Poster: Ensuring Memory Safety for the Transition from C/C++ to Rust

Oliver Braunsdorf
LMU Munich
Bundeswehr University Munich
oliver.braunsdorf@ifi.lmu.de

Konrad Hohentanner
Fraunhofer AISEC
konrad.hohentanner@aisec.fraunhofer.de

Johannes Kinder
LMU Munich
johannes.kinder@lmu.de

***Abstract*—In mixed-language applications, where Rust is combined with C/C++ code, vulnerabilities in C/C++ can undermine Rust's memory-safety guarantees and can lead to faults that arise even in safe Rust code. Memory-safety sanitizers can be applied to both languages during compilation and linking to produce an application with whole-program memory-safety. However, memory-safety sanitizers introduce a significant performance penalty. We present SafeFFI, an approach to reduce performance overhead of sanitizers by eliding checks. The main contribution is a concept and implementation to translate memory-safety information between the static memory-safety enforcement mechanisms of the Rust compiler and the dynamic run-time checks of the sanitizer. Results show that we can significantly reduce the sanitizer's overhead while maintaining whole-program memory safety.**

I. Introduction

Rust is a modern programming language that offers significant advantages for system software development, especially regarding memory safety. However, adopting a new programming language can only happen gradually, replacing a large amount of existing legacy code step by step. This is why today, we see the combination of C/C++ code and Rust within the same address space in projects that use Rust's Foreign Function Interface (FFI) (see Figure 1). However, memory-management faults in C/C++ can cause memory-safety violations, like Buffer-Overflow or Double-Free, even in safe Rust code [1]. Compiler-based memory-safety sanitizers can help securing Mixed-Language Applications (MLAs). But the additional instructions that the sanitizers introduce to guarantee memory-safety can have a significantly negative performance impact. These additional sanitizer instructions are necessary for securing unsafe Rust code and C/C++ code but bring no additional benefit to those Rust code parts that are already proven to be safe by the Rust compiler.

In this paper, we present SafeFFI, a concept and implementation to optimize away unnecessary sanitizer instructions to improve the performance of sanitized mixed-language code while still maintaining the memory-safety guarantees of Rust's type system. We modified the Rust compiler and the underlying LLVM framework to implement a compatibility layer between the memory-safety information available in Rust's type system and the representation of memory-safety metadata in LLVM's sanitizers. We adapted two existing memory-safety sanitizers—SoftBound/CETS [2] and the Hardware-assisted AddressSanitizer (HWASAN)—to utilize SafeFFI to sanitize only necessary parts of MLAs that consist of Rust and C code.

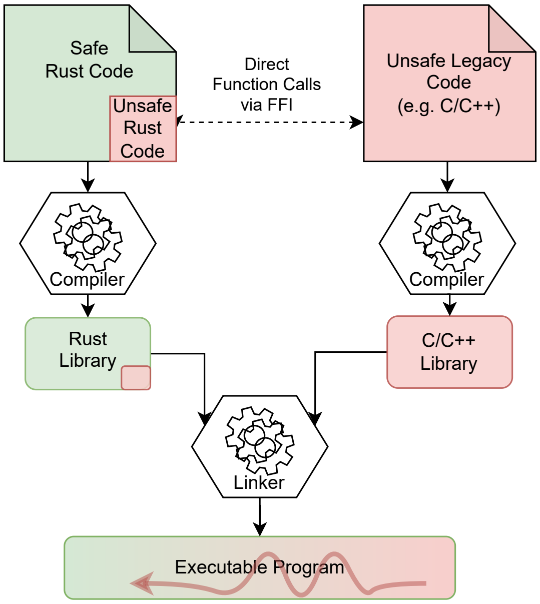


Fig. 1. Building mixed-language applications with C/C++ and Rust can lead to vulnerabilities

II. Background

Rust statically guarantees memory safety for accessing the following pointer types that we will refer to as *safe pointers*:

- `&T`: a shared, immutable reference to a stack object
- `&mut T`: an exclusive, mutable reference to a stack object
- `Box<T>` a pointer to a heap object
- `[T; N]`: a static array with size known at compile-time
- pointers to functions, closures etc.

For dereferencing safe pointers, Rust's type system guarantees the following memory-safety properties:

1) Dereferencing only accesses memory within the bounds of the pointed-to object (spatial safety).
2) The accessed object is always valid, i.e., allocated and not yet freed (temporal safety).

However, the only pointer type that can be passed between Rust and C is the *raw pointer* type (`*mut T` or `*const T`)

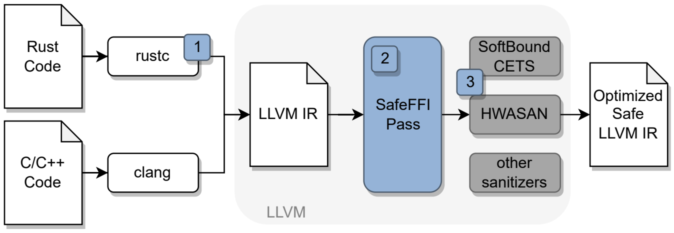


Fig. 2. SafeFFI's modular architecture allows to use different memory-safety sanitizers for different use cases

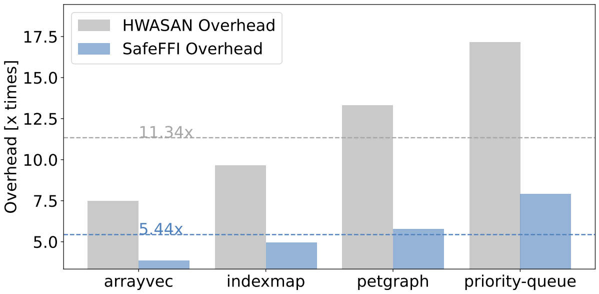


Fig. 3. SafeFFI reduces the sanitizer's performance overhead significantly.

because its binary layout resembles a C pointer. Rust gives no guarantees for dereferencing those raw pointers, which why this is only allowed in code blocks marked with the `unsafe` keyword.

III. Approach and Architecture

Our approach aims to utilize Rust's existing static compiler checks to guarantee memory safety for accessing safe pointers and using dynamic run-time checks of the sanitizer to guarantee memory safety for raw pointers. Thereby, performance overhead for instrumenting safe pointer accesses can be avoided.

For exchanging data between Rust and C, developers have to cast safe pointers to raw pointers before passing them to C functions and vice versa after receiving raw pointers from C functions. Thus, the main contribution of SafeFFI is a concept and algorithm that elides sanitizer checks for safe pointers but maintains Rust's memory-safety guarantees for those safe pointers even if they are casted to or from raw pointers and used in potentially vulnerable C or unsafe Rust code.

To implement this concept, we also contribute a modular architecture that enables the use of different sanitizers as a memory-safety backend (see Figure 2).

Since the Rust compiler is build on top of the LLVM compiler framework, we can utilize existing LLVM sanitizers. We implemented our approach with HWASAN and SoftBound/CETS. The implementation consists of the following steps

1) *Pointer Annotation*: Step 1 is implemented in the Rust compiler where we have detailed type information available to differentiate between safe and raw pointers, which is lost during lowering from Rust's Mid-Level Intermediate Representation (MIR) to LLVM IR. We annotate safe pointers, raw pointers, and casts between them by attaching LLVM Metadata Nodes to the respective LLVM IR Values. In those annotations, we include the size of the pointed-to memory-objects which is necessary to convert the spatial safety requirements of the Rust compiler to the spatial memory-safety metadata of the sanitizer.

2) *Intra-Procedural Static Analysis*: In the SafeFFI LLVM Pass, we conduct a static taint analysis to determine which LLVM values inside a function are safe pointers or raw pointers. All pointers are initialized as "safe". When the analysis encounters a "raw" pointer annotation, the pointer is tainted accordingly as "raw". If a pointer is casted from raw to safe, we remove the "raw" taint and additionally insert a sanitizer check instruction. With this instruction, SafeFFI guarantees temporal safety, and uses the size value attached in the annotation to ensure that Rust's spatial safety guarantee upholds. Thereby, we ensure that every untainted pointer can be treated as a safe Rust pointer and is guaranteed to adhere to Rust's memory-safety rules.

3) *Optimization* We slightly modify the sanitizers to query our analysis results and elide creation, propagation, and validation of the sanitizer's memory-safety metadata for all safe pointers.

IV. Evaluation

For our first evaluation, we focused on HWASAN as this sanitizer is maintained inside the LLVM project and works more stable for Rust code than the SoftBound/CETS prototype.

A. Performance

For evaluating the run-time performance, we selected four real-world Rust crates that contain micro-benchmarks and measured them in three configurations: baseline (without sanitizer), with HWASAN, and with a SafeFFI-optimized HWASAN. The results in Figure 3 show that SafeFFI reduces HWASAN's run-time overhead from 11x to 5x on average.

B. Security

To validate that our optimizations do not reduce the sanitizer's capabilities, we reproduced eight crates with known vulnerabilities with and without SafeFFI. HWASAN detected seven out of the eight vulnerabilities in our dataset. With our SafeFFI optimizations additionally enabled, we can also detect all seven vulnerabilities that HWASAN detected. This shows evidence that SafeFFI successfully maintains the security guarantees of the utilized sanitizer while reducing its performance overhead.

References

[1] Z. Li, J. Wang, M. Sun, and J. C. S. Lui, "Detecting cross-language memory management issues in rust," in *Computer Security – ESORICS 2022*, ser. Lecture Notes in Computer Science, V. Atluri, R. Di Pietro, C. D. Jensen, and W. Meng, Eds. Springer Nature Switzerland, 2022, pp. 680–700.

[2] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: compiler enforced temporal safety for c," *SIGPLAN Not.*, vol. 45, no. 8, p. 31–40, jun 2010. [Online]. Available: https://doi.org/10.1145/1837855.1806657

SafeFFI - Ensuring Memory Safety for the Transition from C to Rust

Oliver Braunsdorf[1,3], Konrad Hohentanner[2], Johannes Kinder[1]

[1] LMU Munich, [2] Fraunhofer AISEC, [3] Bundeswehr University Munich – FI CODE

Current Situation

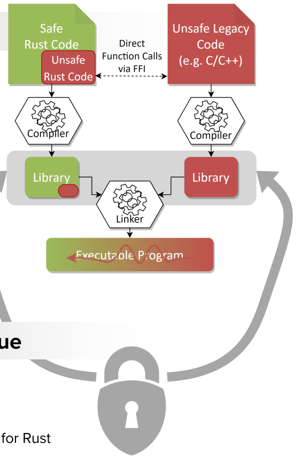
- Rust brings advantages: strong type system, memory-safety, performance
- However: **organizational issues slow down adoption** of Rust in industry, e.g.:
  - Hiring / training employees
  - Adapting development and test processes

➡ **Transition** can only happen **gradually**: replace C code with Rust step-by-step

➡ **Mixed-Language Applications (MLA)**: both languages in one address space using Foreign Function Interface (FFI)

Problem

- In MLAs **memory-safety** guarantees of Rust can be **undermined** by
  - **Vulnerabilities in C**, e.g. buffer overflow
  - **Vulnerabilities in** unsafe **FFI code** due to mistakes in handling pointers
  - **Disregarding of Rust's pointer aliasing rules**, which can lead to new violations like double free [1]



Sanitizers to the Rescue

- **LLVM's memory-safety sanitizers** can be used for both C and Rust (**ASAN**, **HWASAN**, ...)
- We also ported **Softbound/CETS** [2] for Rust

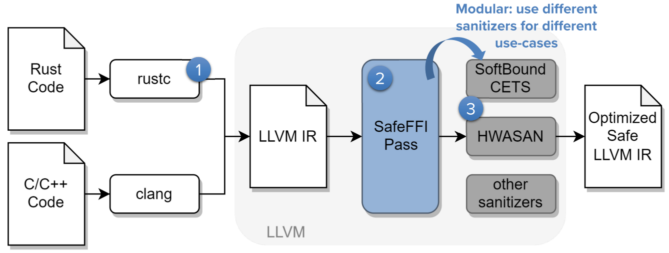
But: Sanitizers cause high run-time and memory overhead

Approach: Only Utilize Sanitizer for C and Unsafe Rust Code

- Definition
  - Safe pointers: pointers whose memory safety is guaranteed by Rust, e.g., Rust references `&T` / `&mut T`
  - Raw pointers: pointers which give no guarantee when dereferencing `*T` / `*mut T`
- **Goal**: Use sanitizer only for raw pointers, elide sanitizer instructions for safe pointers, maintain whole-program memory safety
- **Observation**: Casts between safe and raw pointers are the boundary between the two memory-safety mechanisms

Contribution: Translate **Rust's static memory safety** information to **dynamically checked sanitizer metadata** to bridge the two memory-safety mechanisms

Architecture



1 **Initial Pointer Annotation**

- Annotate pointers during lowering from Rusts MIR to LLVM IR, attach size metadata
- Function arguments
- Raw pointers as return value from function calls
- Casts between safe and raw
  - Generate dummy instructions if cast is a no-op in LLVM

2 **Static Analysis**

- Intra-procedural taint analysis to propagate safe/raw pointer annotations
- Insert run-time checks for safe pointers casted from raw pointers
- Ensures that pointers marked as `safe` adhere to Rust's memory-safety guarantees

3 **Optimization**

- Raw pointers: translate pointer annotations to sanitizer-specific metadata format; continue to use sanitizer instrumentation
- Safe pointers: elide any sanitizer instructions for creating, propagating, or validating metadata

Code Example

```
pub fn bar(safe: &mut MyStruct) {
   ...
}

pub fn c_wrapper(raw_ptr: *mut MyStruct) {
    unsafe {
        c_function(raw_ptr as *mut c_void);
    }
}

pub fn foo(
    a: &mut MyStruct,
    b: *mut MyStruct,
    condition: bool)
{
    bar(a);
    a.elem = 42;
    c_wrapper(b);
    let b_safe: &mut MyStruct = unsafe { &mut *b };
    bar(b_safe);
    if condition {
        let a_raw = a as *mut MyStruct;
        c_wrapper(a_raw);
    }
    bar(a);
}
```

checking safe pointer `a´ here is unnecessary

inserts sanitizer instructions

Rust compiled to LLVM IR with sanitizer

```
define internal void @foo(
%MyStruct* %a, %MyStruct* %b) {
start:
  call void @sanitizer_propagate(%a)
  call void @bar(%MyStruct* %a)
  %0 = getelementptr MyStruct* %a,i32 3
  call void @sanitizer_check(%a)
  store i32 42, i32* %0
  call void @sanitizer_propagate(%b)
  call void @c_wrapper(%MyStruct* %b)


  call void @sanitizer_propagate(%b)
  call void @bar(%MyStruct* %b)
  br i1 %condition, label %bb2, label %bb3

bb2:

  call void @sanitizer_propagate(%a)
  call void @c_wrapper(%MyStruct* %a)
  br label %bb3

bb3:

  call void @sanitizer_propagate(%a)
  call void @bar(%MyStruct* %a)
  ret void
}
```

Activate SafeFFI

```
define internal void @foo(
%MyStruct* %a safe(16), %MyStruct* %b raw(16) ) {
start:
  ~~call void @sanitizer_propagate(%a)~~
● call void @bar(%MyStruct* %a)
● %0 = getelementptr MyStruct* %a,i32 3
  ~~call void @sanitizer_check(%a)~~
● store i32 42, i32* %0
  call void @sanitizer_propagate(%b)
● call void @c_wrapper(%MyStruct* %b)
  call llvm.donothing()  !raw-to-safe(%b, 16)
● call safeffi_ensure(%b, 16)
● ~~call void @sanitizer_propagate(%b)~~
● call void @bar(%MyStruct* %b)
  br i1 %condition, label %bb2, label %bb3
●
bb2:
  call llvm.donothing()  !safe-to-raw(%a, 16)
  call void @sanitizer_propagate(%a)
● call void @c_wrapper(%MyStruct* %a)
  br label %bb3
●
bb3:
  call safeffi_ensure(%a, 16)
  ~~call void @sanitizer_propagate(%a)~~
● call void @bar(%MyStruct* %a)
  ret void
●
```

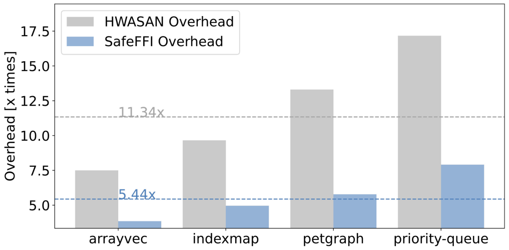
Result: Maintaining Memory Safety

Spatial Violations	HWASAN	HWASAN+ SafeFFI
CVE-2019-15551	✓	✓
CVE-2020-35862	✓	✓
CVE-2021-28032	x	x
CVE-2020-35892	✓	✓

Temporal Violations	HWASAN	HWASAN+ SafeFFI
CVE-2018-20995	✓	✓
CVE-2019-16139	✓	✓
CVE-2020-35887	✓	✓
CVE-2020-35892	✓	✓

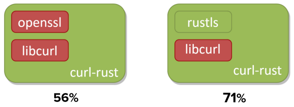
Detection of memory-safety violations in real-world Rust crates

Result: Improving Performance



Result: Incentivizing the Transition to Rust

Case study curl-rust:
How many sanitizer checks can be saved when replacing more C code with Rust?



Incentive: Rewriting in Rust regains performance

References

[1] Zhuohua Li, Jincheng Wang, Mingshen Sun, John C. S. Lui: Detecting Cross-language Memory Management Issues in Rust. ESORICS (3) 2022: 680-700

[2] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, Steve Zdancewic: CETS: compiler enforced temporal safety for C. ISMM 2010: 31-40

Contact

oliver.braunsdorf@ifi.lmu.de   konrad.hohentanner@aisec.fraunhofer.de   johannes.kinder@lmu.de