

Study of Rust's attack surface against multiple fault injection

Etienne Boespflug, Tommy Prats, Timothé Baleras, Laurent Mounier

VERIMAG — Université Grenoble Alpes

JAIF 2026

- Motivation
- The Rust language
- Lazart for Rust
- FISSC in Rust
- The attack surface of idiomatic Rust
- Conclusion and perspectives

- **Rust:** a systems programming language, increasingly used in embedded and critical software

- **Rust**: a systems programming language, increasingly used in embedded and critical software
- It provides **memory safety by construction**
 - Memory safety is not fault resistance

- **Rust**: a systems programming language, increasingly used in embedded and critical software
- It provides **memory safety by construction**
 - Memory safety is not fault resistance
- Existing work: *fault tolerance*, or a *single* fault [SSV+25; KLL26; Led22]
 - No study of **multi-fault** attacks on Rust to our knowledge

- **Rust**: a systems programming language, increasingly used in embedded and critical software
- It provides **memory safety by construction**
 - **Memory safety is not fault resistance**
- Existing work: *fault tolerance*, or a *single* fault [SSV+25; KLL26; Led22]
 - No study of **multi-fault** attacks on Rust to our knowledge
- **Our contributions**
 - **Lazart for Rust** (and a new **Change Pointer**) fault model, for Rust *and* C

- **Rust**: a systems programming language, increasingly used in embedded and critical software
- It provides **memory safety by construction**
 - **Memory safety is not fault resistance**
- Existing work: *fault tolerance*, or a *single* fault [SSV+25; KLL26; Led22]
 - No study of **multi-fault** attacks on Rust to our knowledge
- **Our contributions**
 - 1 **Lazart for Rust** (and a new **Change Pointer**) fault model, for Rust *and* C
 - 2 A **comparison with C**, on FISSC [DPP+16]

- **Rust**: a systems programming language, increasingly used in embedded and critical software
- It provides **memory safety by construction**
 - **Memory safety is not fault resistance**
- Existing work: *fault tolerance*, or a *single fault* [SSV+25; KLL26; Led22]
 - No study of **multi-fault** attacks on Rust to our knowledge
- **Our contributions**
 - ① **Lazart for Rust** (and a new **Change Pointer**) fault model, for Rust *and* C
 - ② A **comparison with C**, on FISSC [DPP+16]
 - ③ The attack surface of **idiomatic Rust's** constructs, and the checks it emits by default

- **Rust**: a systems programming language, increasingly used in embedded and critical software
- It provides **memory safety by construction**
 - **Memory safety is not fault resistance**
- Existing work: *fault tolerance*, or a *single* fault [SSV+25; KLL26; Led22]
 - No study of **multi-fault** attacks on Rust to our knowledge
- **Our contributions**
 - 1 **Lazart for Rust** (and a new **Change Pointer**) fault model, for Rust *and* C
 - 2 A **comparison with C**, on FISSC [DPP+16]
 - 3 The attack surface of **idiomatic Rust's** constructs, and the checks it emits by default
 - 4 A **Rust-native FISSC**, for the community

- Motivation
- The Rust language
- Lazart for Rust
- FISSC in Rust
- The attack surface of idiomatic Rust
- Conclusion and perspectives

- **Statically typed, compiled** systems language [MK14]
 - Performance and low-level control comparable to C/C++
 - Strong **compile-time** guarantees in safe Rust (no use-after-free, double free, data race)

- **Statically typed, compiled** systems language [MK14]

- Performance and low-level control comparable to C/C++
- Strong **compile-time** guarantees in safe Rust (no use-after-free, double free, data race)

- **Ownership**: every value has a unique owner, whose lifetime the compiler tracks

- Borrows follow **aliasing XOR mutability**: many &T or one &mut T
- Verified statically by the **borrow checker**

```
{  
    let mut v = vec![1, 2, 3]; // v owns the vector  
    let r1 = &v; let r2 = &v; // N readers (&T): OK  
    let w_err = &mut v;      // error[E0502]: v already borrowed  
    dbg!(r1, r2);  
    let w = &mut v;           // 1 writer (&mut T): OK (NLL)  
    w.push(4);  
    let v2 = v;               // ownership moved to v2  
    v.len();                  // error[E0382]: v was moved  
                                // v2 out of scope: vector dropped  
}
```

- **Statically typed, compiled systems language** [MK14]
 - Performance and low-level control comparable to C/C++
 - Strong **compile-time** guarantees in safe Rust (no use-after-free, double free, data race)
- **Ownership**: every value has a unique owner, whose lifetime the compiler tracks
 - Borrows follow **aliasing XOR mutability**: many &T or one &mut T
 - Verified statically by the **borrow checker**
- **Other constructs** reducing safety errors
 - Algebraic data types, exhaustive pattern matching
 - Explicit error handling: Option<T>, Result<T,E>
 - Bounds checks on array and slice accesses

```
enum Cmd { Read(u8), Write(u8, u8), Reset }
```

```
match cmd {  
  Cmd::Read(a)    => read(a),  
  Cmd::Write(a, x) => write(a, x),  
}  
// error[E0004]: Reset not covered
```

```
let key = load_key()?;           // Result<Key, E>: Err returned  
let b: Option<u8> = buf.get(i); // None if i is out of bounds  
let b = buf[i];                 // bounds check: panics if i >= len
```

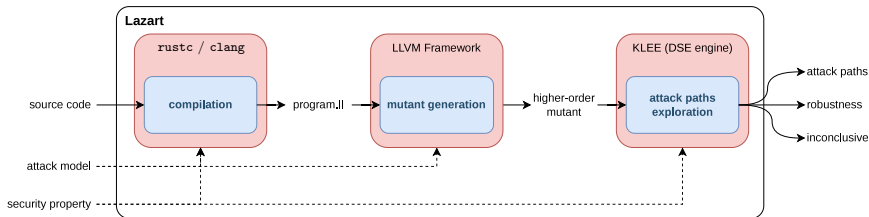
- **Statically typed, compiled systems language** [MK14]
 - Performance and low-level control comparable to C/C++
 - Strong **compile-time** guarantees in safe Rust (no use-after-free, double free, data race)
- **Ownership**: every value has a unique owner, whose lifetime the compiler tracks
 - Borrows follow **aliasing XOR mutability**: many &T or one &mut T
 - Verified statically by the **borrow checker**
- **Other constructs** reducing safety errors
 - Algebraic data types, exhaustive pattern matching
 - Explicit error handling: `Option<T>`, `Result<T,E>`
 - Bounds checks on array and slice accesses
- What the compiler cannot prove safe: mutable aliasing, FFI, low-level access
 - Explicit **unsafe** blocks, audited manually [CSX+23]
 - Safe APIs over unsafe code, formally studied [JJK+17]

```
let p = &mut x as *mut u32; // raw pointer: safe to create
*p = 1;                      // error[E0133]: deref requires unsafe

unsafe {
    *p = 1;                  // aliasing unchecked
    read_volatile(0x4000_0000 as *const u32); // low-level (MMIO)
    memcmp(a.as_ptr(), b.as_ptr(), n);        // FFI: C function
}

// std::Vec<T>: safe API, unsafe implementation
pub fn push(&mut self, value: T) {
    if self.len == self.capacity() { self.grow(); }
    unsafe { ptr::write(self.as_mut_ptr().add(self.len), value); }
    self.len += 1;
}
```

- Motivation
- The Rust language
- **Lazart for Rust**
- FISSC in Rust
- The attack surface of idiomatic Rust
- Conclusion and perspectives



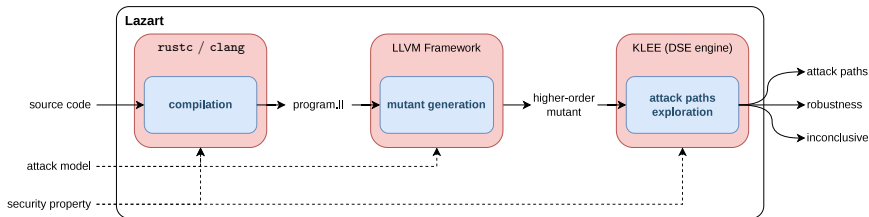
- **Multi-fault analysis** [PMP+14; Boe23] by **dynamic symbolic execution** (KLEE) on **LLVM IR** (which rustc emits)

Test Inversion (TI)

```
br i1 %c, label %else, label %then
```

Data Load Mutation (DL)

```
%n = load i32, i32* %faulted_value, align 4
```



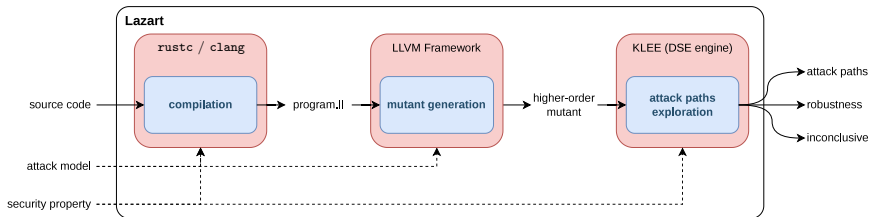
- **Multi-fault** analysis [PMP+14; Boe23] by **dynamic symbolic execution** (KLEE) on **LLVM IR** (which rustc emits)
- Faults are **embedded** inside the mutant: it carries the nominal *and* the faulted behaviour of every injection point
 - One symbolic boolean per IP, bounded by a **fault limit**

Test Inversion (TI)

```
br i1 %c, label %else, label %then
```

Data Load Mutation (DL)

```
%n = load i32, i32* %faulted_value, align 4
```



- **Multi-fault** analysis [PMP+14; Boe23] by **dynamic symbolic execution** (KLEE) on **LLVM IR** (which rustc emits)
- Faults are **embedded** inside the mutant: it carries the nominal *and* the faulted behaviour of every injection point
 - One symbolic boolean per IP, bounded by a **fault limit**
- The user supplies an **attacker model**: fault models, and an **attack objective** (the oracle)
 - Output: **all attack paths** up to the fault limit

Test Inversion (TI)

```
br i1 %c, label %else, label %then
```

Data Load Mutation (DL)

```
%n = load i32, i32* %faulted_value, align 4
```

- **LLVM 9** → **LLVM 14**, and the cargo/rustc pipeline

- LLVM 9 → LLVM 14, and the cargo/rustc pipeline
- **Test Inversion extended** to select and switch
 - ⇒ rustc makes extensive use of select and switch (clang emitted *none*, neither on FISSC nor on Lazard's test suite)

A three-arm match...

```
match c {  
  Color::Red   => 1,  
  Color::Green => 2,  
  Color::Blue  => 3,  
}
```

...lowered to a switch

```
switch i8 %disc, label %unreachable [  
  i8 0, label %red  
  i8 1, label %green  
  i8 2, label %blue  
]
```

- **LLVM 9** → **LLVM 14**, and the cargo/rustc pipeline
- **Test Inversion extended** to select and switch
 - ⇒ rustc makes extensive use of select and switch (clang emitted *none*, neither on FISSC nor on Lazard's test suite)
- **Several other fixes** linked to LLVM-IR emitted by rustc

A three-arm match...

```
match c {  
  Color::Red   => 1,  
  Color::Green => 2,  
  Color::Blue  => 3,  
}
```

...lowered to a switch

```
switch i8 %disc, label %unreachable [  
  i8 0, label %red  
  i8 1, label %green  
  i8 2, label %blue  
]
```

- LLVM 9 → LLVM 14, and the cargo/rustc pipeline
- **Test Inversion** extended to select and switch
 - ⇒ rustc makes extensive use of select and switch (clang emitted *none*, neither on FISSC nor on Lazard's test suite)
- **Several other fixes** linked to LLVM-IR emitted by rustc
- **Change Pointer (CP)**: a new fault model
 - redirect a reference to *another* legitimate object
 - available in C and Rust

Forcing two &mut onto the same cell

```
1 fn transfer(from: &mut u32, to: &mut u32, amount: u32) {  
2     *from -= amount; // both writes hit the same variable  
3     *to += amount; // once from and to alias  
4 }  
5  
6 fn main() {  
7     let (mut balance, mut escrow) = (100, 0);  
8  
9     transfer(&mut balance,  
10         _LZ__change_ptr_R!(&mut escrow,  
11             &mut balance, "alias", u32), 100);  
12  
13     // no debit  
14     _LZ__ORACLE!(balance == 100 && escrow == 0);  
15 }  
16
```

- Motivation
- The Rust language
- Lazart for Rust
- **FISSC in Rust**
- The attack surface of idiomatic Rust
- Conclusion and perspectives

- **Same algorithm, same countermeasures:** FISSC ported **one-to-one** from C [DPP+16] (aligned IP by IP)
 - Any remaining difference then comes from **the language or the compiler**

Porting FISSC to Rust: isolating the language

- **Same algorithm, same countermeasures:** FISSC ported **one-to-one** from C [DPP+16] (aligned IP by IP)
 - Any remaining difference then comes from **the language or the compiler**
- Some technical differences already presented previously (select/switch etc.)



- **Same algorithm, same countermeasures:** FISSC ported **one-to-one** from C [DPP+16] (aligned IP by IP)
 - Any remaining difference then comes from **the language or the compiler**
- Some technical differences already presented previously (select/switch etc.)
- Other differences **neutralise**:
 - Rust's **checks by default**: `get_unchecked()` in `unsafe`, `-C overflow-checks=off`, `-C panic=abort`

- **Same algorithm, same countermeasures:** FISSC ported **one-to-one** from C [DPP+16] (aligned IP by IP)
 - Any remaining difference then comes from **the language or the compiler**
- Some technical differences already presented previously (select/switch etc.)
- Other differences **neutralise**:
 - Rust's **checks by default**: `get_unchecked()` in `unsafe`, `-C overflow-checks=off`, `-C panic=abort`
 - `-C opt-level=1` vs `clang -O0`: at `-O0` (Rust's iterators and comparisons stay as function calls)
 - `#[inline(never)]` where needed (inlining (*INL*) is itself a countermeasure)

- **Same algorithm, same countermeasures:** FISSC ported **one-to-one** from C [DPP+16] (aligned IP by IP)
 - Any remaining difference then comes from **the language or the compiler**
- Some technical differences already presented previously (select/switch etc.)
- Other differences **neutralise**:
 - Rust's **checks by default**: `get_unchecked()` in `unsafe`, `-C overflow-checks=off`, `-C panic=abort`
 - `-C opt-level=1` vs `clang -O0`: at `-O0` (Rust's iterators and comparisons stay as function calls)
 - `#[inline(never)]` where needed (inlining (*INL*) is itself a countermeasure)
 - **Data Load instrumented in the source** (`mut_dl_sym!`): after SSA, no load is left to fault

- **Same algorithm, same countermeasures:** FISSC ported **one-to-one** from C [DPP+16] (aligned IP by IP)
 - Any remaining difference then comes from **the language or the compiler**
- Some technical differences already presented previously (select/switch etc.)
- Other differences **neutralise**:
 - Rust's **checks by default**: `get_unchecked()` in `unsafe`, `-C overflow-checks=off`, `-C panic=abort`
 - `-C opt-level=1` vs `clang -O0`: at `-O0` (Rust's iterators and comparisons stay as function calls)
 - `#[inline(never)]` where needed (inlining (*INL*) is itself a countermeasure)
 - **Data Load instrumented in the source** (`mut_dl_sym!`): after SSA, no load is left to fault

Equivalent attack surface, up to explainable codegen artefacts (a match partly *by construction*)

Outline

- Motivation
- The Rust language
- Lazart for Rust
- FISSC in Rust
- The attack surface of idiomatic Rust
- Conclusion and perspectives

- **16 idiomatic constructs**, each studied **in isolation**

- Does it **add** attack surface, **block** faults, or both?

Construct	Fault target	Model	min.	LLVM
<i>Interior mutability & reference counting</i>				
RefCell::try_borrow_mut	flag == UNUSED	TI	1F	br
Rc::get_mut	strong == 1	TI	1F	br
Rc::drop	strong == 0	TI	1F	br
Cell	reference	CP	1F	ptr
<i>Runtime validity checks</i>				
str::from_utf8	Ok/Err	TI	1F	br
str slicing (get)	is_char_boundary	TI	1F	select
indexing (bounds check)	bound + panic guard	TI	2F	br
<i>Discriminant, niche & value validity</i>				
? operator	Ok/Err branch	TI	1F	br
match (3 arms)	enum discriminant	TI	1F	switch
Option<T> niche	ptr == null	TI	1F	select
bool validity	invalid byte (load)	DL	1F	load
<i>Drop & unwinding</i>				
Drop (drop flag)	drop-flag test	TI	1F	br
panic / unwinding	access guard	TI	1F	br
<i>Iteration</i>				
Iterator::any	closure predicate	TI	1F	br
<i>Aliasing & control flow</i>				
&mut aliasing	reference	CP	1F	ptr
indirect call (CFI)	function-pointer target	TI	1F	br

- **16 idiomatic constructs**, each studied **in isolation**
 - Does it **add** attack surface, **block** faults, or both?
- **Indexing**: rustc inserts a bounds check and a panic
 - 1 TI drives the index out of bounds... **caught** by the panic
 - 1 more TI on that guard lets the access through → **2F**

```
fn get(a: &[u8], i: usize) -> u8 {  
    a[i]    // F1 (upstream): i >= a.len() -> panic  
}
```

```
%ok = icmp ult i64 %i, %len  
br i1 %ok, label %access, label %panic ; F2: invert
```

- **16 idiomatic constructs**, each studied **in isolation**

- Does it **add** attack surface, **block** faults, or both?

- **Indexing**: rustc inserts a bounds check and a panic

- 1 TI drives the index out of bounds... **caught** by the panic
- 1 more TI on that guard lets the access through → **2F**

- **Option/Result, exhaustive match, ?**: one discriminant test (br, switch)

- 1 TI forces the wrong arm, the discriminant is the **only** guard
→ **1F**

```
let cmd = parse(buf)?;           // Ok/Err test
match cmd {                       // exhaustive, 3 arms
    Cmd::Read => read(),
    Cmd::Write => write(),
    Cmd::Reset => reset(),
}
```

```
br i1 %is_err, label %ret, label %ok      ; ?      : 1 TI
switch i8 %tag, label %unreachable [...] ; match : 1 TI
```

- **16 idiomatic constructs**, each studied **in isolation**

- Does it **add** attack surface, **block** faults, or both?

- **Indexing**: rustc inserts a bounds check and a panic

- 1 TI drives the index out of bounds... **caught** by the panic
- 1 more TI on that guard lets the access through → **2F**

- **Option/Result, exhaustive match, ?**: one discriminant test (br, switch)

- 1 TI forces the wrong arm, the discriminant is the **only** guard
→ **1F**

- **Niche & value validity**

- `Option<&T>`: the discriminant *is* the pointer; 1 TI turns `None` into `Some(null)`
- `bool`: 1 DL forges a byte outside `{0,1}`, an invariant the emitted code **assumes** → **1F**

```
let p: Option<&T> = find(k); // None == null
match p { Some(r) => read(r),
          None    => bail() }

let ok: bool = check(pin); // valid: 0 or 1 only
if ok { grant(); }         // 1 DL: 0x02 loaded
```

```
%c = icmp eq ptr %p, null
%v = select i1 %c, ... ; 1 TI: None -> Some(null)
```

- **16 idiomatic constructs**, each studied **in isolation**

- Does it **add** attack surface, **block** faults, or both?

- **Indexing**: rustc inserts a bounds check and a panic

- 1 TI drives the index out of bounds... **caught** by the panic
- 1 more TI on that guard lets the access through → **2F**

- **Option/Result, exhaustive match, ?**: one discriminant test (br, switch)

- 1 TI forces the wrong arm, the discriminant is the **only** guard
→ **1F**

- **Niche & value validity**

- `Option<&T>`: the discriminant *is* the pointer; 1 TI turns `None` into `Some(null)`
- `bool`: 1 DL forges a byte outside $\{0, 1\}$, an invariant the emitted code **assumes** → **1F**

```
let p: Option<&T> = find(k); // None == null
match p { Some(r) => read(r),
         None    => bail() }

let ok: bool = check(pin); // valid: 0 or 1 only
if ok { grant(); }         // 1 DL: 0x02 loaded
```

```
%c = icmp eq ptr %p, null
%v = select i1 %c, ... ; 1 TI: None -> Some(null)
```

Rust's guarantees rest on **runtime checks and values**: **one fault breaks them**, two where Rust adds a guard C omits.

Version	1F	2F	3F
YACCA	0	0	1
RACFED	1	3	2

- Control-flow checking countermeasures [VPH+17; GRR+05], written in Rust by Sini et al. [SSV+25] for an automotive controller

Version	1F	2F	3F
YACCA	0	0	1
RACFED	1	3	2

- Control-flow checking countermeasures [VPH+17; GRR+05], written in Rust by Sini et al. [SSV+25] for an automotive controller
- Their fault model: *one* permanent bit-flip on the program counter
 - Ours is **stronger**: several chosen faults

Version	1F	2F	3F
YACCA	0	0	1
RACFED	1	3	2

- Control-flow checking countermeasures [VPH+17; GRR+05], written in Rust by Sini et al. [SSV+25] for an automotive controller
- Their fault model: *one* permanent bit-flip on the program counter
 - Ours is **stronger**: several chosen faults
- The initial **control-flow corruption** (what the countermeasure must detect) is instrumented and **not counted**

Version	1F	2F	3F
YACCA	0	0	1
RACFED	1	3	2

- **RACFED (1 fault)**. The block signatures are precomputed
 - One DL forcing the running signature (`UpdSigBegin`) to its expected value makes `CheckSig` pass
- Control-flow checking countermeasures [VPH+17; GRR+05], written in Rust by Sini et al. [SSV+25] for an automotive controller
- Their fault model: *one* permanent bit-flip on the program counter
 - Ours is **stronger**: several chosen faults
- The initial **control-flow corruption** (what the countermeasure must detect) is instrumented and **not counted**

Version	1F	2F	3F
YACCA	0	0	1
RACFED	1	3	2

- Control-flow checking countermeasures [VPH+17; GRR+05], written in Rust by Sini et al. [SSV+25] for an automotive controller
- Their fault model: *one* permanent bit-flip on the program counter
 - Ours is **stronger**: several chosen faults
- The initial **control-flow corruption** (what the countermeasure must detect) is instrumented and **not counted**

- **RACFED (1 fault)**. The block signatures are precomputed
 - One DL forcing the running signature (UpdSigBegin) to its expected value makes CheckSig pass
- **YACCA (3 faults here)**. Each TI inverts the ERR_CODE update
 - The count depends on how far the target block is from the end of the path
 - A property of *this example*, not of YACCA

Version	1F	2F	3F
YACCA	0	0	1
RACFED	1	3	2

- Control-flow checking countermeasures [VPH+17; GRR+05], written in Rust by Sini et al. [SSV+25] for an automotive controller
- Their fault model: *one* permanent bit-flip on the program counter
 - Ours is **stronger**: several chosen faults
- The initial **control-flow corruption** (what the countermeasure must detect) is instrumented and **not counted**

- **RACFED (1 fault)**. The block signatures are precomputed
 - One DL forcing the running signature (UpdSigBegin) to its expected value makes CheckSig pass
- **YACCA (3 faults here)**. Each TI inverts the ERR_CODE update
 - The count depends on how far the target block is from the end of the path
 - A property of *this example*, not of YACCA

Explicit countermeasures are not spared: a control-flow corruption can still be **hidden by extra faults**.

Outline

- Motivation
- The Rust language
- Lazart for Rust
- FISSC in Rust
- The attack surface of idiomatic Rust
- Conclusion and perspectives

- **Contributions:** extension of Lazart for Rust (and *Change Pointer* model), FISSC port, idiomatic Rust constructs study, and real-world code (RACFED/YACCA [SSV+25], the aes crate)

- **Contributions:** extension of Lazart for Rust (and *Change Pointer* model), FISSC port, idiomatic Rust constructs study, and real-world code (RACFED/YACCA [SSV+25], the aes crate)
- **At the IR level, Rust is attacked like C:** the same br/switch/select/load are faulted, and the FISSC ports give the same results
 - As expected, **compile-time** memory safety does not imply fault resistance

- **Contributions:** extension of Lazart for Rust (and *Change Pointer* model), FISSC port, idiomatic Rust constructs study, and real-world code (RACFED/YACCA [SSV+25], the aes crate)
- **At the IR level, Rust is attacked like C:** the same br/switch/select/load are faulted, and the FISSC ports give the same results
 - As expected, **compile-time** memory safety does not imply fault resistance
- **The checks Rust emits by default** (bounds, UTF-8, char boundary) turn a **1-fault** attack into a **2-fault** one
 - exactly what a C developer could have written **by hand**

- **Contributions:** extension of Lazart for Rust (and *Change Pointer* model), FISSC port, idiomatic Rust constructs study, and real-world code (RACFED/YACCA [SSV+25], the aes crate)
- **At the IR level, Rust is attacked like C:** the same `br/switch/select/load` are faulted, and the FISSC ports give the same results
 - As expected, **compile-time** memory safety does not imply fault resistance
- **The checks Rust emits by default** (bounds, UTF-8, char boundary) turn a **1-fault** attack into a **2-fault** one
 - exactly what a C developer could have written **by hand**
- **Static guarantees do break at runtime**
 - CP forges the aliasing the borrow checker forbids; one fault on a `bool` or an `Option<T>` breaks a validity invariant

- **Contributions:** extension of Lazart for Rust (and *Change Pointer* model), FISSC port, idiomatic Rust constructs study, and real-world code (RACFED/YACCA [SSV+25], the aes crate)
- **At the IR level, Rust is attacked like C:** the same `br/switch/select/load` are faulted, and the FISSC ports give the same results
 - As expected, **compile-time** memory safety does not imply fault resistance
- **The checks Rust emits by default** (bounds, UTF-8, char boundary) turn a **1-fault** attack into a **2-fault** one
 - exactly what a C developer could have written **by hand**
- **Static guarantees do break at runtime**
 - CP forges the aliasing the borrow checker forbids; one fault on a `bool` or an `Option<T>` breaks a validity invariant

- **Contributions:** extension of Lazard for Rust (and *Change Pointer* model), FISSC port, idiomatic Rust constructs study, and real-world code (RACFED/YACCA [SSV+25], the aes crate)
- **At the IR level, Rust is attacked like C:** the same `br/switch/select/load` are faulted, and the FISSC ports give the same results
 - As expected, **compile-time** memory safety does not imply fault resistance
- **The checks Rust emits by default** (bounds, UTF-8, char boundary) turn a **1-fault** attack into a **2-fault** one
 - exactly what a C developer could have written **by hand**
- **Static guarantees do break at runtime**
 - CP forges the aliasing the borrow checker forbids; one fault on a `bool` or an `Option<T>` breaks a validity invariant

Rust developers **and the optimiser** trust these properties (no aliasing, no use-after-free).
Under **fault injection**, they no longer hold.

- **Lift Lazart/KLEE's limitations:** a ulibrust, counterpart of uClibc
 - Reach the genuine core/alloc code, dyn Trait, async, Mutex, atomics

- **Lift Lazart/KLEE's limitations:** a ulibrust, counterpart of uClibc
 - Reach the genuine core/alloc code, dyn Trait, async, Mutex, atomics
- **Other representation levels:** binary [DBF+16], Rust's MIR

- **Lift Lazard/KLEE's limitations:** a ulibrust, counterpart of uClibc
 - Reach the genuine core/alloc code, dyn Trait, async, Mutex, atomics
- **Other representation levels:** binary [DBF+16], Rust's MIR
- **Countermeasures in Rust** against *multiple* faults: [FJM26], and their preservation by rustc compared to C compilers [SCA18; PBM+25]

- **Lift Lazart/KLEE's limitations:** a ulibrust, counterpart of uClibc
 - Reach the genuine core/alloc code, dyn Trait, async, Mutex, atomics
- **Other representation levels:** binary [DBF+16], Rust's MIR
- **Countermeasures in Rust** against *multiple* faults: [FJM26], and their preservation by rustc compared to C compilers [SCA18; PBM+25]

The Rust-native FISSC collection is online

Use it to evaluate **your** tools and countermeasures, and help us extend it.

<https://lazart.gricad-pages.univ-grenoble-alpes.fr/fissc/>

Thanks !

Questions ?

Email : etienne.boespflug@univ-grenoble-alpes.fr

Artifact : gricad-gitlab.univ-grenoble-alpes.fr/securitytools/rust-fi-artefact

Rust FISSC : lazart.gricad-pages.univ-grenoble-alpes.fr/fissc/

- [BBM+23] Etienne Boespflug et al. “A tool assisted methodology to harden programs against multi-faults injections”. In: *arXiv preprint arXiv:2303.01885* (2023).
- [Boe23] Etienne Boespflug. “Outils pour l’analyse de code et de contre-mesures pour l’injection de fautes multiples”. Thesis. Université Grenoble Alpes, 2023.
- [CSX+23] Mohan Cui et al. *Is unsafe an Achilles’ Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming*. 2023. arXiv: 2308.04785 [cs.SE]. URL: <https://arxiv.org/abs/2308.04785>.
- [DBF+16] Robin David et al. “BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-level Analysis”. In: *Proceedings of SANER 2016*. Osaka, Japan, Mar. 2016.
- [DPP+16] Louis Dureuil et al. “FISSC: A Fault Injection and Simulation Secure Collection”. In: *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016*. 2016.
- [FJM26] François de Ferrière, Yves Janin, and Sirine Mechmech. “SecSwift, a Compiler-Based Framework for Software Countermeasures in Cybersecurity”. In: *CGO*. Ed. by Stephen M. Blackburn, Albert Cohen, and Timothy M. Jones. IEEE, 2026, pp. 98–108. DOI: 10.1109/CGO68049.2026.11395204. URL: <https://doi.org/10.1109/CGO68049.2026.11395204>.
- [GRR+05] O. Goloubeva et al. “Improved software-based processor control-flow errors detection technique”. In: *Annual Reliability and Maintainability Symposium, 2005. Proceedings*. 2005, pp. 583–589. DOI: 10.1109/RAMS.2005.1408426.

- [JJK+17] Ralf Jung et al. “RustBelt: Securing the foundations of the Rust programming language”. In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–34.
- [KLL26] Minjae Kim, Eunsung Lee, and Jaeyong Lee. “Hardening Rust’s Result: Simple Source Code Transformations for Fault-Robust Bootloader”. In: *2026 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE. 2026, pp. 1–5.
- [LA04] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation”. In: *CGO*. San Jose, CA, USA, Mar. 2004, pp. 75–88.
- [Led22] Ledger Donjon. *Fault Injection Checks Demo*. GitHub repository. Accessed: 2026-06-01. 2022. URL: https://github.com/Ledger-Donjon/fault_injection_checks_demo.
- [LL24] Alex Le Blanc and Patrick Lam. *Surveying the Rust Verification Landscape*. 2024. arXiv: 2410.01981 [cs.PL]. URL: <https://arxiv.org/abs/2410.01981>.
- [Mic19] Microsoft Security Response Center. *A proactive approach to more secure code*. <https://www.microsoft.com/en-us/msrc/blog/2019/07/a-proactive-approach-to-more-secure-code>. 2019.
- [MK14] Nicholas D Matsakis and Felix S Klock. “The rust language”. In: *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*. 2014, pp. 103–104.
- [PBM+25] Basile Pesin et al. “Formally Verified Hardening of C Programs against Hardware Fault Injection”. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2025. ISBN: 9798400713477. DOI: 10.1145/3703595.3705880.

- [PMP+14] Marie-Laure Potet et al. “Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections”. In: *ICST*. IEEE, 2014, pp. 213–222.
- [SAK24] Diane B. Stephens, Kawkab Aldoshan, and Mustakimur Rahman Khandaker. “Understanding the Challenges in Detecting Vulnerabilities of Rust Applications”. In: *2024 IEEE Secure Development Conference (SecDev)*. 2024, pp. 54–63. DOI: 10.1109/SecDev61143.2024.00012.
- [SCA18] Laurent Simon, David Chisnall, and Ross J. Anderson. “What You Get is What You C: Controlling Side Effects in Mainstream C Compilers”. In: *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018, pp. 1–15. DOI: 10.1109/EUROSP.2018.00009. URL: <https://doi.org/10.1109/EuroSP.2018.00009>.
- [SSV+25] Jacopo Sini et al. “Improving software reliability with Rust: Implementation for enhanced control flow checking methods”. In: *DATE 2025*. IEEE. 2025, pp. 1–7.
- [The] The Chromium Project. *Memory safety*. <https://www.chromium.org/Home/chromium-security/memory-safety/>. Accessed 2026.
- [VPH+17] Jens Vankeirsbilck et al. “Random Additive Signature Monitoring for Control Flow Error Detection”. In: *IEEE Transactions on Reliability* 66.4 (2017), pp. 1178–1192. DOI: 10.1109/TR.2017.2754548.

Backup slides

Fault *tolerance* (random faults, mostly single bit-flips)

- Sini et al. [SSV+25]: RACFED [VPH+17] and YACCA [GRR+05] control-flow checking **written in Rust** instead of C/assembly — overhead and detection latency, by simulation
- Kim et al. [KLL26]: hardening the `Result<T,E>` tag encoding, so that a bit-flip is unlikely to turn a *valid* tag into another one
- Ledger Donjon [Led22]: simulation-based fault injection on Rust in a CI/CD pipeline — **single** fault

Verification of Rust programs

- Proof of the core language and of its unsafe libraries (RustBelt [JJk+17]); deductive proof, bounded model-checking, symbolic execution, abstract interpretation, fuzzing [LL24; SAK24]
- Functional properties, or Rust-specific ones: absence of panic, **unsafe** encapsulation [CSX+23]
→ all assume a **faithful execution model**

The gap

No work considers an **active attacker** performing **multiple** hardware fault injections on Rust code, with fault models far more powerful than random bit-flips.

- **Objective:** redirect a reference to *another legitimate object*
 - aims to defeat the borrow checker's *Aliasing XOR Mutability* guarantees.

Forcing two `&mut` onto the same cell

```
1 fn transfer(from: &mut u32, to: &mut u32, amount: u32) {  
2     *from -= amount; // both writes hit the same variable  
3     *to   += amount; // once from and to alias  
4 }  
5  
6 fn main() {  
7     let (mut balance, mut escrow) = (100, 0);  
8  
9     transfer(&mut balance,  
10         _LZ__change_ptr_R!(&mut escrow,  
11             &mut balance, "alias", u32), 100);  
12  
13     // no debit  
14     _LZ__ORACLE!(balance == 100 && escrow == 0);  
15 }  
16
```

The *Change Pointer* fault model

- **Objective:** redirect a reference to *another legitimate object*
 - aims to defeat the borrow checker's *Aliasing XOR Mutability* guarantees.

Forcing two `&mut` onto the same cell

```
1 fn transfer(from: &mut u32, to: &mut u32, amount: u32) {  
2     *from -= amount; // both writes hit the same variable  
3     *to += amount; // once from and to alias  
4 }  
5  
6 fn main() {  
7     let (mut balance, mut escrow) = (100, 0);  
8  
9     transfer(&mut balance,  
10         _LZ__change_ptr_R!(&mut escrow,  
11             &mut balance, "alias", u32), 100);  
12  
13     // no debit  
14     _LZ__ORACLE!(balance == 100 && escrow == 0);  
15 }  
16
```

A **single fault** forges the aliasing that safe Rust forbids (at compile time), the oracle is unreachable without faults.



The *Change Pointer* fault model

- **Objective:** redirect a reference to *another legitimate object*
 - aims to defeat the borrow checker's *Aliasing XOR Mutability* guarantees.
- **Physically:** a wrong register already holding a pointer, a skipped offset computation, or a bit-flip on the low bits of an address (each lands on *another live object*)

Forcing two `&mut` onto the same cell

```
1 fn transfer(from: &mut u32, to: &mut u32, amount: u32) {  
2     *from -= amount; // both writes hit the same variable  
3     *to += amount; // once from and to alias  
4 }  
5  
6 fn main() {  
7     let (mut balance, mut escrow) = (100, 0);  
8  
9     transfer(&mut balance,  
10         _LZ__change_ptr_R!(&mut escrow,  
11             &mut balance, "alias", u32), 100);  
12  
13     // no debit  
14     _LZ__ORACLE!(balance == 100 && escrow == 0);  
15 }  
16
```

A single fault forges the aliasing that safe Rust forbids (at compile time), the oracle is unreachable without faults.



The *Change Pointer* fault model

- **Objective:** redirect a reference to *another legitimate object*
 - aims to defeat the borrow checker's *Aliasing XOR Mutability* guarantees.
- **Physically:** a wrong register already holding a pointer, a skipped offset computation, or a bit-flip on the low bits of an address (each lands on *another live object*)
- **How:** source and target given by instrumentation
 - a symbolic address would fork KLEE once per memory object
 - implemented with `unsafe`
- Implemented for **Rust and C**

Forcing two `&mut` onto the same cell

```
1 fn transfer(from: &mut u32, to: &mut u32, amount: u32) {  
2     *from -= amount; // both writes hit the same variable  
3     *to += amount; // once from and to alias  
4 }  
5  
6 fn main() {  
7     let (mut balance, mut escrow) = (100, 0);  
8  
9     transfer(&mut balance,  
10         _LZ__change_ptr_R!(&mut escrow,  
11             &mut balance, "alias", u32), 100);  
12  
13     // no debit  
14     _LZ__ORACLE!(balance == 100 && escrow == 0);  
15 }  
16
```

A **single fault** forges the aliasing that safe Rust forbids (at compile time), the oracle is unreachable without faults.



Same code in C and in Rust: same attack surface?

Version	Attacks			
	1F	2F	3F	4F
VP0	12	21	18	7
VP0 – <i>rust</i>	8	20	20	10
VP1–VP7: identical				
MCMPS1–MCMPS3: identical				

Attack objective: return `true`, or leave the `try_counter` untouched.

Symbolic PINs, differing on at least one byte.

Test Inversion, plus Data Load on `memcmps`.

Same code in C and in Rust: same attack surface?

Version	Attacks			
	1F	2F	3F	4F
VP0	12	21	18	7
VP0 – <i>rust</i>	8	20	20	10
VP1–VP7: identical				
MCMPS1–MCMPS3: identical				

Attack objective: return true, or leave the `try_counter` untouched.

Symbolic PINs, differing on at least one byte.

Test Inversion, plus Data Load on `memcmps`.

- **VP0** — without hardened booleans, `rustc` recycles the branch condition (`xor`) into the return value
 - **One** injection point disappears in `verify_pin`; Lazard reports one path per differing byte → **4 attacks fewer**
 - The 8 remaining ones are all in `compare`
 - A codegen choice `clang` could have made too

Same code in C and in Rust: same attack surface?

Version	Attacks			
	1F	2F	3F	4F
VP0	12	21	18	7
VP0 – <i>rust</i>	8	20	20	10
VP1–VP7: identical				
MCMPS1–MCMPS3: identical				

Attack objective: return true, or leave the `try_counter` untouched.
Symbolic PINs, differing on at least one byte.
Test Inversion, plus Data Load on `memcmps`.

- **VP0** — without hardened booleans, `rustc` recycles the branch condition (`xor`) into the return value
 - **One** injection point disappears in `verify_pin`; Lazard reports one path per differing byte → **4 attacks fewer**
 - The 8 remaining ones are all in `compare`
 - A codegen choice `clang` could have made too
- **VP6** — same counts, but `rustc` places some injection points **earlier** in the control flow
 - Those become the ones to protect first, at 4 faults

Equivalent attack surface, up to explainable codegen artefacts (a match partly *by construction*)

Backup — idiomatic Rust constructs

Algebraic data types + exhaustive match

```
match msg {
    // every variant
    Msg::Quit      => quit(),      // must be
    Msg::Move{x, y} => go(x, y),    // covered
    Msg::Write(s)  => log(s),
}
```

Mandatory, explicit error handling

```
fn read(i: &str) -> Result<u32, Error> {
    let v = parse(i)?; // early return on Err
    Ok(v + 1)
}
let x: Option<T> = map.get(key);
```

Automatic validity checks

```
let b = a[i]; // panics if i >= len
let s = str::from_utf8(&buf)?; // UTF-8 valid.
let t = &s[0..n]; // char-boundary check
```

Interior mutability & shared ownership

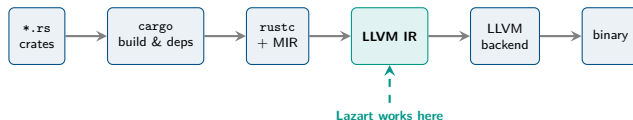
```
let c = Rc::new(RefCell::new(0)); // refcount
*c.borrow_mut() += 1; // borrow flag checked
let d = c.clone(); // *at runtime*
```

Deterministic destruction (Drop)

```
impl Drop for Guard {
    fn drop(&mut self) { self.release(); }
} // run at scope exit *and* while unwinding
```

Each construct removes a class of *programming errors*, and most are enforced by a **runtime check**: a discriminant test, a bound test, a flag or a counter comparison.





- Rust is deployed in operating system kernels, browsers, IoT and embedded systems; certified compilers exist for critical domains (Ferrocene)
- Reported to remove a large share of the memory-safety vulnerabilities of C/C++ code [The; Mic19]
- `rustc` lowers to **LLVM IR** [LA04], like `clang` — but with IR patterns `clang` never produced (`select`, `switch`)

Backup — Lazart injection points and fault models

Injection point in the higher-order mutant

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

Test Inversion (TI)

```
; Nominal behavior
br i1 %4, label %then, label %else
; Faulted behavior
br i1 %4, label %else, label %then
```

Data Load Mutation (DL)

```
; Nominal behavior
%n = load i32* %var, align 4
; Faulted behavior
%n = load i32 %faulted_value, align 4
```

- *Input*: an attacker model = fault models + attack objective (oracle) + fault limit
- *Output*: the set of multi-fault attack paths
- The symbolic inject makes the DSE cover both the nominal and the faulted branch; a single run at limit n covers every order up to n
- IR-level fault models trade *physical fidelity* for *genericity*: a test inversion can be realised by skipping the branch, attacking the program counter, or faulting the condition register
→ “is the program logic sensitive to this inversion?”, independently of the injection technique



MCMP51: a single call to memcmp

```
1 pub fn memcmps(a: &[u8;SIZE], b: &[u8;SIZE],
2   len: i32) -> i32 {
3   let mut result = MASK;
4
5   if !memcmp(a, b, mut_dl_sym!(len) as u64)
6   {
7       // IP0 (DL), IP1 (TI)
8       result = mut_dl_sym!(result)
9         ^ MASK ^ TRUE_VAL; // IP3 (DL)
10   } else {
11       result = mut_dl_sym!(result)
12         ^ MASK ^ FALSE_VAL; // IP4 (DL)
13   }
14   mut_dl_sym!(result) // IP5 (DL)
15 }
16
```

Version	Attacks				Non-redundant			
	1F	2F	3F	4F	1F	2F	3F	4F
MCMP51	16	32	28	8	16	0	0	0
MCMP51 – rust	16	32	28	8	16	0	0	0
MCMP52	4	32	404	1400	4	32	322	194
MCMP52 – rust	4	32	404	1400	4	32	322	194
MCMP53	4	32	216	2262	4	32	134	1468
MCMP53 – rust	4	32	216	2262	4	32	134	1468

- Hardened memcmp: returns true if both arrays are equal, using **hardened booleans** (no three-way comparison)
- Up to four memcmp calls (version 3), XORed with masks, so that several faults are required to return true on unequal arrays
- Fault models: **TI** (return prematurely) and **DL** (on result or on the len parameter)

Why DL faults are instrumented in the source

The mut_dl_sym! macro is called explicitly rather than naming a variable, because:

- it prevents rustc from propagating len in SSA — it would be consumed directly, with **no load instruction** for DL to apply to;
- it prevents the constant folding of result, which would otherwise reduce to a selection between constants depending on the path.

(The macro also takes the IP name and the fault limit, omitted here.)

⇒ with those changes, Rust behaves **exactly** like C, from 1 to 4 faults.



Definition [DPP+16; BBM+23]

The *redundancy analysis* groups together attacks that are **variations of a simpler attack**, obtained by adding extra faults on top of it.

- An n -fault attack is **redundant** if it contains the injection points of a successful attack of lower order
- **Why it matters:** protecting the injection points of the *non-redundant* attacks also makes all their redundant variants require more faults
→ the non-redundant count is the actionable one for a developer placing countermeasures
- This is why VP0 shows 8/20/20/10 attacks but only 8 non-redundant ones: **every higher-order attack is a variation of the eight 1-fault attacks** found in compare
- Note: since `user_pin` and `card_pin` are symbolic, Lazart reports a distinct attack path per input class (differing on the first, second, third or fourth byte) — hence counts that are multiples of 4

- `str::from_utf8`: inverting the *validation* branch is a 1-fault attack, yielding an `&str` that violates its UTF-8 invariant. Inverting the *loop-iteration* branch instead leads **out of bounds** in 2 faults (a `ptr.err` test case reported by KLEE) — arguably an information leak
- **str slicing**: `rustc` compiles the logic of `is_char_boundary` (`idx==0 || idx==len`, plus the continuation-byte test) into `select` instructions — one TI on a `select` edge
- **Checked arithmetic**: compiled out in release, where it wraps as in C — no guard, nothing to fault
- **bool validity**: a DL fault does not *trigger* undefined behaviour — UB is a compile-time notion, with no runtime event to point at. It **breaks an invariant the already-generated code was compiled to assume** (niche layouts, branch simplification), so that code may then behave arbitrarily
- **Drop & unwinding**: one TI on the drop flag gives two distinct attacks (double drop, no drop); the abort path reached on a double panic could also count as an attack on availability
- **Indirect call (CFI)**: one TI changes the call to a wrong-signature target, exactly as in C/C++

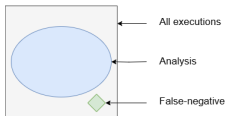
Program (4F)	C		Rust	
	time	paths	time	paths
VP0	0.16 s	130	0.17 s	155
VP1	0.21 s	178	0.17 s	203
VP2	0.42 s	1930	0.55 s	2180
VP3	0.47 s	1930	0.51 s	2180
VP4	0.6 s	2419	0.47 s	2085
VP5	4.95 s	14213	6.6 s	17367
VP6	0.08 s	169	0.16 s	195
VP7	0.24 s	260	0.17 s	286
MCMP51	0.26 s	113	0.32 s	113
MCMP52	2.58 s	4278	3.28 s	4278
MCMP53	7.59 s	12534	9.48 s	12534

Program	Limit	time	paths
RACFED	3F	0.19 s	131
YACCA	3F	0.13 s	29
aes (TI)	2F	2 h 16 min	10374
aes (CP)	2F	0.15 s	2

- One run at limit n covers every attack order up to n
- Cost is driven by the number of paths: it grows with the fault order and the number of countermeasures; Rust explores slightly more paths than C
- rustc: `-C opt-level=1, panic=abort` (except unwinding experiments), `overflow-checks=off, lto=fat, embed-bitcode=yes, codegen-units=1`
- clang: `-O0`; debug information in both languages
- Why `opt-level=1`: zero-cost abstractions only vanish once inlined — at `opt-level=0` a single source-level test can split into several injection points
- Intel Core i5-1335U, 16 GB

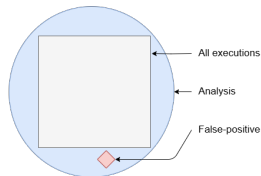
Under-approximation methods

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



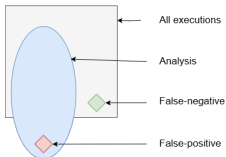
Over-approximation methods

examples: abstract interpretation



Hybrids methods

examples: dynamic-symbolic execution



Exact methods

examples: formal proofs



Listing 1: Function example

```
void example(int x, int y) {  
     $PC_0 \equiv true$   
  
    if (x + y > 2) {  
        z = 2*x + y;  
  
        if (z == 20) {  
            x = z + 3;  
            print(x);  
        }  
        else {  
            y *= 2;  
            assert(x != 0);  
        }  
    }  
    else {  
        foo();  
    }  
}
```

- Symbolic variables maintain a symbolic memory ϕ

Listing 2: Function example

```
void example(int x, int y) {  
     $PC_0 \equiv true$   
     $\phi_0 = \{x \rightarrow x_0, y \rightarrow y_0\}$   
    if (x + y > 2) {  
        z = 2*x + y;  
  
        if (z == 20) {  
            x = z + 3;  
            print(x);  
        }  
        else {  
            y *= 2;  
            assert(x != 0);  
        }  
    }  
    else {  
        foo();  
    }  
}
```

- Symbolic variables maintain a symbolic memory ϕ
- Path Constraint updated at each branch

Listing 3: Function example

```
void example(int x, int y) {  
     $PC_0 \equiv true$   
     $\phi_0 = \{x \rightarrow x_0, y \rightarrow y_0\}$   
    if (x + y > 2) {  
         $PC_1 \equiv x_0 + y_0 > 2$   
        z = 2*x + y;  
  
        if (z == 20) {  
            x = z + 3;  
            print(x);  
        }  
        else {  
            y *= 2;  
            assert(x != 0);  
        }  
    }  
    else {  
        foo();  
    }  
}
```

- Symbolic variables maintain a symbolic memory ϕ
- Path Constraint updated at each branch

Listing 4: Function example

```
void example(int x, int y) {  
     $PC_0 \equiv true$   
     $\phi_0 = \{x \rightarrow x_0, y \rightarrow y_0\}$   
    if (x + y > 2) {  
         $PC_1 \equiv x_0 + y_0 > 2$   
        z = 2*x + y;  
  
        if (z == 20) {  
            x = z + 3;  
            print(x);  
        }  
        else {  
            y *= 2;  
            assert(x != 0);  
        }  
    }  
    else {  
         $PC_4 \equiv x_0 + y_0 \leq 2$   
        foo();  
    }  
}
```

- Symbolic variables maintain a symbolic memory ϕ
- Path Constraint updated at each branch

Listing 5: Function example

```
void example(int x, int y) {  
     $PC_0 \equiv true$   
     $\phi_0 = \{x \rightarrow x_0, y \rightarrow y_0\}$   
  
    if (x + y > 2) {  
         $PC_1 \equiv x_0 + y_0 > 2$   
        z = 2*x + y;  
         $\phi_1 = \{z \rightarrow 2x_0 + y_0, x \rightarrow x_0, y \rightarrow y_0\}$   
  
        if (z == 20) {  
            x = z + 3;  
            print(x);  
        }  
        else {  
            y *= 2;  
            assert(x != 0);  
        }  
    }  
    else {  
         $PC_4 \equiv x_0 + y_0 \leq 2$   
        foo();  
    }  
}
```

- Symbolic variables maintain a symbolic memory ϕ
- Path Constraint updated at each branch

Listing 6: Function example

```
void example(int x, int y) {  
     $PC_0 \equiv true$   
     $\phi_0 = \{x \rightarrow x_0, y \rightarrow y_0\}$   
  
    if (x + y > 2) {  
         $PC_1 \equiv x_0 + y_0 > 2$   
        z = 2*x + y;  
         $\phi_1 = \{z \rightarrow 2x_0 + y_0, x \rightarrow x_0, y \rightarrow y_0\}$   
  
        if (z == 20) {  
             $PC_2 \equiv PC_1 \wedge (2x_0 + y_0 = 20)$   
            x = z + 3;  
  
            print(x);  
        }  
        else {  
            y *= 2;  
  
            assert(x != 0);  
        }  
    }  
    else {  
         $PC_4 \equiv x_0 + y_0 \leq 2$   
        foo();  
    }  
}
```

- Symbolic variables maintain a symbolic memory ϕ
- Path Constraint updated at each branch

Listing 7: Function example

```
void example(int x, int y) {  
     $PC_0 \equiv true$   
     $\phi_0 = \{x \rightarrow x_0, y \rightarrow y_0\}$   
  
    if (x + y > 2) {  
         $PC_1 \equiv x_0 + y_0 > 2$   
        z = 2*x + y;  
         $\phi_1 = \{z \rightarrow 2x_0 + y_0, x \rightarrow x_0, y \rightarrow y_0\}$   
  
        if (z == 20) {  
             $PC_2 \equiv PC_1 \wedge (2x_0 + y_0 = 20)$   
            x = z + 3;  
  
            print(x);  
        }  
        else {  
             $PC_3 \equiv PC_1 \wedge (2x_0 + y_0 \neq 20)$   
            y *= 2;  
  
            assert(x != 0);  
        }  
    }  
    else {  
         $PC_4 \equiv x_0 + y_0 \leq 2$   
        foo();  
    }  
}
```

- Symbolic variables maintain a symbolic memory ϕ
- Path Constraint updated at each branch

Listing 8: Function example

```
void example(int x, int y) {  
     $PC_0 \equiv true$   
     $\phi_0 = \{x \rightarrow x_0, y \rightarrow y_0\}$   
  
    if (x + y > 2) {  
         $PC_1 \equiv x_0 + y_0 > 2$   
        z = 2*x + y;  
         $\phi_1 = \{z \rightarrow 2x_0 + y_0, x \rightarrow x_0, y \rightarrow y_0\}$   
  
        if (z == 20) {  
             $PC_2 \equiv PC_1 \wedge (2x_0 + y_0 = 20)$   
            x = z + 3;  
             $\phi_2 = \{x \rightarrow 2x_0 + y_0 + 3, z \rightarrow 2x_0 + y_0, y \rightarrow y_0\}$   
            print(x);  
        }  
        else {  
             $PC_3 \equiv PC_1 \wedge (2x_0 + y_0 \neq 20)$   
            y *= 2;  
  
            assert(x != 0);  
        }  
    }  
    else {  
         $PC_4 \equiv x_0 + y_0 \leq 2$   
        foo();  
    }  
}
```

- Symbolic variables maintain a symbolic memory ϕ
- Path Constraint updated at each branch

Listing 9: Function example

```
void example(int x, int y) {  
     $PC_0 \equiv true$   
     $\phi_0 = \{x \rightarrow x_0, y \rightarrow y_0\}$   
  
    if (x + y > 2) {  
         $PC_1 \equiv x_0 + y_0 > 2$   
        z = 2*x + y;  
         $\phi_1 = \{z \rightarrow 2x_0 + y_0, x \rightarrow x_0, y \rightarrow y_0\}$   
  
        if (z == 20) {  
             $PC_2 \equiv PC_1 \wedge (2x_0 + y_0 = 20)$   
            x = z + 3;  
             $\phi_2 = \{x \rightarrow 2x_0 + y_0 + 3, z \rightarrow 2x_0 + y_0, y \rightarrow y_0\}$   
            print(x);  
        }  
        else {  
             $PC_3 \equiv PC_1 \wedge (2x_0 + y_0 \neq 20)$   
            y *= 2;  
             $\phi_3 = \{x \rightarrow x_0, y \rightarrow 2y_0, z \rightarrow 2x_0 + y_0\}$   
            assert(x != 0);  
        }  
    }  
    else {  
         $PC_4 \equiv x_0 + y_0 \leq 2$   
        foo();  
    }  
}
```

- Symbolic variables maintain a symbolic memory ϕ
- Path Constraint updated at each branch

Listing 10: Function example

```
void example(int x, int y) {  
     $PC_0 \equiv true$   
     $\phi_0 = \{x \rightarrow x_0, y \rightarrow y_0\}$   
  
    if (x + y > 2) {  
         $PC_1 \equiv x_0 + y_0 > 2$   
        z = 2*x + y;  
         $\phi_1 = \{z \rightarrow 2x_0 + y_0, x \rightarrow x_0, y \rightarrow y_0\}$   
  
        if (z == 20) {  
             $PC_2 \equiv PC_1 \wedge (2x_0 + y_0 = 20)$   
            x = z + 3;  
             $\phi_2 = \{x \rightarrow 2x_0 + y_0 + 3, z \rightarrow 2x_0 + y_0, y \rightarrow y_0\}$   
            print(x);  
        }  
        else {  
             $PC_3 \equiv PC_1 \wedge (2x_0 + y_0 \neq 20)$   
            y *= 2;  
             $\phi_3 = \{x \rightarrow x_0, y \rightarrow 2y_0, z \rightarrow 2x_0 + y_0\}$   
            assert(x != 0);  
        }  
    }  
    else {  
         $PC_4 \equiv x_0 + y_0 \leq 2$   
        foo();  
    }  
}
```

- Symbolic variables maintain a symbolic memory ϕ
- Path Constraint updated at each branch
- SMT solver checks satisfiability

Inputs triggering `assert(x != 0)` :
 $\{x = 0, y > 2, y \neq 20\}$

Satisfiability Modulo Theories (SMT)

Goal: violate assertion `assert(x != 0)`

SMT formula (path constraint):

```
(set-logic QF_LIA)

(declare-const x0 Int)
(declare-const y0 Int)

(assert (= x0 0))
(assert (> (+ x0 y0) 2))
(assert (not (= (+ (* 2 x0) y0) 20)))

(check-sat)
(get-model)
```

Z3 output:

```
# z3 assert.smt2
sat
(model
  (define-fun y0 () Int 3)
  (define-fun x0 () Int 0)
)
```

What is an SMT solver?

Decides satisfiability of logical formulas over theories (arithmetic, bitvectors, arrays...)

Example: Z3, STP, CVC5, Yices, Boolector

Output:

- **sat** – constraints satisfiable
- **unsat** – no model exists
- **unknown** – cannot conclude



DSE limitations:

- **Path explosion problem:** may not terminate in programs with complex branching or loop patterns
- **Limited low-level architectural features:** external function calls, memory model precision, and pointer aliasing
- **SMT solving limitations:** solver timeouts or undecidable queries may result in inconclusive results
- **Concretization:** to handle certain operations, DSE may resort to concretization, which may compromise soundness by missing potential paths

⇒ SE provides *sound* results but may not be *complete* in path enumeration ⇒ DSE may lose soundness sometimes (i.e. some concretizations)

Definition [Godefroid 2011]

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

Listing 11: Nominal behavior

```
normal_behavior()
```

Listing 12: Faulted behavior

```
1 inject = symbolic_bool()
2 if inject and _fault_count <
3     _fault_limit:
4     _fault_count++
5     faulted_behavior()
6 else:
7     normal_behavior()
```

Definition [Godefroid 2011]

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

Listing 13: Nominal behavior

```
normal_behavior()
```

Listing 14: Faulted behavior

```
1 inject = symbolic_bool()
2 if inject and _fault_count <
3   _fault_limit:
4     _fault_count++
5     faulted_behavior()
6 else:
7     normal_behavior()
```

Definition

A faulted path constraint PC_ω^M is *correct* if every model satisfying PC_ω^M gives entries and faults for a (faulted) execution following the path ω .

A faulted path constraint PC_ω^M is *complete* if every pair (entries, faults) following the path ω is a model satisfying PC_ω^M .

The (faulted) path constraint enumeration PC_ω^M is *correct* if and only if, $\forall PC_\omega^M \in \mathcal{E}^M$, PC_ω^M is *correct*.

The (faulted) path constraint enumeration PC_ω^M is *complete* if and only if:

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