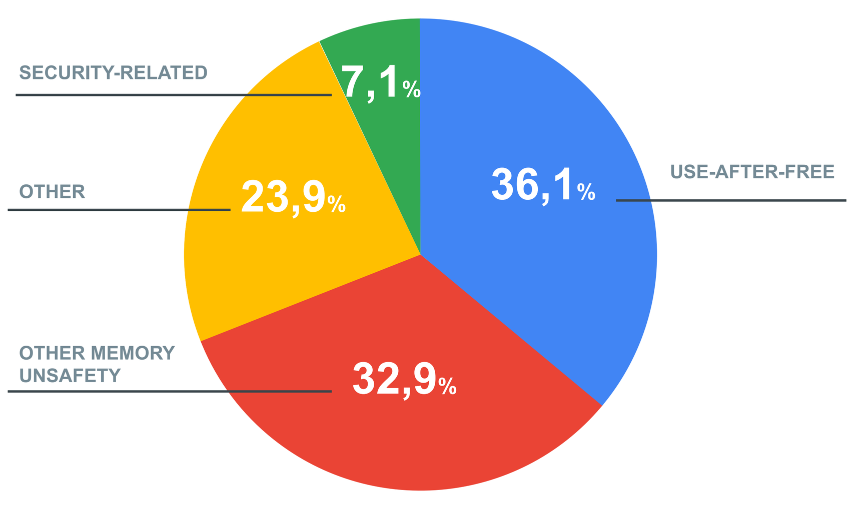
HENRIQUE PRETO, NUNO LOPES

inesc-id.pt

Automatic Translation of C++ to Rust

1. Intro

- Memory safety bugs are a critical class of software security vulnerabilities, responsible for **70% of security vulnerabilities in major Microsoft and Google projects**.
- **C and C++**, while **not memory-safe**, are commonly used to build complex and critical systems software due to their efficiency.
- On the other hand, **Rust** is a **memory-safe** programming language that offers comparable performance to C and C++.
- Fully rewriting older software systems in Rust is not practical.



70% of serious security bugs in Chrome are memory safety bugs

2. GOAL

- Implement a source-to-source compiler that automatically converts a specific subset of **modern C++ code to safe Rust**.
- Reduce the potential for security vulnerabilities while preserving performance and efficiency.

3. C2Rust

- State-of-the-art approaches generate **unsafe Rust**.
- For example, **unsafe pointers** in C are converted to **unsafe pointers** in Rust.
- There is still a question of how to convert **unsafe references** in C++ to **safe references** in Rust.

```
int main() {
    int x = 0;
    int *p = &x;
    if (p) {
        int y = 1;
        p = &y;
    }
    int z = *p;
    return 0;
}
```

A dangling pointer in a C program

```
unsafe fn main_0() -> libc::c_int {
    let mut x = 0 as libc::c_int;
    let mut p: *mut libc::c_int = &mut x;
    if !p.is_null() {
        let mut y = 1 as libc::c_int;
        p = &mut y;
    }
    let mut z = *p;
    return 0 as libc::c_int;
}
```

Unsafe C2Rust translation

4. Why doesn't naive translation work?

- Since C++ is not memory-safe like Rust, a naive translation is not possible.
- Unlike Rust, C++ does not have strict rules governing the **ownership** of memory, **lifetimes**, and **mutability** of references at compile-time, which can lead to memory safety bugs and undefined behaviour.
- Moreover, even memory-safe programs in C++ may not compile in Rust if a naive conversion is followed.

```
int arr[] = {0, 1};
int &first = arr[0];
arr[1] = 0;
first = 1;
```

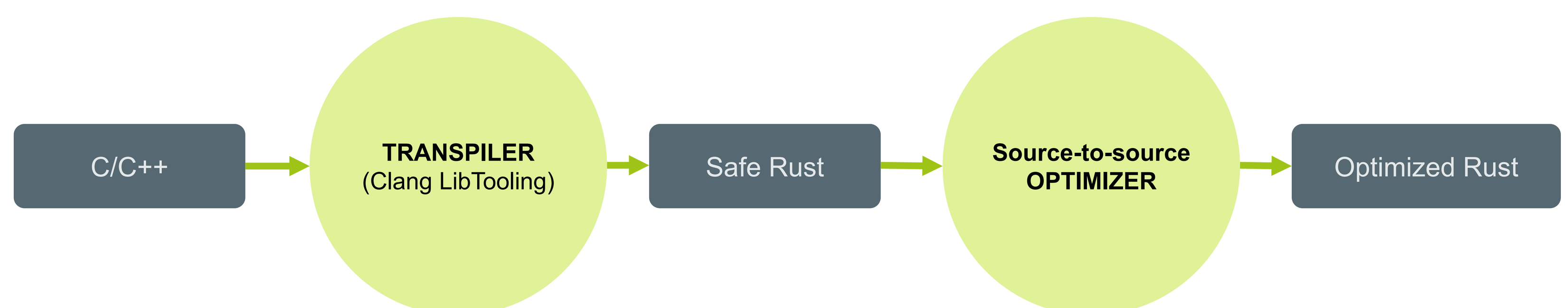
A valid and memory-safe C++ program

```
let mut arr: [i32; 2] = [0, 1];
let first: &mut i32 = &mut arr[0];
arr[1] = 0;
*first = 1; // compile-time error!
```

An invalid Rust program with multiple mutable references

5. Approach

- This work proposes a two-step translation approach.
- First, C++ code is converted into a **safe reference-counted Rust** version, which dynamically checks lifetimes and borrow rules.
- In other words, the transpiler shifts Rust's **borrow checking mechanism from compile-time to run-time**, which may result in a performance cost.
- In the end, a static analysis will be performed to refactor the translated code into a **more idiomatic and optimised Rust** version, when possible to statically prove memory safety.



A two-step approach for converting C++ code into Rust

```
let arr: Rc<RefCell<[Rc<RefCell<i32>>; 2]>> = Rc::new(RefCell::new(
    [Rc::new(RefCell::new(0)), Rc::new(RefCell::new(1))]));
let first: Weak<RefCell<i32>> = Rc::downgrade(&(*arr.borrow())[0]);
*(*arr.borrow())[1].borrow_mut() = 0;
*first.upgrade().expect("err").borrow_mut() = 1;
```

Transpile C++ code into a reference-counted Rust version

```
let arr: RefCell<[RefCell<i32>; 2]> = RefCell::new(
    [RefCell::new(0), RefCell::new(1)]);
let first: &RefCell<i32> = &arr.borrow()[0];
*arr.borrow()[1].borrow_mut() = 0;
*first.borrow_mut() = 0;
```

Refactor the Rust code after validating the lifetime of references

```
let arr: [RefCell<i32>; 2] =
    [RefCell::new(0), RefCell::new(1)];
let first: &RefCell<i32> = &arr[0];
*arr[1].borrow_mut() = 0;
*first.borrow_mut() = 1;
```

Refactor the Rust code after proving the exclusivity of mutable references

This work was partially supported by Google and national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020.