Placement of software countermeasures: a compositional approach

Etienne Boespflug

September 28, 2023

JAIF 2023

VERIMAG - Université Grenoble Alpes (UGA)
name.lastname@univ-grenoble-alpes.fr

Supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) and Arsène (ANR-22-PECY-0004) and SECUREVAL projects (ANR-22-PECY-0005)

Outline

Faults injection - Example on `verify_pin`

PIN verification program from FISSC collection [Dureuil et al., 2016]

```
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;
}
```

- Example of software fault model: *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 collection [Dureuil et al., 2016]

```
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;
}
```

- Example of software fault model: *Test inversion*

  → inverse the branch taken during conditional branching

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

Multi-fault:

→ **countermeasures themselves can be attacked**

→ **require support for models combination**

Lazart results on VerifyPIN collection

Lazart [Potet et al., 2014] is an **LLVM**-level multi-fault robustness evaluation tool based on **Dynamic-Symbolic Execution** (KLEE).

Fault models

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

Lazart results on VerifyPIN collection

Lazart [Potet et al., 2014] is an **LLVM**-level multi-fault robustness evaluation tool based on **Dynamic-Symbolic Execution** (KLEE).

Fault models

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

`verify_pin` version (from FISSC [Dureuil et al., 2016])	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 [Lalande et al., 2014]

Multiple faults and countermeasures placement

- State of the art attacks combine several faults to achieve their goal [Kim and Quisquater, 2007], [Natella et al., 2016], [Wookey/SSTIC20, 2020]

- Try-and-error approaches are unsuitable for multi-fault
  $\rightarrow$ countermeasures themselves can be attacked
  $\rightarrow$ testing all countermeasures placements is unrealistic

- Several tools use *systematic approach*, which could lead to unnecessary protections [Lalande et al., 2014, de Ferrière, 2019]

Probl.

How to help to place countermeasures and give guarantees on the protected program in multi-fault context ?

Placement of software countermeasures

Goal: help to place countermeasures against multi-fault attacks wrt a set of fault models *M*

- Target **robustness** in (*at least*) *N* faults
- Using a catalog of countermeasures schemes with *Injection Point* (IP) granularity

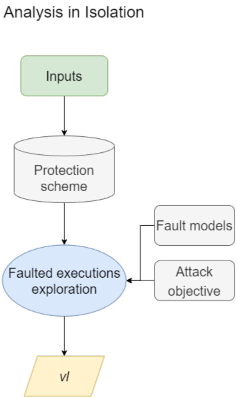
Approach: compositional analysis using:

1. **Isolation analysis** of protection schemes
   → Notion of *adequacy* and *vulnerability level*

2. **Placement algorithms**: select the protection to apply to each IP in the program
   → Using a representative set of attacks on the program wrt to *M*

Outline

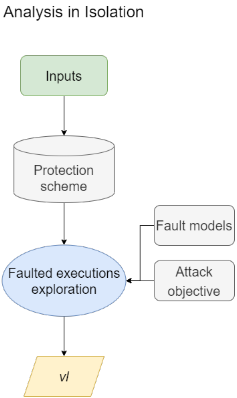
Principle of analysis in isolation



Analysis in isolation: reusable analysis of multi-fault behavior of protection scheme

- *Single fault:* verify that the protection scheme correctly blocks successful attacks for the fault model $m \in M$ (**adequacy**), with m the fault model of the unprotected IP

Principle of analysis in isolation



Analysis in isolation: reusable analysis of multi-fault behavior of protection scheme

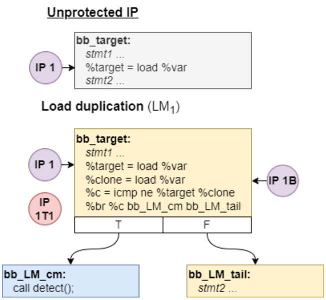
- *Single fault:* verify that the protection scheme correctly blocks successful attacks for the fault model $m \in M$ (**adequacy**), with m the fault model of the unprotected IP
- *Multi fault:* research of the **vulnerability level** (vl) of the protection scheme:

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

  $\rightarrow$ Unprotected IP has $vl = 1$

  $\rightarrow$ Can be computed with Lazart

Analysis in isolation of Load Duplication scheme



Load Duplication: duplication of a `load` instruction

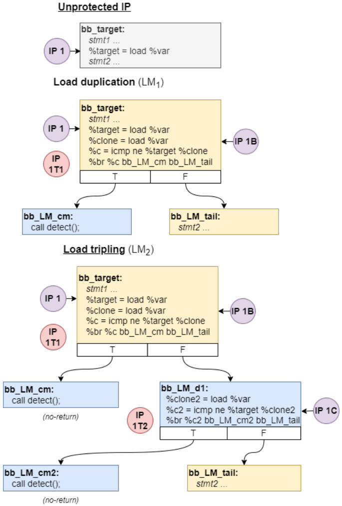
Isolation analysis with *Test Inversion* and *Data Load* fault models

- Explore all faulted paths inside the *Protection Scheme*, using symbolic entries (`%var`), $M = \{TI, DL\}$ and $\phi =$ `%target` stores $v \neq$ `%var`:
  - $T_S(P, M)$: successful undetected attacks
  - $T_C(P, M)$: detected attacks
  - $T_n(P, M)$: nominal case
  - error cases are in T_S or T_C depending on the user

- **Vulnerability Level**: Search of the minimal number of faults required to invalidate the nominal behavior

  $\rightarrow vl =$ minimum number of faults in $T_s(P, M)$

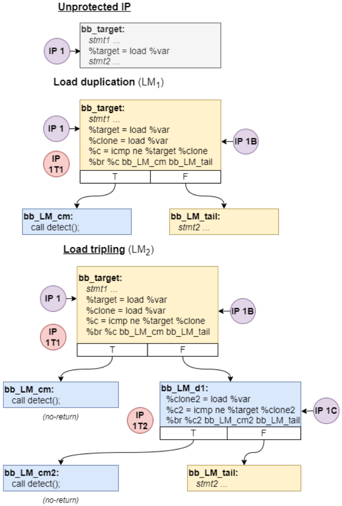
Vulnerability Level (*vl*) for Load Multiplication



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

Table: Vulnerability Level of LM_n

Vulnerability Level (*vl*) for Load Multiplication



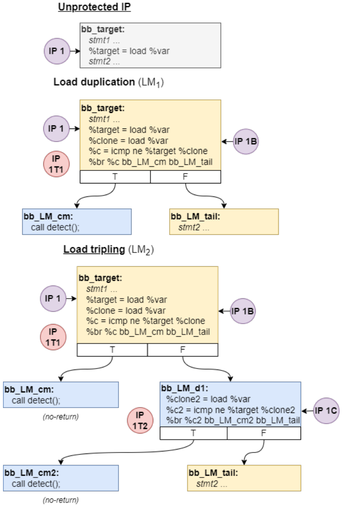
Countermeasure	0-faults	1-fault	2-faults	3-faults	*vl*
LM_0	0	1	0	0	**1**
LM_1	0	0	1	0	**2**
LM_2	0	0	0	1	**3**

Table: Vulnerability Level of LM_n

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

Table: Vulnerability Level of BM_n

Vulnerability Level (*vl*) for Load Multiplication



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

Table: Vulnerability Level of LM_n

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

Table: Vulnerability Level of BM_n

The countermeasures BM_n and TM_n have $vl = 1 + n$ (verified for $n \leq 4$ with Lazart)

Outline

Systematic placement algorithms

Table: Principle of each placement algorithms

Approach	Algorithm	Description
Systematic	**naive**	**All IPs in P are protected with $vl > N$**
Systematic	`atk`	All IPs in attacks are protected with $vl > N$
Systematic	`min`	All IPs in minimal attacks are protected with $vl > N$
Block	`block`	At least one IP per minimal attacks is protected with $vl > N$
Distributed	`opt`	Protection is distributed between the IPs in minimal attacks, to get rid of attacks in less than $N + 1$ faults.

Naive placement algorithm (`naive`): protect **all** IPs in the program with $vl > N$:

1. Compute **required vulnerability levels** (vl_{ip}) for each IP (initialized to 1)
2. Generate P' with protection scheme matching the **required vulnerability levels**

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

$\rightarrow$ *corresponds to standard systematic protection tools*

$\rightarrow$ does not require attacks paths

Systematic placement algorithms

Table: Principle of each placement algorithms

Approach	Algorithm	Description
Systematic	**naive**	**All IPs in P are protected with $vl > N$**
Systematic	`atk`	All IPs in attacks are protected with $vl > N$
Systematic	`min`	All IPs in minimal attacks are protected with $vl > N$
Block	`block`	At least one IP per minimal attacks is protected with $vl > N$
Distributed	`opt`	Protection is distributed between the IPs in minimal attacks, to get rid of attacks in less than $N + 1$ faults.

Naive placement algorithm (`naive`): protect **all** IPs in the program with $vl > N$:

1. Compute **required vulnerability levels** (vl_{ip}) for each IP (initialized to 1)
2. Generate P' with protection scheme matching the **required vulnerability levels**

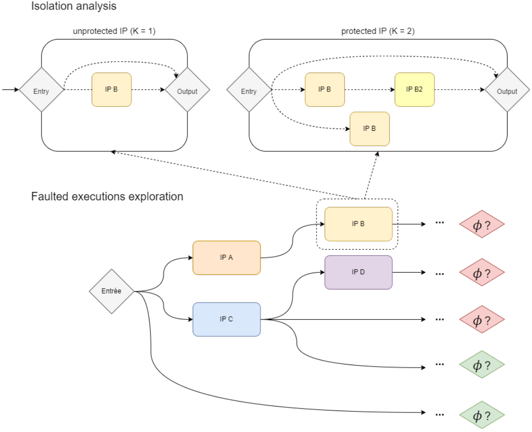
$\Rightarrow$ Using a catalog $\mathcal{C}$ of countermeasures (with computed vl_{ip})

$\rightarrow$ *corresponds to standard systematic protection tools*

$\rightarrow$ does not require attacks paths

$\Rightarrow$ **Use exploration of attack ($T_s(P, M)$) on P, with user-defined ϕ**

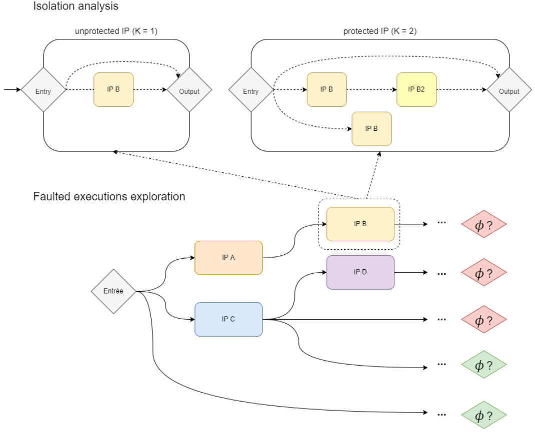
Compositional analysis placement



- Isolation analysis for each considered protection scheme with all studied fault models
- Attacks path exploration on P gives guarantees on which IP violation can lead to an attack

  $\rightarrow$ *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 path exploration on P gives guarantees on which IP violation can lead to an attack

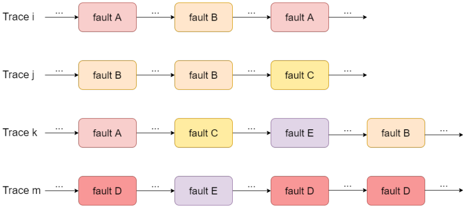
  $\rightarrow$ *Here, IPA can be left unprotected if IPB is protected*

$\Rightarrow$ **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
  $\rightarrow$ usable if the catalog $\mathcal{C}$ does not contains CM for $vl > N$

- An Integer Linear Programming (ILP) optimization problem
  $\rightarrow$ attacks gives constraints on the protection to apply



$C_i = 2x_a + x_b > N$

$C_j = 2x_b + x_c > N$

$C_k = x_a + x_b + x_c + x_e > N$

$C_m = 3x_d + x_e > N$

Research of the **optimal** placement
$\Rightarrow$ 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
  $\rightarrow$ *guarantees on partially protected IPs*

Outline

Experimentation - verify_pin

verify_pin [Dureuil et al., 2016] (**VP**): smart-card PIN verification process

- *fault model:* Test Inversion (TI)

Exp.			Algo.	$\sum$ of protections				Robust
Program	Fault Model	IPs		1-fault	2-faults	3-faults	4-faults	
`VP`	TI	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 - `memcmps3`

memcmps v3 (**MCMPS**): secure version of `memcmp`.

- *fault models:* Test Inversion (TI) + Data Load (DL)

Exp.			Algo.	$\sum$ of protections				Robust
Program	Fault Model	IPs		1-fault	2-faults	3-faults	4-faults	
MCMPS	TI	12	`naive`	12	24	36	48	✓
			`atk`	**0**	**0**	**0**	16	✓
			`min`	**0**	**0**	**0**	16	✓
			`block`	**0**	**0**	**0**	4	✓
			`opt`	**0**	**0**	**0**	**1**	✓
MCMPS	DL	15	`naive`	15	30	45	60	✓
			`atk`	**1**	6	15	32	✓
			`min`	**1**	6	15	32	✓
			`block`	**1**	4	6	8	✓
			`opt`	**1**	**3**	**5**	**7**	✓
MCMPS	TI + DL	27	`naive`	27	54	81	108	✓
			`atk`	**1**	8	24	56	✓
			`min`	**1**	8	24	56	✓
			`block`	**1**	6	9	12	✓
			`opt`	**1**	**3**	**5**	**8**	✓

Experimentations - FU1

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

- *fault models:* Test Inversion (TI) + Data Load (DL)

Exp.			Algo.	$\sum$ of protections				Robust
Program	Fault Model	IPs		1-fault	2-faults	3-faults	4-faults	
`fu1`	TI	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**	✓
	TI+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**	✓

Outline

Conclusion and Future Work

Conclusion:

- Isolation analysis allows to reason about unprotected and protected IP out of the context of a particular program

  → vulnerability level quantifies guarantees of the CM wrt a set of fault models

Conclusion and Future Work

Conclusion:

- Isolation analysis allows to reason about unprotected and protected IP out of the context of a particular program

  $\rightarrow$ vulnerability level quantifies guarantees of the CM wrt a set of fault models

- Placement algorithms gives strong guarantees, even if the trace set is incomplete

  $\rightarrow$ optimality of the placement guaranteed by ILP

Conclusion and Future Work

Conclusion:

- Isolation analysis allows to reason about unprotected and protected IP out of the context of a particular program

  $\rightarrow$ vulnerability level quantifies guarantees of the CM wrt a set of fault models
- Placement algorithms gives strong guarantees, even if the trace set is incomplete

  $\rightarrow$ optimality of the placement guaranteed by ILP

Future Work:

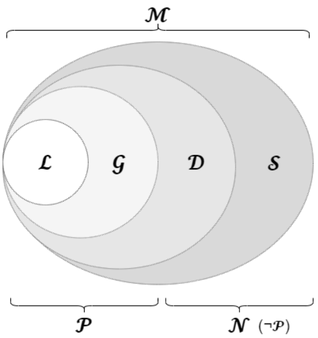
- Study of countermeasures propagating states (SSCF, Swift...)

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

  $\rightarrow$ requires to take into account the several entry and output points of the protection scheme
- Implementation of the approach on binary level

Lazart is planned to be released open-source (Nov 2023)

Future Work - Model protectability



Fault models

$\mathcal{P}$: Protectable

- $\mathcal{L}$: Locally Protectable
- $\mathcal{G}$: Globaly Protectable

$\mathcal{N}$: Unprotectable

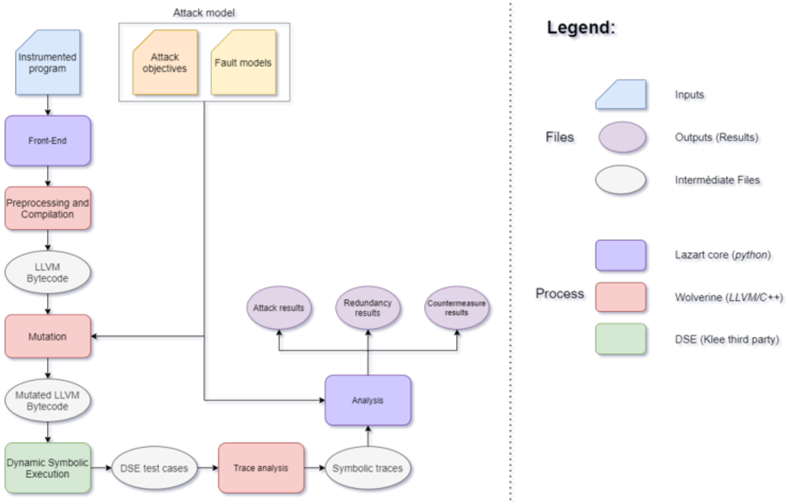
- $\mathcal{D}$: Diluable
- $\mathcal{S}$: Strictly unprotectable

- $\mathcal{L}$: it exists an IP granularity countermeasures with $vl > N$ for all $N > 1$ (Test Inversion, Data Load mutation)
- $\mathcal{G}$: $\exists cm$ such as $cm(P)$ is robust in N faults
- $\mathcal{D}$: $\nexists cm$ such as $cm(P)$ is robust in N faults, but the attacks can be made more difficult
- $\mathcal{S}$: even making the attack more difficult is not possible [Given-Wilson and Legay, 2020]

The End

Thanks for watching

Lazart architecture



Summary

- Robustness of placement depends 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
  $\rightarrow$ if not, attack traces on P' are known
  $\rightarrow$ more robust than P even if trace set is incomplete
- Protection weight: *distributed* $\leq$ *block* $\leq$ *min* $\leq$ *atk* $\leq$ *naive*
  $\rightarrow$ Optimal placement is guaranteed with ILP

Algorithm	Type	Guarantees P' Robust	Optimal	Complexity	Required analysis 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)
  $\rightarrow$ even with optimal algorithm and ILP (1-fault attacks)

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;
}
```

References I

de Ferrière, F. (2019).
A compiler approach to cyber-security.
2019 European LLVM developers' meeting.

Dureuil, L., Petiot, G., Potet, M., Le, T., Crohen, A., and de Choudens, P. (2016).
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.

Given-Wilson, T. and Legay, A. (2020).
Formalising fault injection and countermeasures.
In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–11.

Kim, C. H. and Quisquater, J.-J. (2007).
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.

Lalande, J., Heydemann, K., and Berthomé, P. (2014).
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.

References II

Natella, R., Cotroneo, D., and Madeira, H. S. (2016).

Assessing Dependability with Software Fault Injection: A Survey.

ACM Computing Surveys, 48(3):1–55.

Potet, M.-L., Mounier, L., Puys, M., and Dureuil, L. (2014).

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.

Wookey/SSTIC20 (2020).

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.