Using a Vulnerability Assessment Methodology to build and improve Countermeasures against Multi-Fault Injection

Bruno Ferres[1], Louis Sassier, Etienne Boespflug

with Marie-Laure Potet and Laurent Mounier

[1] bruno.ferres@univ-grenoble-alpes.fr

JAIF2025@Grenoble
October 1st, 2025

Plan

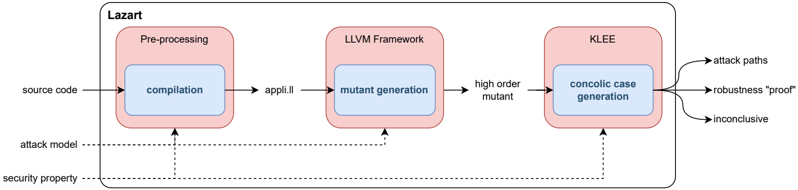
1. Overview of the Methodology

2. Case Study: Protecting Control-Flow against Multiple Faults

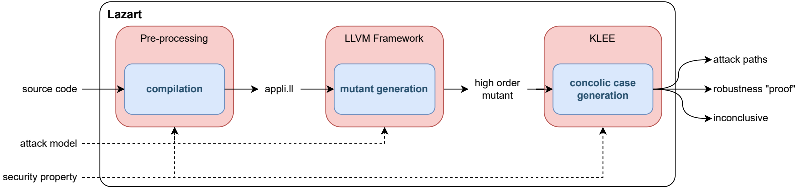
3. Proposing the CFIStack

4. Conclusion and Perspectives

Recalls on Lazart



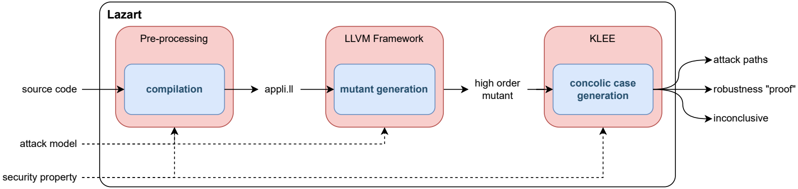
Recalls on Lazart



Multi-fault Analysis

- pre-defined fault models:
  - data load mutation (**DL**)
  - test inversion (**TI**)
  - switch call (**SC**)
  - no call (**NC**)

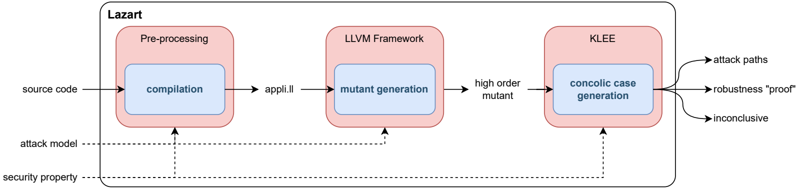
Recalls on Lazart



Multi-fault Analysis

- pre-defined fault models:
  - data load mutation (**DL**)
  - test inversion (**TI**)
  - switch call (**SC**)
  - no call (**NC**)
- **symbolic execution** with fault budget

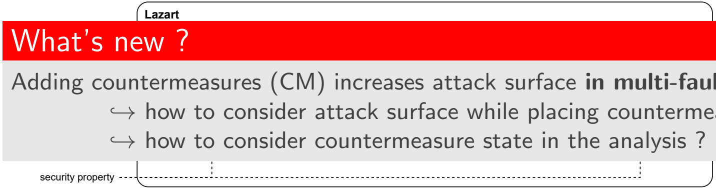
Recalls on Lazart



Multi-fault Analysis

- pre-defined fault models:
  - data load mutation (**DL**)
  - test inversion (**TI**)
  - switch call (**SC**)
  - no call (**NC**)
- **symbolic execution** with fault budget
- objective: help developpers to identify vulnerabilities

Recalls on Lazart



What's new ?

Adding countermeasures (CM) increases attack surface **in multi-fault**. . .

↪ how to consider attack surface while placing countermeasures ?

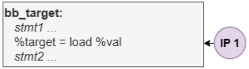
↪ how to consider countermeasure state in the analysis ?

Multi-fault Analysis

- pre-defined fault models:
  - data load mutation (**DL**)
  - test inversion (**TI**)
  - switch call (**SC**)
  - no call (**NC**)
- **symbolic execution** with fault budget
- objective: help developpers to identify vulnerabilities

Protecting Sensitive Schemes

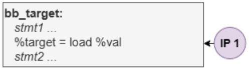
Sensitive Scheme



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

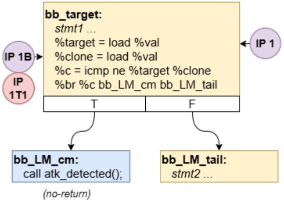
Protecting Sensitive Schemes

Sensitive Scheme



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

Load Duplication



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

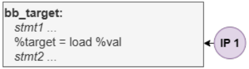
```
bb_LM_cm:
  call atk_detected();
```

(no-return)

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

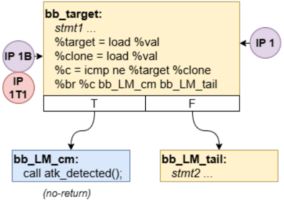
Protecting Sensitive Schemes

Sensitive Scheme



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

Load Duplication



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

```
bb_LM_cm:
  call atk_detected();
```

(no-return)

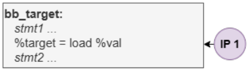
```
bb_LM_tail:
  stmt2 ...
```

```
klee_make_symbolic(&val);
Load_Dupl(val, &target);
oracle(!(target == val));
```

Oracle for load duplication

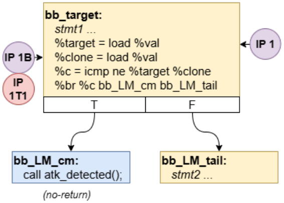
Protecting Sensitive Schemes

Sensitive Scheme



bb_target:
stmt1 ...
%target = load %val
stmt2 ...

Load Duplication



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

bb_LM_cm:
call atk_detected();

(no-return)

bb_LM_tail:
stmt2 ...

```
klee_make_symbolic(&val);
Load_Dupl(val, &target);
oracle(!(target == val));
```

Oracle for load duplication

Analysis Principle

- consider only **nominal behavior** of sensitive scheme (*i.e.* injection points IP) ↪ not of program P
- either preserve nominal, or detect attack

Studying Counter-Measures in Isolation

Robustness levels

Use **symbolic execution** to compute a **robustness level** (*rl*) against **multi-fault attacks**.

Studying Counter-Measures in Isolation

Robustness levels

Use **symbolic execution** to compute a **robustness level** (*rl*) against **multi-fault attacks**.

	Fault model					
	Test inversion		Load modification		Combination	
Countermeasure	*rl*	*Vuln*	*rl*	*Vuln*	*rl*	*Vuln*
Test duplication	1	2	-	-	1	2
Load duplication	-	-	1	1	1	2
Load triplication	-	-	2	1	2	4

Robustness levels of countermeasure schemes (for `max_order=4`)

Vuln: number of attack paths found with **rl + 1** faults

Studying Counter-Measures in Isolation

Robustness levels

Use **symbolic execution** to compute a **robustness level** (*rl*) against **multi-fault attacks**.

	Fault model					
Countermeasure	**Test inversion**		**Load modification**		**Combination**	
	rl	*Vuln*	*rl*	*Vuln*	*rl*	*Vuln*
Test duplication	1	2	-	-	1	2
Load duplication	-	-	1	1	1	2
Load triplication	-	-	2	1	2	4

Robustness levels of countermeasure schemes (for `max_order=4`)

Vuln: number of attack paths found with **rl + 1** faults

↪ **automatic placement of counter-measures against multi-fault**

Plan

1 Overview of the Methodology

2 Case Study: Protecting Control-Flow against Multiple Faults

3 Proposing the CFIStack

4 Conclusion and Perspectives

Minimal Working Example

```
bool is_authentified = false;

void authentify() {
    is_authentified = true;
}

void try_authentify() {
    if (check_password()) {
        authentify();
    }
}

void caller() {
    try_authentify();
}
```

Minimal Working Example

```
bool is_authentified = false;

void authentify() {
    is_authentified = true;
}

void try_authentify() {
    if (check_password()) {
        authentify();
    }
}

void caller() {
    try_authentify();
}
```

Requirements

- ensure **control-flow integrity**

  ↪ both forward and backward edges

  ⇒ check both call and return addresses
- be robust against multiple fault models
  - DL (data load modification)
  - TI (test inversion)
  - SC (switch call)
  - NC (no call)
- handle composition (multiple calls)

Minimal Working Example

```
bool is_authentified = false;

void authentify() {
    is_authentified = true;
}

void try_authentify() {
    if (check_password()) {
        authentify();
    }
}

void caller() {
    try_authentify(); // expected CF
}
```

Requirements

- ensure **control-flow integrity**
  ↪ both forward and backward edges
  ⇒ check both call and return addresses
- be robust against multiple fault models
  - DL (data load modification)
  - TI (test inversion)
  - SC (switch call)
  - NC (no call)
- handle composition (multiple calls)

Minimal Working Example

```
bool is_authentified = false;

void authentify() {
    is_authentified = true;
}

void try_authentify() {
    if (check_password()) {
        authentify();
    }
}

void caller() {
    try_authentify(); // expected CF
}
```

Requirements

- ensure **control-flow integrity**

  ↪ both forward and backward edges

  ⇒ check both call and return addresses
- be robust against multiple fault models
  - DL (data load modification)
  - TI (test inversion)
  - SC (switch call)
  - NC (no call)
- handle composition (multiple calls)

Minimal Working Example

```
bool is_authentified = false;

void authentify() {
    is_authentified = true;
}

void try_authentify() {
    if (check_password()) {
        authentify();
    }
}

void caller() {
    try_authentify(); //applying SC
}
```

Requirements

- ensure **control-flow integrity**

  ↪ both forward and backward edges

  ⇒ check both call and return addresses
- be robust against multiple fault models
  - DL (data load modification)
  - TI (test inversion)
  - SC (switch call)
  - NC (no call)
- handle composition (multiple calls)

Minimal Working Example

```
bool is_authentified = false;

void authentify() {
    is_authentified = true;
}

void try_authentify() {
    if (!check_password()) { //TI
        authentify();
    }
}

void caller() {
    try_authentify();
}
```

Requirements

- ensure **control-flow integrity**

  ↪ both forward and backward edges

  ⇒ check both call and return addresses
- be robust against multiple fault models
  - DL (data load modification)
  - TI (test inversion)
  - SC (switch call)
  - NC (no call)
- handle composition (multiple calls)

Plan

1 Overview of the Methodology

2 Case Study: Protecting Control-Flow against Multiple Faults

3 Proposing the CFIStack

4 Conclusion and Perspectives

Protecting Both Directions with CFIStack

Main Idea

- store **function signature** and **return address** (inspired from **SecSwift** and **shadow stacks**)

↪ check that:

1. we call the **right function**;
2. we jump back to the **right location**

Protecting Both Directions with CFIStack

Main Idea

- store **function signature** and **return address** (inspired from **SecSwift** and **shadow stacks**)

↪ check that:

1. we call the **right function**;
2. we jump back to the **right location**

- using a dedicated stack structure (memory + pointer)

  ↪ with primitives to ensure **integrity**

  - ▶ `push`, `peek_and_check`, `pop`, `check_current`

Protecting Both Directions with CFIStack

Main Idea

- store **function signature** and **return address** (inspired from **SecSwift** and **shadow stacks**)

↪ check that:

1. we call the **right function**;
2. we jump back to the **right location**

- using a dedicated stack structure (memory + pointer)

  ↪ with primitives to ensure **integrity**

  - push, peek_and_check, pop, check_current

```
void try_authentify (addr ret_caller) {
    peek_and_check(sig_try_authentify);
    pop();
    ... // function body
    peek_and_check(ret_caller);
    return;
}

void caller() {
    int mem_current = current;
    push(ret_caller);
    push(sig_try_authentify);
    try_authentify(ret_caller);
ret_caller:
    pop();
    check_current(mem_current);
    ... // remaining code
}
```

*protection

Protecting Both Directions with CFIStack

Main Idea

- store **function signature** and **return address** (inspired from **SecSwift** and **shadow stacks**)

↪ check that:

1. we call the **right function**;
2. we jump back to the **right location**

- using a dedicated stack structure (memory + pointer)

  ↪ with primitives to ensure **integrity**

  - push, peek_and_check, pop, check_current

```
void try_authentify (addr ret_caller) {
     peek_and_check(sig_try_authentify);
     pop();
     ... // function body
     peek_and_check(ret_caller);
     return;
}

void caller() {
     int mem_current = current;
     push(ret_caller);
     push(sig_try_authentify);
     try_authentify(ret_caller);
ret_caller:
     pop();
     check_current(mem_current);
     ... // remaining code
}
```

*protection *duplicable

Protecting Both Directions with CFIStack

Main Idea

- store **function signature** and **return address** (inspired from **SecSwift** and **shadow stacks**)

↪ check that:

1. we call the **right function**;
2. we jump back to the **right location**

- using a dedicated stack structure (memory + pointer)

  ↪ with primitives to ensure **integrity**
  - ▶ `push`, `peek_and_check`, `pop`, `check_current`

<u>Oracle for CFIStack Nominal Behavior:</u>

```
old_cfi_current == cfi_current &&
cmp(cfi_stack, old_cfi_stack, STACK_SIZE)
```

```
void try_authentify (addr ret_caller) {
    peek_and_check(sig_try_authentify);
    pop();
    ... // function body
    peek_and_check(ret_caller);
    return;
}

void caller() {
    int mem_current = current;
    push(ret_caller);
    push(sig_try_authentify);
    try_authentify(ret_caller);
ret_caller:
    pop();
    check_current(mem_current);
    ... // remaining code
}
```

*protection *duplicable

Design Choices

Nothing really new. . .

Design Choices

Nothing really new. . .

. . . but we can play with it !

Design Choices

Nothing really new. . . . . . but we can play with it !

$\hookrightarrow$ we can add **hypotheses** ($\simeq$ contracts) on the hardware, using `Lazart`.

Design Choices

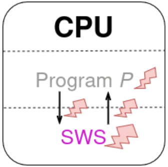
Nothing really new. . . . . . but we can play with it !

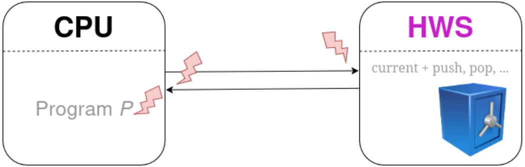
↪ we can add **hypotheses** ($\simeq$ contracts) on the hardware, using `Lazart`.

Proposing SWS *vs.* HWS

- **SWS** (soft): primitives (`push`, . . .) **logic** are vulnerable (*e.g.* TI, DL)
- **HWS** (hard): primitives logic (+ `current`) are considered secure

↪ just change the `Lazart` setup (mainly, IPs for **TI** and **DL**)





Comparing Robustness Levels

Version	Fault model	#attacks		Robustness
		1F	2F	level
SWS	SC		2	1
	SC + DL	132+		**0**
	SC + TI	3		**0**
	SC + DL + TI	135+		**0**
HWS	SC		2	1
	SC + DL		4	1

SWS: Software CFIStack / **HWS**: Hardware CFIStack

Fault models:

- **SC**: Switch Call
- **TI**: Test Inversion
- **DL**: Data Load modification

Comparing Robustness Levels

Version	Fault model	#attacks		Robustness
		1F	2F	level
SWS	SC		2	1
	SC + DL	132+		**0**
	SC + TI	3		**0**
	SC + DL + TI	135+		**0**
HWS	SC		2	1
	SC + DL		4	1

Version	Fault model	#attacks		Robustness
		1F	2F	level
PSWS	SC		2	1
	SC + DL		7+	1
	SC + TI		7+	1
	SC + DL + TI		10+	1

PSWS: Protected SWS (with our methodo.)
↪ add **load-** and **test-duplication** to SWS

SWS: Software CFIStack / **HWS**: Hardware CFIStack

Fault models:

- **SC**: Switch Call
- **TI**: Test Inversion
- **DL**: Data Load modification

Comparing Robustness Levels

Version	Fault model	#attacks		Robustness
		1F	2F	level
SWS	SC		2	1
	SC + DL	132+		**0**
	SC + TI	3		**0**
	SC + DL + TI	135+		**0**
HWS	SC		2	1
	SC + DL		4	1

Version	Fault model	#attacks		Robustness
		1F	2F	level
PSWS	SC		2	1
	SC + DL		7+	1
	SC + TI		7+	1
	SC + DL + TI		10+	1

PSWS: Protected SWS (with our methodo.)
↪ add **load-** and **test-duplication** to SWS

SWS: Software CFIStack / **HWS**: Hardware CFIStack

Fault models:

- **SC**: Switch Call
- **TI**: Test Inversion
- **DL**: Data Load modification

Observations

- HW guarantees impact dev. and analysis
- CM obtained in isolation can be re-used
  ↪ at least for **sequential calls**

Comparing Robustness Levels

Version	Fault model	#attacks		Robustness
		1F	2F	level
SWS	SC		2	1
	SC + DL	132+		**0**
	SC + TI	3		**0**
	SC + DL + TI	135+		**0**
HWS	SC		2	1
	SC + DL		4	1

Version	Fault model	#attacks		Robustness
		1F	2F	level
PSWS	SC		2	1
	SC + DL		7+	1
	SC + TI		7+	1
	SC + DL + TI		10+	1

PSWS: Protected SWS (with our methodo.)
↪ add **load-** and **test-duplication** to SWS

SWS: Software CFIStack / **HWS**: Hardware CFIStack

Fault models:

- **SC**: Switch Call
- **TI**: Test Inversion
- **DL**: Data Load modification

Observations

- HW guarantees impact dev. and analysis
- CM obtained in isolation can be re-used
  ↪ at least for **sequential calls**
- **but can we do better ?**

Feedback and Distribution

Insightful Feedback

Proposed methodology can be used to:

- identify **hotspots** to harden
  ↪ and which ones to consider first

Feedback and Distribution

Insightful Feedback

Proposed methodology can be used to:

- identify **hotspots** to harden
  ↪ and which ones to consider first
- model **hardware guarantees**/requirements
  ↪ as assumptions on the C code

Feedback and Distribution

Insightful Feedback

Proposed methodology can be used to:

- identify **hotspots** to harden
  ↪ and which ones to consider first
- model **hardware guarantees**/requirements
  ↪ as assumptions on the C code
- select **countermeasures** to use
  ↪ placement algorithms

Feedback and Distribution

Insightful Feedback

Proposed methodology can be used to:

- identify **hotspots** to harden
  ↪ and which ones to consider first
- model **hardware guarantees**/requirements
  ↪ as assumptions on the C code
- select **countermeasures** to use
  ↪ placement algorithms

Try it yourself !

Both methodology and example documented in `Lazart` wiki:
https://gricad-gitlab.univ-grenoble-alpes.fr/securitytools/lazart/-/wikis/Countermeasure-hardening-tutorial

Plan

1 Overview of the Methodology

2 Case Study: Protecting Control-Flow against Multiple Faults

3 Proposing the CFIStack

4 Conclusion and Perspectives

Conclusion and Perspectives

Proposed Methodology

- study counter-measures in isolation
- automatic placement algorithms
- can be used to **design countermeasures**
  ↪ including complex CMs with state
- can model **hardware guarantees** by adapting fault models
  ↪ to study variations and/or prototype

Conclusion and Perspectives

Proposed Methodology

- study counter-measures in isolation
- automatic placement algorithms
- can be used to **design countermeasures**
  ↪ including complex CMs with state
- can model **hardware guarantees** by adapting fault models
  ↪ to study variations and/or prototype

Perspectives

- experiment more thoroughly on **imbricated calls**
- **formalize the requirements between HW and SW**
- **try to model more complex countermeasures**

Conclusion and Perspectives

Proposed Methodology

- study counter-measures in isolation
- automatic placement algorithms
- can be used to **design countermeasures**
  ↪ including complex CMs with state
- can model **hardware guarantees** by adapting fault models
  ↪ to study variations and/or prototype

Perspectives

- experiment more thoroughly on **imbricated calls**
- **formalize the requirements between HW and SW**
- **try to model more complex countermeasures**
- study similar usecases **at binary level** (RAR TwinSec)

Conclusion and Perspectives

Proposed Methodology

- study counter-measures in isolation
- automatic placement algorithms
- can be used to **design countermeasures**
  ↪ including complex CMs with state
- can model **hardware guarantees** by adapting fault models
  ↪ to study variations and/or prototype

Perspectives

- experiment more thoroughly on **imbricated calls**
- **formalize the requirements between HW and SW**
- **try to model more complex countermeasures**
- study similar usecases **at binary level** (RAR TwinSec)

To be submitted:

“A Tool-Assisted Methodology to Harden Programs Against Multi-Faults: adding and designing Countermeasures”

Conclusion and Perspectives

Proposed Methodology

- study counter-measures in isolation
- automatic placement algorithms
- can be used to **design countermeasures**
  ↪ including complex CMs with state
- can model **hardware guarantees** by adapting fault models
  ↪ to study variations and/or prototype

Any questions ?

Perspectives

- experiment more thoroughly on **imbricated calls**
- **formalize the requirements between HW and SW**
- **try to model more complex countermeasures**
- study similar usecases **at binary level** (RAR TwinSec)

To be submitted:

“A Tool-Assisted Methodology to Harden Programs Against Multi-Faults: adding and designing Countermeasures”

BACKUPS

Please don't be mean

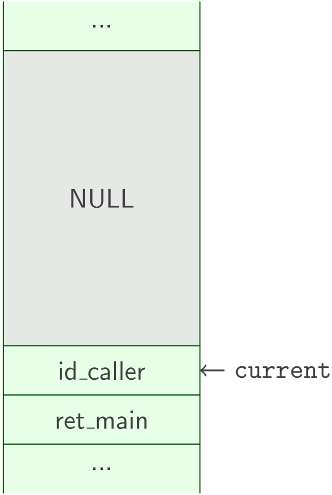
Need to Preserve the Whole Stack ?

```
bool is_authentified = false;

void authentify() {
    is_authentified = true;
}

void try_authentify() {
    if (check_password()) {
        authentify();
    }
}

void caller() {
    try_authentify();
}
```



...

NULL

id_caller ← current

ret_main

...

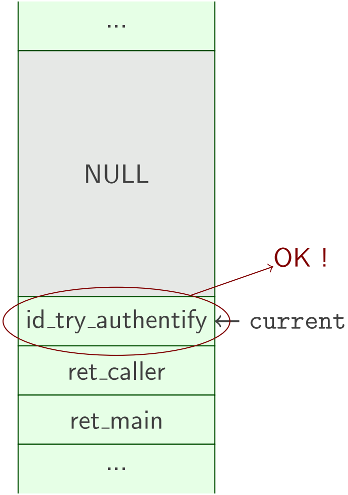
Need to Preserve the Whole Stack ?

```
bool is_authentified = false;

void authentify() {
    is_authentified = true;
}

void try_authentify() {
    if (check_password()) {
        authentify();
    }
}

void caller() {
    try_authentify();
}
```



...

NULL

OK !

id_try_authentify ← current

ret_caller

ret_main

...

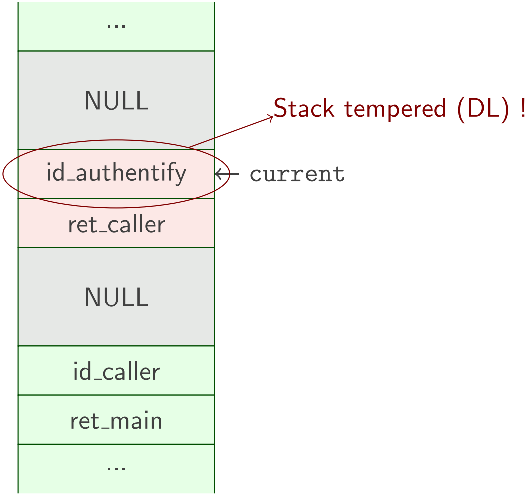
Need to Preserve the Whole Stack ?

```
bool is_authentified = false;

void authentify() {
    is_authentified = true;
}

void try_authentify() {
    if (check_password()) {
        authentify();
    }
}

void caller() {
    try_authentify();
}
```



...

NULL

id_authentify

ret_caller

NULL

id_caller

ret_main

...

Stack tempered (DL) !

current

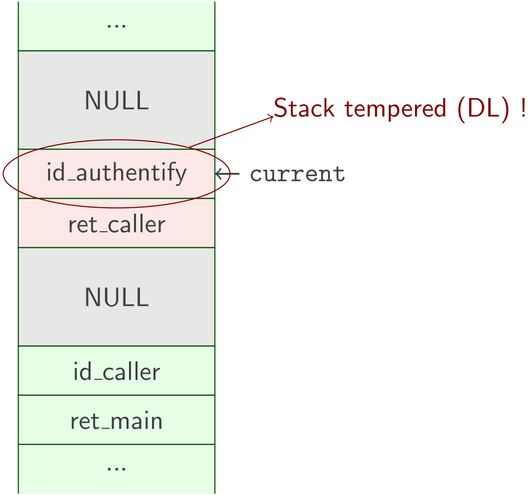
Need to Preserve the Whole Stack ?

```
bool is_authentified = false;

void authentify() {
    is_authentified = true;
}

void try_authentify()
    if (check_password()) {
        authentify();
    }
}

void caller() {
    try_authentify();
}
```



...

NULL

id_authentify

ret_caller

NULL

id_caller

ret_main

...

Stack tempered (DL) !

current

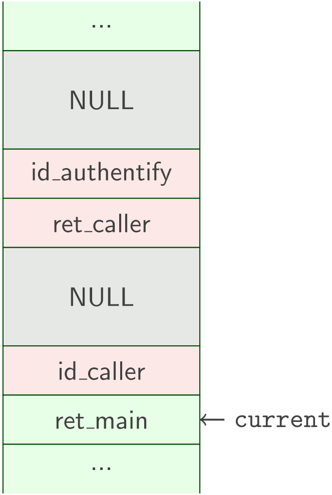
Need to Preserve the Whole Stack ?

```
bool is_authentified = false;

void authentify() {
    is_authentified = true;
}

void try_authentify() {
    if (check_password()) {
        authentify();
    }
}

void caller() {
    try_authentify();
}
```


...

NULL

id_authentify

ret_caller

NULL

id_caller

ret_main ← current

...

Need to Preserve the Whole Stack ?

```
bool is_authentified = false;

void authentify() {
    is_authentified = true;
}
```

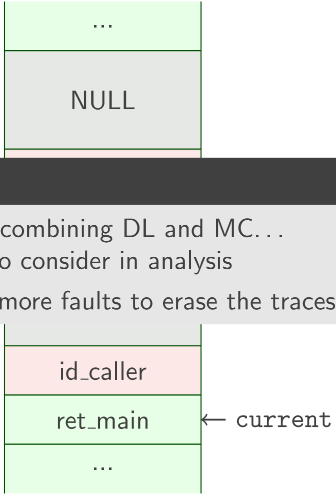
Why do we need such a strong oracle ?

In a multi-fault context, we might temper the stack by combining DL and MC...
↪ could lead to erroneous control-flow, that we need to consider in analysis

↪ by checking the whole stack, it would require twice more faults to erase the traces

```
}

void caller() {
    try_authentify();
}
```



Need to Preserve the Whole Stack ?

```
bool is_authentified = false;

void authentify() {
    is_authentified = true;
}
```

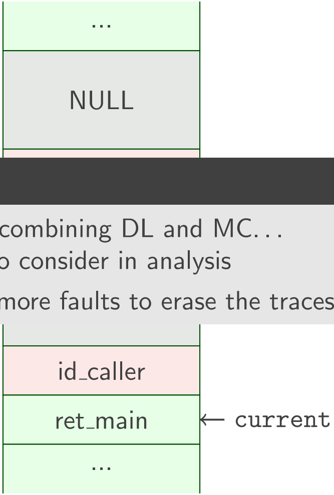
Why do we need such a strong oracle ?

In a multi-fault context, we might temper the stack by combining DL and MC...
↪ could lead to erroneous control-flow, that we need to consider in analysis

↪ by checking the whole stack, it would require twice more faults to erase the traces

```
}

void caller() {
    try_authentify();
}
```



Considered Fault Models for CFIStack Analysis

Variant	Fault model	Target	Function
SWS	SC/NC	try_authentify push pop check_current	caller()
		peek_and_check pop	try_authentify()

	TI	-	pop()
			push()
			peek_and_check()

	DL	mem_current sig_try_authentify	caller()
		sig_authentify	authentify()

		current	pop()
			push()

Variant	Fault model	Target	Function
HWS	SC/NC	try_authentify check_current pop	caller()
	TI	-	-
	DL	mem_current sig_try_authentify	caller()
		sig_authentify	authentify()

Comparing Robustness Levels with NC and SC

Version	Fault model	#attacks 1F	#attacks 2F	Robustness level
SWS	SC		2	1
	SC + DL	132+		**0**
	SC + TI	3		**0**
	SC + DL + TI	135+		**0**
	NC		1	1
	NC + DL	72+		**0**
	NC + TI	3		**0**
	NC + DL + TI	75+		**0**
Protected SWS	SC		2	1
	SC + DL		7+	1
	SC + TI		7+	1
	SC + DL + TI		10+	1
	NC		1	1
	NC + DL		5+	1
	NC + TI		5+	1
	NC + DL + TI		10+	1

Version	Fault model	#attacks 1F	#attacks 2F	Robustness level
HWS	SC		2	1
	SC + DL		4	1
	NC		1	1
	NC + DL		2	1

Toward n-Robustness

Version	Fault model	#IP to protect for 2-robustness
HWS	NC	1
	NC + DL	1
	SC	3
	SC + DL	3
PSWS	NC	1
	NC + DL	3
	NC + TI	4
	NC + TI + DL	6
	SC	3
	SC + DL	5
	SC + TI	6
	SC + TI + DL	8

Insights from the methodology

- which IP to protect ?
- which CM to use ?
- try to **minimize** the protection while **ensuring robustness**

Some feedback to harden using CFIStack variants