2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

MUNICH, GERMANY
OCTOBER 14-15, 2025

Minimally Intrusive Safety and Security Verification of Rust RTIC Applications

Pawel Dzialo*, Ivar Jönsson*†, Malte Munch*, Erik Serrander†, Johan Eriksson†, Per Lindgren*†

Luleå University of Technology*, Grepit AB†

accellera SYSTEMS INITIATIVE

What is RTIC

- Real-Time Interrupt Driven Concurrency
- Rust-based bare-metal scheduling framework/domain specific language
- Scheduling/resource management underpinned by Stack Resource Policy(SRP)
- Guaranteed* free of race conditions/deadlocks, bounded priority inversion
- 600.000+ downloads, one of **the** de-facto libraries for bare-metal Rust

The Gist of RTIC

Rust

```rust
#[shared]
struct Shared {buf: [u8; 64]}
#[local]
struct Local {
  foo_local: u8,
  bar_local: u8
}
#[init]
fn init() -> (Shared, Local) {
  (
    Shared {buf: [0u8; 64]},
    Local {foo_local: 0u8, bar_local: 0u8}
  )
}
```

The Gist of RTIC

Rust

```rust
#[shared]
struct Shared {buf: [u8; 64]}
#[local]
struct Local {
  foo_local: u8,
  bar_local: u8
}
#[init]
fn init() -> (Shared, Local) {
  (
    Shared {buf: [0u8; 64]},
    Local {foo_local: 0u8, bar_local: 0u8}
  )
}
```

The Gist of RTIC

Rust

```rust
#[shared]
struct Shared {buf: [u8; 64]}
#[local]
struct Local {
  foo_local: u8,
  bar_local: u8
}
#[init]
fn init() -> (Shared, Local) {
  (
    Shared {buf: [0u8; 64]},
    Local {foo_local: 0u8, bar_local: 0u8}
  )
}
```

The Gist of RTIC

Rust

```rust
#[task(
  dispatcher = GPIOA,
  priority = 1,
  local = [foo_local],
  shared = [buf]
)]
fn foo(cx: foo::Context) {
  *cx.local.foo_local = 4
  cx.shared.buf.lock(|buf| {
    buf[1] = 2;
  });
}
```

The Gist of RTIC

Rust

```rust
#[task(
  dispatcher = GPIOA,
  priority = 1,
  local = [foo_local],
  shared = [buf]
)]
fn foo(cx: foo::Context) {
  *cx.local.foo_local = 4
  cx.shared.buf.lock(|buf| {
    buf[1] = 2;
  });
}
```

The Gist of RTIC

Rust

```rust
#[task(
  dispatcher = GPIOA,
  priority = 1,
  local = [foo_local],
  shared = [buf]
)]
fn foo(cx: foo::Context) {
  *cx.local.foo_local = 4
  cx.shared.buf.lock(|buf| {
    buf[1] = 2;
  });
}
```

The Gist of RTIC

Rust

```
#[task(
  dispatcher = GPIOA,
  priority = 1,
  local = [foo_local],
  shared = [buf]
)]
fn foo(cx: foo::Context) {
  *cx.local.foo_local = 4
  cx.shared.buf.lock(|buf| {
    buf[1] = 2;
  });
}
```

The Gist of RTIC

Rust

```
#[task(
  dispatcher = GPIOA,
  priority = 1,
  local = [foo_local],
  shared = [buf]
)]
fn foo(cx: foo::Context) {
  *cx.local.foo_local = 4
  cx.shared.buf.lock(|buf| {
    buf[1] = 2;
  });
}
```

The Gist of RTIC

```rust
#[task(
  dispatcher = GPIOA,
  priority = 1,
  local = [foo_local],
  shared = [buf]
)]
fn foo(cx: foo::Context) {
  *cx.local.foo_local = 4
  cx.shared.buf.lock(|buf| {
    buf[1] = 2;
  });
}
```

The Gist of RTIC

Rust

```rust
#[task(
  dispatcher = GPIOA,
  priority = 1,
  local = [foo_local],
  shared = [buf]
)]
fn foo(cx: foo::Context) {
  *cx.local.foo_local = 4
  cx.shared.buf.lock(|buf| {
    buf[1] = 2;
  });
}
```

The Gist of RTIC

Rust

```rust
#[task(
  dispatcher = GPIOA,
  priority = 1,
  local = [foo_local],
  shared = [buf]
)]
fn foo(cx: foo::Context) {
  *cx.local.foo_local = 4
  cx.shared.buf.lock(|buf| {
    buf[1] = 2;
  });
}
```

Program Verification

- Given pre/post conditions and some program
- Total Correctness
  - Safety (e.g. nothing *bad* can happen)
  - Liveness (e.g. *something* will eventually happen)
  - Good things will eventually happen

Rust from a Verification POV

- Freedom from Undefined Behavior *mostly* at compile time
- Run-time verification injected where unsuccessful
  - Array indexing with variable index, e.g. a[i]
  - Division with variable divisor, e.g. 1/x, division by zero is UB
  - Panic (i.e. halt) instead of UB
- What about liveness? Halting is not ideal…

Symbolic Execution

- Method backing our analysis
- Determine the possible paths through piece of code, and values variables may assume
- Each unknown variable starts as unconstrained
- Constraints on variables tighten as we go

Symbolic Execution: An Example

Rust

```
fn simple(t:u8) -> u8 {
    if t > 10 {
        if t == 11 {
            return 11;
        }
        else if t == 3 {
            panic!()
        }
            return 10;
    }
    return 9;
}
```

Symbolic Execution: An Example

```rust
fn simple(t:u8) -> u8 {
  if t > 10 {
    if t == 11 {
      return 11;
    }
    else if t == 3 {
      panic!()
    }
    return 10;
  }
  return 9;
}
```

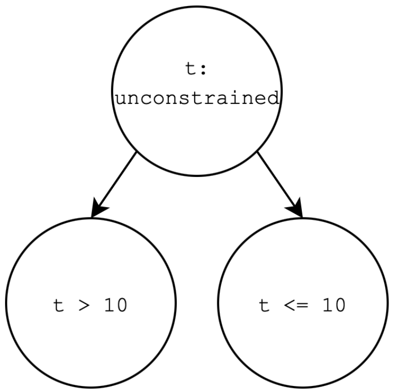
Rust

t: unconstrained

Symbolic Execution: An Example

Rust

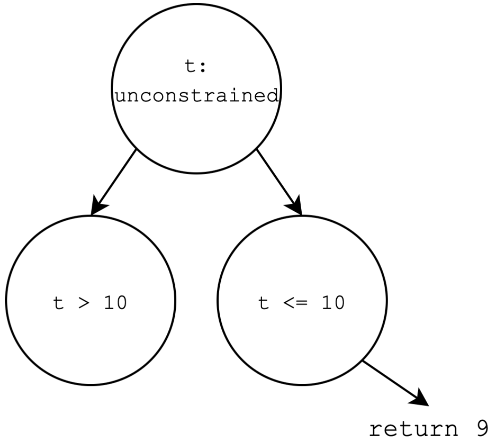
```rust
fn simple(t:u8) -> u8 {
  if t > 10 {
      if t == 11 {
        return 11;
      }
      else if t == 3 {
        panic!()
      }
      return 10;
   }
   return 9;
}
```



Symbolic Execution: An Example

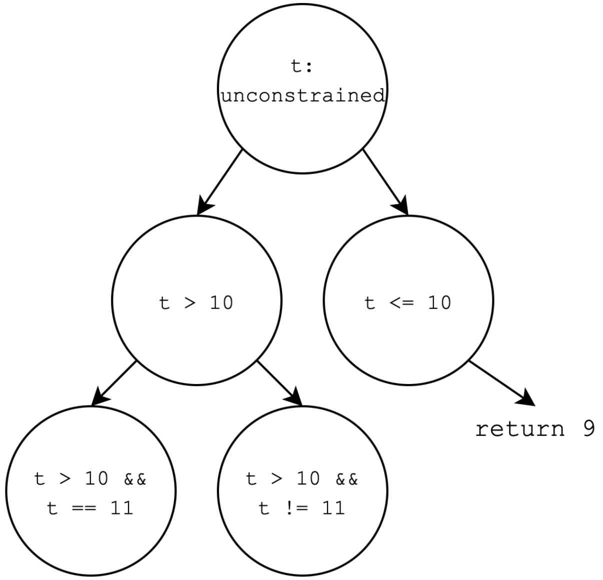
```rust
fn simple(t:u8) -> u8 {
  if t > 10 {
    if t == 11 {
      return 11;
    }
    else if t == 3 {
      panic!()
    }
    return 10;
  }
  return 9;
}
```

Rust



Symbolic Execution: An Example

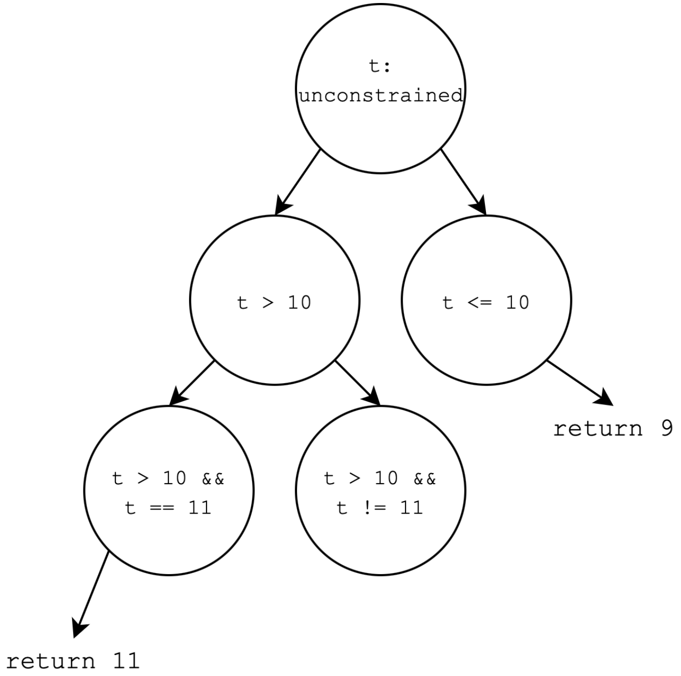
```rust
fn simple(t:u8) -> u8 {
  if t > 10 {
    if t == 11 {
      return 11;
    }
    else if t == 3 {
      panic!()
    }
    return 10;
  }
  return 9;
}
```



Symbolic Execution: An Example

Rust

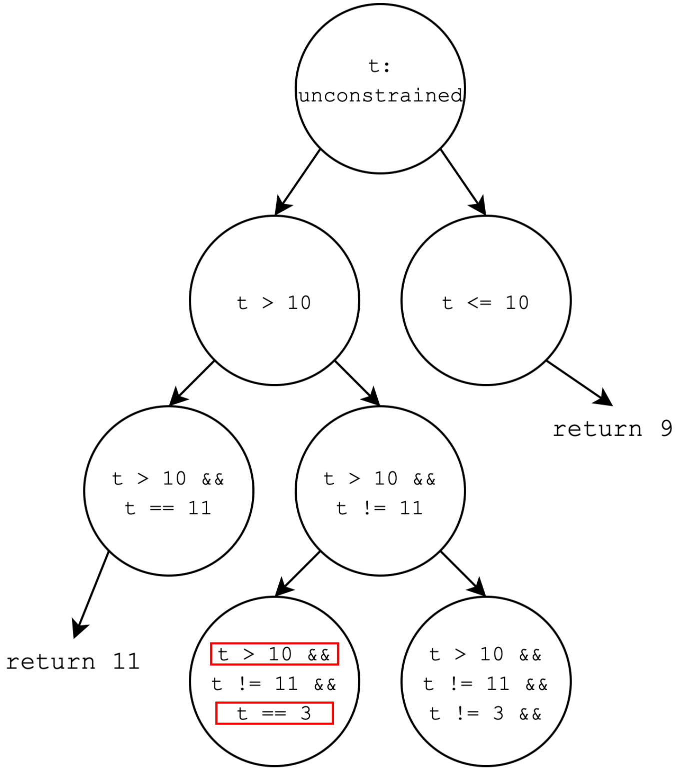
```rust
fn simple(t:u8) -> u8 {
  if t > 10 {
    if t == 11 {
      return 11;
    }
    else if t == 3 {
      panic!()
    }
    return 10;
  }
  return 9;
}
```



Symbolic Execution: An Example

Rust

```rust
fn simple(t:u8) -> u8 {
  if t > 10 {
    if t == 11 {
      return 11;
    }
    else if t == 3 {
      panic!()
    }
    return 10;
  }
  return 9;
}
```

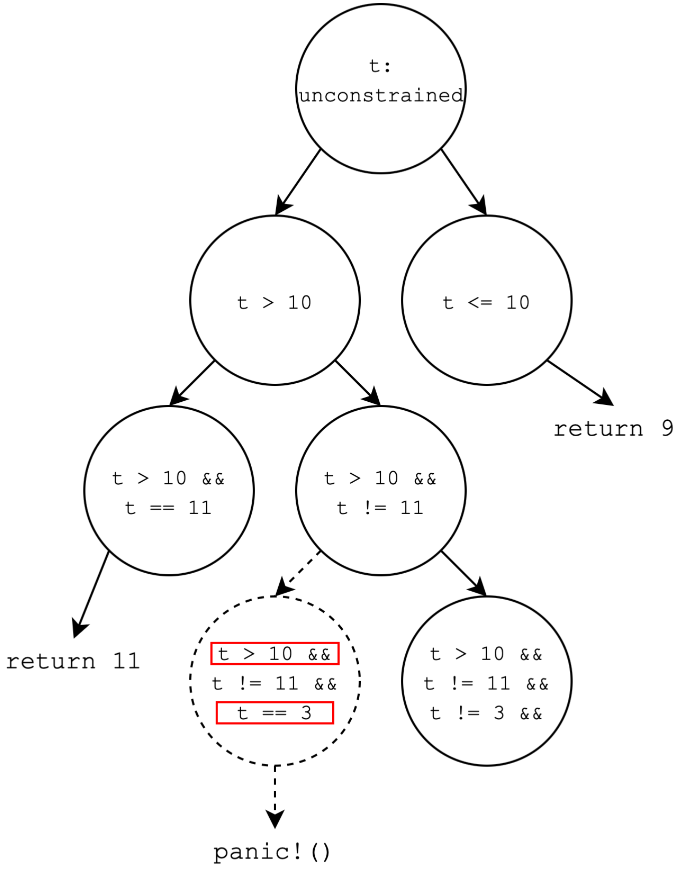


Contradiction!

Symbolic Execution: An Example

Rust

```rust
fn simple(t:u8) -> u8 {
  if t > 10 {
    if t == 11 {
      return 11;
    }
    else if t == 3 {
      panic!()
    }
    return 10;
  }
  return 9;
}
```



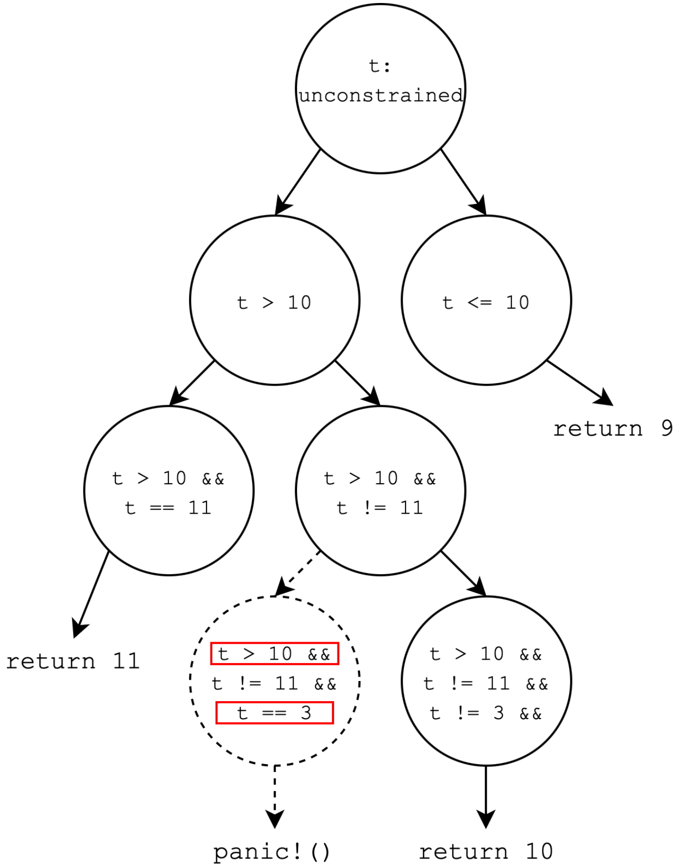
Path to panic is **unfeasible!**

Symbolic Execution: An Example

Rust

```rust
fn simple(t:u8) -> u8 {
  if t > 10 {
    if t == 11 {
      return 11;
    }
    else if t == 3 {
      panic!()
    }
      return 10;
  }
  return 9;
}
```

All paths through *simple* and valid values of *t*



Symbolic Execution in terms of Rust

- Symbolic execution can show the absence of panic (or the inverse)
- Strengthens argument for liveness
- Assertions can constrain variables further
- Showing no panic shows constraints are upheld
- Strengthens argument for safety

Rust

```rust
fn foo() -> u32 {
    let mut x = 0;
    // operate on x...
    assert(40 < x && x < 80);
    return x;
}
```

Symbolic Execution in the Wild

- Exponential growth in paths is real issue in large-scale applications (path explosion)
- Existing options: KLEE, SAGE, Crucible
- Prune paths by some heuristics (may miss paths)
- Executing Intermediate Representation(IR), e.g. LLVM-IR, may also miss paths due to e.g. loop unrolling
- Bare-metal is much simpler, path explosion may not be as much of an issue

Enter Symex

- **Pure** symbolic execution (all paths are explored)
- Executes **General Assembly**(GA)
- Abstraction over commonplace processor operations (simple register-to-register, branches, register-to-memory, etc.)
- Symex lifts an ELF binary to GA through **Translation Layer**

Translation Layer

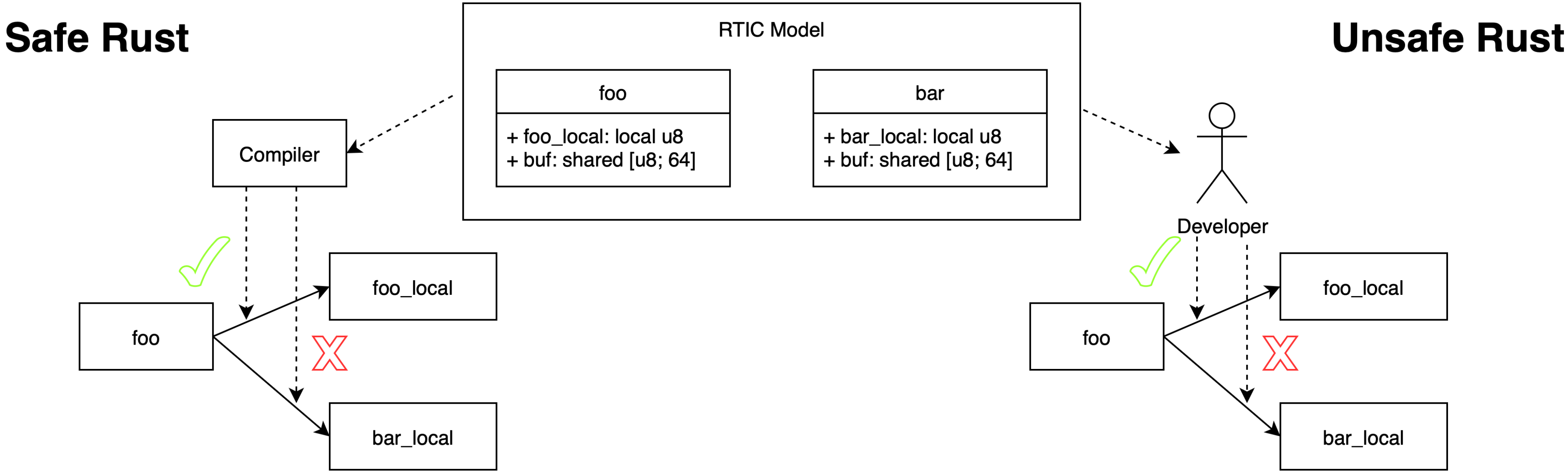
- For a given architecture, maps each ISA instruction to a sequence of GA operations
- Each instruction carries metadata, e.g. Worst-Case Execution Time(WCET)
  - Metadata may be constant (e.g. WCET = 2 cycles)
  - May also be a closure on executor state (e.g. WCET is state dependent)
  - Allows modelling e.g. pipelines, branch predictors
- Translation layers implemented for ARMv6/7 and RV32I

Note on Scheduling Analysis

- WCET for instructions is interesting in terms of SRP
- Methods for scheduling analysis (response time, overall schedulability) are well-known for SRP
  - The only unknown is the WCET of a task (and the critical sections, two sides of the same coin)
  - With Symex determining WCET amounts to adding up instruction costs along the longest path
- Similar for stack depth, track path with largest stack pointer difference

What else can we do

- Ensuring safety of unsafe Rust requires context
- RTIC provides outset for reasoning around unsafe

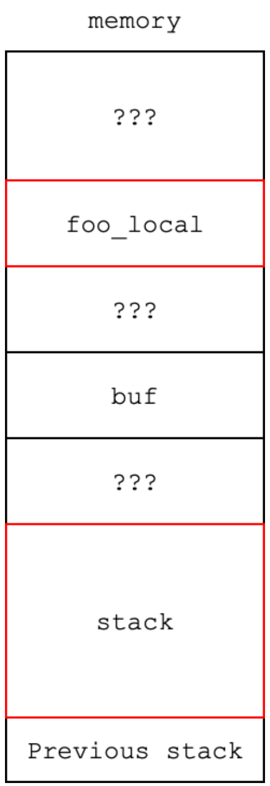


What else can we do

```rust
#[task(..., local = [foo_local], shared = [buf])]
fn foo(cx: foo::Context) {
  let a = *cx.local.foo_local;
  *cx.local.foo_local = a >> 1;
  cx.shared.buf.lock(|buf| {
    buf[1] = a;
  });
  unsafe {
    (_rtic_shared_resource_buf.get_mut() as *mut u32).write(2);
    core::ptr::write(0x13371330 as *mut _, 2);
  }
}
```

What else can we do

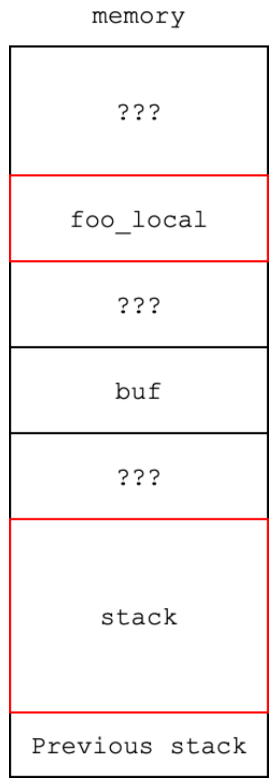
```rust
#[task(..., local = [foo_local], shared = [buf])]
fn foo(cx: foo::Context) {
  let a = *cx.local.foo_local;
  *cx.local.foo_local = a >> 1;
  cx.shared.buf.lock(|buf| {
    buf[1] = a;
  });
  unsafe {
    (_rtic_shared_resource_buf.get_mut() as *mut u32).write(2);
    core::ptr::write(0x13371330 as *mut _, 2);
  }
}
```



All good!

What else can we do

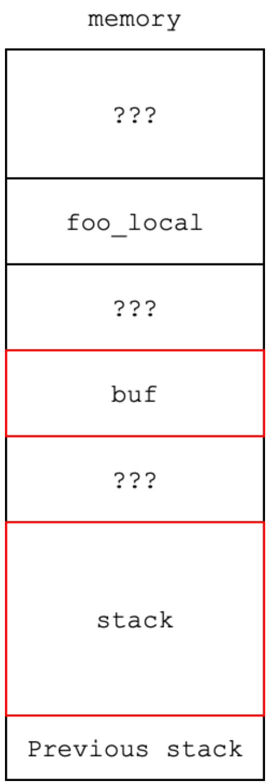
```rust
#[task(..., local = [foo_local], shared = [buf])]
fn foo(cx: foo::Context) {
  let a = *cx.local.foo_local;
  *cx.local.foo_local = a >> 1;
  cx.shared.buf.lock(|buf| {
    buf[1] = a;
  });
  unsafe {
    (_rtic_shared_resource_buf.get_mut() as *mut u32).write(2);
    core::ptr::write(0x13371330 as *mut _, 2);
  }
}
```



Still all good!

What else can we do

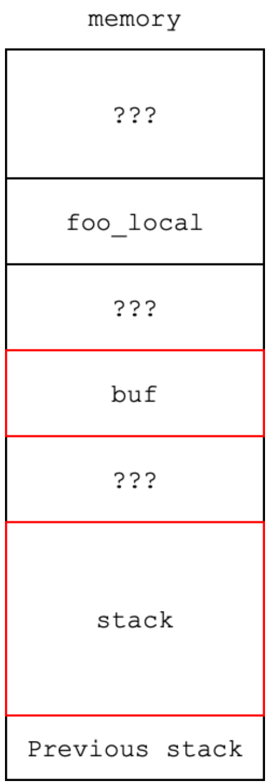
```rust
#[task(..., local = [foo_local], shared = [buf])]
fn foo(cx: foo::Context) {
  let a = *cx.local.foo_local;
  *cx.local.foo_local = a >> 1;
  cx.shared.buf.lock(|buf| {
    buf[1] = a;
  });
  unsafe {
    (_rtic_shared_resource_buf.get_mut() as *mut u32).write(2);
    core::ptr::write(0x13371330 as *mut _, 2);
  }
}
```



Still all good!

What else can we do

```rust
#[task(..., local = [foo_local], shared = [buf])]
fn foo(cx: foo::Context) {
  let a = *cx.local.foo_local;
  *cx.local.foo_local = a >> 1;
  cx.shared.buf.lock(|buf| {
    buf[1] = a;
  });
  unsafe {
     (_rtic_shared_resource_buf.get_mut() as *mut u32).write(2);
    core::ptr::write(0x13371330 as *mut _, 2);
  }
}
```



Potential race condition (**bad**)!

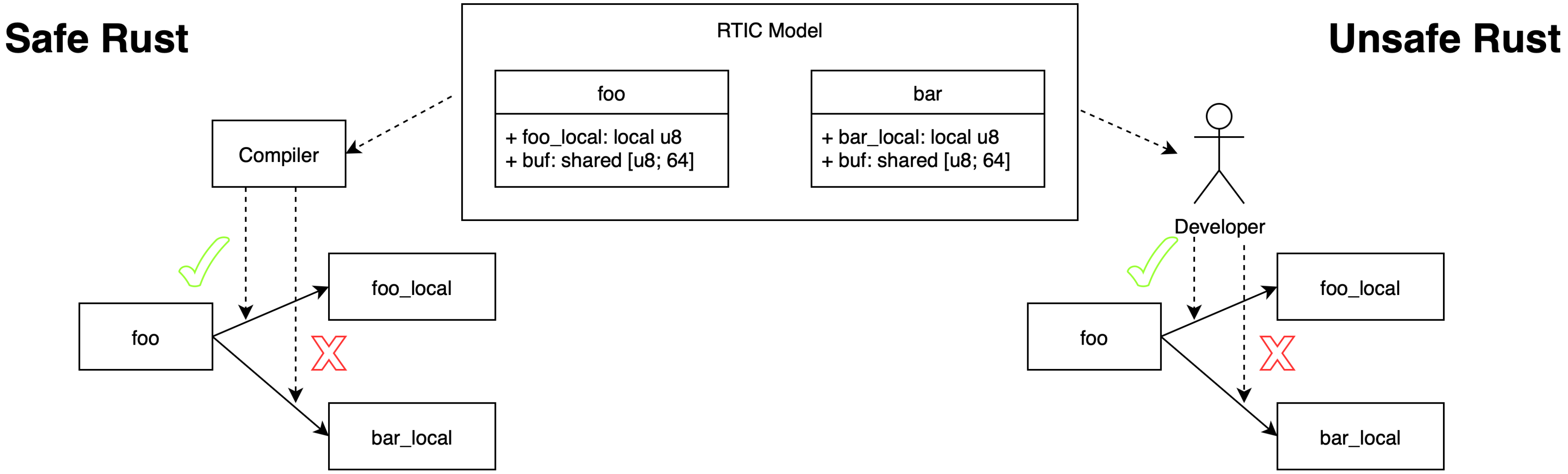
What else can we do

```rust
#[task(..., local = [foo_local], shared = [buf])]
fn foo(cx: foo::Context) {
  let a = *cx.local.foo_local;
  *cx.local.foo_local = a >> 1;
  cx.shared.buf.lock(|buf| {
    buf[1] = a;
  });
  unsafe {
    (_rtic_shared_resource_buf.get_mut() as *mut u32).write(2);
    core::ptr::write(0x13371330 as *mut _, 2);
  }
}
```

Bad past the point of telling…

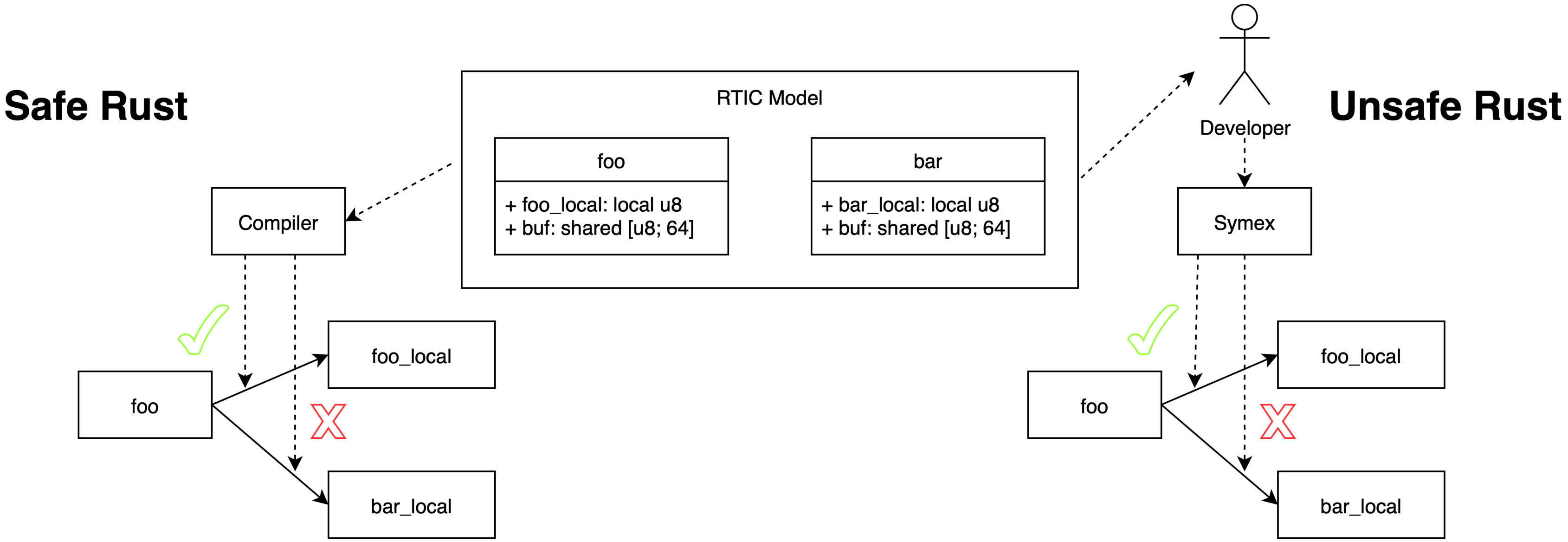
What else can we do

- Mistakes do happen
- Can we replace the developer?



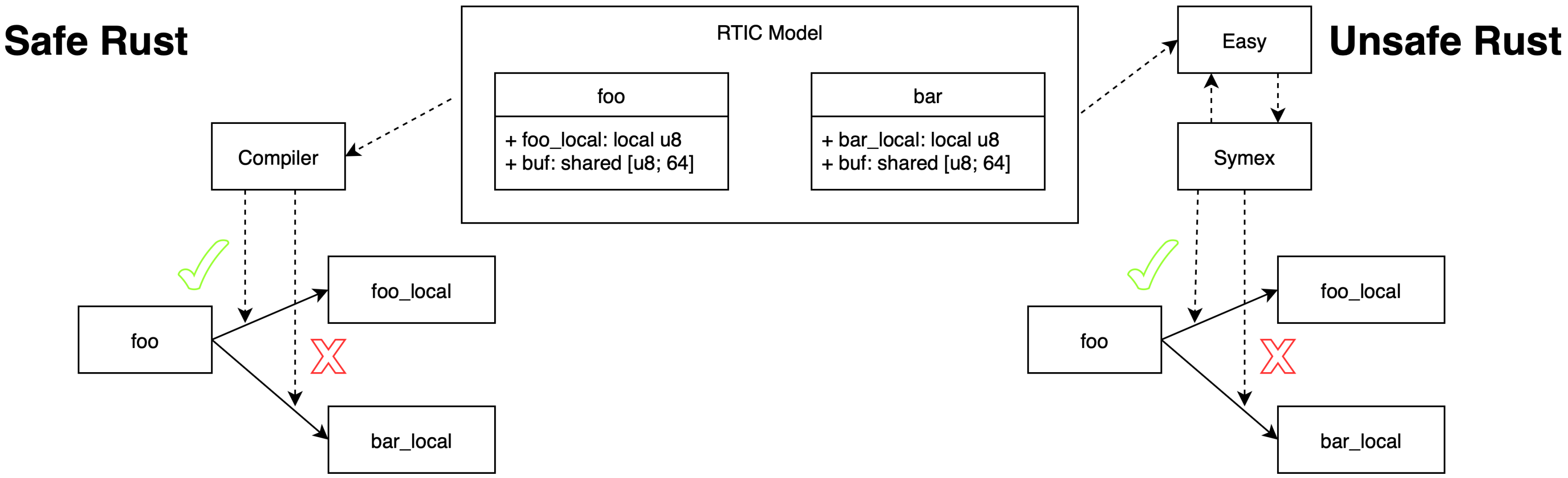
What else can we do

- Resources in RTIC are either statically allocated or on the stack
- Feed this info to Symex, verify all memory access instructions



What else can we do

- **Easy** extracts meaningful info from RTIC model
- **Symex** uses info to verify all memory accesses, absence of panic
- **WCET** info from Symex used for SRP scheduling analysis



EASY report

```
Task: task_3 (priority 2)
wcet: 42.812uS
Time preempted: 13.438uS
Number of preemptions: 1
Time blocked: 2.812uS (by __easy_internal_button_handler_trampoline)
Deadline: 1.000mS
Core util: 0.04281249999999996%
Period: 2.000mS
Response time: 59.062uS
Is schedulable: true
Stack usage: 40
Preempted by:

   Preempted by: button_receiver 1 times
```

```
Worst case utilization 0.04434156249999994%
Worst case stack usage 192 Bytes
```

EASY report, p.2

Rust

```rust
#[task(..., local = [foo_local], shared =
[buf])]
fn foo(cx: foo::Context) {
  let a = *cx.local.foo_local;
  *cx.local.foo_local = a >> 1;
  cx.shared.buf.lock(|buf| {
    buf[1] = a;
  });
  unsafe {
    core::ptr::write(0x13371330 as *mut _, 2);
  }
}
```

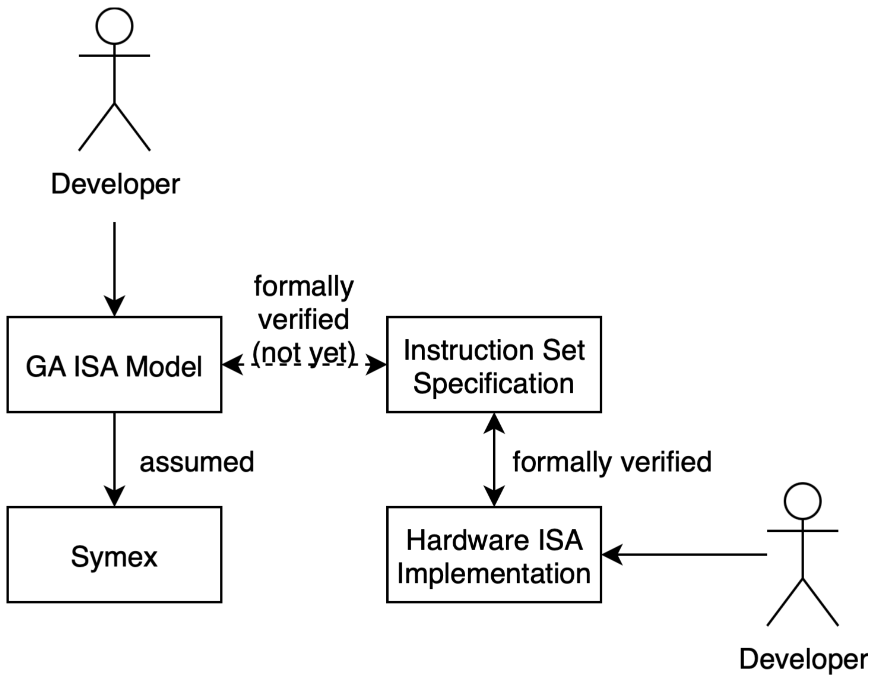
```
Path
Used 64 bytes of stack
Result: Failure Tried to write to 0x13371330, on stack: Some(false) @ PC : 0x81070a → in file /home/pawel/.rustup/toolchains/nightly-x86_64-unknown-linux-gnu/lib/rustlib/src/rust/library/core/src/ptr/mod.rs on line 1932 (__easy_internal_foo_trampoline)

    Critial region
    With priority: 2
    WCET: 11.406uS

        Critial region
        With priority: 3
        WCET: 2.813uS

pawel@archlinux ~/f/e/symex-demo (master) $
```

Preliminary testing

- For ARMv7, hardware-in-the-loop confirms instruction level estimates
  - Correct instruction results
  - Safe execution time bounds
- Colleagues at Grepit are trying to pass their production system through the tool
  - 20k LoC w/o dependencies
  - 1M+ LoC w/ dependencies
  - ~30 mins, albeit still work in progress

Future Work

- Further temporal modelling
  - Difficult to derive accurate figures for ARM due to lacking specification
  - RISC-V models can be derived directly from HDL
- Functional verification of GA ISA models against formal specifications (e.g. SAIL models)



Questions

- Email: pawel.dzialo@ltu.se