Evaluation of Rust for Embedded Systems in terms of Security, Performance, and Usability

Ankita Kumari
ge93vep@mytum.de

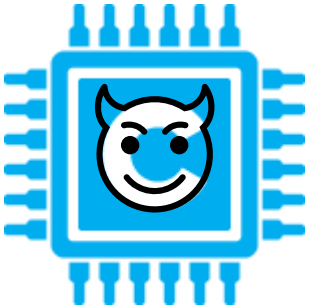
Advisor: Andreas Finkenzeller

Department of Electrical and Computer Engineering
Technical University of Munich

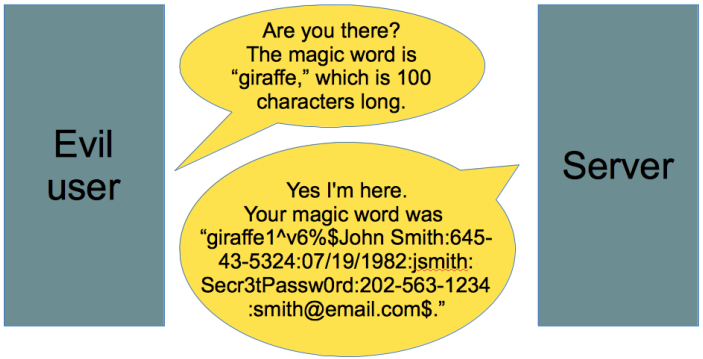
Contents

Introduction

The Heartbleed Vulnerability [1]



C for Embedded Systems



Example of Buffer overread [2]

Motivation

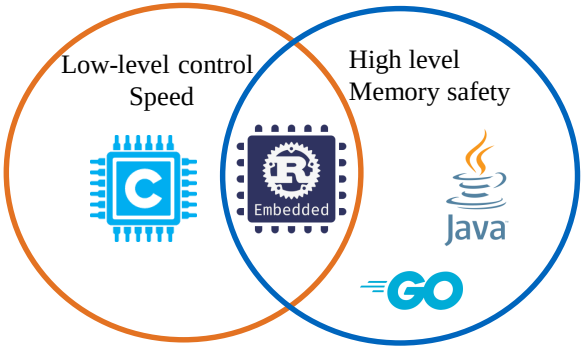
- Defense mechanism
  - Address Space Layout Randomisation
  - Secure Coding
- Performance overhead

- Garbage Collector
- Does not port to assembly

- Memory safety
- Bad cross-compiling

Aim

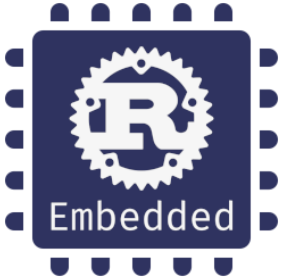
- Rust [3]



- **Evaluate claims of Rust for Embedded Systems**
  - Security provided by Rust
  - Performance of Rust compared to C
  - Compatibility of Rust with C libraries (Usability)

Introduction to Rust

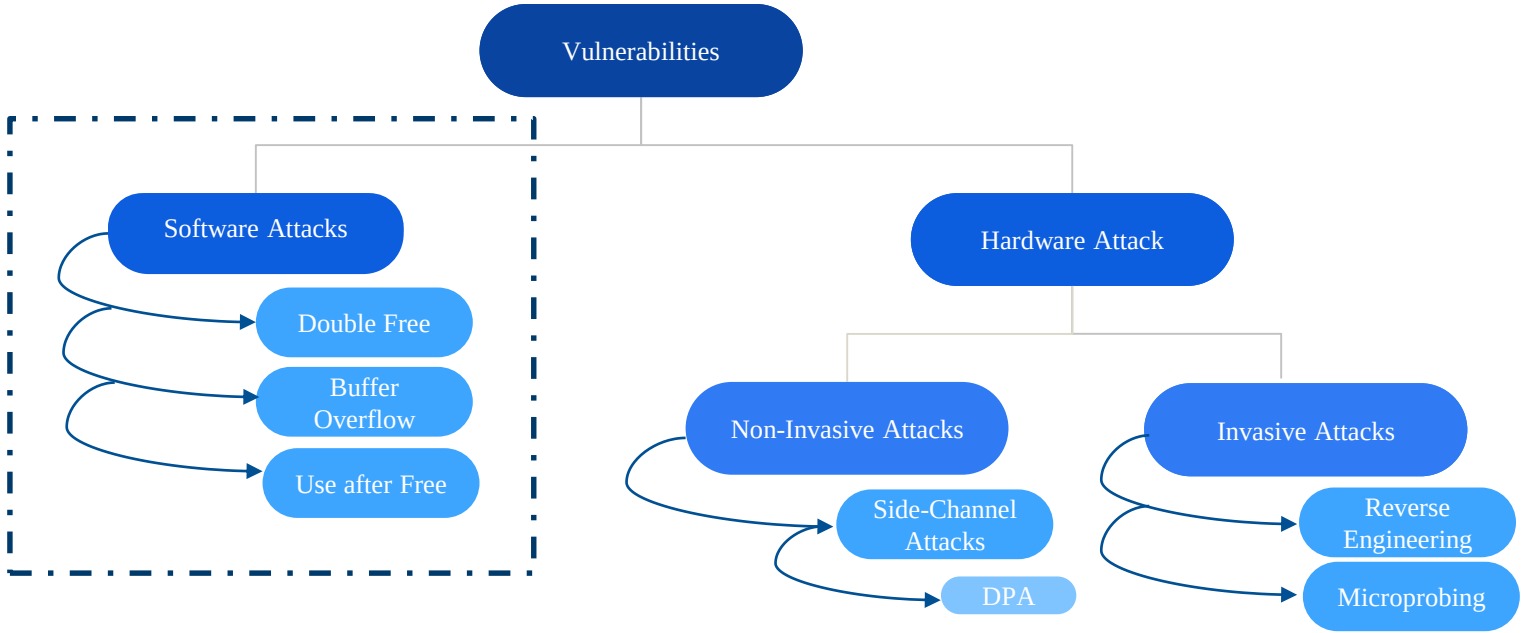
- Suitable for embedded programming:
  - Produces predictable assembly
  - No garbage collection



Memory Safety
Type Safety

Ownership	Every value in Rust has a single owner at a given time
Borrowing	Used to pass the reference but keep the ownership
Mutability	Immutable by default. Makes sharing of data easier. Prevents overwriting

Security Vulnerabilities in Embedded System



Software Attacks - Buffer Overflow

```
let mut a: [char; 2] = ['1', '2'];
let b: [char; 3] = ['1', '2', '3'];
a.copy_from_slice(&b);
```

→ Rust panics and terminates the program.

```
let buffer : [u8;30];
let s: [u8;21];
buffer = &s;
```

→

```
error[E0308]: mismatched types
  --> src/main.rs:28:13
   |
28 |     buffer = &s;
   |              ^^ expected an array with a fixed size of 30 elements, found one with 21 elements

error: aborting due to previous error; 3 warnings emitted
```

Heartbleed Vulnerability in Rust

```
memcpy(output, input, payload);
```

- If `payload` larger than the buffer `input`, additional data copied to the `output`

```
let payload:usize = ((input[1] as usize) << 8) + input[2] as usize;
let mut output: [u8;payload+PADDING];                    //needs constant size of array
output.extend(input[3..payload].iter().cloned());  //copy of payload string happens here
```

- *Payload* is upper index of the slice to be copied
- Rust panics if tried to access out of bound value from *p* [1]

Memory Exploits Using Unsafe Keyword

Unsafe Keyword

- Calling functions from other languages
- Bypass the compiler checks
- Interact with hardware
- Access Unions

1. Dereferencing a dangling pointer

```
let p;
{
   let mut x: u8 = 1;
   p = x as *const i32;
}
unsafe{
   hprintln!("{}",*p);
  }
```

➢ *unsafe* allows dereferencing the pointer even when it is freed

Buffer Overread and Immutability

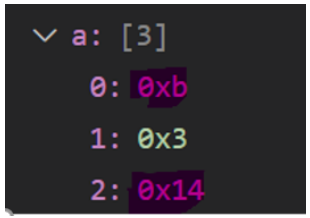
2. Overread

```
let mut a: [u8; 3] = [2,3,20];
let ptr: *mut [u8;3] = &mut a;
let big_ptr: *mut [u8;20] = ptr as *mut [u8;20];
unsafe {
  let mut new_array: [u8;20] = *big_ptr;
  new_array[10] = 10;
        }
```

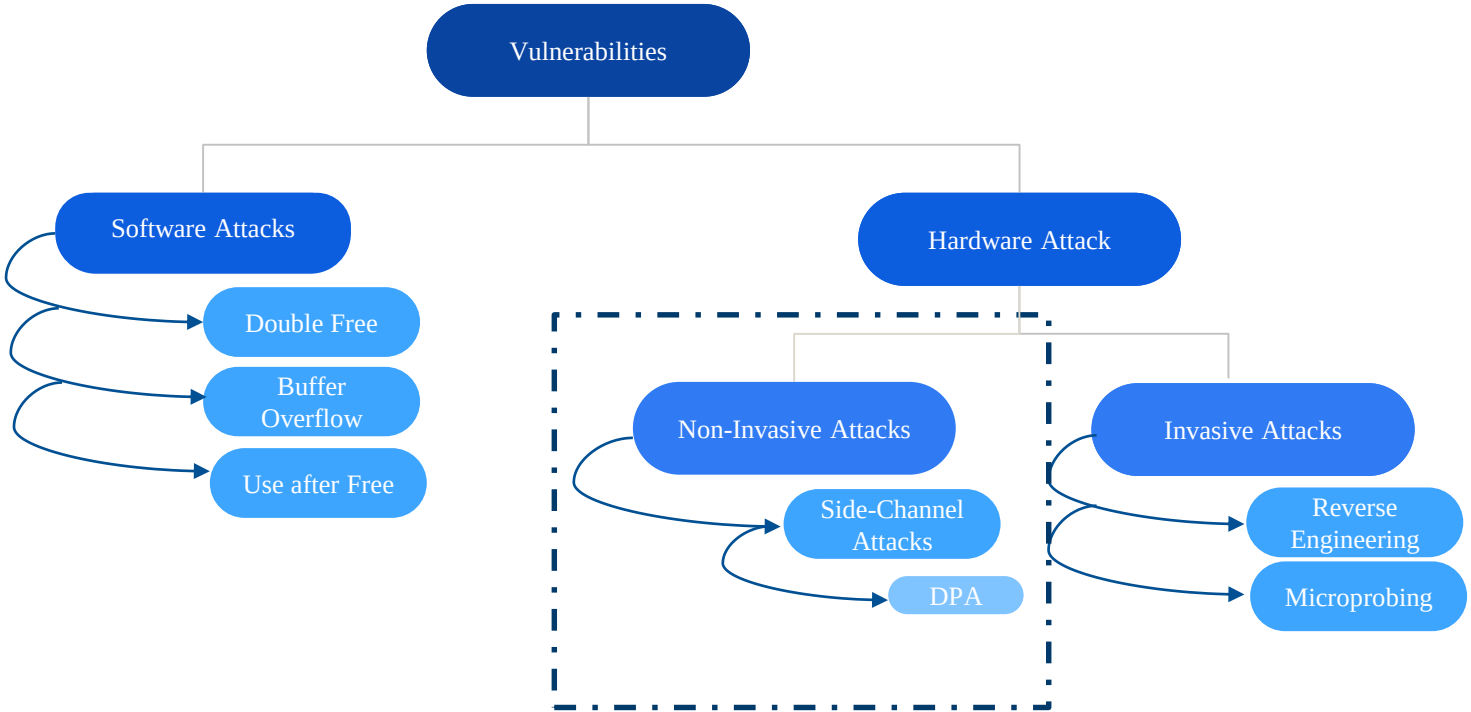
- `*big_ptr` copy the contents of `big_ptr` to `new_array`. **Overread** by reading the memory past the length of `a`

3. Immutability

```
let a: [u8; 3] = [2,3,20];
let ptr: *const [u8;3] = &a;
let big_ptr: *mut [u8;20] = ptr as *mut [u8;20];
unsafe {
  let mut new_array: &mut [u8;20] = &mut *big_ptr;
  new_array[0] = 11;
 }
hprintln!("{}",a[0]); \\ prints 11
```

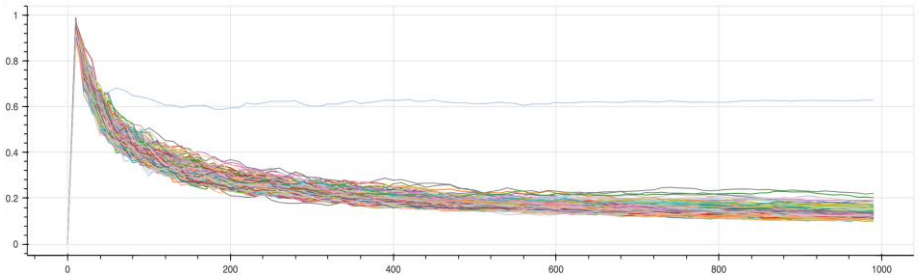


Security Vulnerabilities in Embedded System

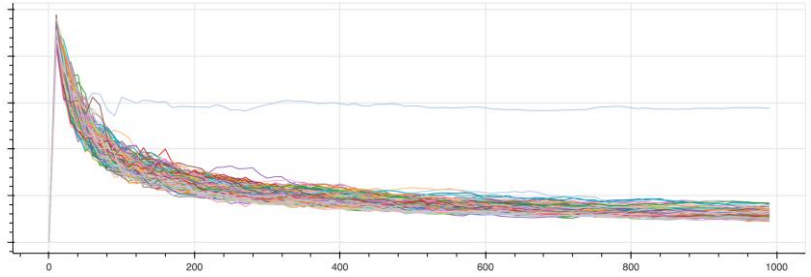


Non-Invasive Attacks - DPA

- Power analysis of 255 different values of the key position 0 for 1000 traces



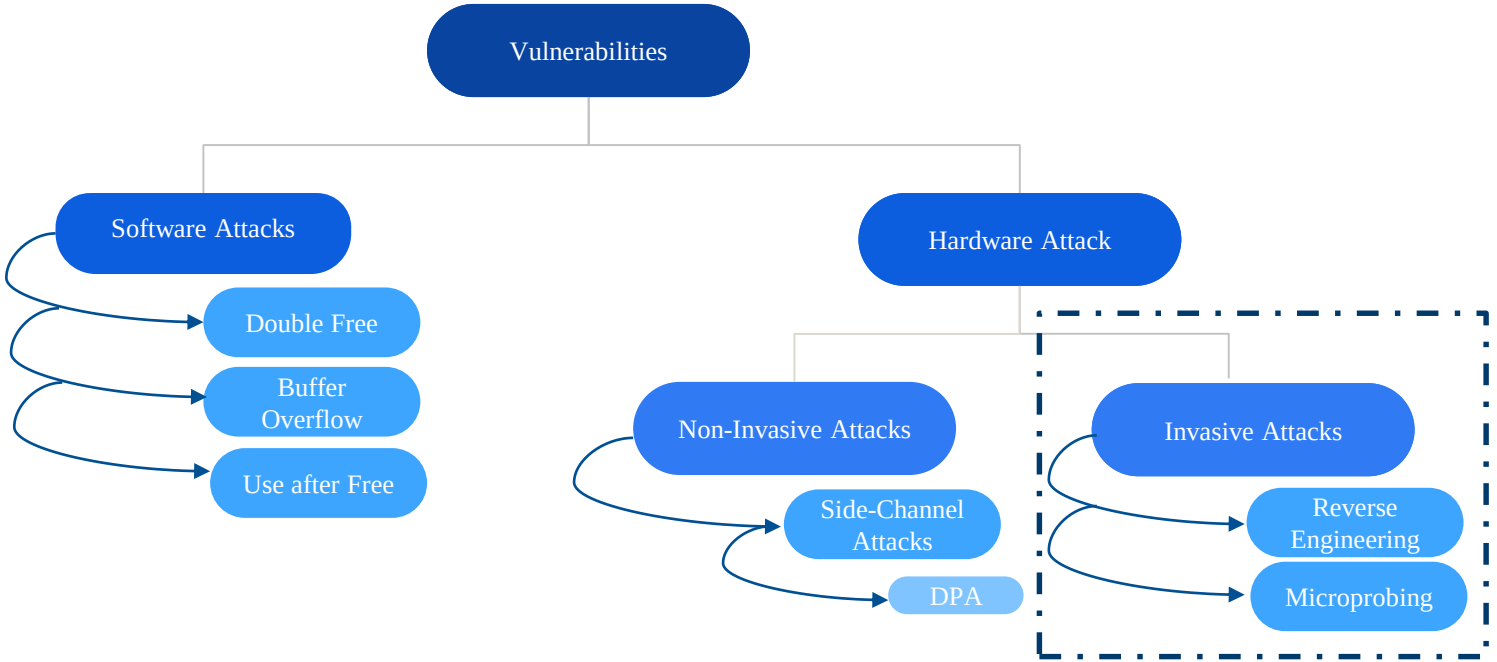
Rust



C

- No additional safety against the side-channel attacks
- Key read out in less than 100 power traces.

Security Vulnerabilities in Embedded System



Secure Coding patterns against Invasive Attacks

- **Secure Boolean**
  - Non-zero values result in True
  - 255 possibilities to flip a bit
- **Solution**
  - Same hamming weight and distance
  - Use volatile

```
enum SecBool {
       SecTrue = 0x9999, \\Hamming Weight(HW)=8
       SecFalse = 0x3C3C, \\ HW = 8
       SecInit = 0x5A5A  \\ HW = 8
}
match unsafe { *core::ptr::read_volatile(&val)}
 {
  SecBool::SecTrue=>  hprintln!("TRUE“),
  SecBool::SecFalse =>hprintln!("FALSE),
    _ => panic!()
  };
```

Secure Coding patterns against Invasive Attacks

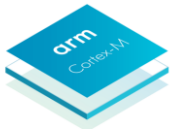
- Secure Return parameter
- Secure Branch Handling
- Secure Loop

Performance Evaluation

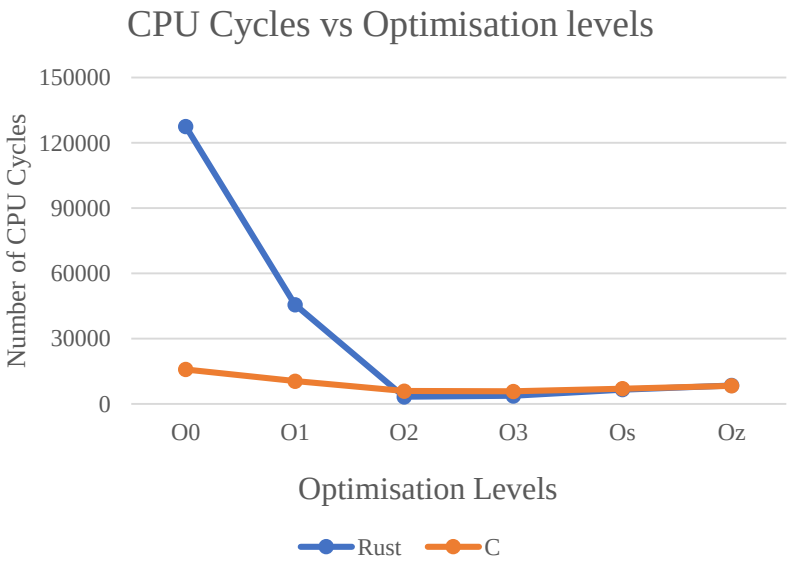
- AES algorithm → Used for the performance evaluation

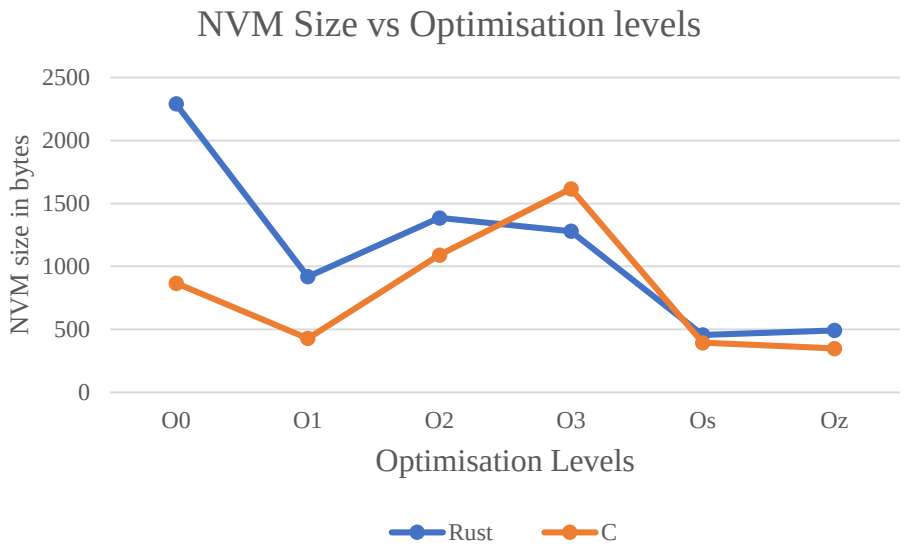
Optimization Level	O0, O1, O2, O3, Os, Oz

- Evaluation is done based on
  - Clock Cycles
  - NVM Size
  - Stack Usage

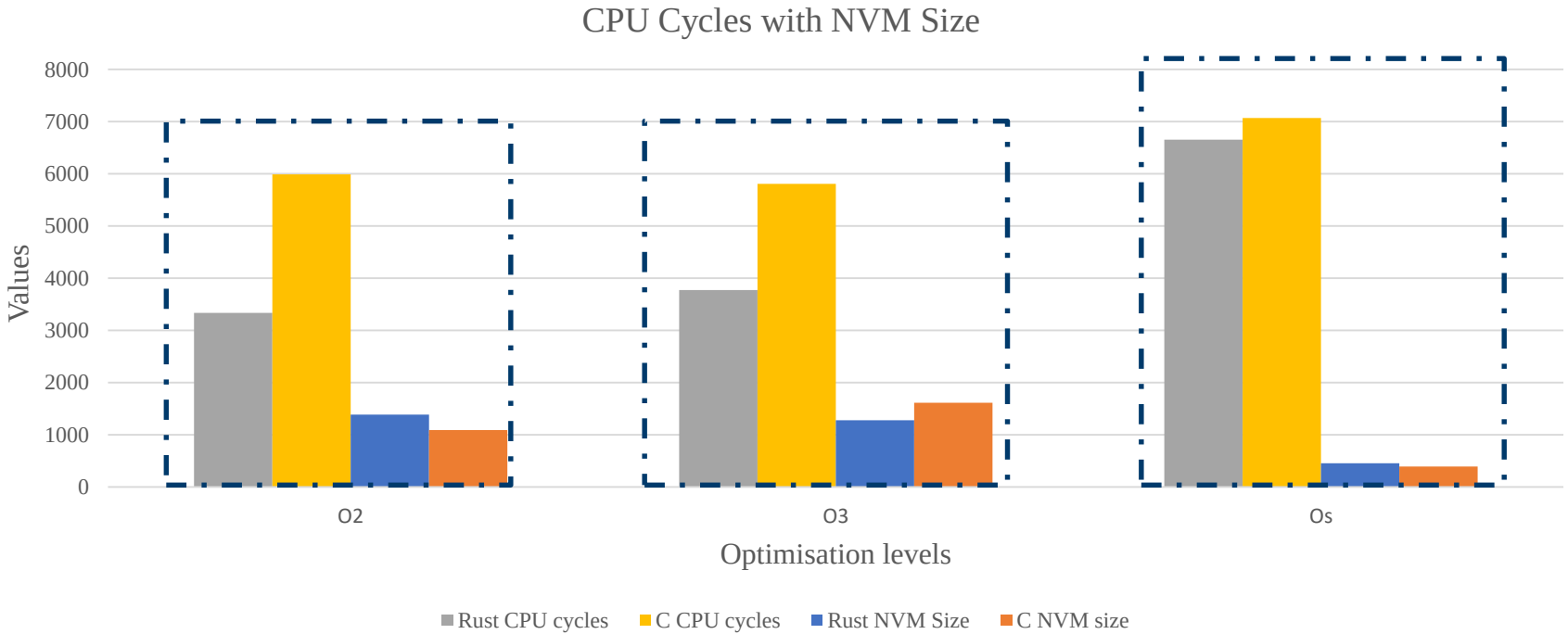


Evaluation of Clock cycles and NVM Size

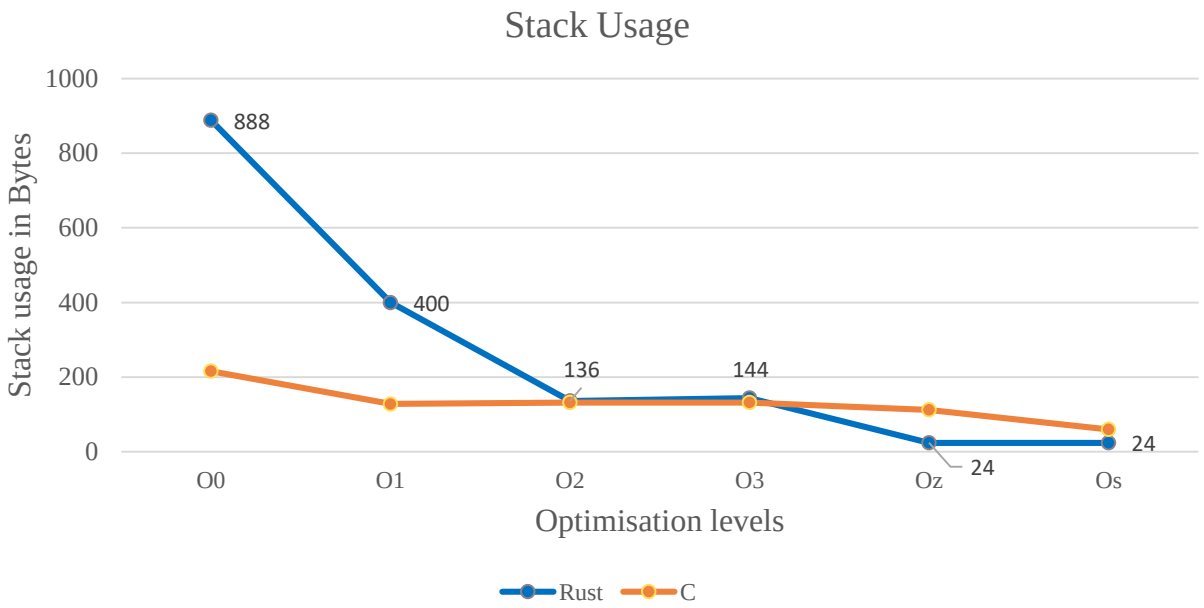




CPU Cycles varying with NVM Size



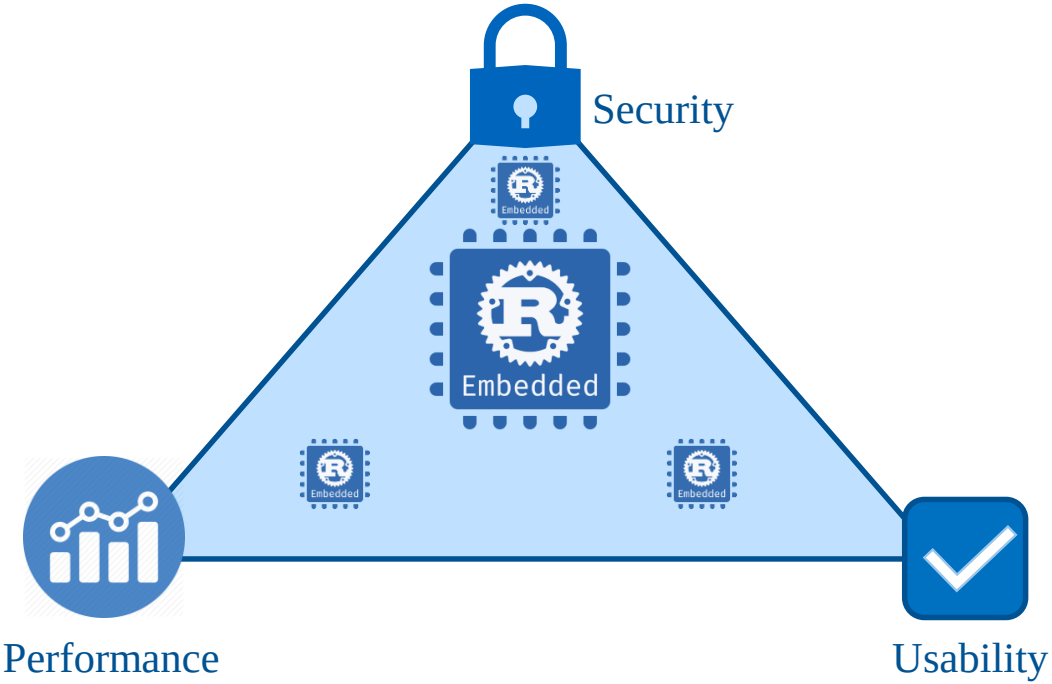
Stack Usage


Stack Usage
1000
800
600
400
200
0
Stack usage in Bytes
888
400
136
144
24
24
O0
O1
O2
O3
Oz
Os
Optimisation levels
Rust
C

Summary

	Security		Performance			Usability		
	Software Attacks	Hardware attacks	Execution Speed	NVM Size	RAM consumption	Inter-portability	Embedded Support	Learning Curve
Rust	**++**	-	**++**	-	**++**	++	**++**	-
C	--	-	+	+	+	++	+	++

Conclusion


Security
Embedded
Embedded
Embedded
Embedded
Performance
Usability

Future Work

- Analysis of Oz optimization level
- Dynamic memory allocation and run-time checks
- Using Rust for multi-threaded programming

References

[1] "The Heartbleed Bug", https://heartbleed.com/#:~:text=The%20Heartbleed%20Bug%20is%20a,used%20to%.

[2] "Buffer Overread" https://www.vox.com/2014/6/19/18076318/heartbleed

[3] 'The Rust Programming Language'. [Online]. Available: https://doc.rustlang.org/book/ [Accessed: 17-Feb-2016].

Questions?