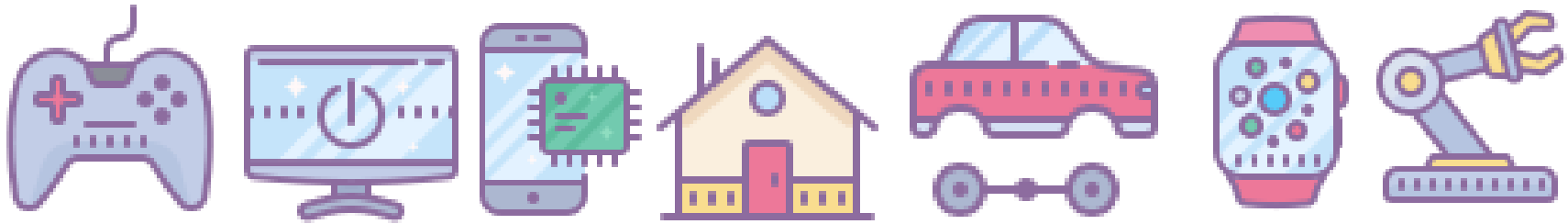
Inception: System-Wide Security Testing of Real-World Embedded Systems Software

Nassim Corteggiani (Maxim Integrated / EURECOM)

Giovanni Camurati (EURECOM)
Aurélien Francillon (EURECOM)

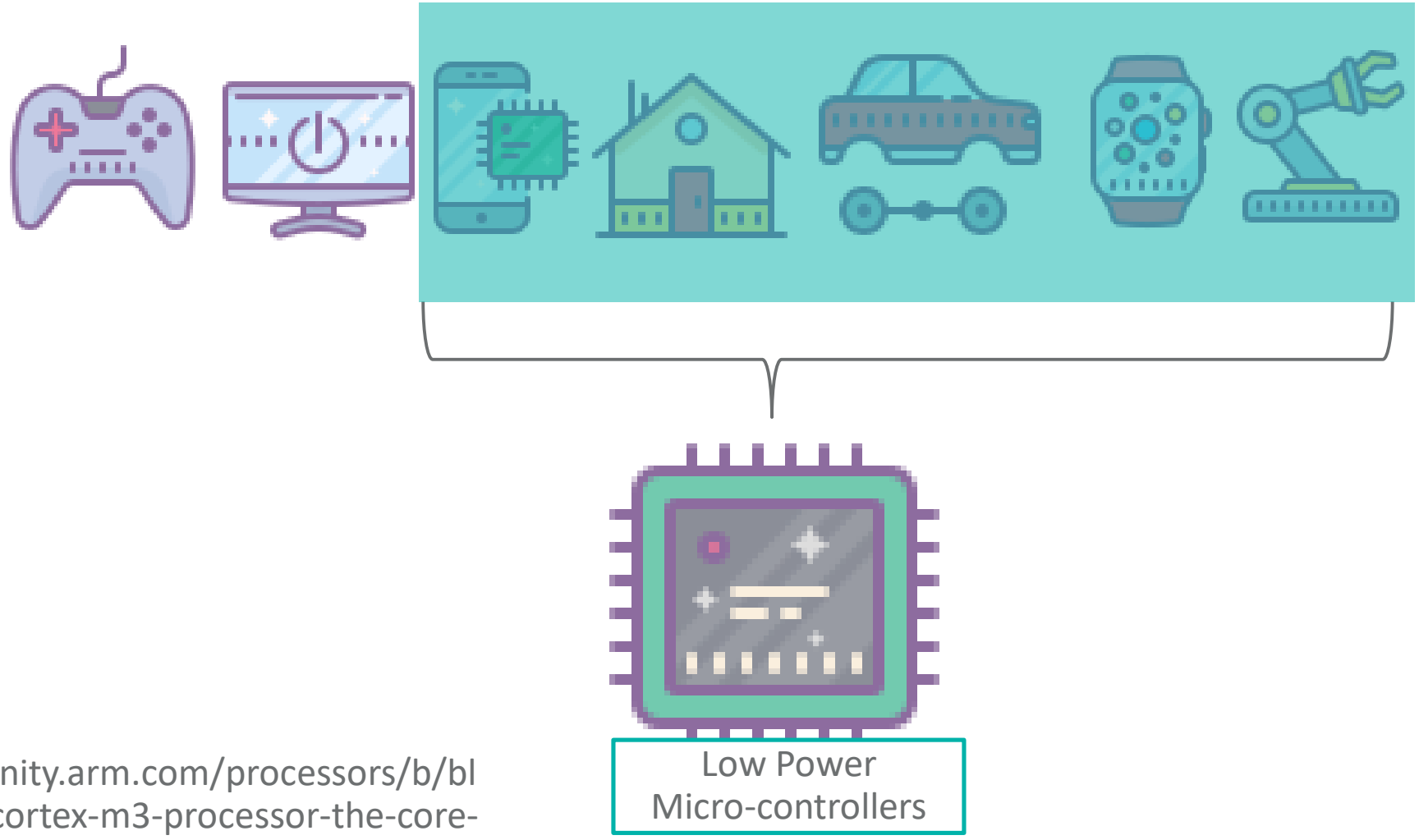
08/15/18

Embedded Systems Are Everywhere



[1] https://community.arm.com/processors/b/blog/posts/arm-cortex-m3-processor-the-core-of-the-iot

Embedded Systems Are Everywhere

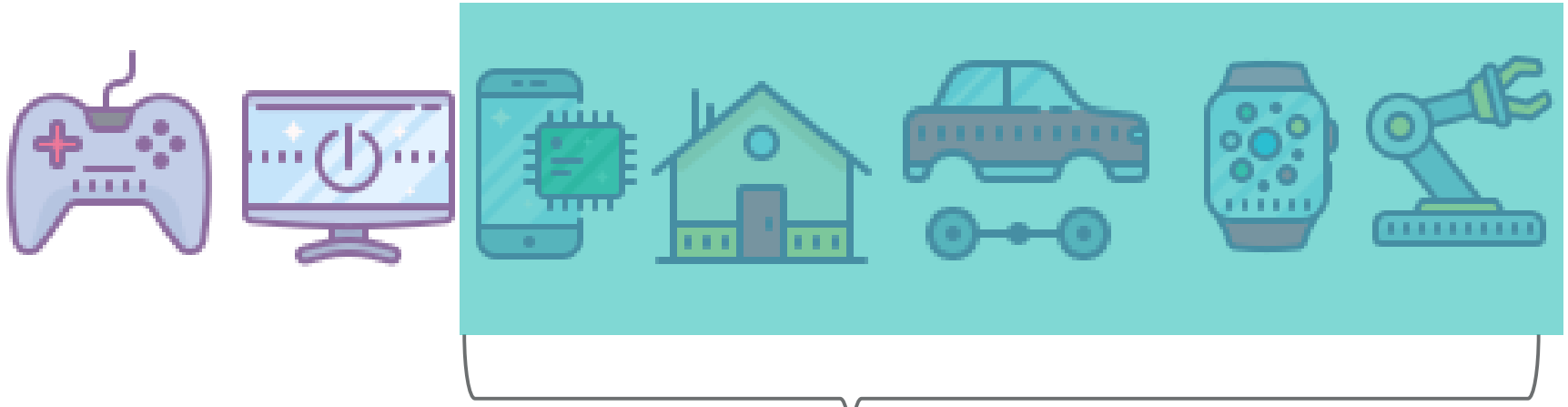


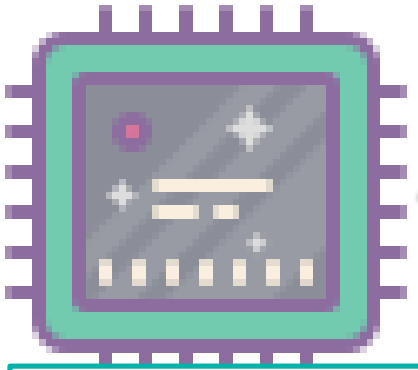
[1] https://community.arm.com/processors/b/blog/posts/arm-cortex-m3-processor-the-core-of-the-iot

EURECOM
Sophia Antipolis

maxim integrated™

Embedded Systems Are Everywhere





Low Power Micro-controllers

Over 32 billions of ARM Cortex M3 shipped in 2018 [1]

Cover a wide range of fields

[1] https://community.arm.com/processors/b/blog/posts/arm-cortex-m3-processor-the-core-of-the-iot

Why the Security of Such Systems Matters?

Why the Security of Such Systems Matters?

- Highly connected -> large scale attacks

Why the Security of Such Systems Matters?

- Highly connected -> large scale attacks

- Difficulty to patch the code
  - > Mask ROM → mask applied on the chip during the fabrication
  - > Off-line devices

Why the Security of Such Systems Matters?

- Highly connected -> large scale attacks

- Difficulty to patch the code
  - > Mask ROM → mask applied on the chip during the fabrication
  - > Off-line devices

- Store sensitive data
  - > Bitcoin wallet
  - > Payment terminal

Why the Security of Such Systems Matters?

- Highly connected -> large scale attacks

- Difficulty to patch the code
  - > Mask ROM → mask applied on the chip during the fabrication
  - > Off-line devices

- Store sensitive data
  - > Bitcoin wallet
  - > Payment terminal

- Drive sensitive hardware system
  - > Physical damage
  - > Production line outage
  - > Signaling systems (red light)

Exemple of Recent Security Issues

Recent attacks

Exemple of Recent Security Issues

Recent attacks

- Nintendo Switch Tegra X1 bootrom exploit 2018
  - > buffer overflow in the USB stack embedded in the mask ROM
  - > Cannot be patched
  - > Give access to the entire software stack

How Can We Test Such Firmware Programs?

- Symbolic Execution
  - > High path coverage
  - > Return test case for bugs

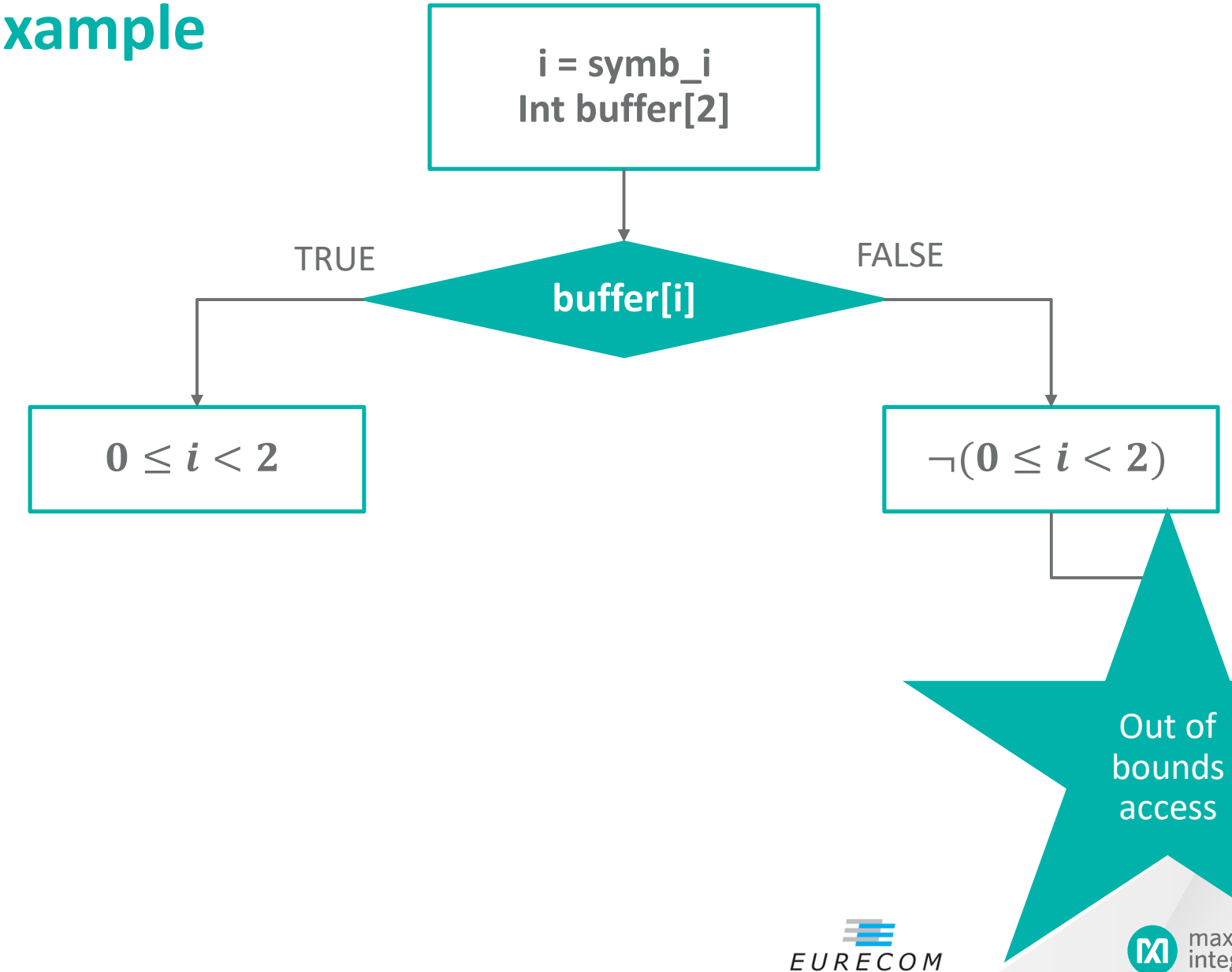
Symbolic Execution Example

i = symb_i
Int buffer[2]

```
i = <input>

int buffer[2] = {0, 1};
```

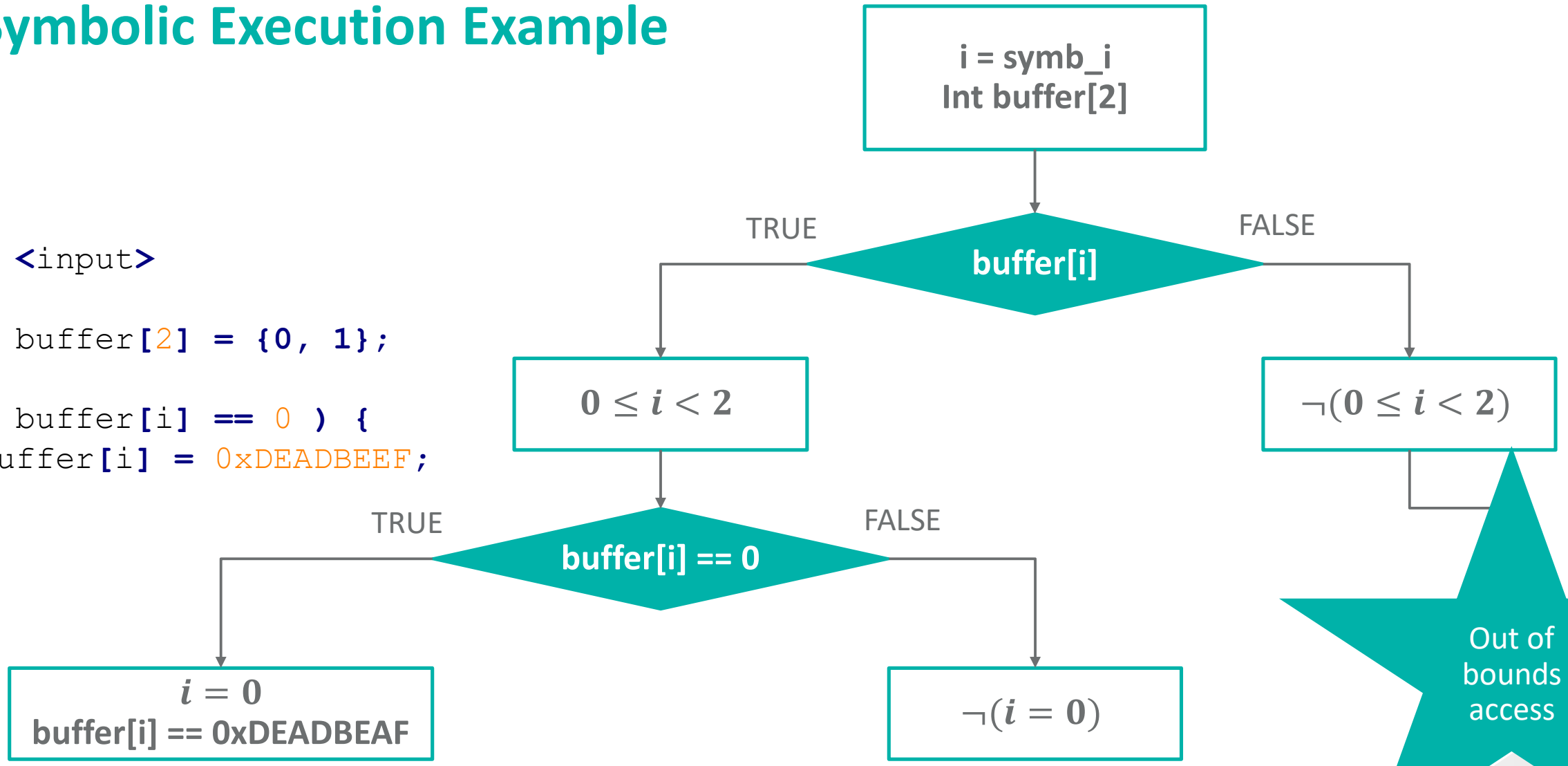
Symbolic Execution Example

```
i = <input>

int buffer[2] = {0, 1};

if( buffer[i] == 0 ) {
```



Symbolic Execution Example

```
i = <input>

int buffer[2] = {0, 1};

if( buffer[i] == 0 ) {
  buffer[i] = 0xDEADBEEF;
}
```



Building A Symbolic Executor For Firmware Programs

Klee as a basis

- Inception is based on Klee a symbolic virtual machine:

Building A Symbolic Executor For Firmware Programs

Klee as a basis

- Inception is based on Klee a symbolic virtual machine:
  - > Widely deployed, efficient and based on the LLVM framework.

Building A Symbolic Executor For Firmware Programs

Klee as a basis

- Inception is based on Klee a symbolic virtual machine:
  - > Widely deployed, efficient and based on the LLVM framework.
  - > Find memory safety violations

Building A Symbolic Executor For Firmware Programs

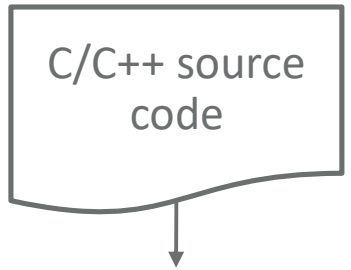
Klee as a basis

- Inception is based on Klee a symbolic virtual machine:
  - > Widely deployed, efficient and based on the LLVM framework.
  - > Find memory safety violations
  - > High code coverage

Building A Symbolic Executor For Firmware Programs

Klee as a basis

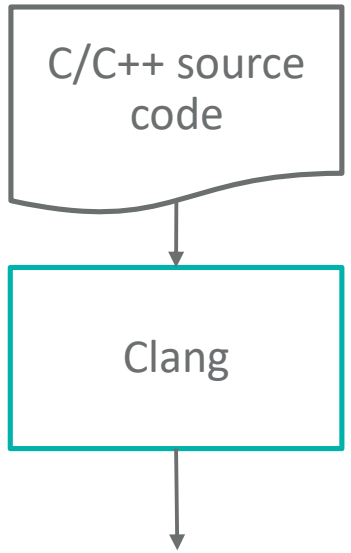
- Inception is based on Klee a symbolic virtual machine:
  - > Widely deployed, efficient and based on the LLVM framework.
  - > Find memory safety violations
  - > High code coverage



Building A Symbolic Executor For Firmware Programs

Klee as a basis

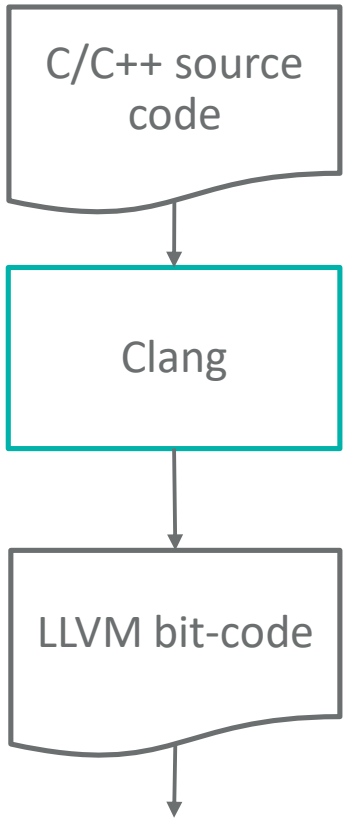
- Inception is based on Klee a symbolic virtual machine:
  - > Widely deployed, efficient and based on the LLVM framework.
  - > Find memory safety violations
  - > High code coverage



Building A Symbolic Executor For Firmware Programs

Klee as a basis

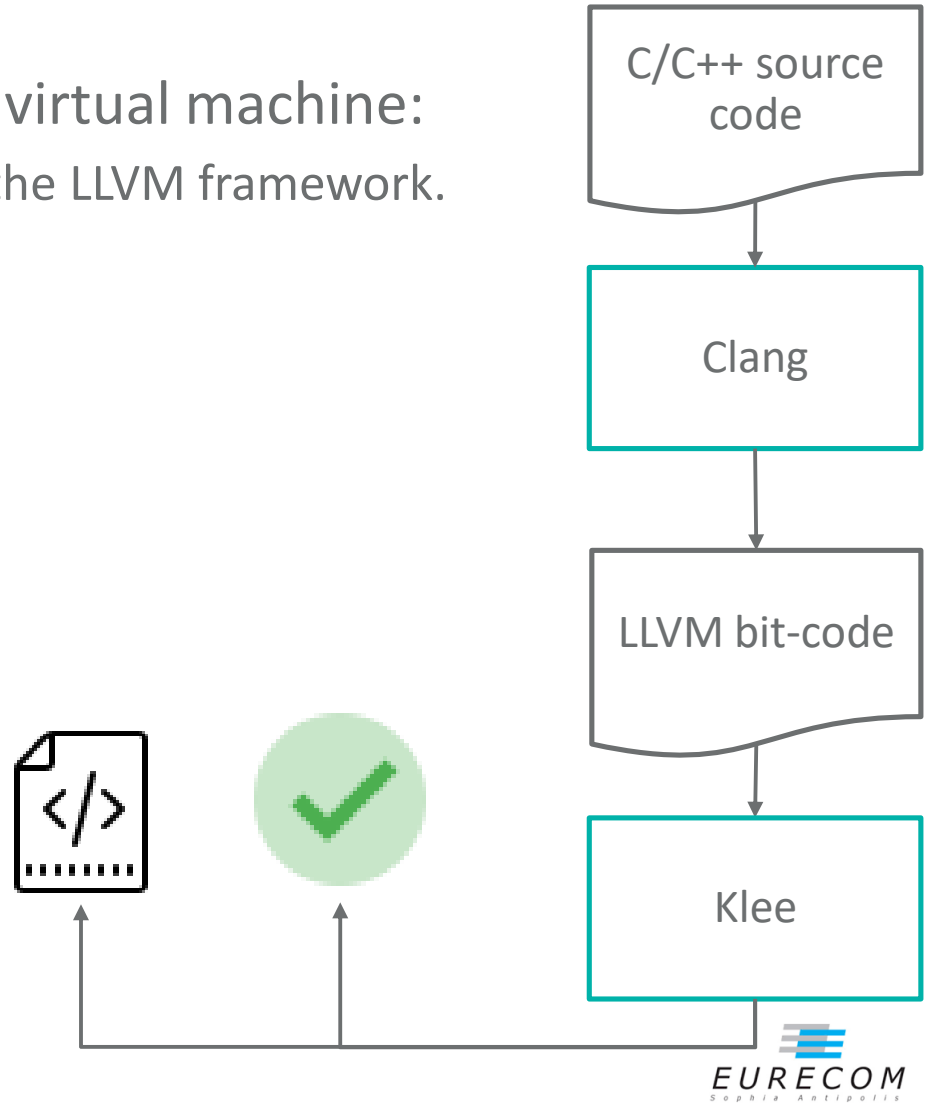
- Inception is based on Klee a symbolic virtual machine:
  - > Widely deployed, efficient and based on the LLVM framework.
  - > Find memory safety violations
  - > High code coverage



Building A Symbolic Executor For Firmware Programs

Klee as a basis

- Inception is based on Klee a symbolic virtual machine:
  - > Widely deployed, efficient and based on the LLVM framework.
  - > Find memory safety violations
  - > High code coverage



Why testing source code instead of binary code ?

Source VS Binary

Why testing source code instead of binary code ?

Source

```
char b1[2];
char b2[2];
char getElement(int index)
{
 return b1[index];
}
```

VS

Binary

```
b1:   .space  2
b2:   .space  2
getElement(int):
   ldr r2, .L3
   add r3, r2, r0
   ldrb r0, [r3]
   bx lr
.L3: .word b1
```

Why testing source code instead of binary code ?

Source

```
char b1[2];
char b2[2];
char getElement(int index)
{
 return b1[index];
}
```

VS

Binary

```
b1:   .space  2
b2:   .space  2
getElement(int):
   ldr r2, .L3
   add r3, r2, r0
   ldrb r0, [r3]
   bx lr
.L3: .word b1
```


EURECOM Sophia Antipolis

maxim integrated™

Why testing source code instead of binary code ?

Source (Klee/Clang…) VS Binary (SE2, angr, BAP)

```
char b1[2];
char b2[2];
char getElement(int index)
{
 return b1[index];
}
```

```
define i8 @getElement(i32 %index){
entry:
%0 = load i32* %index.addr
%1 = getelementptr inbounds
[2 x i8]* @b1, i32 0, i32 %0
%2 = load i8* %1
ret i8 %2
}
```

```
b1:   .space  2
b2:   .space  2
getElement(int):
   ldr r2, .L3
   add r3, r2, r0
   ldrb r0, [r3]
   bx lr
.L3: .word b1
```

```
define i8 @getElement(i32 index) {
entry:
 store i32 %index, i32* @R0
 store i32 268436792, i32* @R2
 %R2_1 = load i32* @R2
 %R0_1 = load i32* @R0
 %R2_2 = add i32 %R2_1, %R0_1
 %R3_0 = inttoptr i32 %R2_2 to i32*
 %R3_1 = bitcast i32* %R3_0 to i8*
 %R3_2 = load i8* %R3_1
 %R3_3 = zext i8 %R3_2 to i32
```

Why testing source code instead of binary code ?

Source (Klee/Clang…) VS Binary (SE2, angr, BAP)

```
char b1[2];
char b2[2];
char getElement(int index)
{
 return b1[index];
}
```

```
define i8 @getElement(i32 %index){
entry:
%0 = load i32* %index.addr
%1 = getelementptr inbounds
[2 x i8]* @b1, i32 0, i32 %0
%2 = load i8* %1
ret i8 %2
}
```

```
b1:   .space  2
b2:   .space  2
getElement(int):
   ldr r2, .L3
   add r3, r2, r0
   ldrb r0, [r3]
   bx lr
.L3: .word b1
```

```
define i8 @getElement(i32 index) {
entry:
 store i32 %index, i32* @R0
 store i32 268436792, i32* @R2
 %R2_1 = load i32* @R2
 %R0_1 = load i32* @R0
 %R2_2 = add i32 %R2_1, %R0_1
 %R3_0 = inttoptr i32 %R2_2 to i32*
 %R3_1 = bitcast i32* %R3_0 to i8*
 %R3_2 = load i8* %R3_1
 %R3_3 = zext i8 %R3_2 to i32
```

Source vs Binary

- When source available testing binary is possible however:
  - > Types are lost
  - > Corruption will be detected later if at all
  - > Worse on embedded systems
    - See: Muench et. al. What you corrupt is not what you crash, NDSS 2018

- Goal of Inception: improve testing for firmware during development
  - > Limit requirements on code

Major Challenges For Symbolic Execution of Firmware Programs

Is C/C++ Support Enough To Test Real World Firmware ?

- Number of functions including assembly instructions in real world embedded software

EURECOM Sophia Antipolis

maxim integrated™

Major Challenges For Symbolic Execution of Firmware Programs

Is C/C++ Support Enough To Test Real World Firmware ?

- Assembly code :
  - > Multithreading
  - > Optimizations
  - > Side channel counter-measures
  - > Hardware features e.g. ultra low power mode

- Number of functions including assembly instructions in real world embedded software

EURECOM Sophia Antipolis

maxim integrated™

Major Challenges For Symbolic Execution of Firmware Programs

Is C/C++ Support Enough To Test Real World Firmware ?

Challenge 1 :

Firmware source code contains a mix of C/C++, assembly and binary

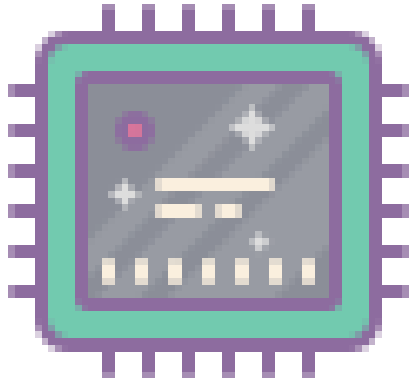
Major Challenges For Symbolic Execution of Firmware Programs

Hardware environment

Major Challenges For Symbolic Execution of Firmware Programs

Hardware environment

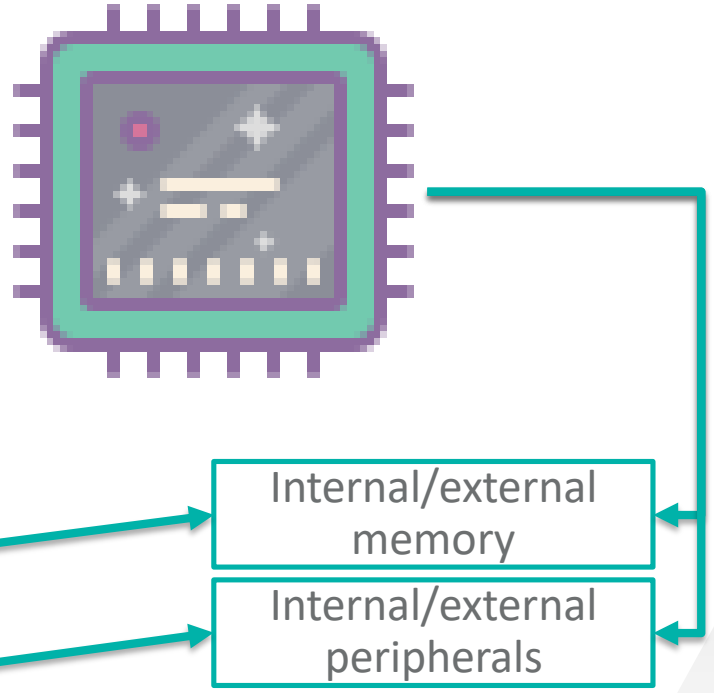
- Hardware interactions
  - > Memory Mapped I/O



Major Challenges For Symbolic Execution of Firmware Programs

Hardware environment

- Hardware interactions
  - > Memory Mapped I/O
    - Memory
    - Peripherals




```
#define UART_STATUS 0x40000000
#define UART_DATA   0x40000004

char* RX_BUFFER = 0x20000000;
while(!*UART_STATUS) {
  char* data = (char*)UART_DATA;
  strncpy(RX_BUFFER++, data, 4);
}
```

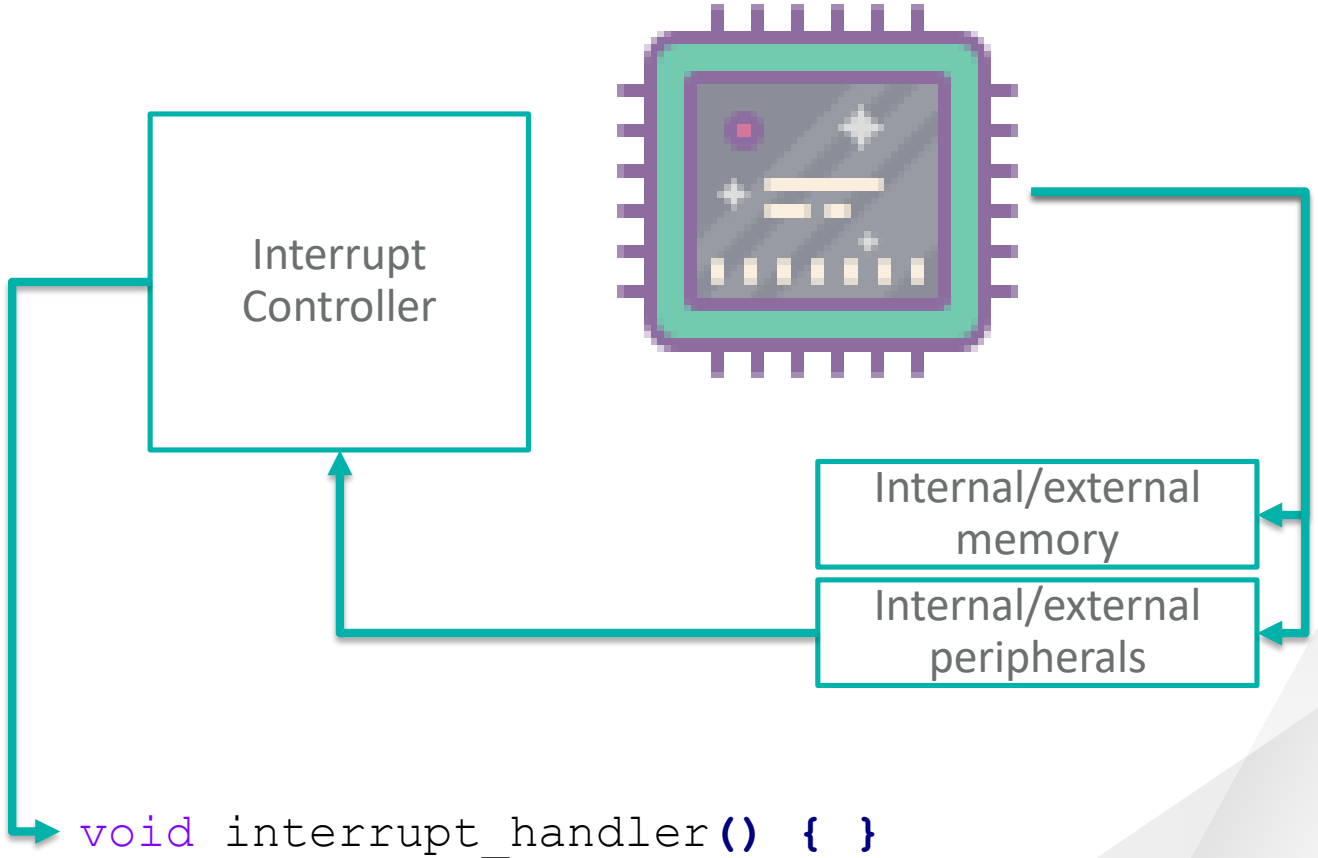

EURECOM Sophia Antipolis

maxim integrated™

Major Challenges For Symbolic Execution of Firmware Programs

Hardware environment

- Hardware interactions
  - > Memory Mapped I/O
    - Memory
    - Peripherals
  - > Interrupt driven programs




```
#define UART_STATUS 0x40000000
#define UART_DATA   0x40000004

char* RX_BUFFER = 0x20000000;
while(!*UART_STATUS) {
  char* data = (char*)UART_DATA;
  strncpy(RX_BUFFER++, data, 4);
}
```

Major Challenges For Symbolic Execution of Firmware Programs

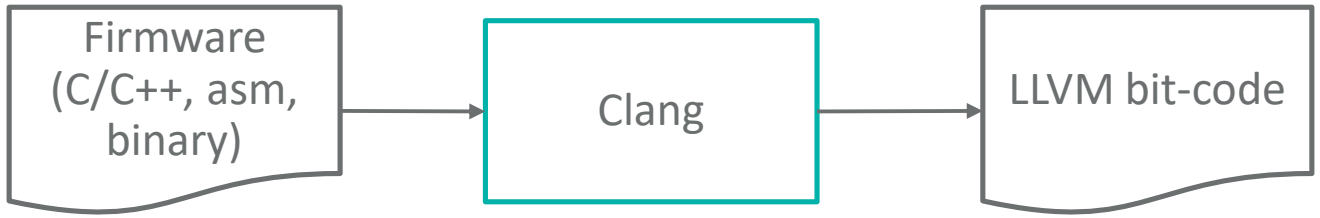
Hardware environment

Challenge 2 :

Firmware programs highly interact with their hardware environment

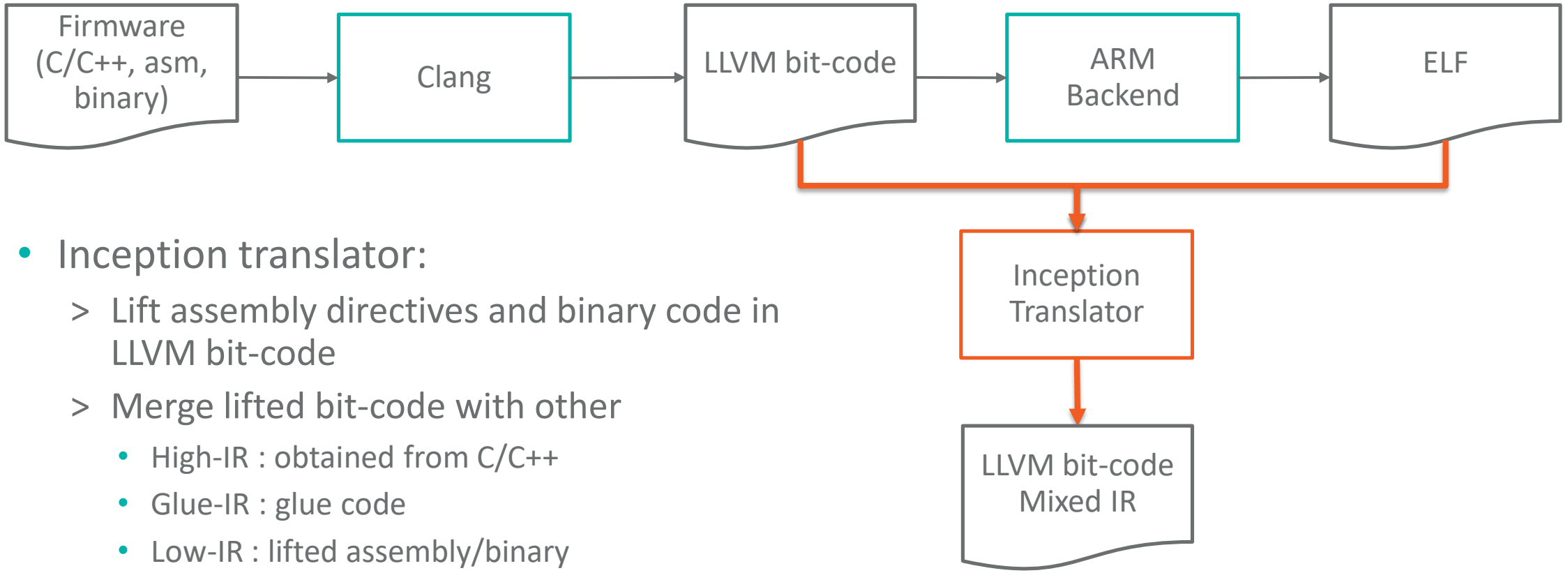
Building A Symbolic Executor For Firmware Programs

Building A Symbolic Executor For Firmware Programs



Building A Symbolic Executor For Firmware Programs

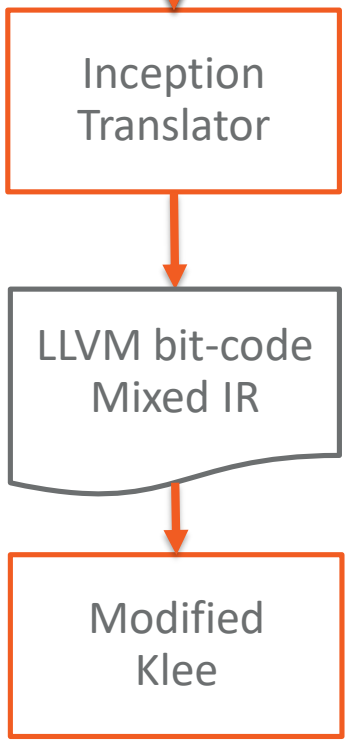
Building A Symbolic Executor For Firmware Programs



- Inception translator:
  - > Lift assembly directives and binary code in LLVM bit-code
  - > Merge lifted bit-code with other
    - High-IR : obtained from C/C++
    - Glue-IR : glue code
    - Low-IR : lifted assembly/binary

Building A Symbolic Executor For Firmware Programs

- Inception translator:
  - > Lift assembly directives and binary code in LLVM bit-code
  - > Merge lifted bit-code with other
    - High-IR : obtained from C/C++
    - Glue-IR : glue code
    - Low-IR : lifted assembly/binary
  - > Support Cortex M3 ISA



Challenge 1 : Supporting C/C++/Asm/Binary code

Inception-translator

Inception Translator : Merging High-IR and Low-IR

```
int a = 4;
boo(a);
```

```
<boo>:
1000045C: 80 B4 push {r7}
1000045E: 83 B0 sub sp, #0xc
```

Inception Translator : Merging High-IR and Low-IR

```
int a = 4;
boo(a);
```

```
<boo>:
1000045C: 80 B4 push {r7}
1000045E: 83 B0 sub sp, #0xc
```

```
%a = alloca i32
store i32 4, i32* %a
%0 = load i32* %a
%call = call i32
@boo(i32 %0)
ret void }
```

High IR

Inception Translator : Merging High-IR and Low-IR

```
int a = 4;
boo(a);
```

```
<boo>:
1000045C: 80 B4 push {r7}
1000045E: 83 B0 sub sp, #0xc
```

```
%a = alloca i32
store i32 4, i32* %a
%0 = load i32* %a
%call = call i32
@boo(i32 %0)
ret void }
```

High IR

```
"boo+0": ; preds = %entry
%R7_1 = load i32* @R7
%SP1 = load i32* @SP
%SP2 = sub i32 %SP1, 4
%SP3 = inttoptr i32 %SP2 to i32*
store i32 %R7_1, i32* %SP3
store i32 %SP2, i32* @SP
%SP4 = load i32* @SP
%SP5 = add i32 %SP4, -13
%SP6 = add i32 %SP5, 1
```

Low IR

Inception Translator : Merging High-IR and Low-IR

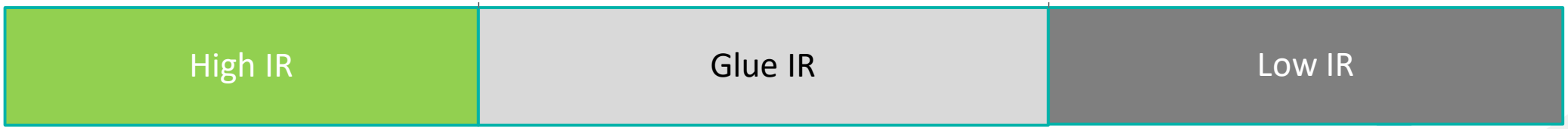
```
int a = 4;
boo(a);
```

```
<boo>:
1000045C: 80 B4 push {r7}
1000045E: 83 B0 sub sp, #0xc
```

```
%a = alloca i32
store i32 4, i32* %a
%0 = load i32* %a
%call = call i32
@boo(i32 %0)
ret void }
```

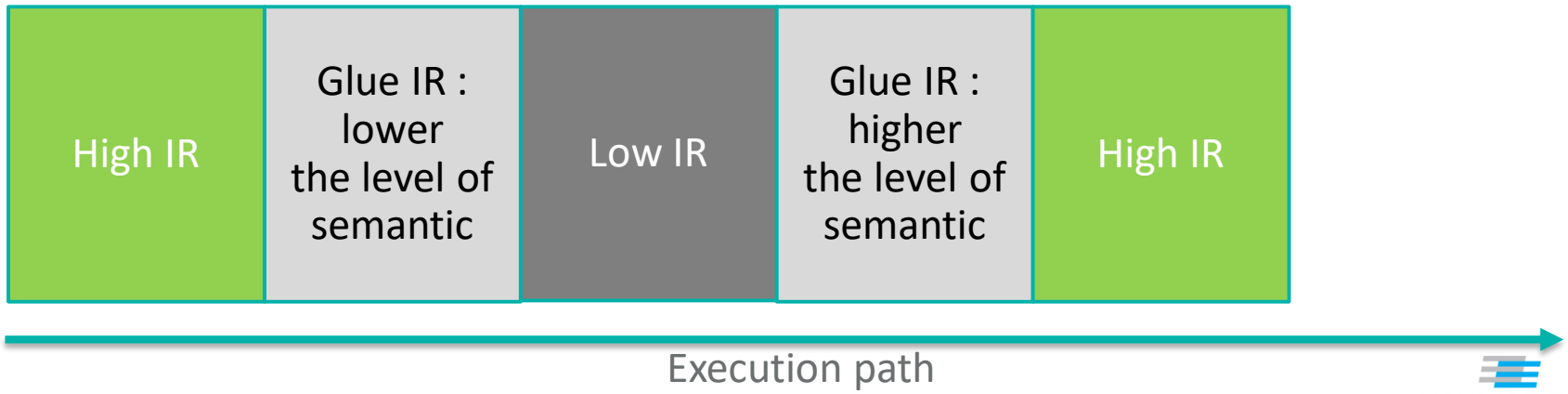
```
define i32 @boo(i32 %a){
entry:
store i32 %a, i32* @R0
br label %"boo+0"
```

```
"boo+0": ; preds = %entry
%R7_1 = load i32* @R7
%SP1 = load i32* @SP
%SP2 = sub i32 %SP1, 4
%SP3 = inttoptr i32 %SP2 to i32*
store i32 %R7_1, i32* %SP3
store i32 %SP2, i32* @SP
%SP4 = load i32* @SP
%SP5 = add i32 %SP4, -13
%SP6 = add i32 %SP5, 1
```



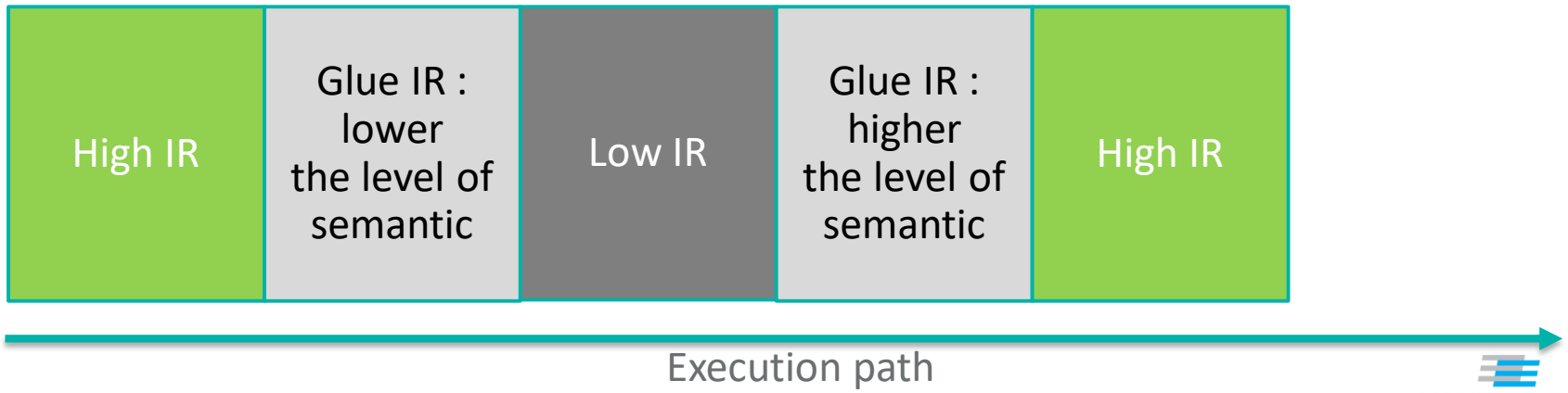
Unified Memory Layout

Unified Memory Layout



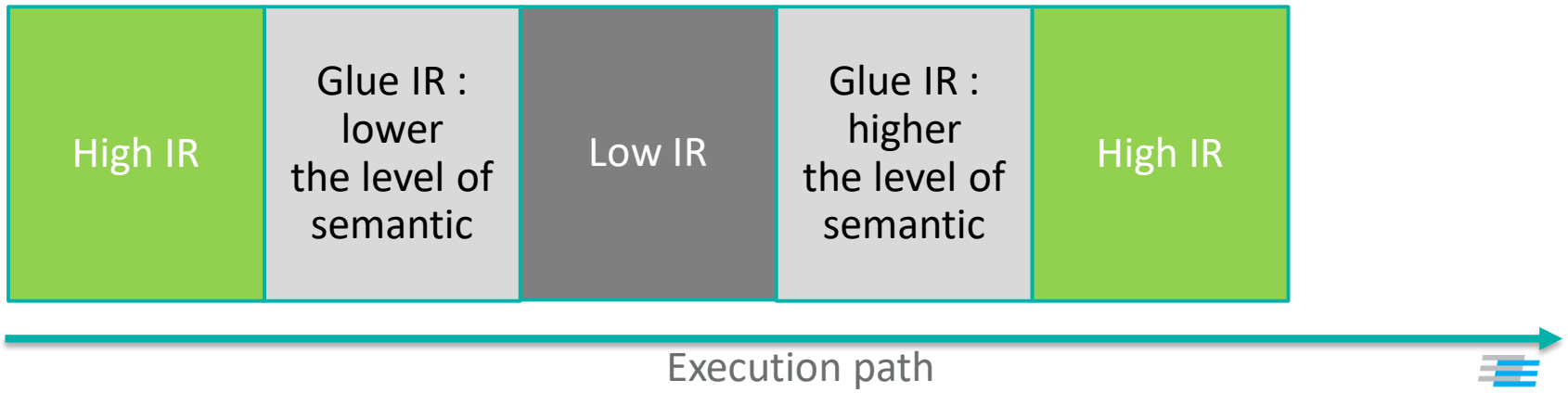
Unified Memory Layout

- Allocate Low IR memory : stack, virtual CPU registers, heap



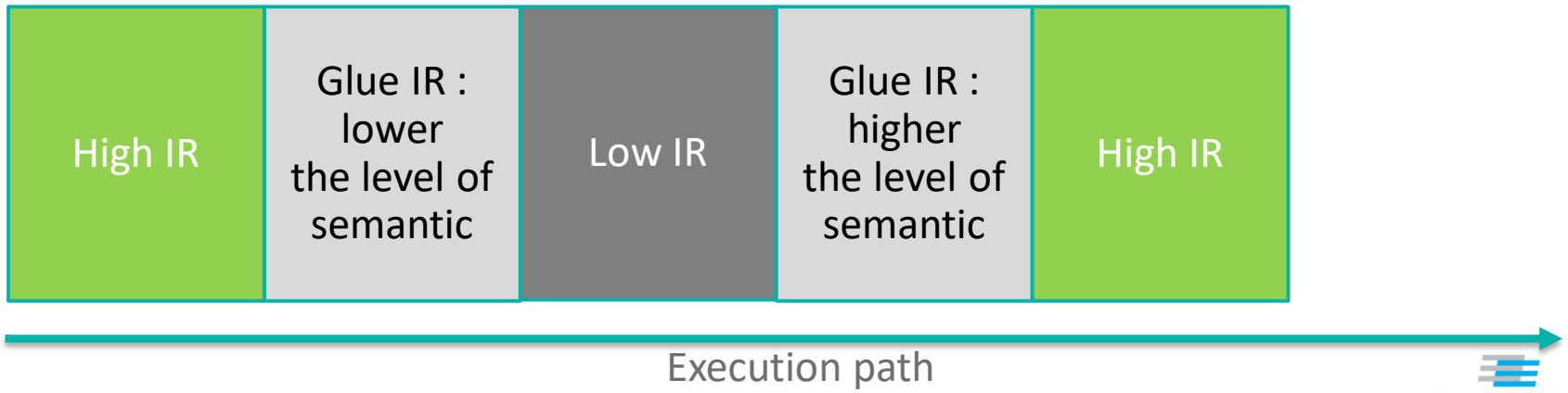
Unified Memory Layout

- Allocate Low IR memory : stack, virtual CPU registers, heap
- Fill gaps in global data sections
  - When no C/C++ symbols point to this area



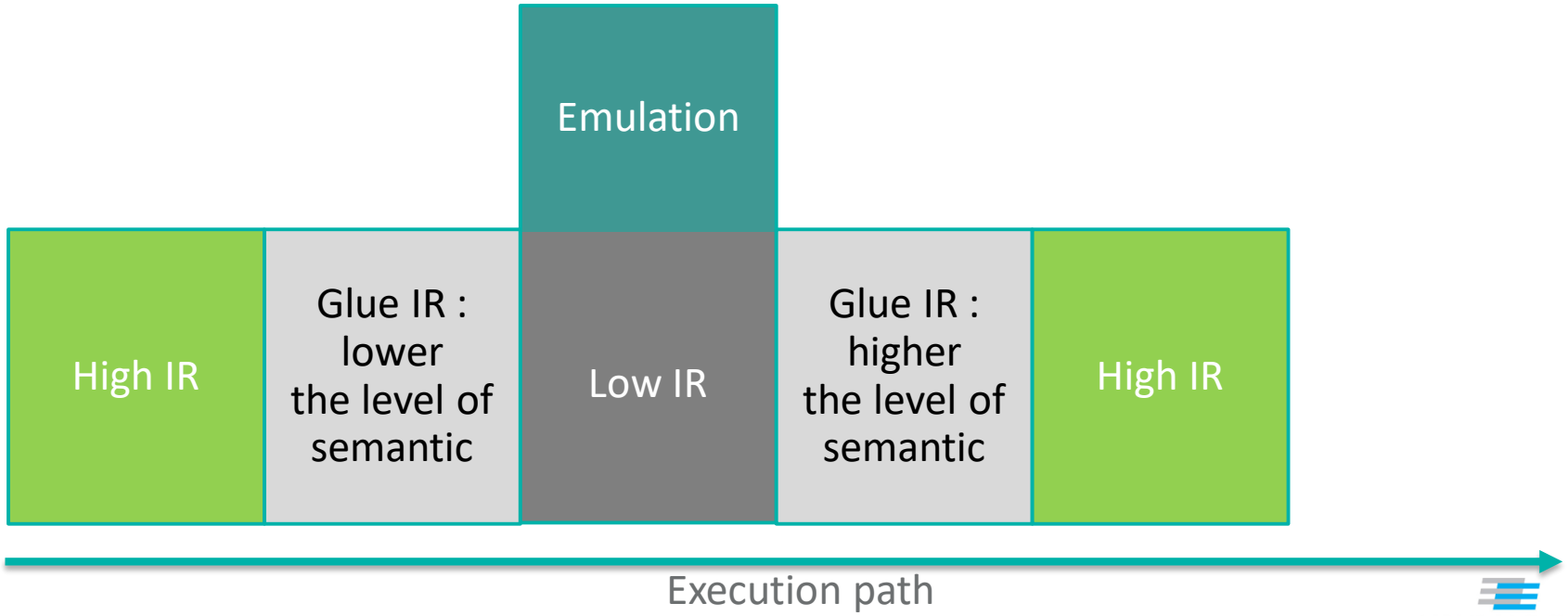
Unified Memory Layout

- Allocate Low IR memory : stack, virtual CPU registers, heap
- Fill gaps in global data sections
  - When no C/C++ symbols point to this area
- Allocate High-IR objects at location defined in the ELF symbols table



Low IR Hardware Mechanisms Emulation

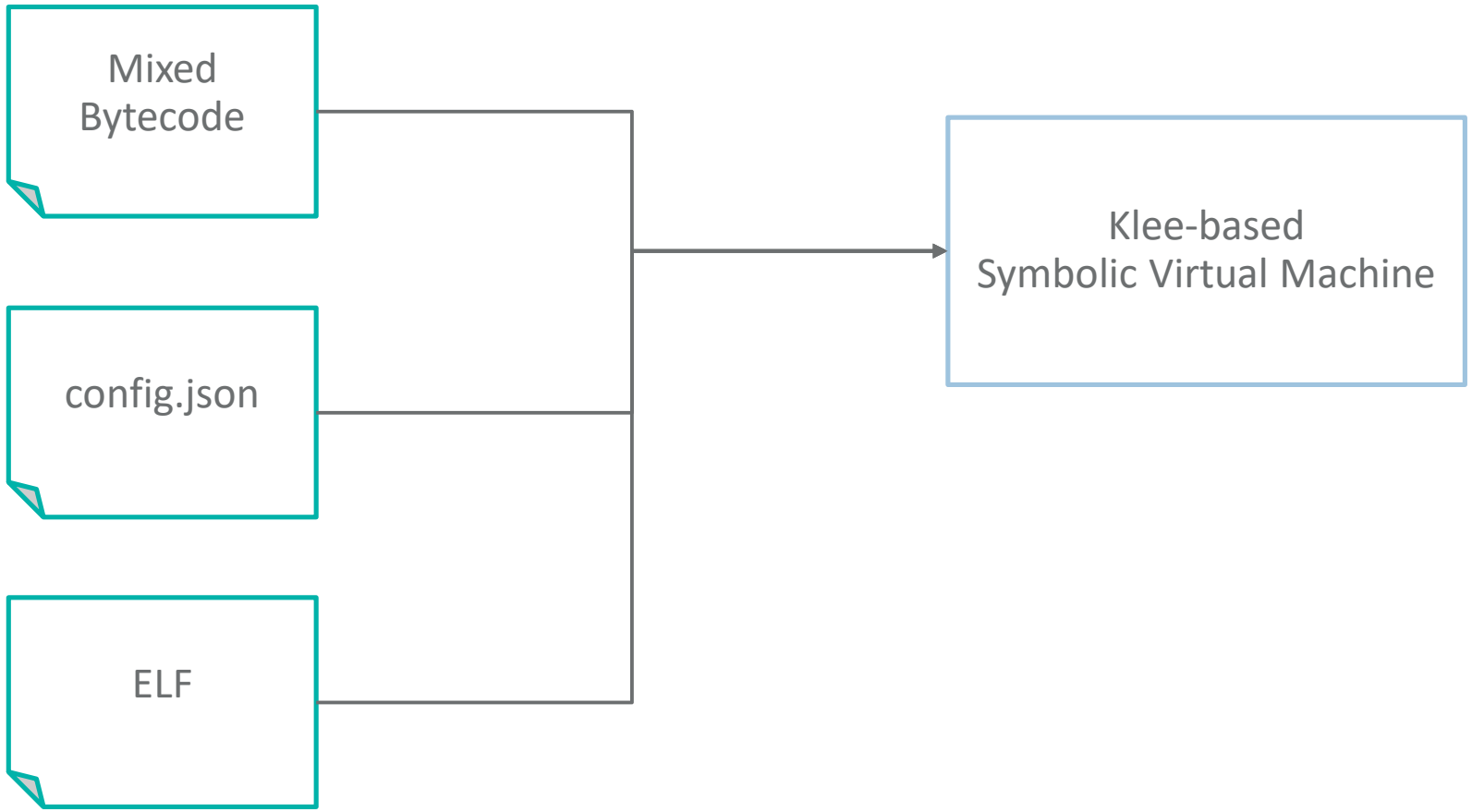
- Challenge we solved:
  - > Indirect calls (Indirect Call Promotion)
  - > Seamless hardware mechanisms (Context switching)
  - > Supervisor call
  - > Update specific registers values (LR, MSP, PSP, BASEPRI, ITSTATE, …)



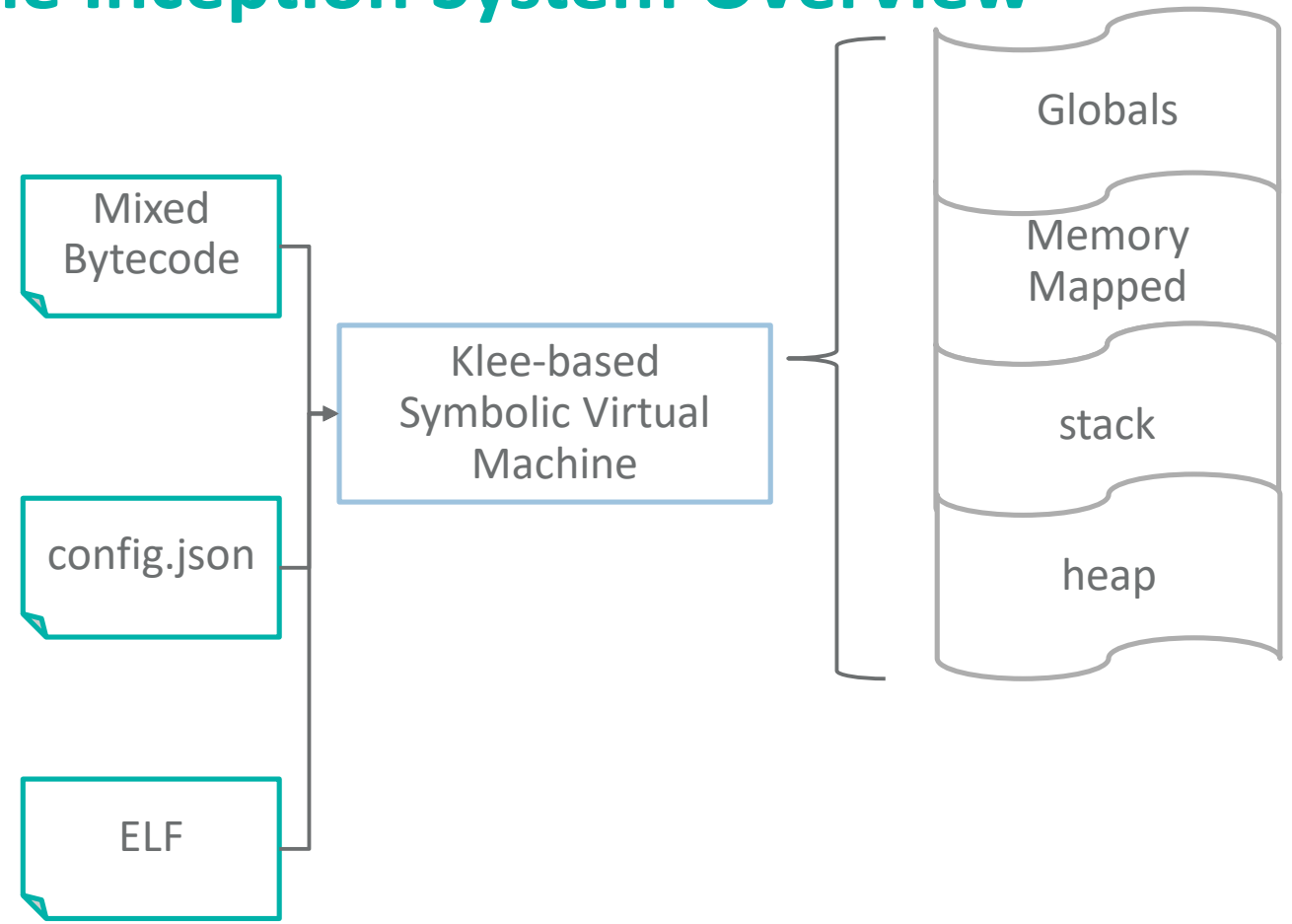
Challenge 2 : Hardware interactions

Inception-analyzer

The Inception System Overview



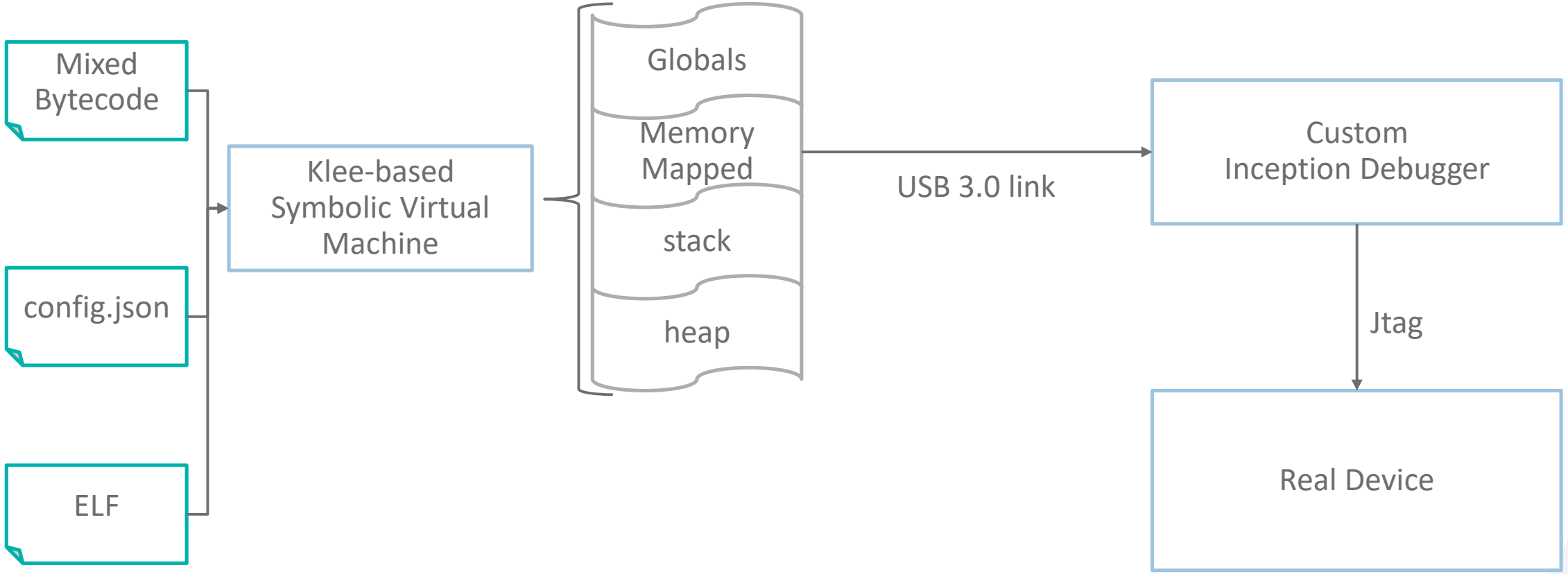
The Inception System Overview



Data are allocated according to the information present in the symbol table

User configuration (config.json):
- Local memory
- Redirected memory
- Symbolic memory

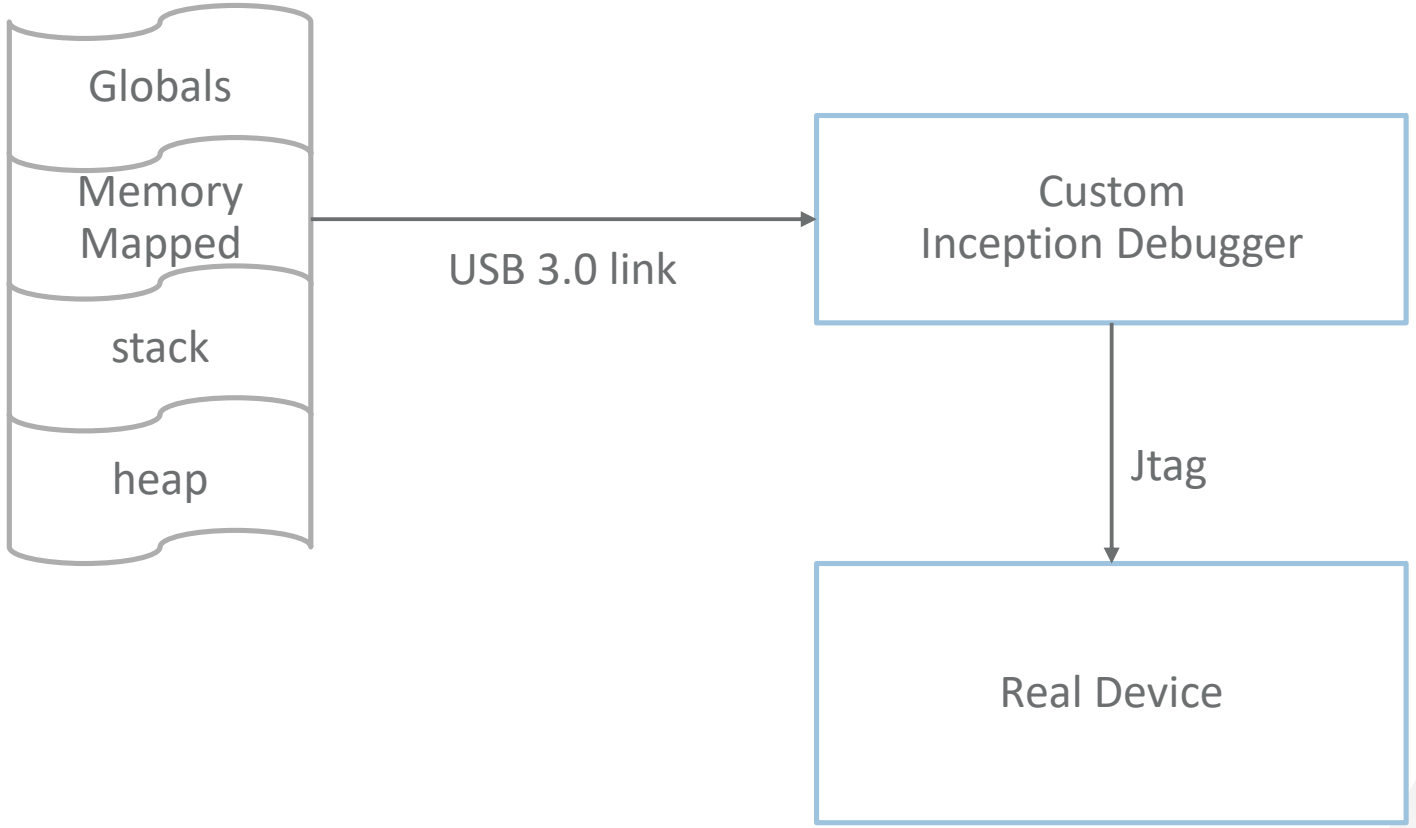
The Inception System Overview: Inception debugger



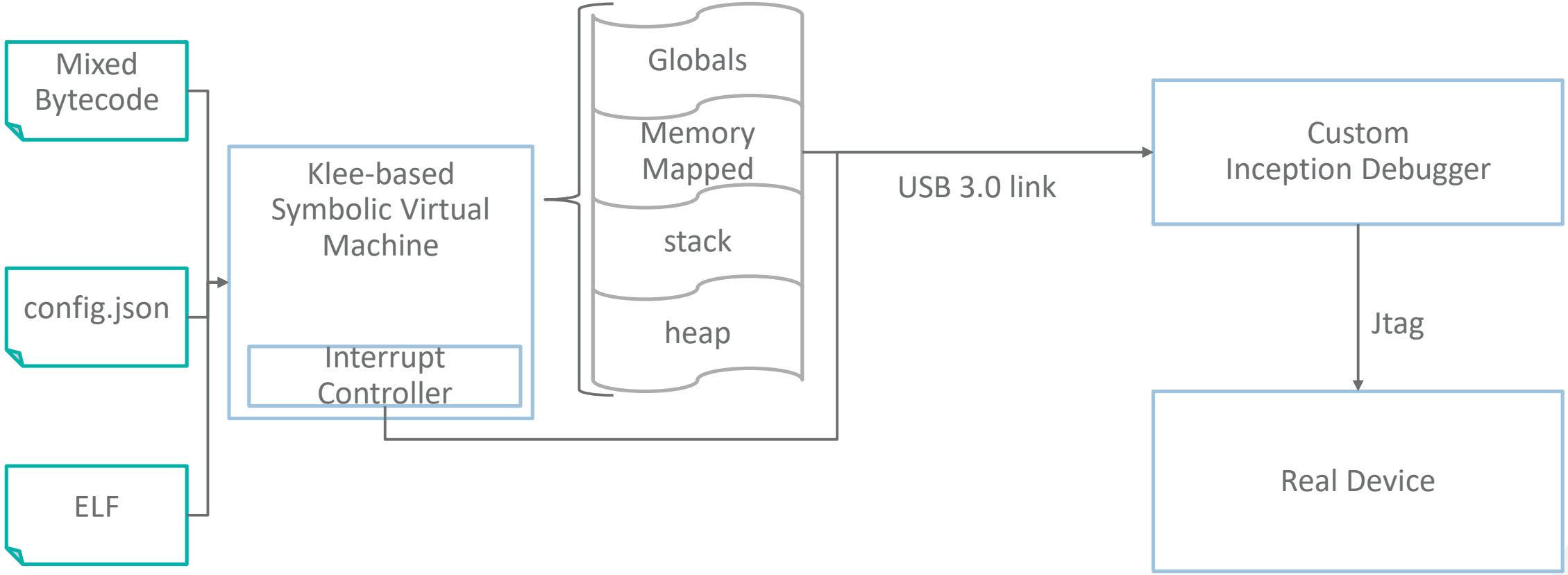
The Inception System Overview: Inception debugger

- Inspired by Surrogates and Avatar

Zaddach et. al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares, NDSS 2014
Koscher et. al. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems, WOOT 2015



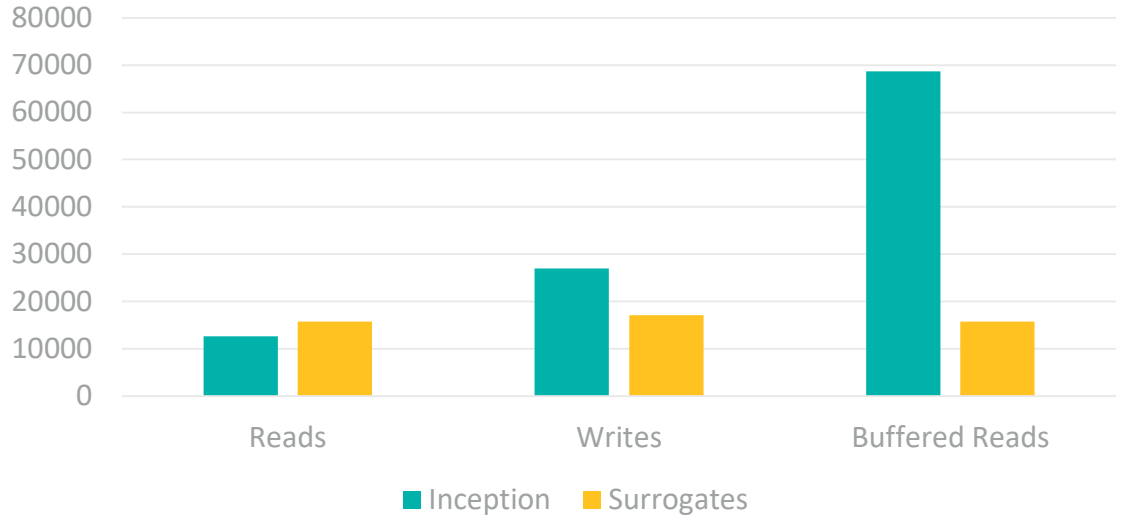
The Inception System Overview: Inception debugger



Evaluation

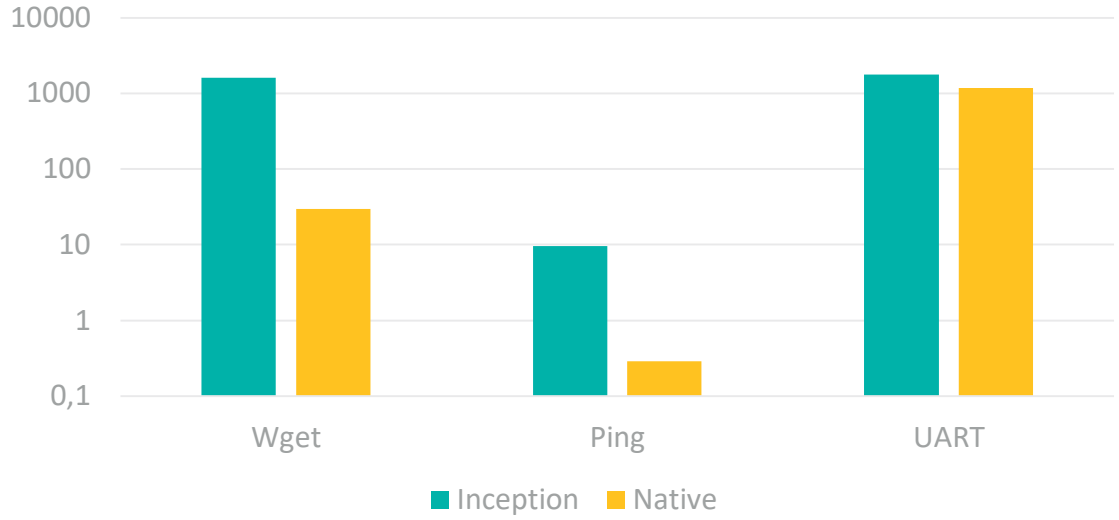
Performance

Average IO per second



Average time to complete 1×10^6 read or write requests for SURROGATES and Inception.

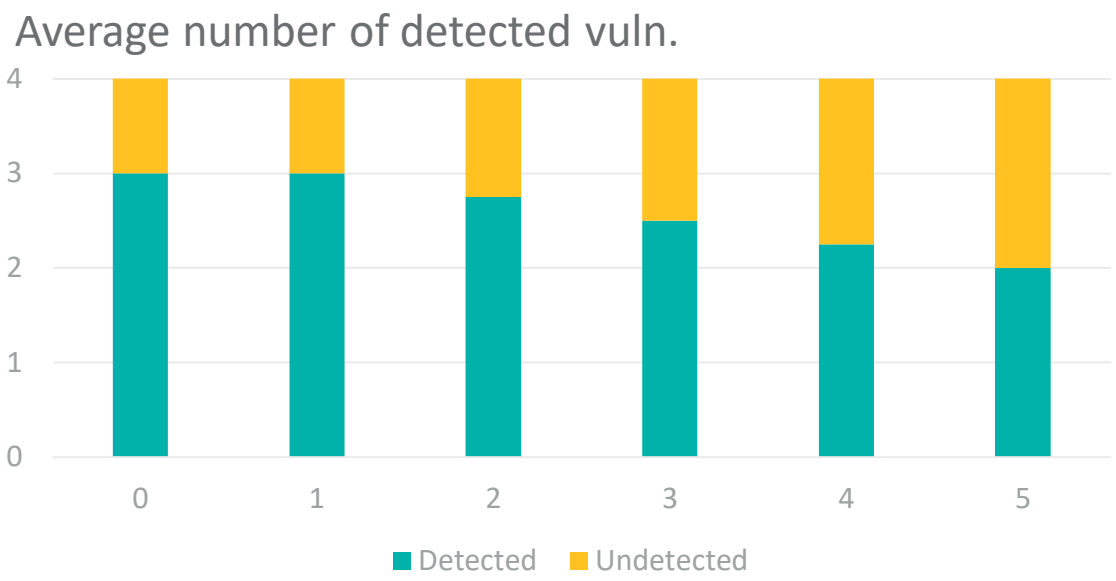
Average runtime [ms]



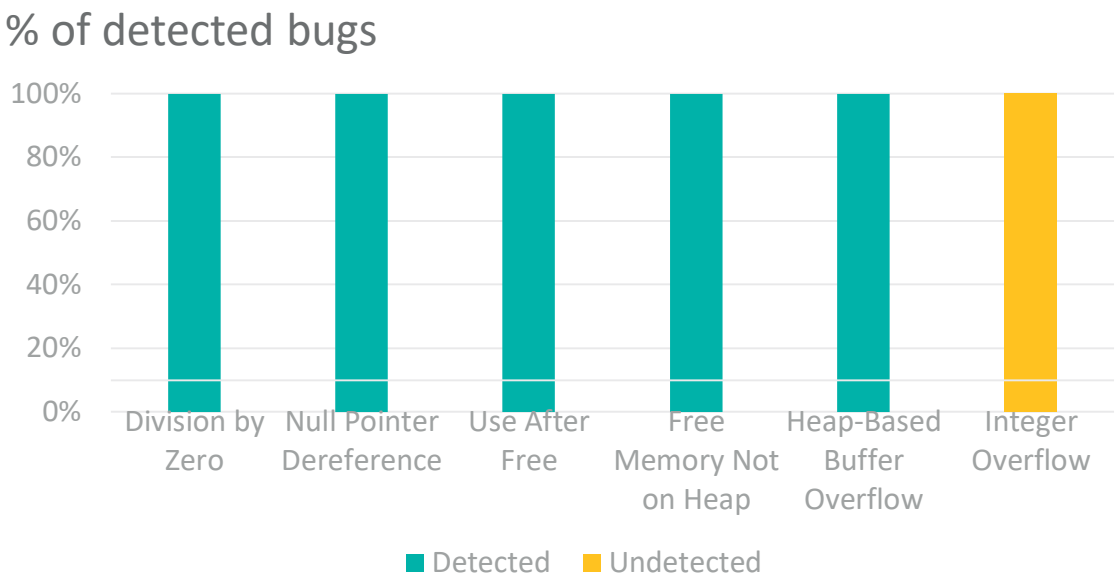
Performance comparison between native execution and Inception.

* Current bottleneck is bit-code execution

Bug Detection



Evolution of corruption detection vs. number of assembly functions in the EXPACT XML parser (4 vulnerabilities [1], symbolic inputs, and a timeout of 90s).



Corruption detection of real-world security flaws based on FreeRTOS and the Juliet 1.3 test suites.

[1] MUENCH et. al. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In NDSS 2018.

Verification

- Intensive verification of the lifter and the modified Klee
  - > 53K tests comparison between Inception and native
  - > 1562 tests based on NIST Juliet 1.3 tests suite
  - > 40 tests based on the Klockwork tests suite
  - > Several demos for the STM32 L152RE and the LPC1850 DB1 boards
  - > 1 Mbed TLS test suite
  - > Several embedded operating systems (FreeRTOS, mini-arm-os)

Conlusion

- Extends analysis to mixed languages: assembly, C/C++, binary
- Fit well in chip life-cycle :
  - > test without hardware
  - > FPGA-based design
  - > silicium
- Already used on proprietary real world Mask ROM code at Maxim
  - > Bugs found before mask manufacturing
- Inception is open-sourced:
  - > Getting started at https://inception-framework.github.io/inception/
  - > Github and docker

Questions?

Free icons license : https://icons8.com