Tools for code and countermeasures analysis against multiple faults attacks

PhD Thesis defense

Etienne Boespflug

April 28, 2023

Supervised by Marie-Laure Potet and Laurent Mounier

Rapporteurs :	Karine HEYDEMANN	-	Maître de conférences (Sorbonne Université)
	Julien SIGNOLES	-	Ingénieur de recherche (CEA Saclay)
Directeur :	Marie-Laure POTET	-	Professeur des universités (Grenoble INP)
Examinateurs :	Vincent BEROULLE	-	Professeur des universités (Grenoble INP)
	Thomas JENSEN	-	Directeur de recherche (INRIA de Rennes)
Invités :	David FÉLIOT	-	Ingénieur (CEA Leti)
	Laurent MOUNIER	-	Maître de conférences (Université Grenoble Alpes)

Supported by SECURIOT-2-AAP FUI 23, by ANR-15-IDEX-02 and LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01)

Outline

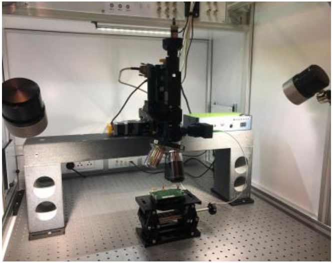
1. Context
2. The Lazart tool
3. Countermeasures placement
4. Countermeasures optimization
5. Conclusion and future work

Fault Injection

Fault-injection attacks

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

Figure: Laser fault injection bench [1]



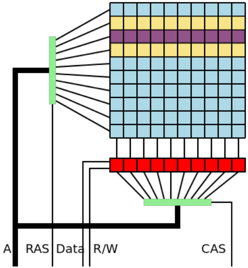
Goal: modify device behavior/state to break security property and gain advantage

Fault Injection

Fault-injection attacks

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

Figure: Rowhammer principle [2]



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

```c
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

```c
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

Multiple Faults

State of the art attacks combine several faults to achieve their goal. Most robustness evaluation tools consider only single fault

- Need to help developer and auditor in multiple faults
- Standard analysis cannot be trivially applied
  → faults can induce modification in the CFG or the data flow

Multiple Faults

State of the art attacks combine several faults to achieve their goal. Most robustness evaluation tools consider only single fault

- Need to help developer and auditor in multiple faults
- Standard analysis cannot be trivially applied
  → faults can induce modification in the CFG or the data flow
- Exploration of faulted execution is subject to *combinatory explosion* of paths due to faults

Multiple Faults

State of the art attacks combine several faults to achieve their goal. Most robustness evaluation tools consider only single fault

- Need to help developer and auditor in multiple faults
- Standard analysis cannot be trivially applied
  → faults can induce modification in the CFG or the data flow
- Exploration of faulted execution is subject to *combinatory explosion* of paths due to faults

Probl. 1

Need of automated tools to evaluate programs against multiple faults attacks

Robustness evaluation in multiple faults

- 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

- How to count attacks in case of multiple faults ?

Robustness evaluation in multiple faults

- 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

- How to count attacks in case of multiple faults ?

  ⇒ 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 [2]

Robustness evaluation in multiple faults

- 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

- How to count attacks in case of multiple faults ?
  ⇒ 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 [2]

Robustness evaluation in multiple faults

- 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

- How to count attacks in case of multiple faults ?
  ⇒ 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 [2]

Probl. 2

How to evaluate programs and countermeasures in multiple-fault context ?

Countermeasures evaluation challenge in multiple faults

- Try-and-error approaches are unsuitable for multi-faults

  → countermeasures themselves can be attacked

  → brute-forcing all countermeasures combinations is unrealistic

Probl. 3

How to help to place countermeasures and give guarantees on the protected program in multiple faults context ?

Countermeasures evaluation challenge in multiple faults

- Try-and-error approaches are unsuitable for multi-faults

  → countermeasures themselves can be attacked

  → brute-forcing all countermeasures combinations is unrealistic

Probl. 3

How to help to place countermeasures and give guarantees on the protected program in multiple faults context ?

- → most tools use systematic (everywhere) placement approach

Probl. 4

How to ensure that all added countermeasures are necessary ?

Contributions of this Thesis

Contributions:

- Extension of the tool Lazart and of robustness analysis for multiple faults

  → Problematic *P1*, *P2*

Problematics:

- *P1*: Multi-faults tools
- *P2*: Countermeasures evaluation / comparison
- *P3*: Countermeasures placement
- *P4*: Countermeasures necessity

Contributions of this Thesis

Contributions:

- Extension of the tool Lazart and of robustness analysis for multiple faults

  → Problematic *P1*, *P2*

- Isolation analysis and placement algorithms

  → Problematic *P2*, *P3*

Problematics:

- *P1*: Multi-faults tools
- *P2*: Countermeasures evaluation / comparison
- *P3*: Countermeasures placement
- *P4*: Countermeasures necessity

Contributions of this Thesis

Contributions:

- Extension of the tool Lazart and of robustness analysis for multiple faults

  → Problematic *P1*, *P2*

- Isolation analysis and placement algorithms

  → Problematic *P2*, *P3*

- Optimization of detector-based countermeasures

  → Problematic *P2*, *P4*

Problematics:

- *P1*: Multi-faults tools
- *P2*: Countermeasures evaluation / comparison
- *P3*: Countermeasures placement
- *P4*: Countermeasures necessity

Outline

1 Context

2 The Lazart tool

3 Countermeasures placement

4 Countermeasures optimization

5 Conclusion and future work

Lazart

LAZART

Lazart [3] 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

Lazart

Lazart [3] 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

Lazart

LAZART

Lazart [3] 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 (user-defined)

→ cover most of high-level fault models

DSE and Fault Injection attacks

An *Injection Point* (IP) is mutated using a *symbolic boolean* determining if an injection occurs, forking execution into the nominal and faulted behavior

Listing: Nominal behavior

```
normal_behavior()
```

Listing: Fault behavior

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

Only faults (+ values) and some entries (user-defined) are symbolic

DSE and Fault Injection attacks

An *Injection Point* (IP) is mutated using a *symbolic boolean* determining if an injection occurs, forking execution into the nominal and faulted behavior

Listing: Nominal behavior

```
normal_behavior()
```

Listing: Fault behavior

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

Only faults (+ values) and some entries (user-defined) are symbolic

- Dynamic Symbolic Execution = Symbolic Execution + concretization

  → *correct* (except for some concretizations)

  → not guaranteed to be *complete* on path enumeration

→ Lazart tries to give as much information as possible (timeout, coverage, errors...)

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

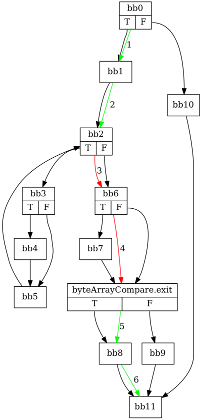
Attack analysis - `verify_pin`

- 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();

Figure: The 2-faults attack (Test Inversion)



CFG for 'verifyPIN' function

Attack analysis - `verify_pin`

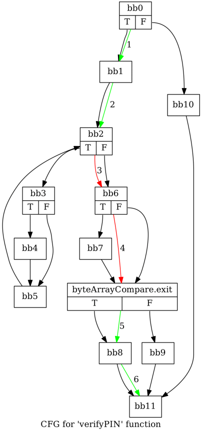
- Analysis parameters:
  - **Inputs**: Incorrect PIN
  - **Attack objective**: being authenticated with a false PIN
  - **Fault model**: up to N *test inversions*

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)



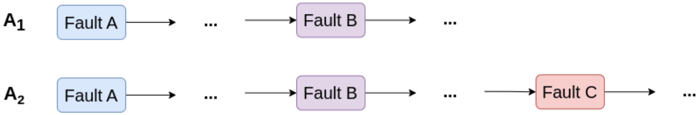
Redundancy / Equivalence

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'



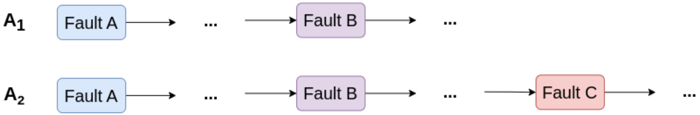
Redundancy / Equivalence

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

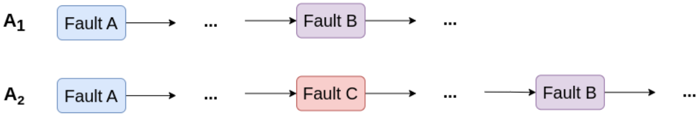
Definition (Redundancy prefix)

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

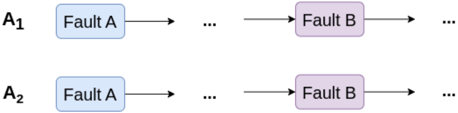
- **Redundancy** and **equivalence** aims to filter attacks for the user in multiple faults

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



Experimentation

Tested programs:

- *Verify_PIN* (**vp**): smart-card PIN verification process

  ⇒ *model*: test inversion
- *RSA Cipher* (**rsa**): implementation of RSA encryption scheme.

  ⇒ *model*: data load mutation
- *Firmware Updater* (**fu**): updates a firmware from remote source

  ⇒ *models*: test inversion and data load mutation

Experimentation

Tested programs:

- *Verify_PIN* (**vp**): smart-card PIN verification process

  ⇒ *model*: test inversion
- *RSA Cipher* (**rsa**): implementation of RSA encryption scheme.

  ⇒ *model*: data load mutation
- *Firmware Updater* (**fu**): updates a firmware from remote source

  ⇒ *models*: test inversion and data load mutation

- Simplifies the number of attacks to consider

  ⇒ by a factor 200 in a set of 15 examples from FISSC
- DSE is the main limit factor

  → with redundancy analysis matching on some examples

Table: Attack analysis on example programs

Program			Attacks					Minimal attacks (with equivalence)				
Nom	LoCs	IPs	1F	2F	3F	4F	Total	1F	2F	3F	4F	Total
vp0	25	4	12	21	18	7	**58**	3	0	0	0	**3**
vp4	45	11	34	118	180	147	**479**	3	0	1	0	**4**
vp7	78	8	4	36	116	173	**329**	1	2	0	0	**3**
rsa0	65	15	7	37	151	425	**620**	7	0	0	0	**7**
fu1	93	23	1	72	915	8191	**9179**	1	8	9	13	**31**
fu2	126	7	17	119	425	1031	**1592**	17	2	10	53	**82**

Summary

My contributions:

- **Python API :**
  - **Manipulation of traces, analysis and models**
  - **Fine fault space specification**
  - **User accessibility: error handling, control on attack objective**

Summary

My contributions:

- Python API
  - Manipulation of traces, analysis and models
  - Fine fault space specification
  - User accessibility: error handling, control on attack objective
- **Rewriting of Wolverine (mutation tool):**
  - **Fault model combination**
  - **Models data (+symbolic) and jump**
  - **Automated countermeasures (TM, LM..)**
  - **Switch to LLVM 9 (KLEE 2)**

Summary

My contributions:

- Python API
  - Manipulation of traces, analysis and models
  - Fine fault space specification
  - User accessibility: error handling, control on attack objective
- Rewriting of Wolverine (mutation tool):
  - Fault model combination
  - Models data (+symbolic) and jump
  - Automated countermeasures (TM, LM..)
  - Switch to LLVM 9 (KLEE 2)
- **Analysis:**
  - **Redundancy / Equivalence**
  - **Hotspots analysis**
  - **Countermeasure analysis**

Summary

My contributions:

- Python API
  - Manipulation of traces, analysis and models
  - Fine fault space specification
  - User accessibility: error handling, control on attack objective
- Rewriting of Wolverine (mutation tool):
  - Fault model combination
  - Models data (+symbolic) and jump
  - Automated countermeasures (TM, LM..)
  - Switch to LLVM 9 (KLEE 2)
- Analysis:
  - Redundancy / Equivalence
  - Hotspots analysis
  - Countermeasure analysis
- **Combination of Lazart with static analysis (Frama-C) on Wookey boot loader [Lacombe 2023]**
- **Filed at APP**

Outline

1. Context

2. The Lazart tool

3. Countermeasures placement

4. Countermeasures optimization

5. Conclusion and future work

Placement of software countermeasures

```
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;
}
```

- **Goal**: help to place countermeasures in multiple fault context (P2 and P3)

  → several fault models can be considered

  → countermeasures can be attacked with each fault model

  → countermeasures can introduce attacks

  → using a defined attack objective

Placement of software countermeasures

```
bool compare(uchar* a1, uchar* a2, size_t size)
{
    bool ret = true;
    size_t i = 0;
    for(; i < size; i++) {
        if(i >= size) killcard(); // Duplication (true)
        if(i >= size) killcard(); // Triplication (true)

        uchar a1_dup = a1[i];  // Load duplication
        if(a1_dup != a1[i]) killcard();
        uchar a2_dup = a2[i];  // Load duplication
        if(a1_dup != a1[i]) killcard();

        if(a1[i] != a2[i]) {
            if(i >= size)
                  killcard(); // Duplication (true)
            if(i >= size)
                  killcard(); // Triplication (true)
            ret = false;
        }
    }
    if(i != size) killcard(); // Duplication (false)
    if(i != size) killcard(); // Triplication (false)

    return ret;
}
```

- **Goal**: help to place countermeasures in multiple fault context (P2 and P3)

  → several fault models can be considered

  → countermeasures can be attacked with each fault model

  → countermeasures can introduce attacks

  → using a defined attack objective

Placement of software countermeasures

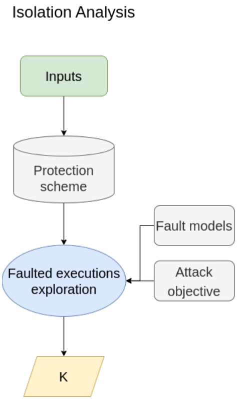
Goal: help to place countermeasures in multiple fault context

- Target robustness in *N* faults
- Give guarantees even if trace exploration is not complete
- Using a catalog of countermeasures schemes with *Injection Point* (IP) granularity

Approach:

- Compositional analysis using:
  - *Isolation analysis* of countermeasures schemes
    → Notion of protection coefficient
  - Exploration of attacks traces on the program P
- Placement algorithms

Principle of analysis in isolation



- Analysis of countermeasures scheme in isolation
- Focus on countermeasures with IP granularity

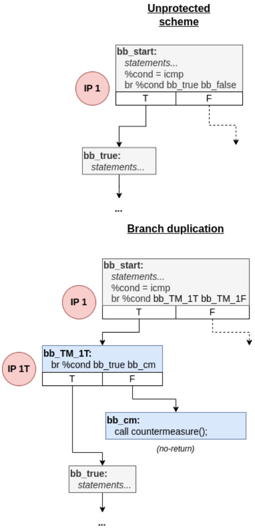
  → A **protection scheme** describe how an IP is protected
- Research of the *protection coefficient* (K) of the protection scheme:

  → e.g. *How many faults are required to induce an abnormal behavior (not detected) for the protected IP ?*

  → Unprotected IP has a $K = 1$

  → Can be computed with Lazart

Analysis in isolation of Branch duplication scheme



Unprotected scheme

IP 1

```
bb_start:
  statements...
  %cond = icmp
  br %cond bb_true bb_false
T F

bb_true:
  statements...
...
```

Branch duplication

IP 1

```
bb_start:
  statements...
  %cond = icmp
  br %cond bb_TM_1T bb_TM_1F
T F
```

IP 1T

```
bb_TM_1T:
  br %cond bb_true bb_cm
T F

bb_cm:
  call countermeasure();
```

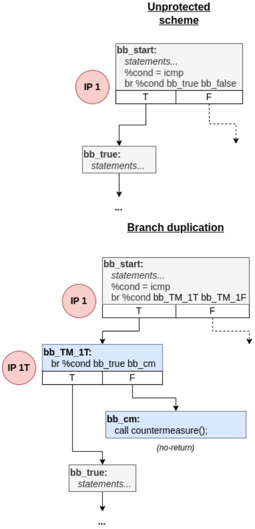
(no-return)

```
bb_true:
  statements...
...
```

Branch Multiplication (BM_n): n-plication of a conditional branch

Isolation analysis with *Branch Inversion* fault model

Analysis in isolation of Branch duplication scheme



Unprotected scheme

bb_start:
statements...
%cond = icmp
br %cond bb_true bb_false

bb_true:
statements...

...

Branch duplication

bb_start:
statements...
%cond = icmp
br %cond bb_TM_1T bb_TM_1F

bb_TM_1T:
br %cond bb_true bb_cm

bb_cm:
call countermeasure();

(no-return)

bb_true:
statements...

...

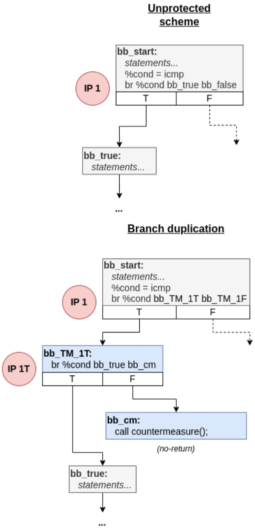
Branch Multiplication (BM_n): n-plication of a conditional branch

Isolation analysis with *Branch Inversion* fault model

Need to define:

- Input(s) of the scheme
- Output(s) of the scheme
- Entry point(s)
- Output point(s)
- Attack surface
- Nominal behavior

Analysis in isolation of Branch duplication scheme



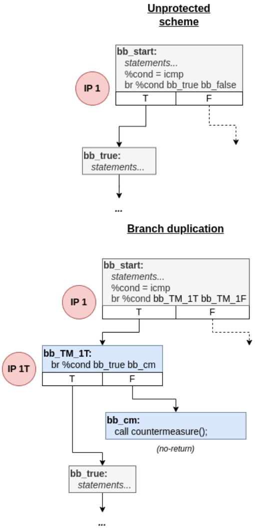
Branch Multiplication (BM_n): n-plication of a conditional branch

Isolation analysis with *Branch Inversion* fault model

Need to define:

- Input(s) of the scheme → **the %cond temporary**
- Output(s) of the scheme → **the destination branch**
- Entry point(s)
- Output point(s)
- Attack surface
- Nominal behavior

Analysis in isolation of Branch duplication scheme



Unprotected scheme

IP 1

bb_start:
statements...
%cond = icmp
br %cond bb_true bb_false

T F

bb_true:
statements...

...

Branch duplication

IP 1

bb_start:
statements...
%cond = icmp
br %cond bb_TM_1T bb_TM_1F

T F

IP 1T

bb_TM_1T:
br %cond bb_true bb_cm

T F

bb_cm:
call countermeasure();

(no-return)

bb_true:
statements...

...

Branch Multiplication (*BM_n*): n-plication of a conditional branch

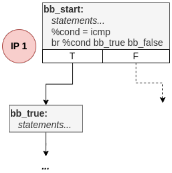
Isolation analysis with *Branch Inversion* fault model

Need to define:

- Input(s) of the scheme → **the `%cond` temporary**
- Output(s) of the scheme → **the destination branch**
- Entry point(s) → **the `br` instruction (`bb_start`)**
- Output point(s) → **the destination block (`bb_true`)**
- Attack surface
- Nominal behavior

Analysis in isolation of Branch duplication scheme

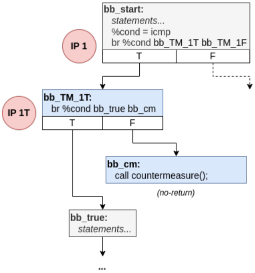
Unprotected scheme



bb_start:
statements...
%cond = icmp
br %cond bb_true bb_false

bb_true:
statements...

Branch duplication



bb_start:
statements...
%cond = icmp
br %cond bb_TM_1T bb_TM_1F

bb_TM_1T:
br %cond bb_true bb_cm

bb_cm:
call countermeasure();

(no-return)

bb_true:
statements...

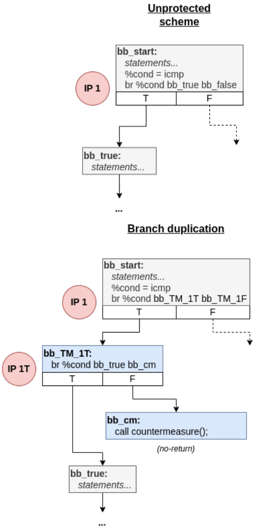
Branch Multiplication (BM_n): n-plication of a conditional branch

Isolation analysis with *Branch Inversion* fault model

Need to define:

- Input(s) of the scheme → **the `%cond` temporary**
- Output(s) of the scheme → **the destination branch**
- Entry point(s) → **the `br` instruction (`bb_start`)**
- Output point(s) → **the destination block (`bb_true`)**
- Attack surface → ***IP* 1 and *IP* 1*T* with BI fault model**
- Nominal behavior

Analysis in isolation of Branch duplication scheme



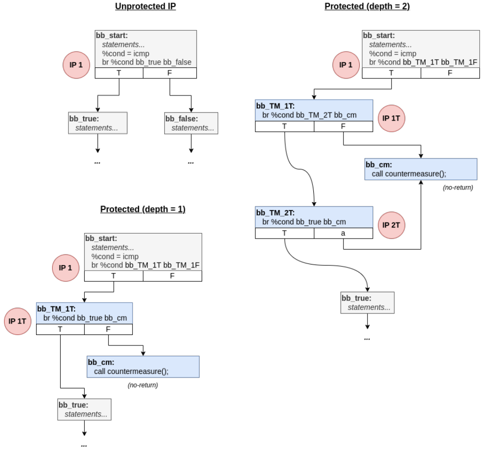
Branch Multiplication (BM_n): n-plication of a conditional branch

Isolation analysis with *Branch Inversion* fault model

Need to define:

- Input(s) of the scheme → **the** `%cond` **temporary**
- Output(s) of the scheme → **the destination branch**
- Entry point(s) → **the** `br` **instruction (**`bb_start`**)**
- Output point(s) → **the destination block (**`bb_true`**)**
- Attack surface → ***IP* 1 and *IP* 1*T* with BI fault model**
- Nominal behavior → **reach** `bb_true` **if and only if %cond is true**

Analysis in isolation of BM schemes



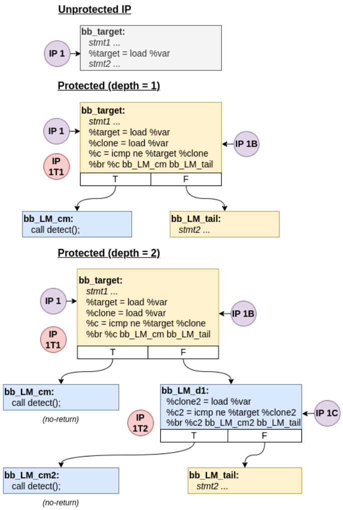
Branch Multiplication (BM_n): n-plication of a conditional branch

Isolation analysis with *Branch Inversion* fault model

Countermeasure	0-faults	1-fault	2-faults	3-faults	K
BM_0	0	1	0	0	**1**
BM_1	0	0	1	0	**2**
BM_2	0	0	0	1	**3**

Table: *BM* isolation analysis

Analysis in isolation of LM schemes



Unprotected IP

```
bb_target:
  stmt1 ...
  %target = load %var
  stmt2 ...
```

Protected (depth = 1)

```
bb_target:
  stmt1 ...
  %target = load %var
  %clone = load %var
  %c = icmp ne %target %clone
  %br %c bb_LM_cm bb_LM_tail
```

```
bb_LM_cm:
  call detect();
```

```
bb_LM_tail:
  stmt2 ...
```

Protected (depth = 2)

```
bb_target:
  stmt1 ...
  %target = load %var
  %clone = load %var
  %c = icmp ne %target %clone
  %br %c bb_LM_cm bb_LM_tail
```

```
bb_LM_cm:
  call detect();
```

(no-return)

```
bb_LM_d1:
  %clone2 = load %var
  %c2 = icmp ne %target %clone2
  %br %c2 bb_LM_cm2 bb_LM_tail
```

```
bb_LM_cm2:
  call detect();
```

(no-return)

```
bb_LM_tail:
  stmt2 ...
```

Load Multiplication (LM_n): n-plication of a `load` instruction (and checks)

Isolation analysis with *Data Load* and *Branch Inversion* fault models

- Input: the value stored in `%var` memory cell
- Output: the value loaded in `%target`
- Nominal behavior: `%target` stores `%var`'s value

Countermeasure	0-faults	1-fault	2-faults	3-faults	K
LM_0	0	1	0	0	**1**
LM_1	0	0	1	0	**2**
LM_2	0	0	0	1	**3**

Table: *LM* isolation analysis

Placement algorithms principles

GOAL: generate a P' program which is robust to *N* faults

⇒ Give guarantees that P' is *more robust than P* even if trace set is *incomplete* or if catalog is incomplete

Placement algorithms principles

GOAL: generate a P' program which is robust to *N* faults

⇒ Give guarantees that P' is *more robust than P* even if trace set is *incomplete* or if catalog is incomplete

Basic structure of placement algorithms:

1. Obtain set of attack traces

   ⇒ Computed with all fault models and the user-defined attack objectives
2. Compute **required protection coefficient** (K_{ip}) for each IP (initialized to 1)
3. Generate P' with protection scheme matching the **required protection coefficients**

   ⇒ Using a catalog $\mathcal{C}$ of countermeasures (with computed K_{ip})

Placement algorithms principles

GOAL: generate a P' program which is robust to N faults

⇒ Give guarantees that P' is *more robust than P* even if trace set is *incomplete* or if catalog is incomplete

Basic structure of placement algorithms:

1. Obtain set of attack traces

   ⇒ Computed with all fault models and the user-defined attack objectives
2. Compute **required protection coefficient** (K_{ip}) for each IP (initialized to 1)
3. Generate P' with protection scheme matching the **required protection coefficients**

   ⇒ Using a catalog $\mathcal{C}$ of countermeasures (with computed K_{ip})

Three approaches:

- *Systematic* placement: protect all IPs of a set with K > N
- *Block* placement: protect at least one IPs of all attacks with K > N
- *Distributed* placement: protect IPs such as for each trace, the sum of K of each IP is greater than N

Systematic placement algorithms

```
def placement_min(C: Catalog, P: Program, M: AttackModel, n: int):
    # Get sucessfull non-detected attacks.
    attacks = T_s(P, M, n)
    # Filter with minimals attacks.
    minimals = RedundacyAnalysis(attacks).minimals()

    # Initial protection factors Kn at 1 for all IP.
    required_kn = { IPA: 1, IPB: 1, ..., IPN: 1 }

    # Apply ponderation of n for all IP in traces
    for attack in minimals:
        for IP in attack:
            required_kn[IP] = n + 1 # Make IP robust en n faults.

    # Generation of P’
    P‘ = P
    for IP, kn in required_kn:
        S = C.get_cm(IP.model(), kn) # Select protection scheme from catalog
        P‘ = S(P‘, IP) # Apply local protection
    return P‘
```

Systematic algorithms:

- Naive placement (`naive`): protect all IP with K > N

  → *corresponds to standards systematic protection tools*

  → do not require attacks paths
- Attack placement (`atk`): protect all IP in attacks with K > N
- Minimal placement (`min`) (**on left)**: protect all IP in minimal attacks with K > N

Systematic placement algorithms

```
def placement_min(C: Catalog, P: Program, M: AttackModel, n: int):
    # Get sucessfull non-detected attacks.
    attacks = T_s(P, M, n)
    # Filter with minimals attacks.
    minimals = RedundacyAnalysis(attacks).minimals()

    # Initial protection factors Kn at 1 for all IP.
    required_kn = { IPA: 1, IPB: 1, ..., IPN: 1 }

    # Apply ponderation of n for all IP in traces
    for attack in minimals:
        for IP in attack:
            required_kn[IP] = n + 1 # Make IP robust en n faults.

    # Generation of P’
    P‘ = P
    for IP, kn in required_kn:
        S = C.get_cm(IP.model(), kn) # Select protection scheme from catalog
        P‘ = S(P‘, IP) # Apply local protection
    return P‘
```

Systematic algorithms:

- Naive placement (`naive`): protect all IP with K > N

  → *corresponds to standards systematic protection tools*

  → do not require attacks paths
- Attack placement (`atk`): protect all IP in attacks with K > N
- Minimal placement (`min`) (**on left)**: protect all IP in minimal attacks with K > N

Guarantees:

- **Robust in *N* faults** if the catalog provides required *K* for each IP (complete catalog) and if the entry trace set is complete

Systematic placement algorithms

```
def placement_min(C: Catalog, P: Program, M: AttackModel, n: int):
    # Get sucessfull non-detected attacks.
    attacks = T_s(P, M, n)
    # Filter with minimals attacks.
    minimals = RedundacyAnalysis(attacks).minimals()

    # Initial protection factors Kn at 1 for all IP.
    required_kn = { IPA: 1, IPB: 1, ..., IPN: 1 }

    # Apply ponderation of n for all IP in traces
    for attack in minimals:
        for IP in attack:
            required_kn[IP] = n + 1 # Make IP robust en n faults.

    # Generation of P’
    P‘ = P
    for IP, kn in required_kn:
        S = C.get_cm(IP.model(), kn) # Select protection scheme from catalog
        P‘ = S(P‘, IP) # Apply local protection
    return P‘
```

Systematic algorithms:

- Naive placement (`naive`): protect all IP with K > N

  → *corresponds to standards systematic protection tools*

  → do not require attacks paths
- Attack placement (`atk`): protect all IP in attacks with K > N
- Minimal placement (`min`) **(on left)**: protect all IP in minimal attacks with K > N

Guarantees:

- **Robust in *N* faults** if the catalog provides required *K* for each IP (complete catalog) and if the entry trace set is complete
- **At least as robust as** *P* otherwise

  → if the catalog is incomplete, *vulnerable traces in P' are known*

Block placement algorithm

```
def placement_bloc_h(C: Catalog, P: Program, M: AttackModel, n: int):
    # Get sucessfull non-detected attacks.
    attacks = T_s(P, M, n)
    # Filter with minimals attacks.
    minimals = RedundacyAnalysis(attacks).minimals()

    # Initial protection factors Kn at 1 for all IP.
    required_kn = { IPA: 1, IPB: 1, ..., IPN: 1 }

    # For all attacks by faults count.
    for order in 1 to n:
        # Loop trought order-faults attacks by number of associated redundant attacks.
        for attack in minimals.where(order=order).sort_by(Minimals):
            if is_protected(attack, required_kn):
                continue
            # Make attack robust in n faults
            IP = select IP in attack with most occurence
            required_kn[IP] = n + 1

    # Generation of P’
    P‘ = P
    for IP, kn in required_kn:
        S = C.get_cm(IP.model(), kn) # Select protection scheme from catalog
        P‘ = S(P‘, IP) # Apply local protection
    return P‘
```

Protection of at least one IP per minimal attack with K > N

→ heuristic based

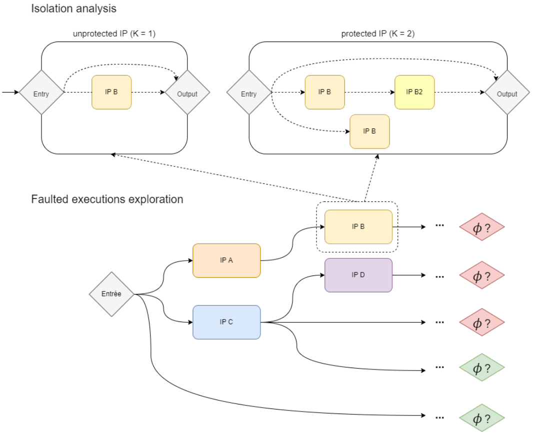
Guarantees:

- **Robust in *N* faults** if the catalog provides required *K* for each IP (complete catalog) and if the entry trace set is complete
- **At least as robust as** *P* otherwise

  → if the catalog is incomplete, *vulnerable traces in P' are known*

How to be sure than no attack paths is introduced by non-protected IPs ?

Compositional analysis placement

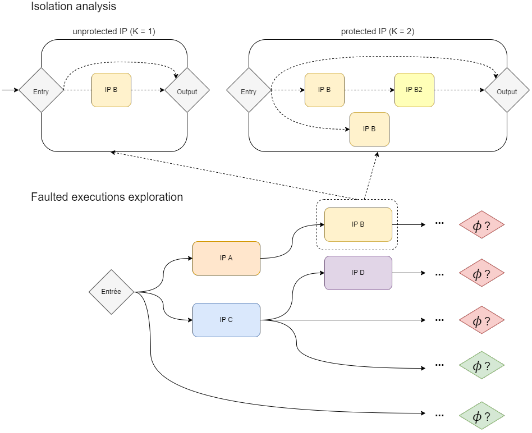


- Isolation analysis for each considered protection scheme with all studied fault models

- Attacks traces gives guarantees on which IP violation can lead to an attack

  → *Here, IPA can be left unprotected if IPB is protected*

Compositional analysis placement



- Isolation analysis for each considered protection scheme with all studied fault models

- Attacks traces gives guarantees on which IP violation can lead to an attack

  → *Here, IPA can be left unprotected if IPB is protected*

⇒ **Protection can be distributed between the IPs**

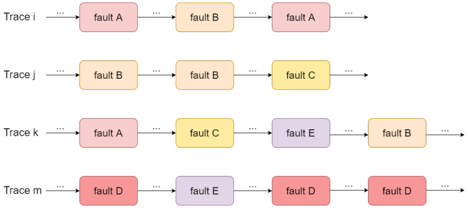
Optimal distributed placement

- Distribute protections of IPs inside minimal attacks traces to ensure at least N + 1 faults are required to obtain attacks

  → usable if the catalog $\mathcal{C}$ does not contains CM for K > N

- An Integer Linear Programming (ILP) optimization problem

  → attacks gives constraints on the protection to apply



$C_i = 2x_a + x_b \geq N + 1$

$C_j = 2x_b + x_c \geq N + 1$

$C_k = x_a + x_b + x_c + x_e \geq N + 1$

$C_m = 3x_d + x_e \geq N + 1$

Research of the **optimal** placement

⇒ minimize the protection weight $Z = x_a + x_b + \ldots + x_p$

- require to ensure that all states produced by the protected IPs are studied in trace exploration fault models

  → *guarantees on partially protected IPs*

Optimal distributed placement

```
1   def placement_rep_opt(C: Catalog, P: Program, M:
        AttackModel, n: int):
2       # Get sucessfull non-detected attacks.
3       attacks = T_s(P, M, n)
4       # Filter with minimals attacks.
5       minimals = RedundacyAnalysis(attacks).
          minimals()
6
7       # Initial protection factors Kn at 1 for all
           IP.
8       required_kn = { IPA: 1, IPB: 1, ..., IPN: 1
          }
9
10      constraints = [] # constraints for ILP
11      for attack in minimals:
12          constraints += compute_constraint(attack
          )
13
14      required_kn = solve_ilp(constraints,
          required_kn)
15
16      # Generation of P’
17      P‘ = P
18      for IP, kn in required_kn:
19          S = C.get_cm(IP.model(), kn) # Select
          protection scheme from catalog
20          P‘ = S(P‘, IP) # Apply local protection
21      return P‘
```

Optimal placement using ILP problem encoding:

1. Encode constraints from attack traces (lines 10-12)
2. Solve ILP (line 14)
3. Generate P’ (lines 17-20)

Guarantees:

- **Robust in *N* faults** if the catalog provides required *K* for each IP (complete catalog) and if the entry trace set is complete

  → and **optimal** (i.e. minimal set of protections wrt $\mathcal{C}$)
- **At least as robust as** *P* otherwise

  → if the catalog is incomplete, *vulnerable traces in P' are known*

Experimentation - verify_pin

verify_pin [1] (**VP**): smart-card PIN verification process

- *fault model:* branch inversion
- *inputs:* input PINs are different and symbolic
- *attack objective:* being authenticated or do not decrement the try counter
- *countermeasures:*
  - *integrated:* hardened boolean, fixed time loop
  - *placement:* branch multiplication (BM)

Exp.			Algo.	$\sum$ of protections				Robust
Program	Fault Model	IPs		1-fault	2-faults	3-faults	4-faults	
`vp2b`	BI	8	naive	8	16	24	32	✓
			atk	**3**	8	12	16	✓
			min	**3**	8	12	16	✓
			block	**3**	**6**	**9**	**12**	✓
			opt	**3**	**6**	**9**	**12**	✓

Experimentations - FU1

firmware_updater v1 (**fu1**): updates a firmware from remote source

- *fault model:* branch inversion + data load
- *inputs:* input PINs are different and symbolic
- *attack objective:* load a corrupted firmware or avoid load
- *countermeasures:*
  - *integrated:* systematic tests duplication
  - *placement:* branch multiplication (BM) and load multiplication (LM)

	Exp.		Algo.	$\sum$ of protections				Robust
Program	Fault Model	IPs		1-fault	2-faults	3-faults	4-faults	
`fu1`	BI	42	naive	42	84	126	168	✓
			atk	**0**	28	42	88	✓
			min	**0**	28	42	72	✓
			block	**0**	14	21	28	✓
			opt	**0**	**7**	**14**	**21**	✓
	DL	2	naive	2	4	6	8	✓
			atk	1	4	6	8	✓
			min	**1**	**2**	**3**	**4**	✓
			block	**1**	**2**	**3**	**4**	✓
			opt	**1**	**2**	**3**	**4**	✓
	BI+DL	44	naive	44	88	132	176	✓
			atk	**1**	32	60	96	✓
			min	**1**	32	60	80	✓
			block	**1**	16	24	32	✓
			opt	**1**	**9**	**17**	**25**	✓

Summary

- Robustness of placement depend on the property of the catalog $\mathcal{C}$

- P' is guaranteed to be robust for N faults if the required protection coefficients (K) are available

  → if not, attack traces on P' are known

  → more robust than P even if trace set is incomplete

- Protection weight: *distributed* $\leq$ *block* $\leq$ *min* $\leq$ *atk* $\leq$ *naive*

  → Optimal placement is guaranteed with ILP

Algorithme	Type	Guarantees P'		Complexity	Required analysis		
		Robust	Optimal		AA	Red	HS
naive	syst.	✓	-	$O(t)$	✓	-	-
atk	syst.	✓	-	$O(t)$	✓	-	-
min	syst.	✓	-	$O(t)$	✓	✓	-
block	block	✓	-	$O(t)$	✓	✓	✓
opt	distributed	✓	✓	NP-Complete	✓	✓	-

- Placement algorithm is fast compared to trace generation (DSE)

  → even with optimal algorithm and ILP (1-fault attacks)

Outline

1. Context

2. The Lazart tool

3. Countermeasures placement

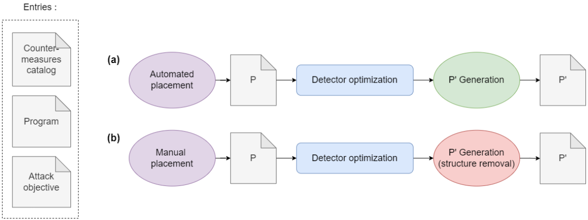
4. Countermeasures optimization

5. Conclusion and future work

Countermeasures optimization approach

- Can some part of the countermeasure be removed without adding new attacks ?

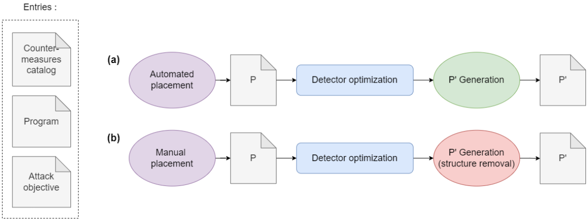
  ⇒ Probl. P2 & P4



Countermeasures optimization approach

- Can some part of the countermeasure be removed without adding new attacks ?

  ⇒ Probl. P2 & P4



- **Approach:**

  ⇒ focus on the subclass of detector-based countermeasures

  ⇒ based on the execution with non-blocking detectors

Definitions - Detectors and bodies

We divide a **detector-based countermeasure** in two parts:

- Detectors are control point in the program corresponding to sanity checks about the current state
- The countermeasure's bodies: shadow variables, parameters, additional computation etc.

```
bool compare(uchar* a1, uchar* a2,
    size_t size, size_t size_dup) {
    size_t i = 0u;
    bool result = true;
    bool result_dup = true;

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

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

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

    return result;
}
```

Objectives:

- determine if some detector could be removed, without adding any attack paths for the considered attack model.

Definitions - Detectors and bodies

We divide a **detector-based countermeasure** in two parts:

- Detectors are control point in the program corresponding to sanity checks about the current state
- The countermeasure's bodies: shadow variables, parameters, additional computation etc.

Examples

- *Test duplication* (**TD**)
- *Load Multiplication* (**LM**)
- *SecSwift Control Flow* (**SSCF**)[4]: associates an unique identifier to each basic block and uses a xor-based mechanism to ensure that the correct branch has been taken
- **LBH** [2]: introduce step counters to protect against C-level instruction skips. Each counter verification is a `detector`

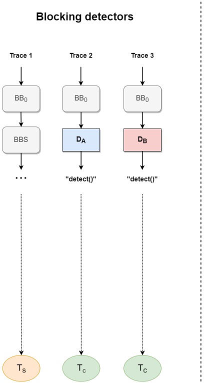
```
bool compare(uchar* a1, uchar* a2,
    size_t size, size_t size_dup) {
    size_t i = 0u;
    bool result = true;
    bool result_dup = true;

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

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

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

    return result;
}
```

Objectives:

- determine if some detector could be removed, without adding any attack paths for the considered attack model.

Countermeasure optimization - Idea: don’t stop execution after detection

- Detector are considered as a structure `if(cond) killcard();` that can be attacked and `killcard` an atomic detection function

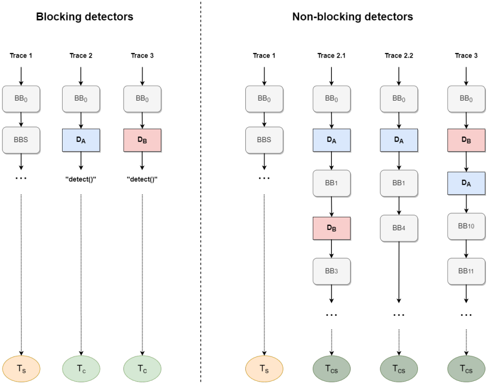


Countermeasure optimization - Idea: don't stop execution after detection

- Detector are considered as a structure `if(cond) killcard();` that can be attacked and `killcard` an atomic detection function
- *Intuition:* Explore the program executions by noting at each detector location if the corresponding detector would be triggered

  ⇒ all variation of detectors are considered in one single exploration.

  ⇒ require side-effect free property on detectors

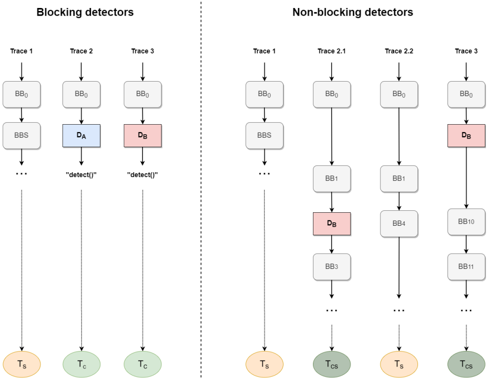


Countermeasure optimization - Idea: don’t stop execution after detection

- Detector are considered as a structure `if(cond) killcard();` that can be attacked and `killcard` an atomic detection function
- *Intuition:* Explore the program executions by noting at each detector location if the corresponding detector would be triggered

  ⇒ all variation of detectors are considered in one single exploration.

  ⇒ require side-effect free property on detectors

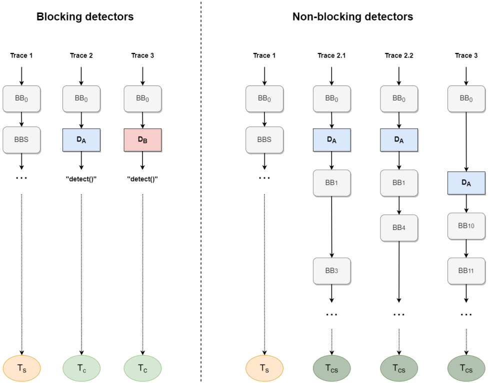


Countermeasure optimization - Idea: don't stop execution after detection

- Detector are considered as a structure `if(cond) killcard();` that can be attacked and `killcard` an atomic detection function
- *Intuition:* Explore the program executions by noting at each detector location if the corresponding detector would be triggered

  ⇒ all variation of detectors are considered in one single exploration.

  ⇒ require side-effect free property on detectors



Countermeasure optimization - Principle

- **An optimization problem:** Search the minimal set of detectors $\mathcal{D}$ such as at least one detector cover each trace of the set of detected attack traces

  $\Rightarrow$ using non-blocking detectors

- Exploration space can be reduced by *classification* step:
  - If $\forall t \in T'$, $d_i \notin \{cm(t)\}$, d_i is **inactive** (should be removed)
  - If $\exists t \in T'$, $cm(t) = \{d_i\}$, d_i is **necessary** (should be kept)
  - Other detectors are **repetitive**
    $\Rightarrow$ focus on traces containing only **repetitive** detectors

Methodology - Step 1 - Test Duplication results in 2 faults

VerifyPIN + Test Duplication:

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

Table: Detectors classification in 2 faults

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

```
bool verify_pin(uchar* user_pin) {
    if(try_counter > 0)      D0 & D1
        if(compare(user_pin, card_pin, PIN_SIZE)
              ) {  D8
            try_counter = 3;
            return true;
        } else {  D9
            try_counter--;
            return false;
        }
    return false;
}
```

```
bool compare(uchar* a1, uchar* a2, size_t size)
      {
    bool result = true;
    size_t i = 0;
    for(; i < size; i++) {  D2 & D3
        if(a1[i] != a2[i]) {  D4 & D5
            result = false;
        }
    }

    if(i != size)  D6 & D7
        countermeasure(100);  D100

    return result;
}
```

Step 4 - Removed CCPs for verifyPIN (2 faults)

The removed and kept detectors and bodies for *Test duplication* on `verify_pin` with 2 faults

```
bool verify_pin(uchar* user_pin) {
    bool c_1;
    bool c_2;
    if(c_1 = try_counter > 0) {
        if(!c_1)
            killcard();

        if(c_2 = compare(user_pin, card_pin, PIN_SIZE)) {
            if(!c_2)
                countermeasure();
            try_counter = 3;
            return true;
        } else {
            if(c_2)
                countermeasure();
            try_counter--;
            return false;
        }
    } else
        if(c_1)
            countermeasure();

    return false;
}
```

```
bool compare(uchar* a1, uchar* a2, size_t size) {
    bool result = true;
    size_t i = 0;
    bool c_1;
    bool c_2;
    bool c_3;

    for(; c_1 = i < size; i++) {
        if(!c_1)
            countermeasure();
        if(c_2 = a1[i] != a2[i]) {
            if(!c_2)
                countermeasure();
            result = false;
        } else
            if(c_2)
                countermeasure();
    }
    if(c_1)
        countermeasure();

    if(c_3 = i != size) {
        if(!c_3)
            countermeasure();
        countermeasure();

    } else
        if(c_3)
           countermeasure();

    return result;
}
```

Experimentations

Program	Detectors	1 attack	2 attacks	3 attacks
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%

Three countermeasures experimented:

- *Test duplication* (**TD**): presented previously
- *SecSwift Control Flow* (**SSCF**)[4]: associates an unique identifier to each basic block and uses a xor-based mechanism to ensure that the correct branch has been taken
- **LBH** [2]: introduce step counters to protect against C-level instruction skips. Each counter verification is a `detector`

Outline

1. Context

2. The Lazart tool

3. Countermeasures placement

4. Countermeasures optimization

5. Conclusion and future work

Conclusion

Robustness evaluation with Lazart

- filter of multi-fault attacks: equivalence and redundancy
- combination of fault models
- user accessibility, case studies

Countermeasures evaluation

- isolation analysis
- placement algorithms

  → gives strong guarantees, even if the trace set is incomplete

  → allows combination of fault model in multiple faults
- detector optimization algorithm

  → up to 80% of `detectors` removed

Publications:

- *Combining Static Analysis and Dynamic Symbolic Execution in a Toolchain to detect Fault Injection Vulnerabilities*
  Guilhem Lacombe, David Féliot, Etienne Boespflug and Marie-Laure Potet, *Journal of Cryptography Engineering 2023*
- *Combining Static Analysis and Dynamic Symbolic Execution in a Toolchain to detect Fault Injection Vulnerabilities*
  Guilhem Lacombe, David Féliot, Etienne Boespflug and Marie-Laure Potet, *PROOFS Workshop 2021*
- *Countermeasures Optimization in Multiple Fault-Injection Context*
  Etienne Boespflug, Cristian Ene, Marie-Laure Potet and Laurent Mouniter, *FDTC 2020*

Future Works

Tools:

- Extension of fault models in Lazart
- Extension of automated countermeasures in Lazart
- Validate contribution on more example programs
- Combination with static analysis
- "fault-aware" dynamic-symbolic execution engine

Future Works

Tools:

- Extension of fault models in Lazart
- Extension of automated countermeasures in Lazart
- Validate contribution on more example programs
- Combination with static analysis
- "fault-aware" dynamic-symbolic execution engine

Countermeasure placement:

- Study of countermeasures without IP granularity
- Study of countermeasures propagating states (SSCF, Swift...)

  → may require to consider two isolation analysis cases: sane CM's inputs and corrupted CM's inputs
- Study of more complex CFG fault models

  → requires to take into account the several entry and output points of the protection scheme
- Extension of notion of *adequation*, *perfection* of CMs and *protectability* of fault models

Future Works

Tools:

- Extension of fault models in Lazart
- Extension of automated countermeasures in Lazart
- Validate contribution on more example programs
- Combination with static analysis
- "fault-aware" dynamic-symbolic execution engine

Countermeasure placement:

- Study of countermeasures without IP granularity
- Study of countermeasures propagating states (SSCF, Swift...)

  $\longrightarrow$ may require to consider two isolation analysis cases: sane CM's inputs and corrupted CM's inputs
- Study of more complex CFG fault models

  $\longrightarrow$ requires to take into account the several entry and output points of the protection scheme
- Extension of notion of *adequation*, *perfection* of CMs and *protectability* of fault models

Countermeasure optimization:

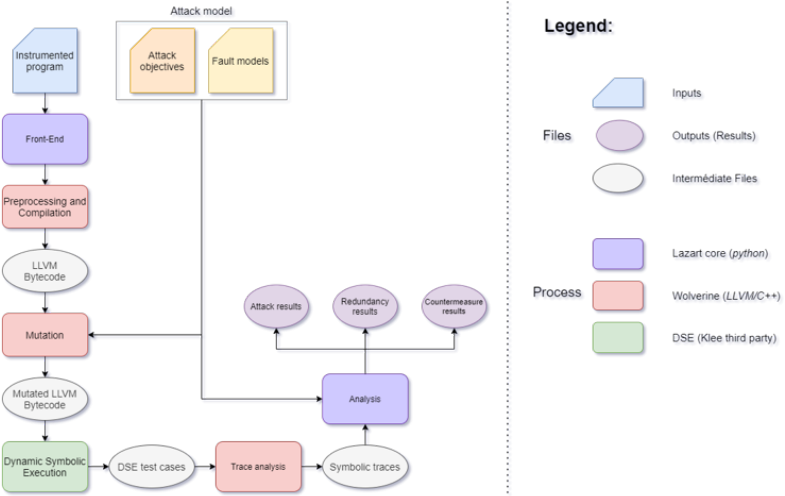
- Switch to Lazart 4
- Fully symbolic version (relax constraints on detectors)

  $\longrightarrow$ internal states may be difficult to consider
- Study of other countermeasures

The End

Thanks for watching

Lazart architecture



Attack model

Attack objectives

Fault models

Instrumented program

Front-End

Preprocessing and Compilation

LLVM Bytecode

Mutation

Mutated LLVM Bytecode

Dynamic Symbolic Execution

DSE test cases

Trace analysis

Symbolic traces

Analysis

Attack results

Redundancy results

Countermeasure results

Legend:

Files

Inputs

Outputs (Results)

Intermédiate Files

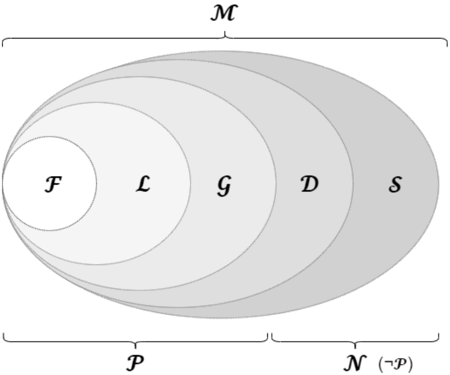
Process

Lazart core (*python*)

Wolverine (*LLVM/C++*)

DSE (Klee third party)

Model protectability



Modèles:

- $\mathcal{P}$: Protégeables
  - $\mathcal{F}$: Parfaitement protégeables
  - $\mathcal{L}$: Localement protégeables
  - $\mathcal{G}$: Globalement protégeables
- $\mathcal{N}$: Non-protégeables
  - $\mathcal{D}$: Diluables
  - $\mathcal{S}$: Strictement non-protégeables

- notion of *adequation* of CMs
- notion of *perfection* of CMs
- notion of protectability of models

Future work - Fully symbolic CCPO

- The methodology requires properties for detectors → a full symbolic version can be used

- Each detectors forks the execution:

```
if(sym_bool()) // Is the detector active ?
{
    // Local code with detector
}
else
{
    // Local unprotected code.
}
```

- Issues:
  - Paths explosion
  - Some countermeasure structures depend on the presence of detectors (can require specific instrumentation for CMs)

Detector requirements

Let V_P be the state of the non-protected program and V_C the state of the countermeasures.

The detector has limitation for the read and write operation on the states V_P et V_C.

Listing: Generic detector example

```
...
stm1; // can: read(VP, VC) & write(VP, VC)

// Detector:
if(cond) { // read(VP, VC)
    stm2; // read(VP, VC) & write(VP, VC)
    killcard();
}

stm3; // read(VP, VC) & write(VP, VC)
...
```

memcmps3 program

Listing: Analysis's main

```c
// main.c
#include "lazart.h"
#include "memcmps.h"

#define SIZE 4

int main()
{
    // Inputs
    uint8_t a1[SIZE];
    _LZ__SYM(a1, SIZE); // Symbolic array
    uint8_t a2[SIZE];
    _LZ__SYM(a2, SIZE); // Symbolic array

    bool equals = true;
    for(size_t i = 0; i < SIZE; ++i)
        if(a1[i] != a2[i])
            equals = false;
    _LZ__ORACLE(!equal); // Consider only different inputs

    BOOL res = memcmps(a1, a2, SIZE); // Call studied function

    _LZ__ORACLE(res == TRUE); // Attack objective
}
```

Listing: memcmps3 program

```c
// memcmps.h
typedef BOOL uint16_t;
#define TRUE    0x1234u
#define FALSE   0x5678u
#define MASK    0xABCDu

// memcmps.c
#include "memcmps.h"

BOOL memcmps(uint8_t* a, uint8_t* b, size_t len)
{
  BOOL result = FALSE;

  if (!memcmp(a, b, len)) {
    result ^= MASK;             // result = FALSE ^ MASK
    if (!memcmp(a, b, len)) {
      result ^= FALSE ^ TRUE;   // result = MASK ^ TRUE
      if (!memcmp(a, b, len)) {
        result ^= MASK;         // result = TRUE
      }
    }
  }

  return result;
}
```

memcmps3 analysis file

```
#!/usr/bin/python3
from lazart.lazart import *

attack_model = functions_list(["memcmps"], [ti_model(), data_model({ "vars": { "result": "
      __sym__", "len":  "__sym__" } })])

a = Analysis(["memcmps.c", "main.c"], # Input files
    attack_model, # Attack model
    flags=AnalysisFlag.AttacksAnalysis, # Analysis type
    compiler_args="-Wall",
    max_order=4,
    path="my_analysis")

execute(a)

```

Listing: memcmps3 program

Countermeasure optimization

- The program P contains a set of *detector* $\mathcal{D}$.

  Stopping traces: $s_0 \dots s_n d_i$, where s_i are nominal or faulty transitions, and d_i the triggered detector

  Non-stopping traces: $s_0 \dots s_n d_i s_0^2 \dots s_n^2 d_j \dots$ with d_i detectors triggering

  $\rightarrow$ Stopping traces are prefix of non-stopping ones. One stopping trace can lead to several non-stopping traces

- **Goal** $\Rightarrow$ find the minimal set of detectors in which at least one detector is kept for each trace

- Only traces which validate the attack objective ϕ and in which at least one detector is triggered are considered. This trace set is denoted $\mathcal{T}_P$.

  The detector *selection* is an optimization problem, searching a minimal set of detectors covering each traces of $\mathcal{T}_P$ with at least one trigger.

  Exploration space can be reduced:

  - If $\forall t \in T', d_i \notin \{\mathit{triggered}(t)\}$, d_i is inactive (should be removed)
  - If $\exists t \in T', \mathit{triggered}(t) = \{d_i\}$, d_i is necessary (should be kept)

Experimentation results - Playing with the attack objectives

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

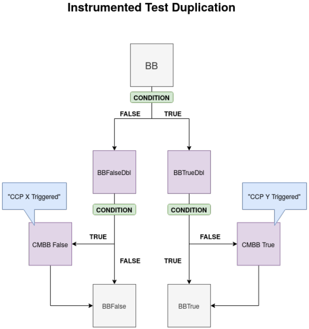
- ϕ_{auth}: being authenticated with a false PIN.
- ϕ_{ptc}: do not decrement the try counter with a false PIN.

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

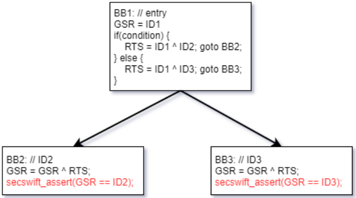
Property	1 fault	2 faults	3 faults
ϕ_{auth}	83%	72%	18%
ϕ_{ptc}	72%	63%	9%
$\phi_{auth \wedge ptc}$	83%	72%	18%
$\phi_{auth \vee ptc}$	72%	63%	9%
ϕ_{true}	18%	9%	9%

Test Duplication

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



SecSwift Control-Flow



```
BB1: // entry
GSR = ID1
if(condition) {
    RTS = ID1 ^ ID2; goto BB2;
} else {
    RTS = ID1 ^ ID3; goto BB3;
}
```

```
BB2: // ID2
GSR = GSR ^ RTS;
secswift_assert(GSR == ID2);
```

```
BB3: // ID3
GSR = GSR ^ RTS;
secswift_assert(GSR == ID3);
```

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

- 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 [2]

```
#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

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.

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.

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.

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 II

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.

Eli Biham and Adi Shamir.

Differential fault analysis of secret key cryptosystems.

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

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.

References III

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.

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.

References IV

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.

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.

References V

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.

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.

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.

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.

References VI

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.

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.

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.

References VII

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.

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.

References VIII

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.

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.

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.

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.

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 IX

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.