

# Firmware Surgery

Cutting, Patching and Instrumenting Firmware  
for Debugging the Undebuggable

Henrik Ferdinand Nölscher – DEEPSEC21  
@slckcc

# whoami

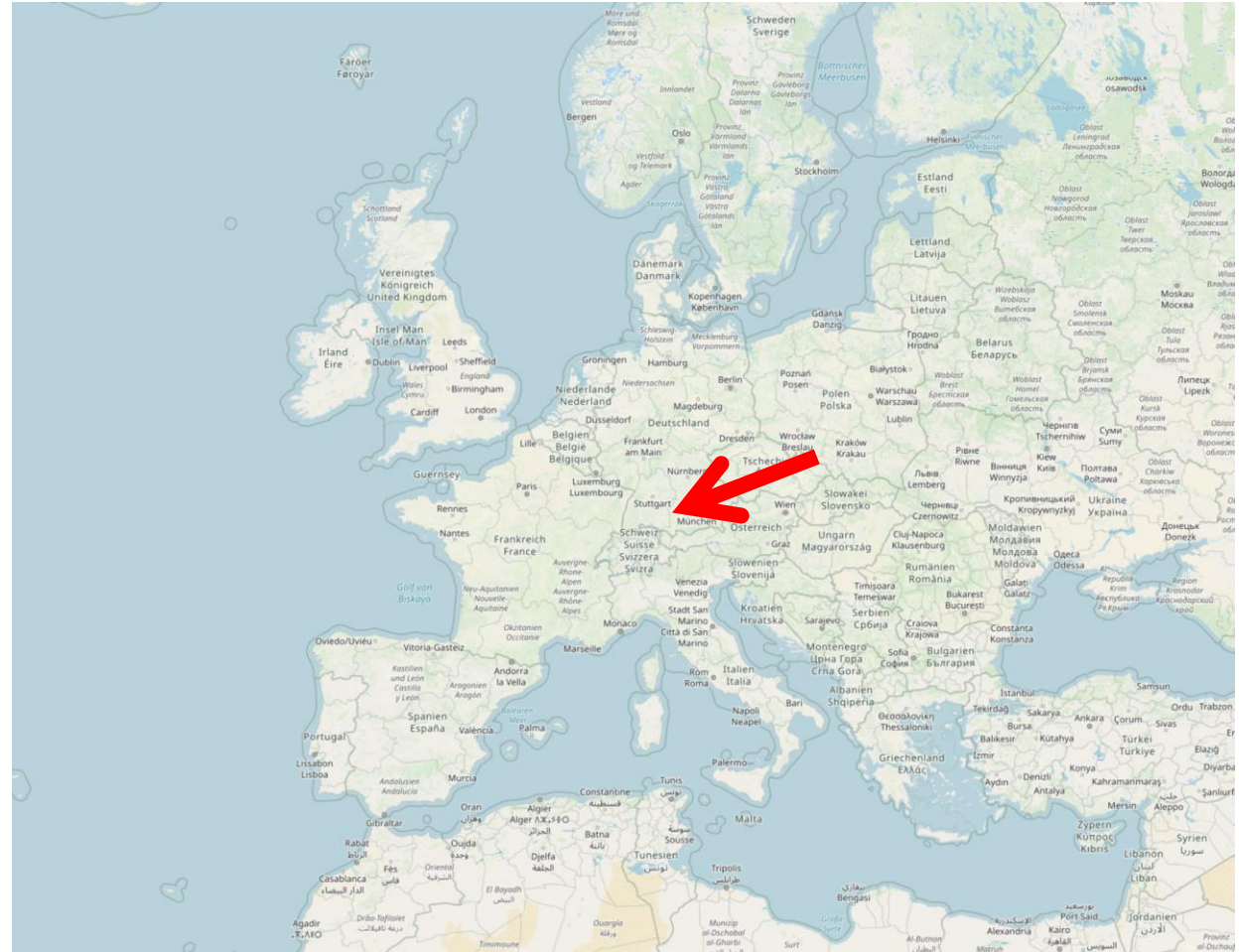
Henrik Ferdinand Nölscher

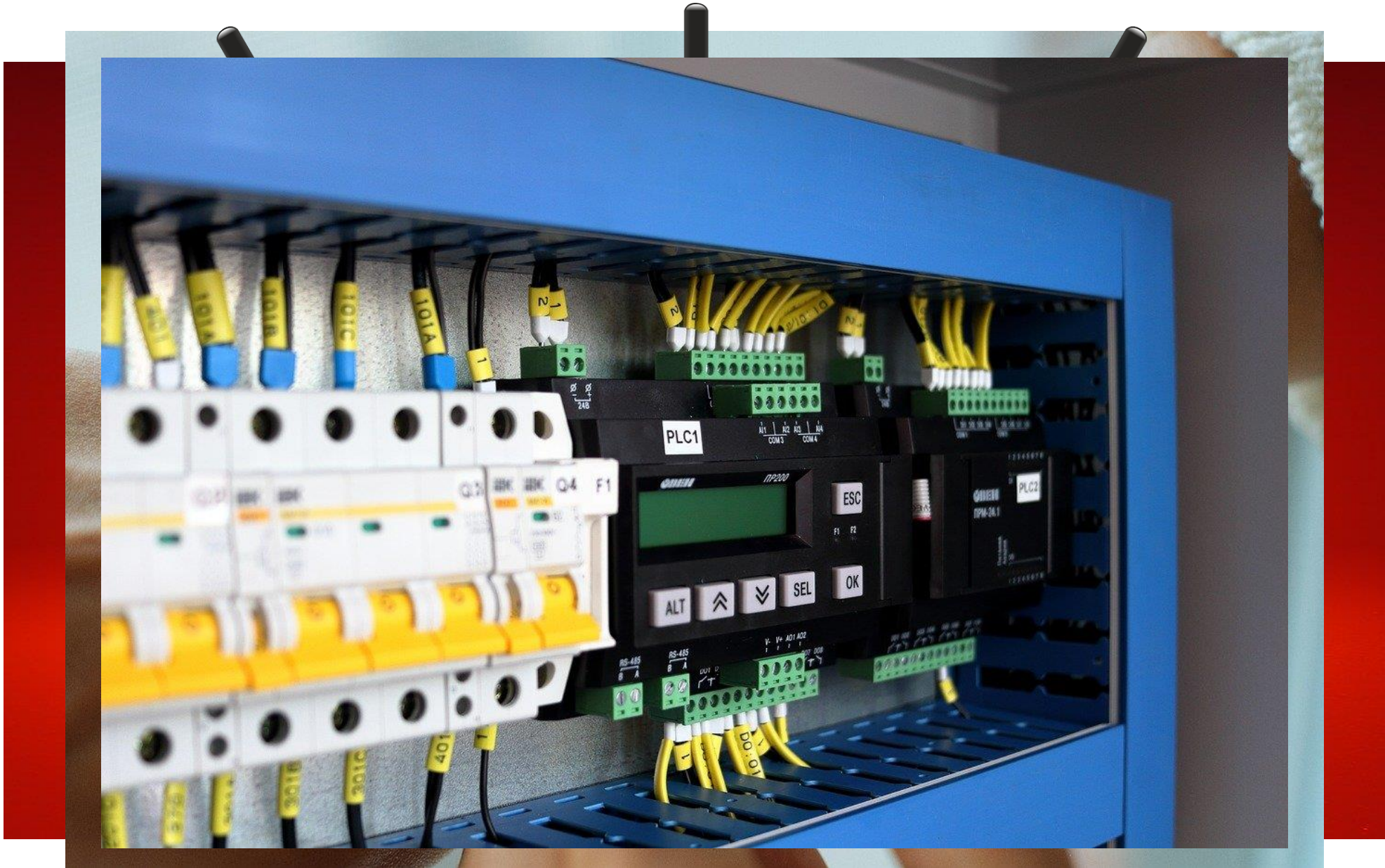


I analyze embedded systems

Published & Presented at Usenix, Defcon etc.

Embedded Linux Security, Fault Injection and  
Firmware Reverse Engineering





# Target Device Types

Type I

Rich OS-based devices  
Run some kind of operating system kernel and one or multiple applications atop such kernel.

Type II

Embedded OS-based devices  
Feature a (real-time) operating system, including lightweight abstractions for task scheduling and messaging

Type III

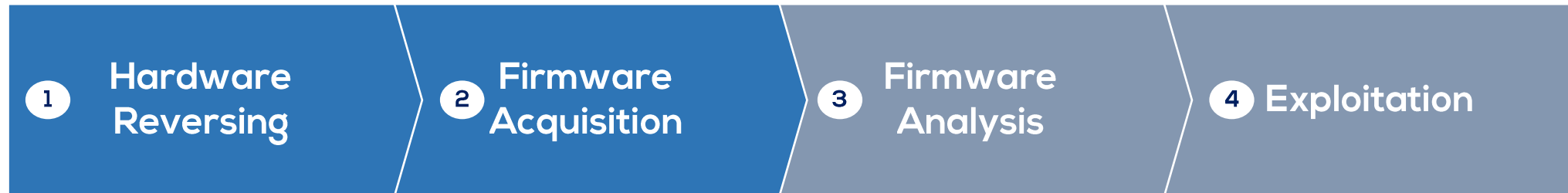
Bare-Metal Devices.  
Run a single, binary firmware image with few or no separation between tasks.

**Our Focus**



\*Type Definition from: Muench et al., What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices

# “Embedded Security Testing is Easy”



## Equipment

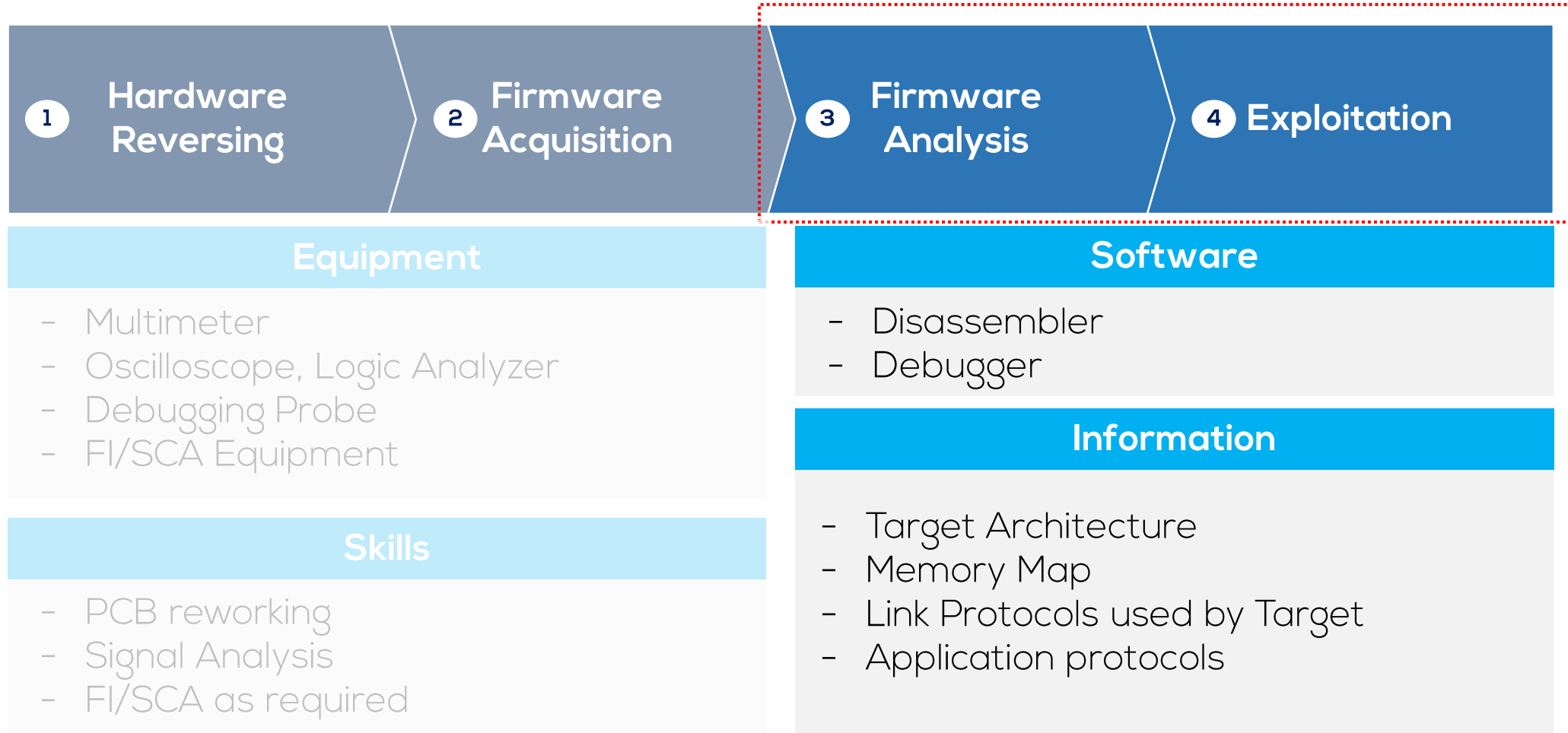
- Multimeter
- Oscilloscope, Logic Analyzer
- Debugging Probe
- FI/SCA Equipment

## Skills

- PCB reworking
- Signal Analysis
- FI/SCA as required

# "Embedded Security Testing is Easy"

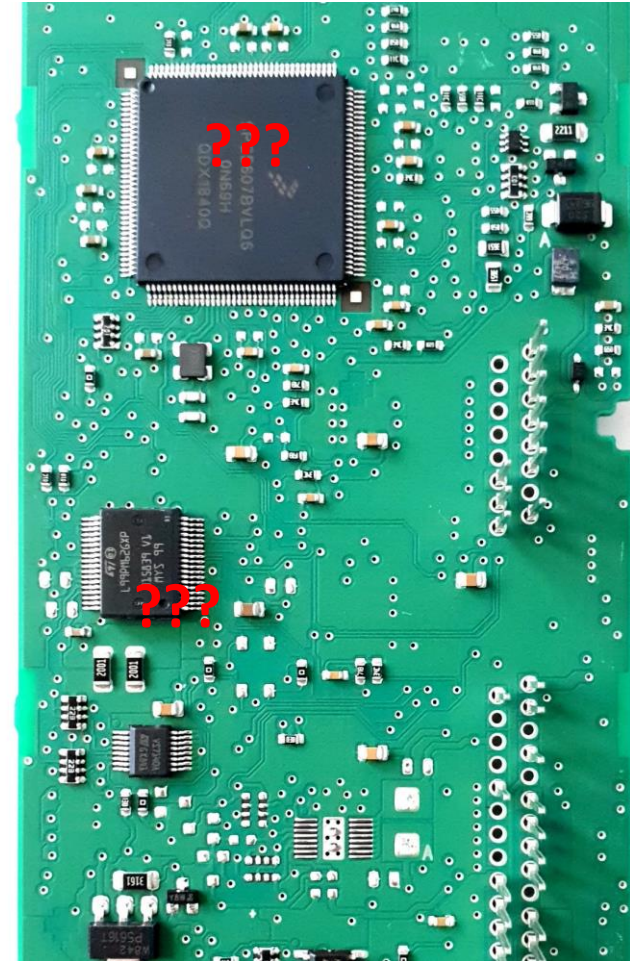
Our Focus



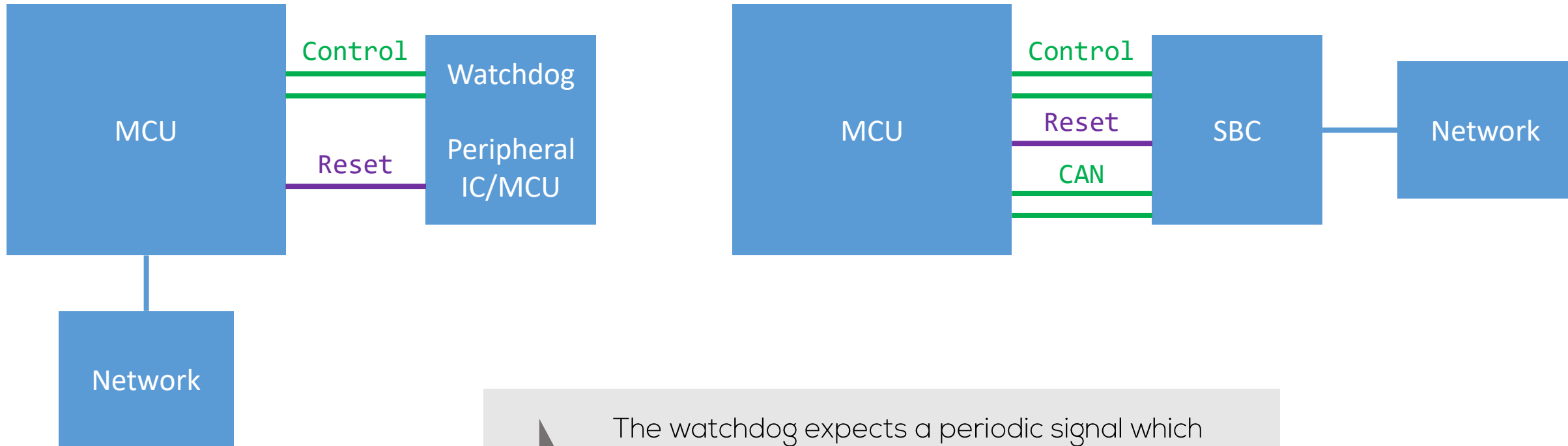
# “Embedded Security Testing is Hard!”

What if:

- The system's safety features work against us?
- We lack equipment?
- We lack critical information?



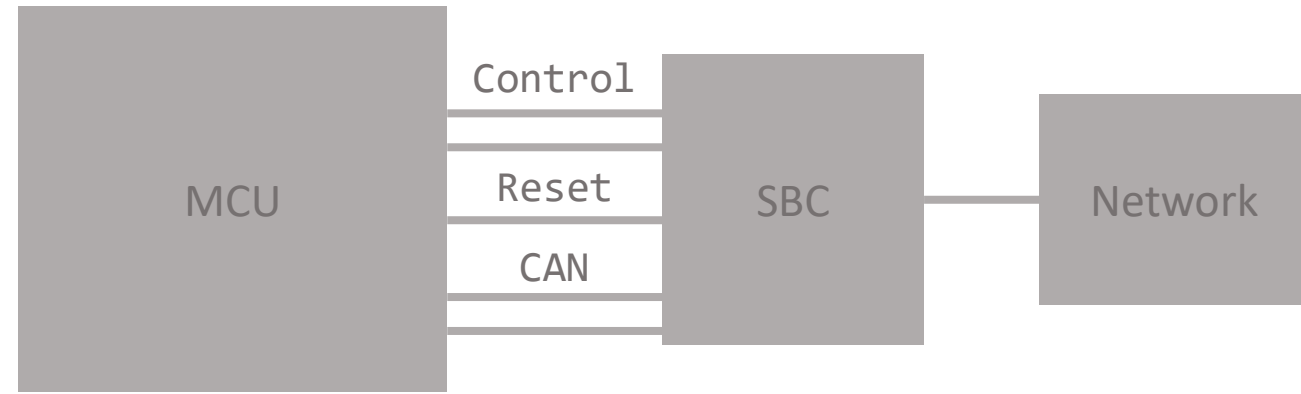
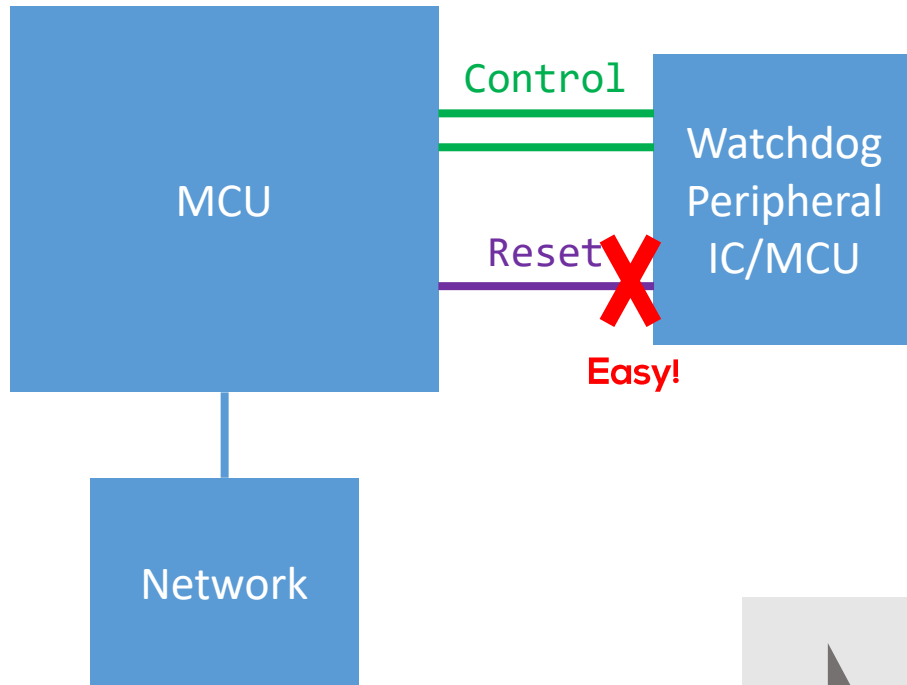
# Watchdogs & SBCs



The watchdog expects a periodic signal which indicates that the MCU is working correctly.

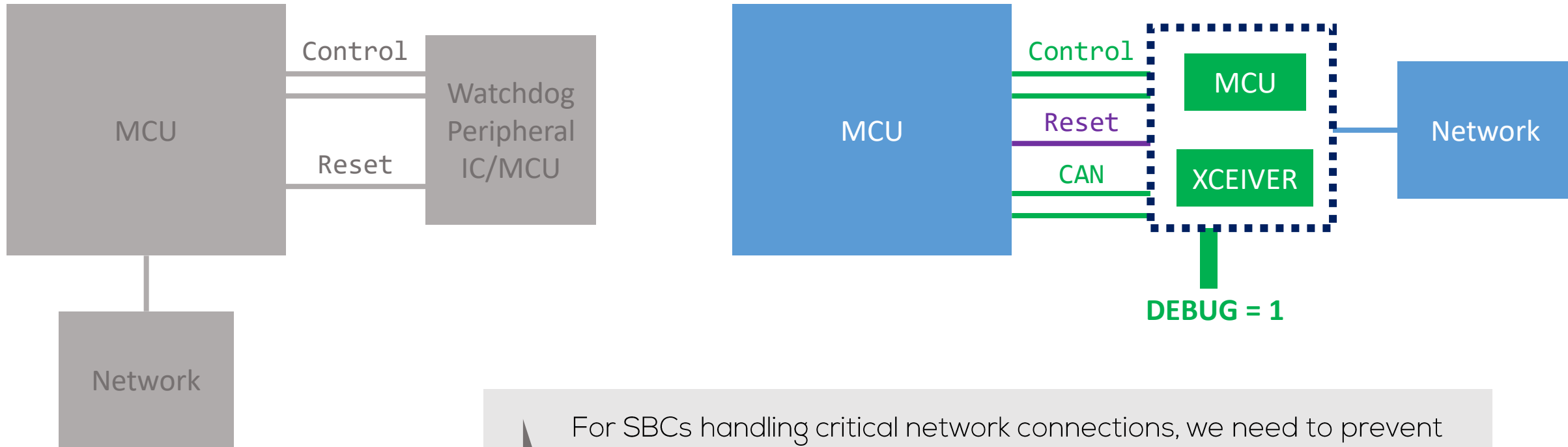
During interactive debugging, this missing signal will reset the device, destroying our session!

# Watchdogs & SBCs



Peripheral MCUs or Watchdogs can be kept quiet by cutting the connection controlling the reset line of our target MCU

# Watchdogs & SBCs



For SBCs handling critical network connections, we need to prevent entering a fail state instead of disabling the reset output!

Available Options include:

1) Entering special mode 2) Man-in-the-Middle 3) Emulating SBC

# “Embedded Security Testing is Hard!”

What if:

- The system's safety features work against us?
- We lack equipment?
- We lack critical information?

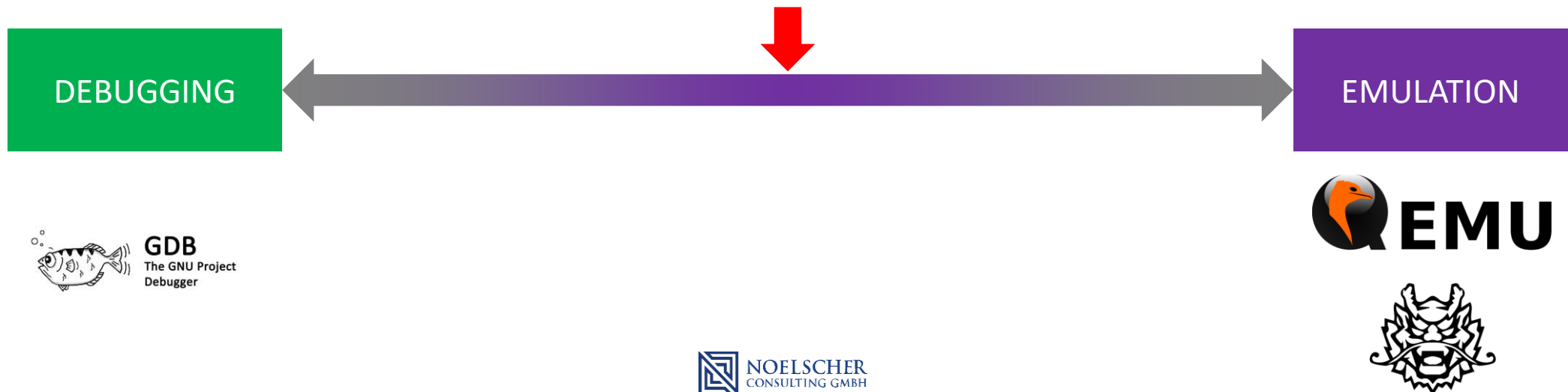


# Missing Debug Access

Think of the following Scenarios:

1. Debug pins are used for something else or not present at all
2. Debugging is very hard because of timing constraints: Radio, USB!
3. You “only” get code execution, dump the Firmware but you can’t open the debug port

How can we solve this?



# Flopz – Firmware Liberation on Python

**Flopz** patches & instruments raw firmware, adding trace capabilities and backdoors.

It supports:

- PPC (VLE, only)
- ARM Thumb Mode
- RISC-V RV32I
- IA-32

Written from scratch in pure python

## Prerequisites:

- Knowledge of target architecture
- Firmware r/w access
- Channel for transferring Data
- Host computer with Python3
- External disassembler



**Our goal: Make firmware instrumentation as easy as possible!**

# Why Flopz?

## Why not use Keystone?

- Keystone is based on LLVM and supports a broad range of Architectures
- Adding architectures beyond LLVM's support can be challenging
- Keystone solves the problem of generating instructions, but (as far as I know) doesn't provide helpers for patching, dynamic payload generation etc. by design

## Why not use rasm2?

- Like many other tools, rasm2 turns strings of assembly into bytes
- However, we'd like to have a model of our system in high-level code, including Registers, Functions, Peripherals in order to increase payload reusability

# Why Flopz?

```
from r2.r_asm import *

def asm(a, arch, op):
    a.use(arch)
    code = a.massemble(op)
    print "HEX: %s"%code.buf_hex

a = RAsm()
asm(a, 'x86.olly', 'mov eax, 33')
```

```
from flopz.arch.ppc.vle.e200z0 import E200Z0
from flopz.core.shellcode import Shellcode
from flopz.arch.ppc.vle.instructions import *
from flopz.arch.ppc.vle.auto_instructions import *

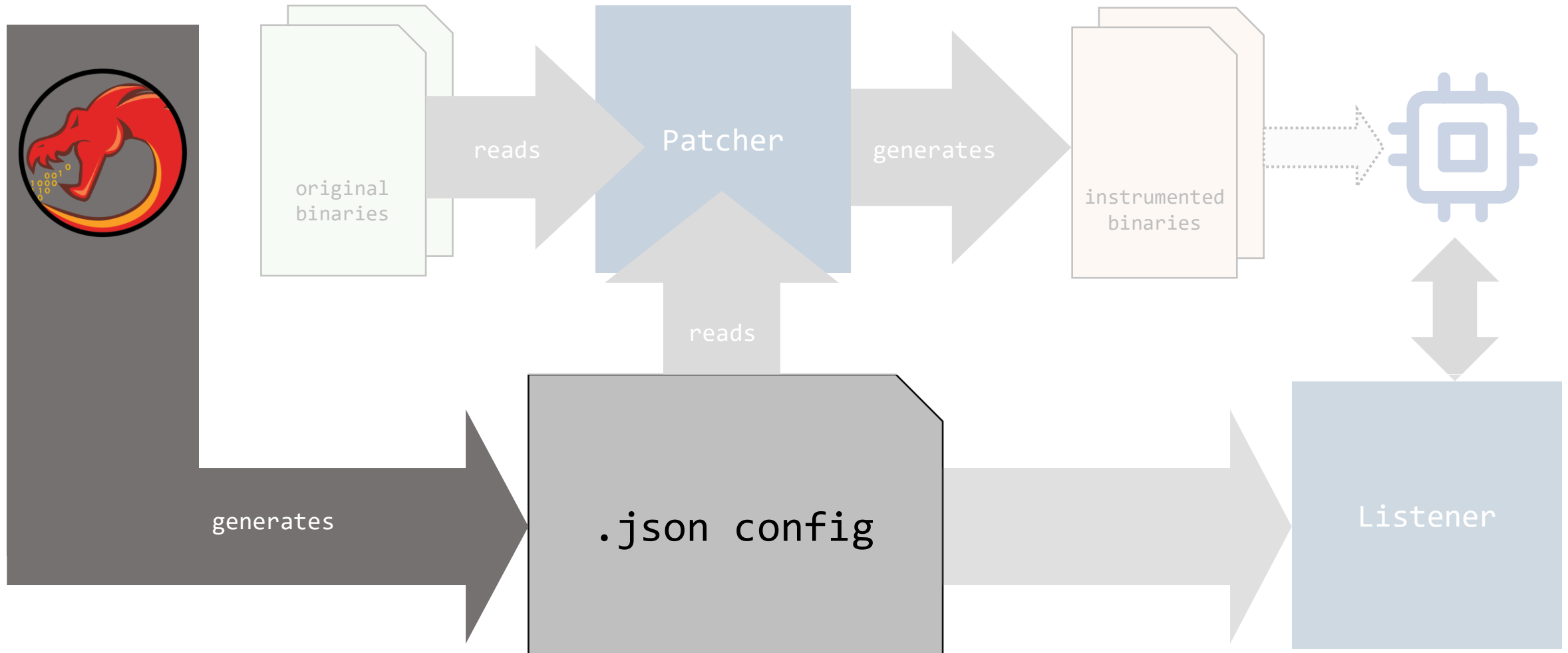
core = E200Z0()
Shellcode(address=0x123, instructions=[
    AutoLoadI(core.r5, 0xfffffc108),
    Selwz(core.r6, core.r5, 2) # r6 = [r5+2]
]).bytes()
```

# Why Flopz?

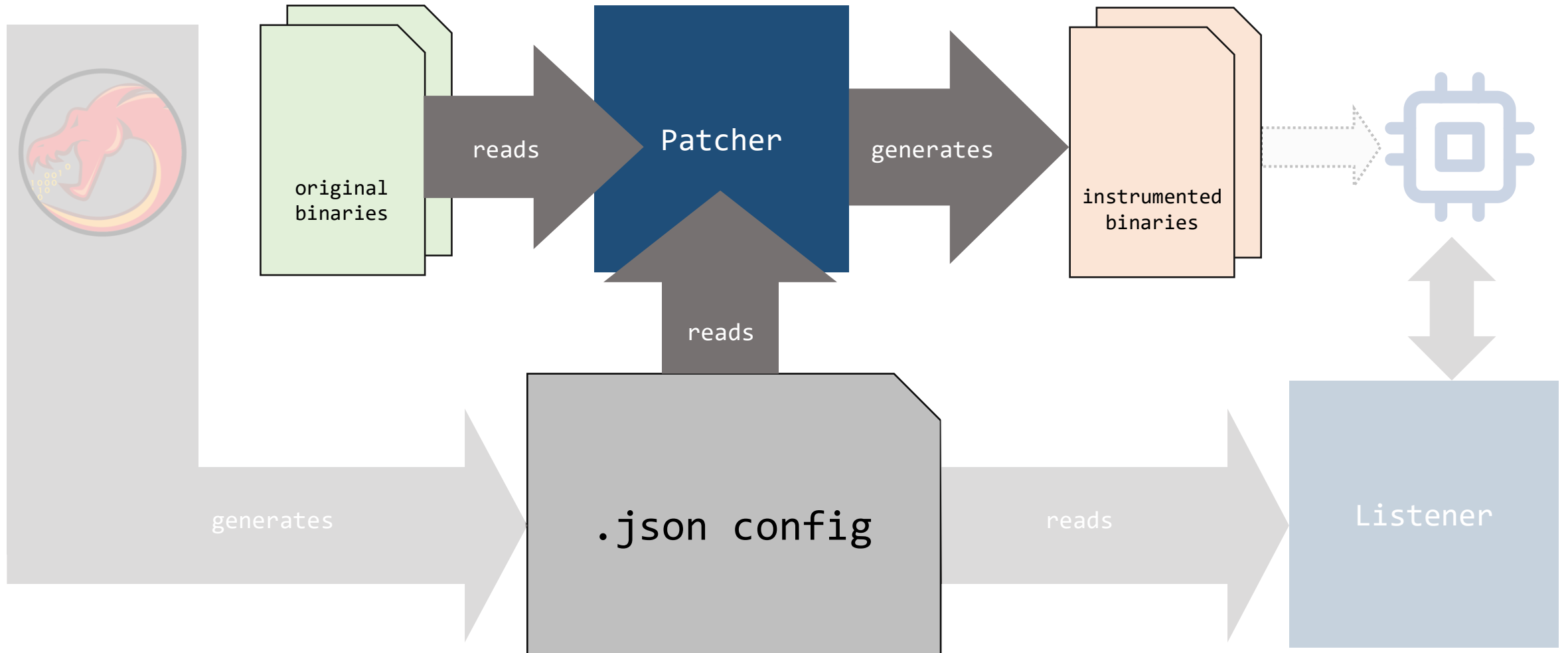
```
from flopz.arch.ppc.vle.e200z0 import E200Z0
from flopz.core.shellcode import Shellcode
from flopz.arch.ppc.vle.instructions import *
from flopz.arch.ppc.vle.auto_instructions import *

core = E200Z0()
Shellcode(address=0x123, instructions=[
    LabelRef("bail_out", lambda addr: SeBlt(addr)),
    AutoLoadI(core.r5, 0xfffffc108),
    Selwz(core.r6, core.r5, 2), # r6 = [r5+2]
    Label("bail_out"),
]).bytes()
```

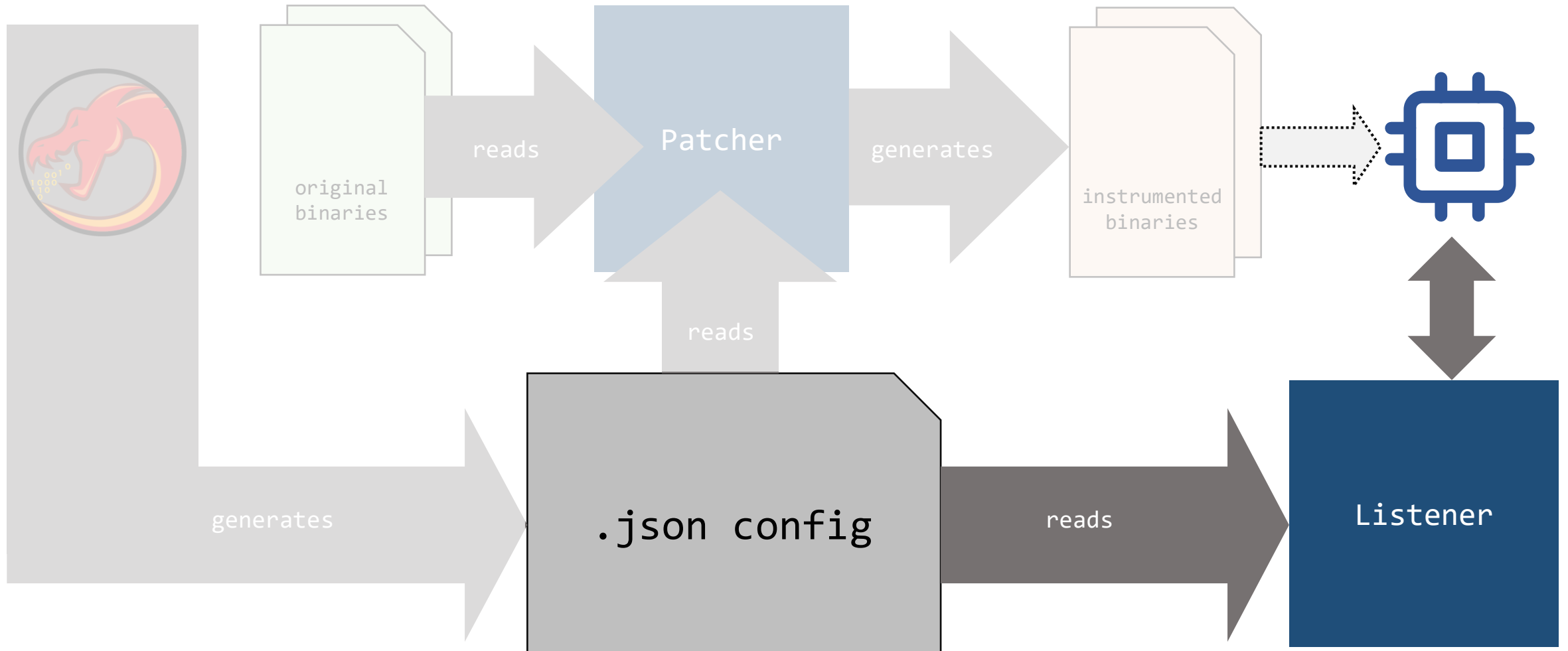
# Flopz Architecture



# Flopz Architecture



# Flopz Architecture



# Instrumentation Internals

1. How to gain control over execution cleanly?
2. Where to put the extra data?
3. How to communicate with the target?

# Internals: Redirecting Control Flow

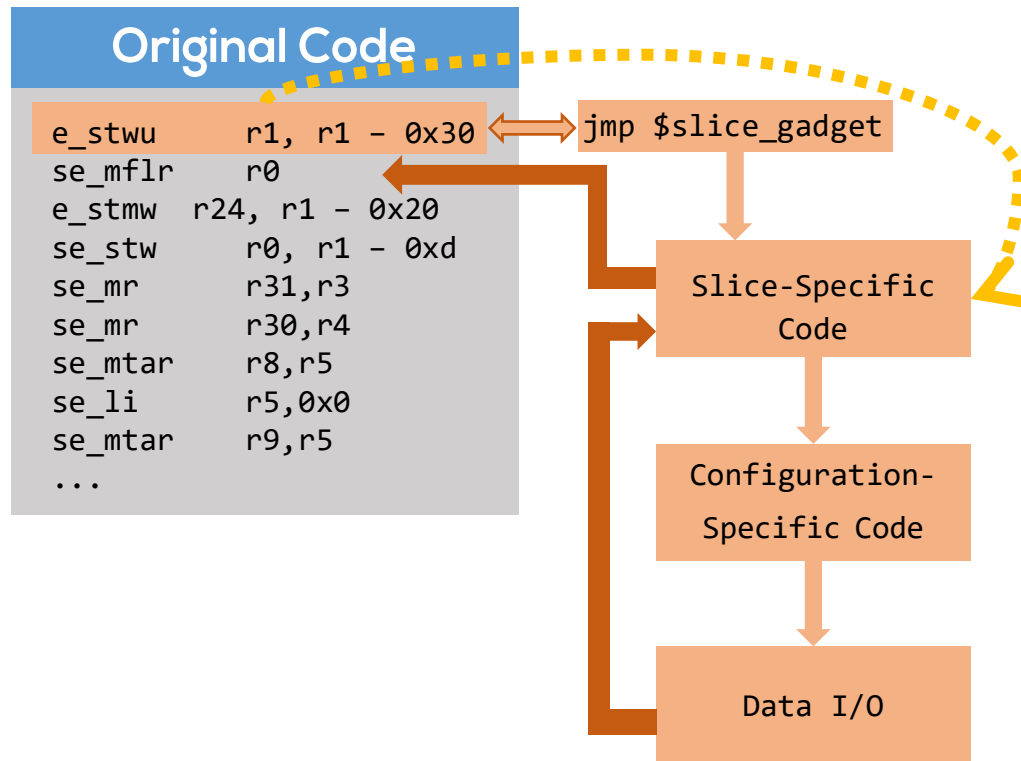
“The Trampoline”:

## Original Code

```
e_stwu    r1, r1 - 0x30
se_mflr   r0
e_stmw    r24, r1 - 0x20
se_stw    r0, r1 - 0xd
se_mr     r31, r3
se_mr     r30, r4
se_mtar   r8, r5
se_li     r5, 0x0
se_mtar   r9, r5
...
```

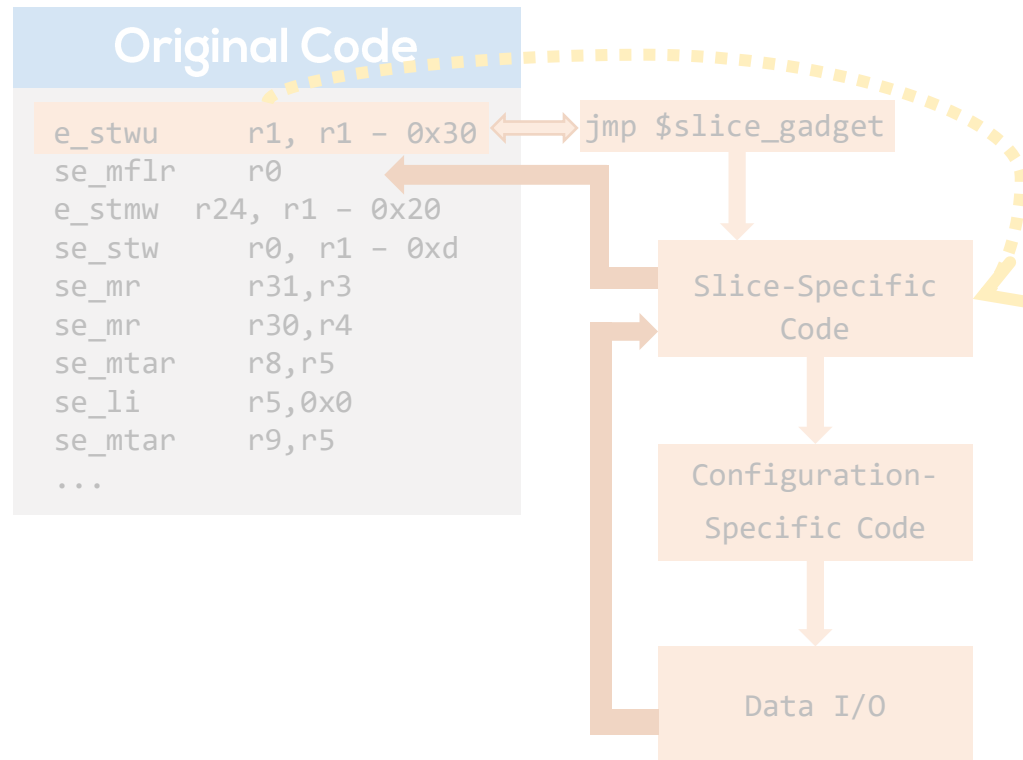
# Internals: Redirecting Control Flow

“The Trampoline”:

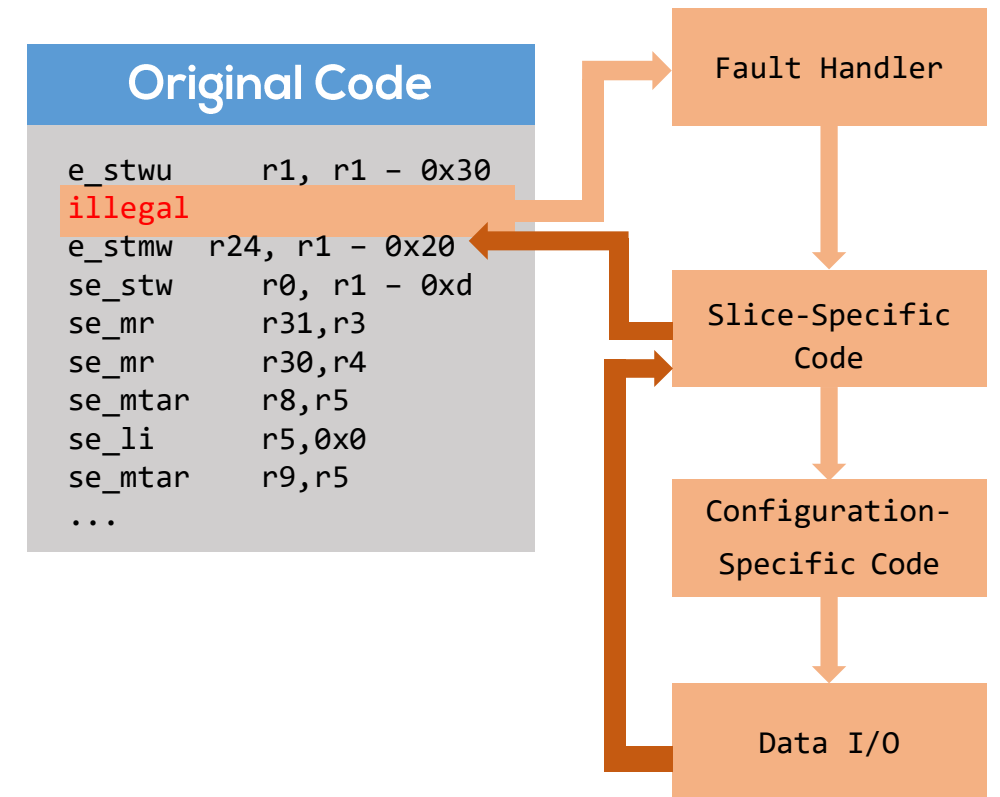


# Internals: Redirecting Control Flow

"The Trampoline":

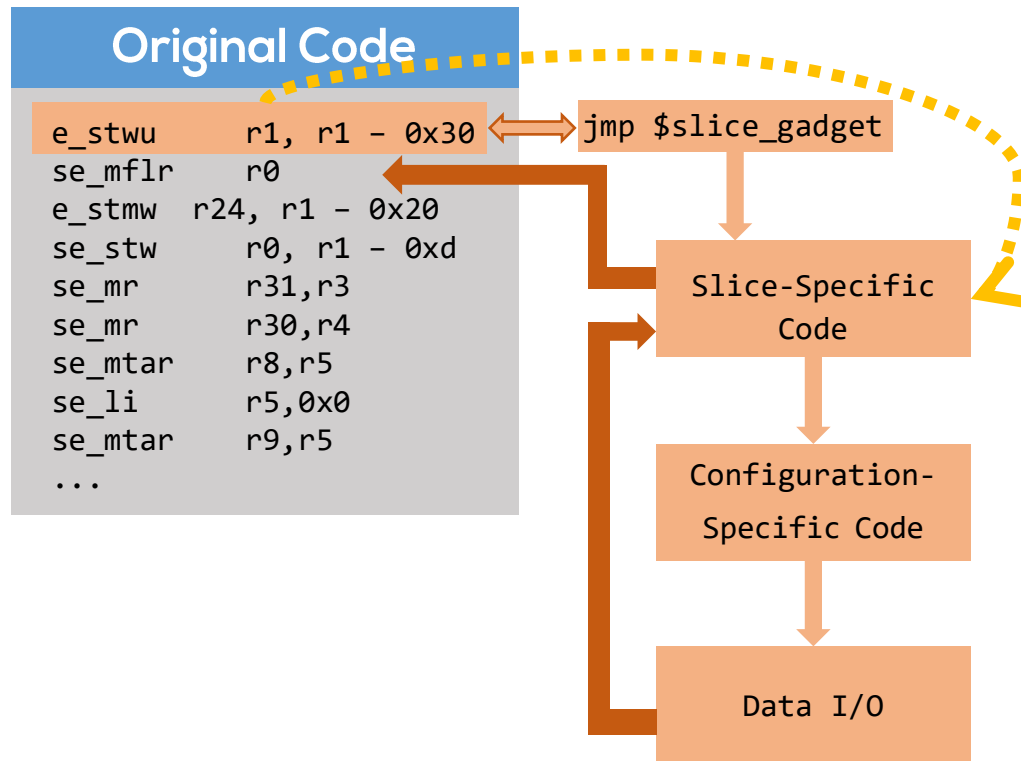


"The Corruption":

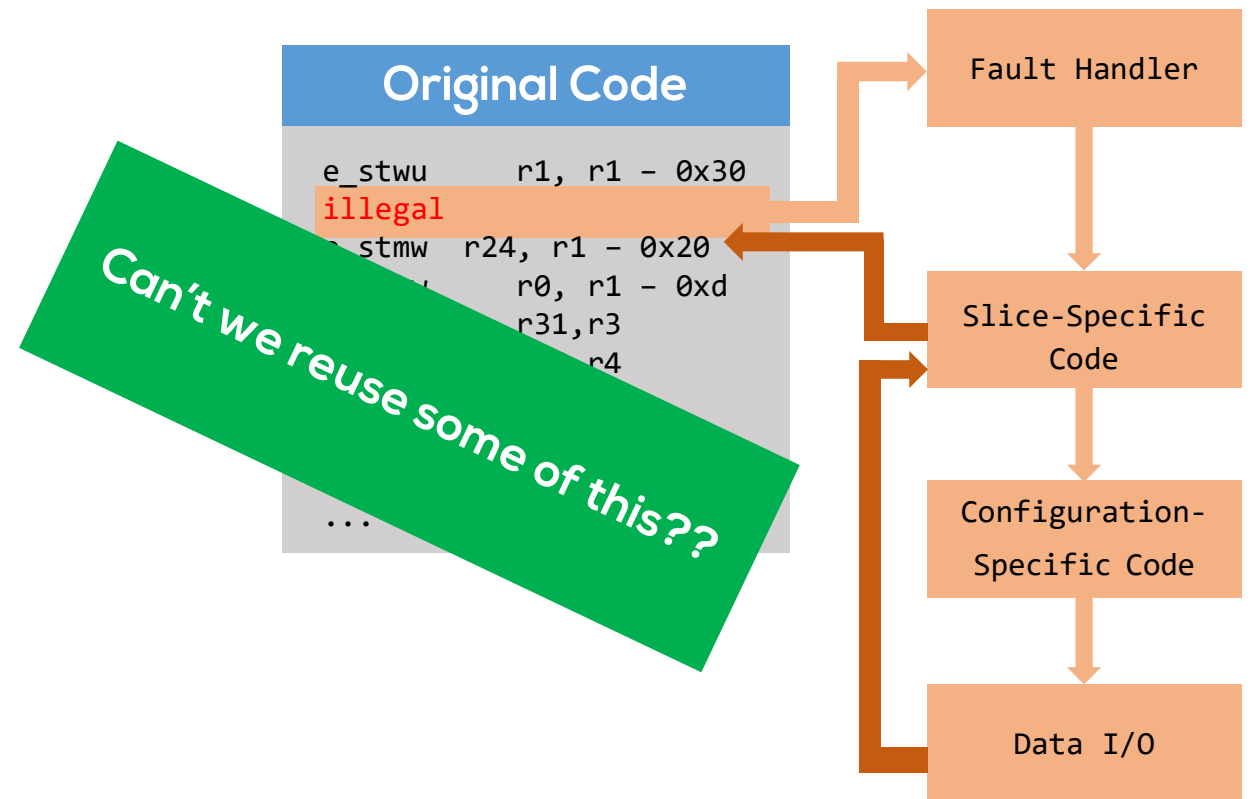


# Internals: Redirecting Control Flow

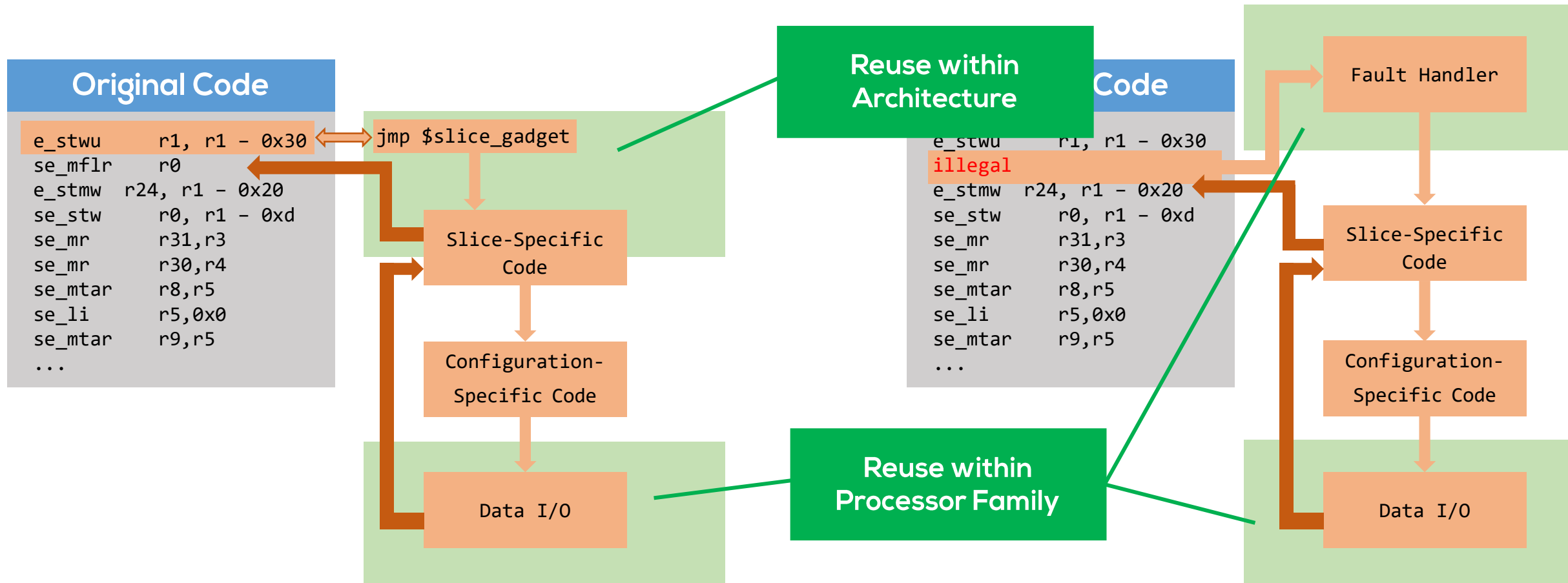
## "The Trampoline":



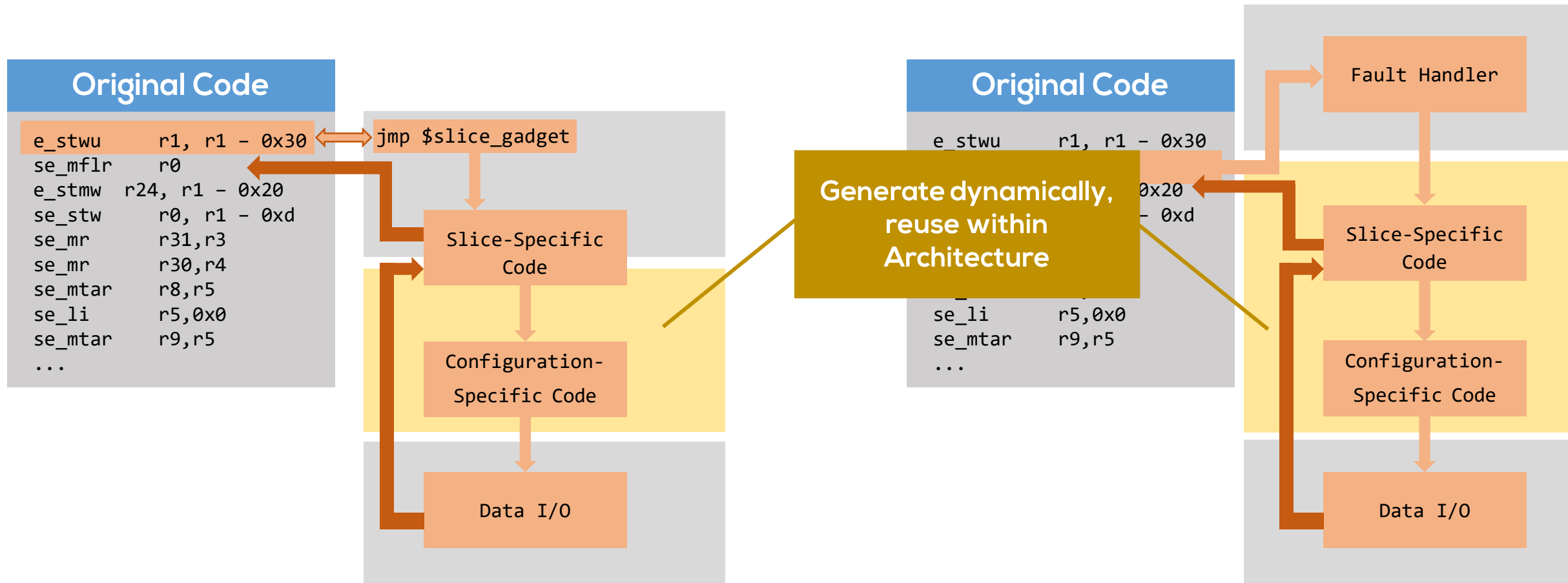
## "The Corruption":



# Internals: Redirecting Control Flow



# Internals: Redirecting Control Flow

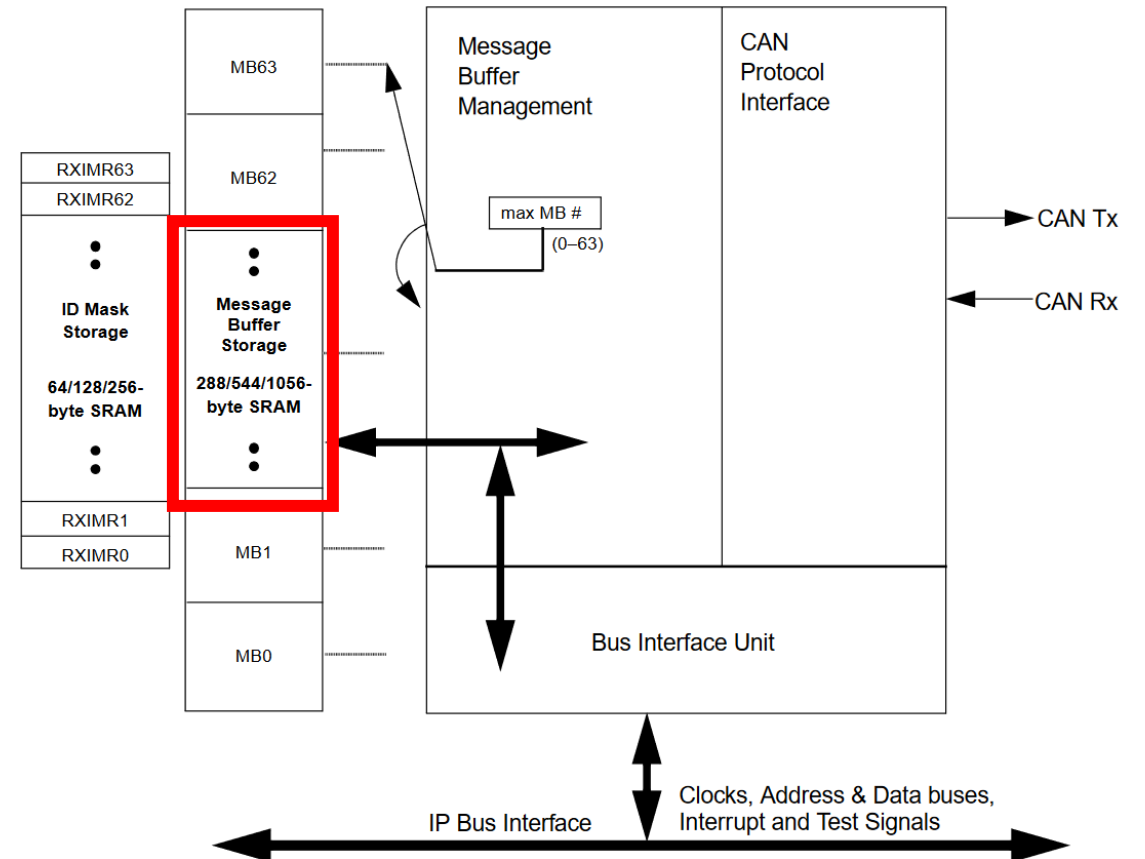


# Instrumentation Internals

1. How to gain control over execution cleanly?
2. Where to put the extra data?
3. How to communicate with the target?

# Internals: Storing Data

- For extra instructions, the user selects a range of Flash.
- But we need to set up some kind of stack somewhere, even if it's small
- RAM is usually very limited
- Solution: Peripheral Memory



# Instrumentation Internals

1. How to gain control over execution cleanly?
2. Where to put the extra data?
3. How to communicate with the target?

# Data Channels

(Q|D) SPI

Ethernet

CAN(-FD) / Flexray

UART

PWM

**slow, but commonly present and  
require low amount of pins**

# Data Channels

(Q|D) SPI

Ethernet

CAN(-FD) / Flexray

**faster, but more complex to set up - more things can fail and more pins are required**

UART

PWM

# Data Channels

(Q|D) SPI

Ethernet

CAN(-FD) / Flexray

faster, but more complex to set up – more things can fail and more pins are required

UART

PWM

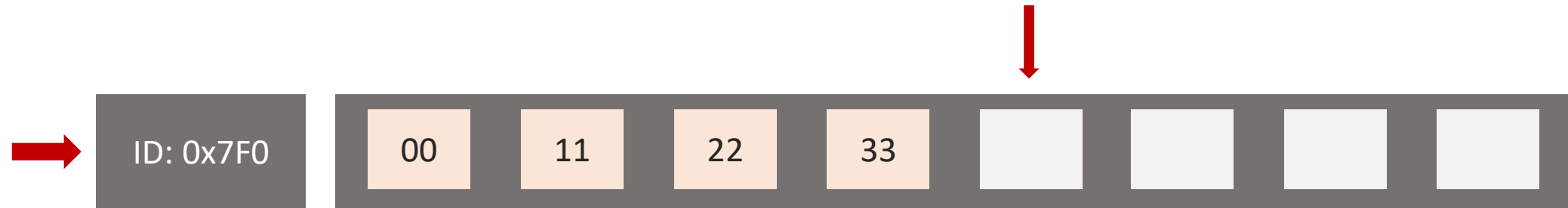


Any kind of interface can be (ab)used to serve as a data channel. Generally, the “best” interface is the fastest one that works – every target is different!

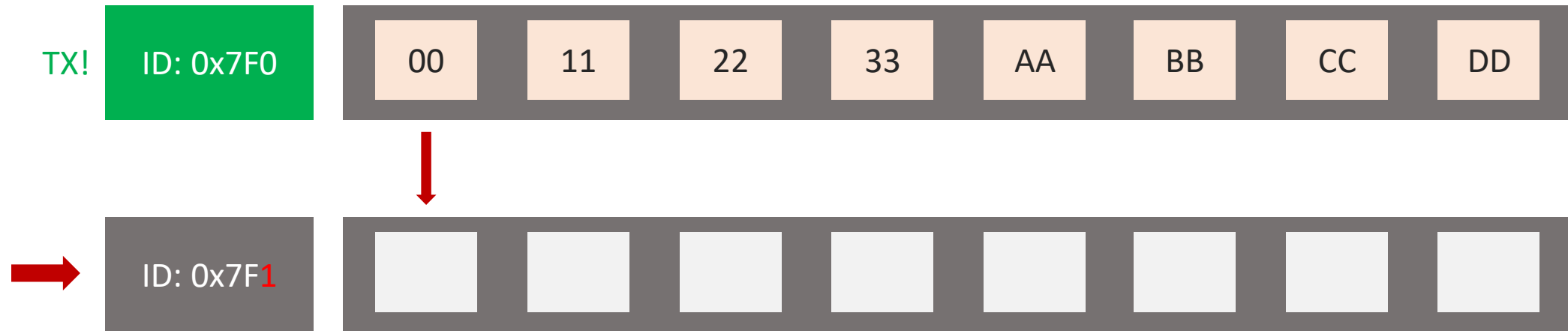
# Data Channels: Async Transfer



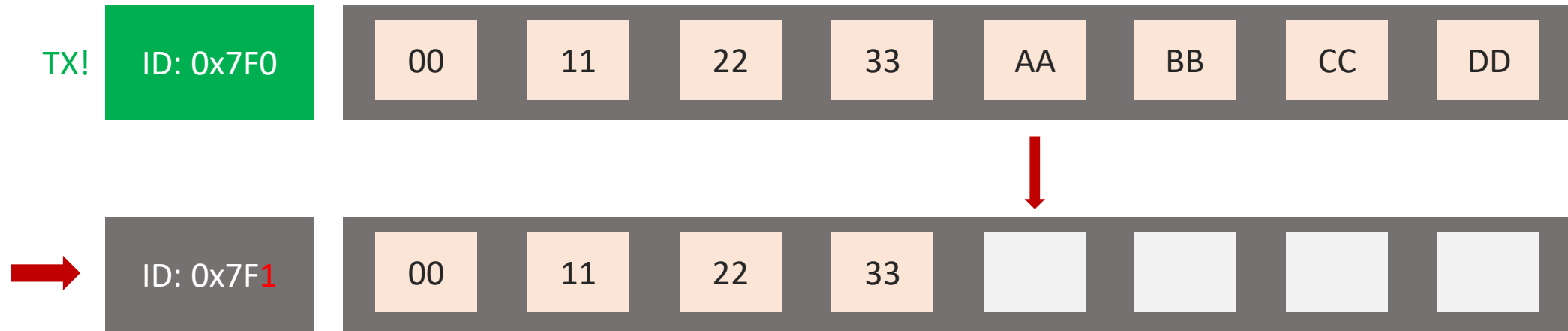
# Data Channels: Async Transfer



# Data Channels: Async Transfer



# Data Channels: Async Transfer

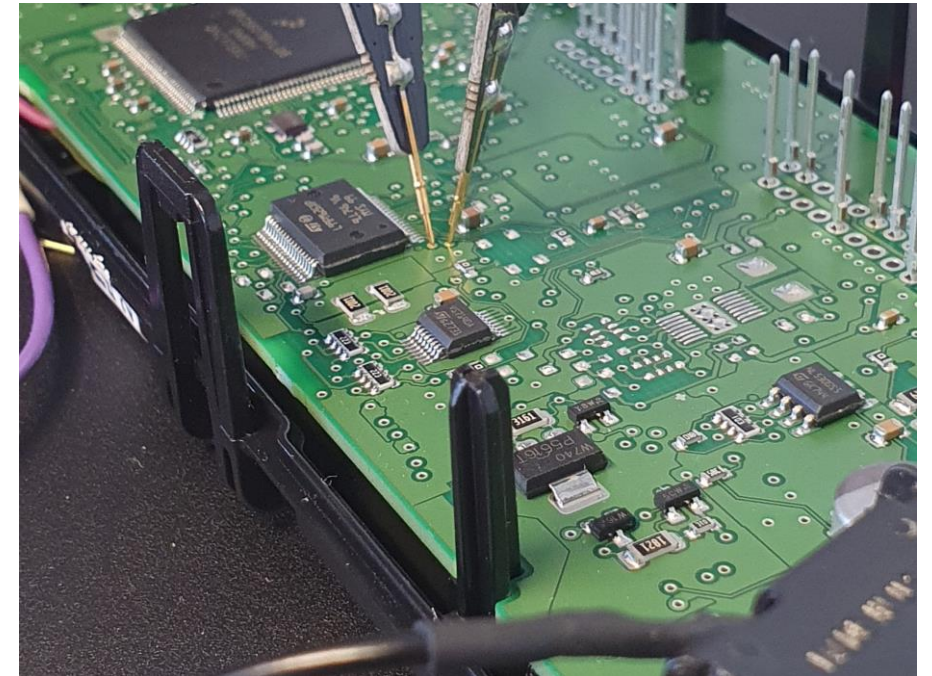


# Flopz Listener

- Handles *Links* such as UART or CAN
- Feeds data to target-defined *Protocols*
- Protocols generate *Events* that are handled by the Logger (non-interactive) or by the Listener (interactive)
- Finally, a Logger parses and stores Events in a specific format

# Case Study: MPC56XX aka Bolero

- Premium-Segment Body Application
- Has SBC
- Large share of generated code
- RTOS-enabled
- VLE-only\* processor core



# Demo Time

# Finding Bugs

Heap Corruption

Double Free

Stack Corruption

Static Function  
Pointer Corruption

## In General:

- Always define fuzz entry & exit, crash condition
- Snapshot fuzzing is broken by specific peripheral interaction
- Instrumenting all fault handlers is important
- As always, crashes don't necessarily imply bugs

# Finding Bugs

Heap Corruption

Double Free

Stack Corruption

Static Function  
Pointer Corruption

## Red Zones:

Instrument malloc/free in order to prepend & append canary buffers around allocated memory.

Validate memory and function arguments at every call.

# Finding Bugs

Heap Corruption

Double Free

Stack Corruption

Static Function  
Pointer Corruption

## Red Zones:

Instrument all function entries and exits.

Append & Prepend canary values to stack memory, validate at function exit.

# Finding Bugs

Heap Corruption

Double Free

Stack Corruption

Static Function  
Pointer Corruption

## Problem:

Safety-Critical Type II / Type III devices typically don't use dynamic stack/heap allocation at all or in a limited amount of cases!

# Finding Bugs

Heap Corruption

Double Free

Stack Corruption

Static Pointer  
Corruption

## Periodic Validation:

At target-specific intervals or points in time, iterate over all function pointer locations and validate them.

In the general case, false positives can not be ruled out using this method.

In practice, false positives are reduced by limiting the values that indicate corruption.

# Summary

**Flopz** is an assembler toolkit for embedded security applications written in python

Flopz is used to dynamically **patch and instrument black-box firmware images**

Limiting factors are the target's usable flash size and the speed of the data channel

Flopz is gradually being open sourced. All requests are welcome.



<https://github.com/Flopz-Project/flopz>



@s1ckcc