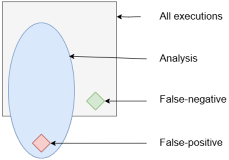
Over-approximation methods

examples: abstract interpretation



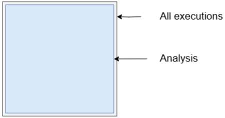
Hybrids methods

examples: dynamic-symbolic execution



Exact methods

examples: formal proofs



Dynamic Symbolic Execution

Formal method based on execution of the program using *symbolic* variables

Listing: Function `example`

```
void example(int x, int y) {
        σ = { x → x0, y → y0 }
        PC0 ≡ true
    if (x + y > 2) {   PC1 ≡ x0 + y0 > 2
        z = 2*x + y;
                σ = {z → 2*x0 + y0, x → x0, y → y0 }
        if (z == 20) {
                PC2 ≡ x0 + y0 > 2 ∧ 2 * x0 + y0 = 20
            x = z + 3;
                σ = { z → 2 * x0 + y0, x →
                2 * x0 + y0 + 3, y → y0 }
            print(x);
        }
        else {  PC3 ≡ x0 + y0 > 2 ∧ 2 * x0 + y0 ≠ 20
            y *= 2;           σ = { z →
                2 * x0 + y0, x → x0, y → 2 * y0 }
            assert(x != 0);
        }
    }
    else {  PC4 ≡ x0 + y0 <= 2
        foo();
    }
}
```

- Variables can be evaluated symbolicaly, maintaining a symbolic memory state σ
- Each time a condition is encountered, the execution is forked with the Path Constraint (PC) updated

  $\rightarrow$ a SMT solver is called to check if the PC of the path is satisfiable

Find entry that trigger the assert (x != 0) $\rightarrow$ solve the PC: [5, 10]

DSE and Fault Injection attacks

Definition (Godefroid 2011)

A path constraint PC_ω is *correct* if every model satisfying PC_ω gives entries for an execution following the path ω.
A path constraints PC_ω is *complete* if every entries following the path ω is a model satisfying PC_ω.

Listing: Nominal behavior

```
normal_behavior()
```

Listing: Faulted behavior

```
inject = symbolic_bool()
if inject and _fault_count <=
        _fault_limit:
    _fault_count++
    faulted_behavior()
else:
    normal_behavior()
```

DSE and Fault Injection attacks

Definition (Godefroid 2011)

A path constraint PC_ω is *correct* if every model satisfying PC_ω gives entries for an execution following the path ω.
A path constraints PC_ω is *complete* if every entries following the path ω is a model satisfying PC_ω.

Listing: Nominal behavior

```
normal_behavior()
```

Listing: Faulted behavior

```
inject = symbolic_bool()
if inject and _fault_count <=
        _fault_limit:
    _fault_count++
    faulted_behavior()
else:
    normal_behavior()
```

Definition

A faulted path constraint PC_ω^M is *correct* if every model satisfying PC_ω^M gives entries and faults for a (faulted) execution following the path ω.
A faulted path constraint PC_ω^M is *complete* if every pair (entries, faults) following the path ω is a model satisfying PC_ω^M.
The (faulted) path constraint enumeration PC_ω^M is *correct* if and only if, $\forall PC_\omega^M \in \mathcal{E}^M$, PC_ω^M is *correct*.
The (faulted) path constraint enumeration PC_ω^M is *complete* if and only if:

- $\forall PC_\omega^M \in \mathcal{E}^M$, PC_ω^M is *complete*, **and**,
- for all path $\omega \in \Omega^M$, $\exists PC_\omega^M \in \mathcal{E}^M$ such as PC_ω^M gives entries and faults for a (faulted) execution following the path ω.

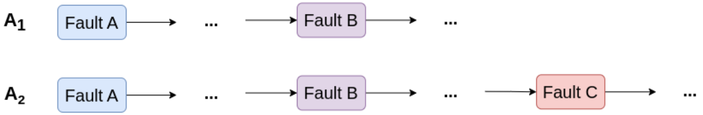
Redundancy / Equivalence

Attack traces are represented as a sequence of *nominal* and *faulted* transitions

- **Redundancy** and **equivalence** aims to filter attacks for the user in multiple faults

Definition (Redundancy prefix)

An attack a' is *redundant by prefix* wrt an attack a if the word of faulted transition of a is a **proper prefix** of the faulted transition word of a'



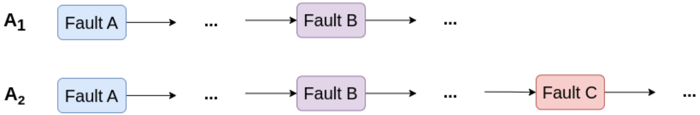
Redundancy / Equivalence

Attack traces are represented as a sequence of *nominal* and *faulted* transitions

- **Redundancy** and **equivalence** aims to filter attacks for the user in multiple faults

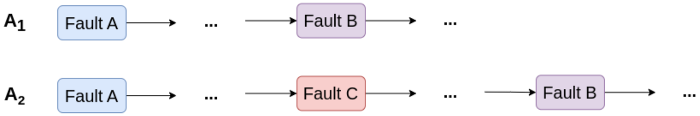
Definition (Redundancy prefix)

An attack a' is *redundant by prefix* wrt an attack a if the word of faulted transition of a is a **proper prefix** of the faulted transition word of a'



Definition (Redundancy subword)

An attack a' is *redundant by subword* wrt an attack a if the word of faulted transition of a is a **strict subword** of the faulted transition word of a'



Redundancy / Equivalence

Attack traces are represented as a sequence of *nominal* and *faulted* transitions

- **Redundancy** and **equivalence** aims to filter attacks for the user in multiple faults

Definition (Equivalence)

An attack a is **equivalent** to an attack a' if their sequence of transitions are equal

Definition (Fault-equivalence)

An attack a is **equivalent** to an attack a' if their sequence of faulted transitions are equal

Outline

Lazart overview

Lazart [12] is an LLVM-level multi-fault robustness evaluation tool based on Dynamic-Symbolic Execution (KLEE)

$\rightarrow$ Help developer to develop secure code

$\rightarrow$ Help auditor to find vulnerabilities

$\rightarrow$ Help for evaluation of countermeasures schemes

Lazart overview

LAZART

Lazart [12] is an LLVM-level multi-fault robustness evaluation tool based on Dynamic-Symbolic Execution (KLEE)

$\rightarrow$ Help developer to develop secure code

$\rightarrow$ Help auditor to find vulnerabilities

$\rightarrow$ Help for evaluation of countermeasures schemes

Handling multiple faults:

- Support for fault models combination
- Fine description of fault space
- Notion of redundancy and equivalence

Lazart overview

LAZART

Lazart [12] is an LLVM-level multi-fault robustness evaluation tool based on Dynamic-Symbolic Execution (KLEE)

→ Help developer to develop secure code

→ Help auditor to find vulnerabilities

→ Help for evaluation of countermeasures schemes

Handling multiple faults:

- Support for fault models combination
- Fine description of fault space
- Notion of redundancy and equivalence

Fault models

- Test/Branch inversion
- Data mutation (`load`) (symbolic)
- Jump
- Switch call

→ cover most of high-level fault models

Attack analysis - `verify_pin`

- 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

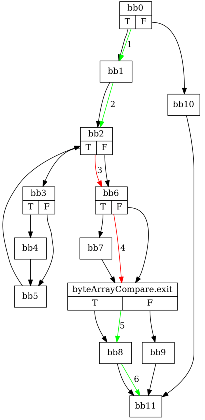
Attack analysis - `verify_pin`

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();



CFG for 'verifyPIN' function

Attack analysis - `verify_pin`

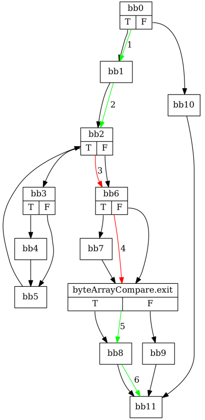
- Analysis parameters:
  - **Inputs**: Incorrect PIN
  - **Attack objective**: being authenticated with a false PIN
  - **Fault model**: up to N *test inversions*

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();

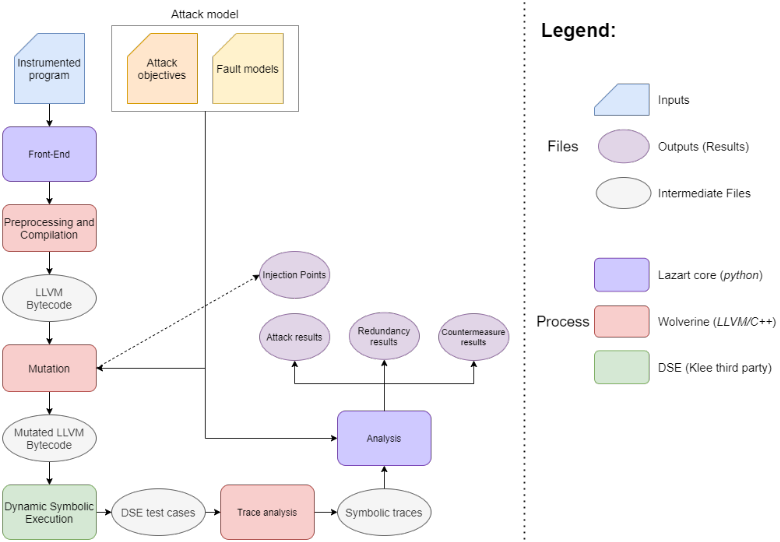
→ How to simplify the attacks presented to the user in multiple faults ?

Figure: The 2-faults attack (Test Inversion)



CFG for 'verifyPIN' function

Lazart: source level analysis for multiple faults injection



Program instrumentation

Symbolic input are used for:

- Symbolic inputs (using `klee_make_symbolic`).
- Attack objective (predicate verified by `_LZ__ORACLE`
- Faults (symbolic boolean determining if the fault is activated).

Fault can be defined:

- Using Python analysis script (see later).
- By program instrumentation (for some fault models).

Listing: Instrumentation example with Lazart

```c
int foo(int a, int b)
{
    if(b == 0)
        return a;
    return a / b;
}

int main() {
    int a;
    int b;
    // (memory pointer, size, name):
    klee_make_symbolic(&a, sizeof(a), "a");
    klee_make_symbolic(&b, sizeof(b), "b");

    int result = foo(a, b);

    _LZ__ORACLE(result > 0); // attack objective
        .
}
```

Analysis script example

```python
#!/usr/bin/python3

from lazart.lazart import *

# Parse CLI parameters
params = install_script()

data_sym = data_model({"vars": { # Explicit value description
    "len": 0, # len is faulted using fixed value.
    "res": "__sym__" # res is faulted using symbolic value.
}})


# Create analysis or load from path if available
a = read_or_make(["src/verify_pin.c", "src/main.c"], # Source files
    functions_list(["verify_pin", "compare"], [ti_model(), data_sym]), # Attack model
    path="results", # Pass in which analysis file will be stored
    flags=AnalysisFlag.AttackAnalysis, # Attack analysis (default)
    compiler_args="-Wall", # Compiler arguments
    params=params, # Pass CLI params to the analysis
    klee_args=""
)

execute(a) # Execute analysis, print results and generate report

verify.attack_analysis(a)
verify.traces_parsing(a)
```

Injecting a fault - mutation function

```
int lz_mutation_data_i32(int original, int fault_limit, int (*predicate)(int),
     const char* ip_name)
{
    int inject, value;
    klee_make_symbolic(&inject, sizeof(inject), "inject");
    if (inject && fault < fault_limit){
        klee_make_symbolic(&value, sizeof(value), "value");
        klee_assume(predicate(value));
        fault++;
        printf("[FAULT] at %s from %d to %d\n", ip_name, original,
             klee_get_value_i32(value));
        return value;
    }
    return original;
}
```

- 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

```
[...]
%6 = load i32* %tmp, align 4
%7 = sext i32 %6 to i64
%8 = call i32 @memcmp(i8* %4, i8* %5, i64 %7) #5
%9 = icmp ne i32 %8, 0
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

```
[...]
%6 = alloca i32
%7 = load i32* %tmp, align 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
store i32 %funCall4, i32* %6, align 4
%8 = load i32* %6, align 4
%9 = sext i32 %8 to i64
%10 = call i32 @memcmp(i8* %4, i8* %5, i64 %9)
      #5
%11 = icmp ne i32 %10, 0
%12 = sext i1 %11 to i32
%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
%13 = icmp ne i32 %funCall5, 0
br i1 %13, label %bb26, label %bb25
```

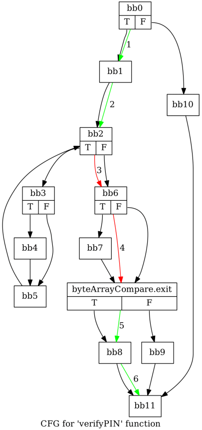
Processing KLEE's results

- 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 -> b
yteArrayCompare.exit), BB(byteArrayCompare.exit), BB(bb8), BB(bb11), ]: Correct)
>>>
```

Figure: VerifyPIN 2-fault attack graph



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

Lazart's other features:

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

```yaml
fault-models:
  - &ti
    type: test-inversion
  - &dl
    type: data
    var:
      - tmp: 0
      - x: symbolic
      - y: symbolic
fault-scope:
  functions:
    - __all__:
      - *ti
    - foo:
      - type: data
        all: symbolic
    - bar:
      - *dl
```

Demo

LIVE DEMONSTRATION

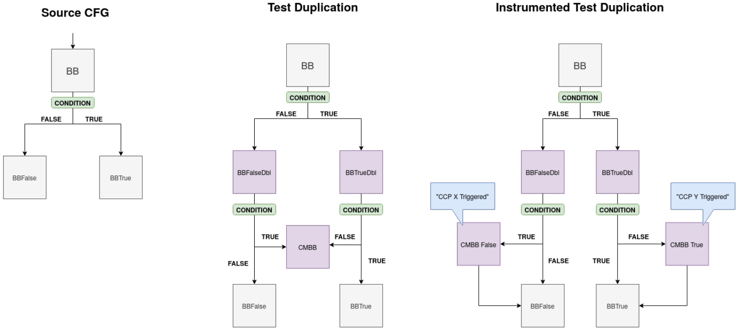
Outline

Countermeasures

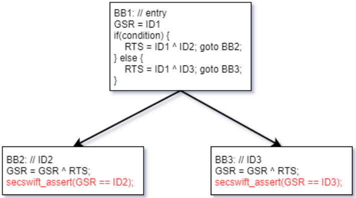
- **Countermeasures** are software modification that doesn't change the software nominal behavior (without fault) but improve software security in case of faults.
- Comparing protected program in multiple fault is not trivial.
- Software countermeasures, three classes:
  - Detective countermeasures: some checks are placed in the program to verify some security properties. Detection is represented in Lazart using `_LZ__CM();` function.
  - Infective countermeasures: in case of fault, the result is made unusable.
  - Others: complete program modifications etc.

Test Duplication

The *Test Duplication* generates two detectors for each conditional branch.



SecSwift Control-Flow



SecSwift ControlFlow is one of the 3 parts of SecSwift[16]

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

LBH's countermeasure [5]

```c
#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 **detectors**
- Analysis allows to know where the counter verification can be removed

Conclusion

- Multi fault make difficult to analyze programs (longer analysis times).
- Fault models depends on the representation level (physical, µarchitectural, binary, software, logcal...).
- Lazart uses DSE at LLVM level to produce multiple fault attacks tests cases.

References I

Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens.

FISSC: A Fault Injection and Simulation Secure Collection.

In *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*, pages 3–11, 2016.

Chong Hee Kim and Jean-Jacques Quisquater.

Fault attacks for crt based rsa: New attacks, new results, and new countermeasures.

In *IFIP International Workshop on Information Security Theory and Practices*, pages 215–228. Springer, 2007.

Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira.

Assessing Dependability with Software Fault Injection: A Survey.

ACM Computing Surveys, 48(3):1–55, February 2016.

Wookey/SSTIC20.

Inter-cesti: Methodological and technical feedbacks on hardware devices evaluations.

https://www.sstic.org/media/SSTIC2020/SSTIC-actes/inter-cesti_methodological_and_technical_feedbacks/SSTIC2020-Article-inter-cesti_methodological_and_technical_feedbacks_on_hardware_devices_evaluations-benadjila.pdf, 2020.

Jean-François Lalande, Karine Heydemann, and Pascal Berthomé.

Software countermeasures for control flow integrity of smart card C codes.

In *Pr. of the 19th European Symposium on Research in Computer Security, ESORICS 2014*, pages 200–218, 2014.

References II

Chris Lattner and Vikram Adve.
LLVM: A compilation framework for lifelong program analysis and transformation.
In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

J. Balasch, B. Gierlichs, and I. Verbauwhede.
An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus.
In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 105–114, 2011.

Johannes Blömer, Martin Otto, and Jean-Pierre Seifert.
A new crt-rsa algorithm secure against bellcore attacks.
In *10th ACM conference on Computer and communications security*, CCS '03, page 311–320, New York, NY, USA, 2003. Association for Computing Machinery.

I. Verbauwhede, D. Karaklajic, and J. Schmidt.
The fault attack jungle - a classification model to guide you.
In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 3–8, 2011.

Lionel Rivière, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage.
High precision fault injections on the instruction cache of armv7-m architectures.
In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*, pages 62–67. IEEE, 2015.

J. Laurent, V. Beroulle, C. Deleuze, and F. Pebay-Peyroula.
Fault injection on hidden registers in a risc-v rocket processor and software countermeasures.
In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 252–255, 2019.

References III

Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil.

Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections.

In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 213–222. IEEE, 2014.

Eli Biham and Adi Shamir.

Differential fault analysis of secret key cryptosystems.

In *Annual international cryptology conference*, pages 513–525. Springer, 1997.

Shoei Nashimoto, Naofumi Homma, Yu-ichi Hayashi, Junko Takahashi, Hitoshi Fuji, and Takafumi Aoki.

Buffer overflow attack with multiple fault injection and a proven countermeasure.

Journal of Cryptographic Engineering, 7(1):35–46, 2017.

Niek Timmers, Albert Spruyt, and Marc Witteman.

Controlling pc on arm using fault injection.

In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35. IEEE, 2016.

François de Ferrière.

A compiler approach to cyber-security.

2019 European LLVM developers' meeting, 2019.

Jörn-Marc Schmidt, Michael Hutter, and Thomas Plos.

Optical fault attacks on aes: A threat in violet.

In *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 13–22. IEEE, 2009.

References IV

Michael Hutter and Jörn-Marc Schmidt.
The temperature side channel and heating fault attacks.
In *International Conference on Smart Card Research and Advanced Applications*, pages 219–235. Springer, 2013.

Paul Kocher, Joshua Jaffe, and Benjamin Jun.
Differential power analysis.
In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi.
Introduction to differential power analysis.
Journal of Cryptographic Engineering, 1(1):5–27, 2011.

Karine Gandolfi, Christophe Mourtel, and Francis Olivier.
Electromagnetic analysis: Concrete results.
In *International workshop on cryptographic hardware and embedded systems*, pages 251–261. Springer, 2001.

Paul C Kocher.
Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems.
In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

Kris Tiri.
Side-channel attack pitfalls.
In *2007 44th ACM/IEEE Design Automation Conference*, pages 15–20. IEEE, 2007.

References V

Dmitri Asonov and Rakesh Agrawal.
Keyboard acoustic emanations.
In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 3–11. IEEE, 2004.

Haritabh Gupta, Shamik Sural, Vijayalakshmi Atluri, and Jaideep Vaidya.
A side-channel attack on smartphones: Deciphering key taps using built-in microphones.
Journal of Computer Security, 26(2):255–281, 2018.

Dan Boneh, Richard A DeMillo, and Richard J Lipton.
On the importance of checking cryptographic protocols for faults.
In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.

François Poucheret, Karim Tobich, Mathieu Lisarty, L Chusseauz, B Robissonx, and Philippe Maurine.
Local and direct em injection of power into cmos integrated circuits.
In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 100–104. IEEE, 2011.

Brice Colombier, Alexandre Menu, Jean-Max DUTERTRE, Pierre-Alain Moëllic, Jean-Baptiste Rigaud, and Jean-Luc Danger.
Laser-induced Single-bit Faults in Flash Memory: Instructions Corruption on a 32-bit Microcontroller.
In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–10, McLean, United States, May 2019. IEEE.

References VI

Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu.
Flipping bits in memory without accessing them: An experimental study of dram disturbance errors.
ACM SIGARCH Computer Architecture News, 42(3):361–372, 2014.

Mark Seaborn and Thomas Dullien.
Exploiting the dram rowhammer bug to gain kernel privileges.
Black Hat, 15:71, 2015.

Bilgiday Yuce, Patrick Schaumont, and Marc Wittenman.
Fault attacks on secure embedded software: Threats, design, and evaluation.
In *Journal of Hardware and Systems Security*, pages 111–130, 2018.

Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria.
Fault model analysis of laser-induced faults in sram memory cells.
In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 89–98. IEEE, 2013.

Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache.
Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures.
Proceedings of the IEEE, 100(11):3056–3076, 2012.

Alfredo Benso, Paolo Prinetto, Maurizio Rebaudengo, and M Sonza Reorda.
Exfi: a low-cost fault injection system for embedded microprocessor-based boards.
ACM Transactions on Design Automation of Electronic Systems (TODAES), 3(4):626–634, 1998.

Giorgis Georgakoudis, Ignacio Laguna, Dimitrios S Nikolopoulos, and Martin Schulz.
Refine: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed.
In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

Anna Thomas and Karthik Pattabiraman.
Llfi: An intermediate code level fault injector for soft computing applications.
In *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.

Vincent Werner.
Optimiser l'identification et l'exploitation de vulnérabilité à l'injection de faute sur microcontröleurs.
PhD thesis, Université Grenoble Alpes, 2022.

Olivier Faurax, Laurent Freund, Assia Tria, Traian Muntean, and Frédéric Bancel.
A generic method for fault injection in circuits.
In *2006 6th International Workshop on System on Chip for Real Time Applications*, pages 211–214. IEEE, 2006.

Hoang M Le, Vladimir Herdt, Daniel Große, and Rolf Drechsler.
Resilience evaluation via symbolic fault injection on intermediate code.
In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 845–850. IEEE, 2018.

Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar Iyer.
Symplfied: Symbolic program-level fault injection and error detection framework.
In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 472–481. IEEE, 2008.

References VIII

Guilhem Lacombe, David Feliot, Etienne Boespflug, and Marie-Laure Potet.

Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities.

In *PROOFS WORKSHOP (SECURITY PROOFS FOR EMBEDDED SYSTEMS)*, 2021.