SafeFFI: Efficient Sanitization at the Boundary Between Safe and Unsafe Code in Rust and Mixed-Language Applications

Oliver Braunsdorf[1], Tim Lange[1], Konrad Hohentanner[2], Julian Horsch[2], and Johannes Kinder[1]

[1]Ludwig-Maximilians-Universität München, Germany
[2]Fraunhofer AISEC, Germany

Abstract

Unsafe Rust code is necessary for interoperability with C/C++ libraries and implementing low-level data structures, but it can cause memory safety violations in otherwise memory-safe Rust programs. Sanitizers can catch such memory errors at run time, but introduce many unnecessary checks even for memory accesses guaranteed safe by the Rust type system. We introduce SafeFFI, a system for optimizing memory safety instrumentation in Rust binaries such that checks occur at the boundary between unsafe and safe code, handing over the enforcement of memory safety from the sanitizer to the Rust type system. Unlike previous approaches, our design avoids expensive whole-program analysis; hence, it incurs significantly less compile-time overhead (2.01× compared to over 5.91×). On a collection of popular Rust crates, SafeFFI reduces sanitizer checks by up to 79.63%, while still detecting all memory safety violations in our dataset of known vulnerable Rust code.

1 Introduction

Memory corruptions caused by unsafe programming languages such as C and C++ remain a major source of critical software vulnerabilities. Memory bugs regularly take top spots in the lists of most dangerous [32] and known exploited weaknesses [31], and studies by Google [13,45] and Microsoft [29] indicate that around 70% of their severe bugs are caused by memory unsafety.

In recent years, the Rust programming language has gained traction as a safe-by-construction solution for newly developed software. Securing existing C/C++ programs by rewriting them in safe languages requires substantial development effort, however. Despite recent efforts to translate legacy code bases to Rust [9, 10, 51], we can expect C/C++ code to be relied on for the foreseeable future. Thus, in practice, Rust applications may actually be *mixed language applications* (MLAs) of Rust code linked against C/C++ code—or vice versa—via a foreign function interface (FFI). In this situation, the safety guarantees of Rust may be compromised by unsafe operations in foreign code. But even pure Rust code can contain *unsafe code*, marked via the `unsafe` keyword, which allows developers to violate Rust's strict typing rules to implement efficient algorithms and data structures or hardware interactions. Thus, be it from foreign functions or explicitly marked unsafe code regions, Rust bears the risk of memory safety violations originating in unsafe code that potentially affect the whole code base, even safe code [17,28].

Both C/C++ and unsafe Rust code can be protected by applying memory safety *sanitizers* [43, 46] to the whole code base, which transparently introduce run-time checks for memory operations. While sanitizers such as AddressSanitizer (ASan) [41] and Hardware-assisted AddressSanitizer (HWASan) [42] do not require any source code changes and can detect most memory safety violations [46], they introduce a significant run-time overhead by inserting checks for *every* pointer dereference in the code. However, many of these checks are unnecessary: a pointer dereference in Rust is guaranteed safe as long as the pointer is safe and cannot be affected by unsafe code. Previous solutions for selective sanitization of Rust code [5,30] use whole-program static points-to analysis to classify pointers as provably safe or potentially unsafe and then elide checks for an under-approximation of safe pointers. While theoretically sound, this approach incurs significant compile-time overhead and misses optimization opportunities. In contrast, our approach relies on efficient local reasoning about pointer types and their safety guarantees.

In this paper, we present SafeFFI, a new approach to reduce the overhead of memory safety sanitizers in Rust applications and MLAs consisting of C and Rust code. SafeFFI utilizes the fact that Rust's strong type system enforces guarantees for pointer types such as reference types (`&T`) and box types (`Box<T>`), while raw pointers (`*const T`) are unchecked. A key insight in our work is that the cast from a raw pointer type to a safe pointer type forms the boundary between sanitizer-enforced memory safety and type-system-enforced memory safety. SafeFFI hoists and bundles checks into a single dynamically-checked precondition, which is propagated

Published in 35th USENIX Security Symposium, Baltimore, MD, USA, August 12 – 14, 2026

statically through the type system. This way, we free the memory sanitizer from checking *every* pointer dereference. Therefore, for most patterns of Rust programs, we can elide a significant number of sanitizer checks, leading to improved run-time performance.

We only require local reasoning for hoisting memory safety checks, hence, we avoid expensive whole-program static analysis leading to better compile-time performance and enabling SafeFFI to scale well on large software projects. In summary, we make the following contributions:

- A novel concept and algorithm for combining sanitizers for unsafe languages and type information from strongly-typed languages to enforce memory safety in mixed code. Our algorithm avoids expensive whole-program static analysis and exposes bugs early by placing checks at the location where the type system expects safety guarantees to hold.
- A modular architecture, independent of the underlying sanitizer, based on a modified Rust compiler that allows for analyses across multiple intermediate representations of Rust and LLVM to implement the concept, including an efficient algorithm for finding potentially deallocating functions.
- A systematic evaluation using LLVM's widely-used sanitizers ASan and HWASan on popular Rust crates, known vulnerable Rust code, and a set of real-world benchmarks. SafeFFI effectively reduces the number of sanitizer checks by up to 79.63% and consequently reduces the run-time overhead of ASan from 2.71× to 2.44× and of HWASan from 3.18× to 2.29× while achieving better compile-time overheads (2.01×) compared to other state-of-the-art approaches (5.91×).

2 Background

We now introduce the necessary background on memory safety in C/C++ and existing methods for run-time sanitization (§2.1), followed by revisiting the memory safety guarantees in Rust and the impact of unsafe code (§2.2).

2.1 Memory Safety and C/C++

C and C++ remain widely used in security-critical software but inherently lack memory safety. Memory safety violations are classified into *spatial* (e.g., buffer overflows) and *temporal* (e.g., use-after-free) errors. To catch memory safety bugs, sanitizers [43, 46] typically instrument code at compile time with additional check logic to detect violations at run time. Furthermore, sanitizers often use symbol interposition to redirect function calls to instrumented versions of a library function, e.g., for malloc, free, or memcpy. This technique is called *Interception* and is especially helpful for sanitizing third party libraries without recompiling. Sanitizers use different types of metadata to track memory state and detect violations, typically categorized as being either object-based [4, 14, 22, 25, 41, 42] or pointer-based [16, 20, 34, 37]. Object-based metadata, such as redzones [41] or memory tags [42], is associated with memory allocations and marks memory regions as valid or invalid. Pointer-based metadata augments pointers with additional information, such as bounds [34, 37] and references to temporal metadata [35]. Since it is associated with pointers, it allows for tracking the validity of memory accesses based on the pointer's state, which can enable the detection of more bug categories, such as sub-object memory violations [46].

Two widely used memory safety sanitizers are ASan [41] and HWASan [42], available in LLVM and GCC. ASan uses shadow memory to maintain metadata for application memory and instruments code to place red zones around memory objects, catching out-of-bounds errors. It also poisons freed memory and delays reuse, improving detection for use-after-free errors at the cost of higher memory overhead. HWASan reduces overhead by using tagged pointers and memory tags (typically 16:1 granularity). Each block carries a tag in shadow memory, and the upper pointer bits hold a matching tag; a mismatch signals a violation. Employing different tags for freed memory and for different objects allows HWASan to detect more types of temporal and spatial bugs than ASan. But, due to the limited tag size, tag collisions can occur, making all detections in HWASan probabilistic. On ARM, HWASan commonly leverages top-byte-ignore (TBI) to store tags in the pointer's top byte, primarily improving performance by not requiring pointers to be stripped before usage, but also compatibility with non-instrumented code.

2.2 Memory Safety and Rust

Rust is a modern systems programming language that aims to eliminate memory safety bugs through a strong static type system. This type system is built on the principles of ownership, borrowing, and lifetimes [19] which guarantee spatial and temporal memory safety for safe pointer-like types. *Spatial safety* in Rust is enforced either at run time or at compile-time. For statically-sized memory objects, the size of the memory object is known at compile-time. The Rust compiler tracks the object's type and its corresponding size for every safe pointer derived from the memory object. Thus, it can statically verify that a pointer dereference is within the bounds of the pointed-to memory object. For dynamically-sized types, the compiler generates run-time bounds checks when indexing or dereferencing through a safe pointer. *Temporal safety* is guaranteed by the type system which enforces that each value is owned by exactly one variable. The compiler takes care of generating code for the deallocation of the value at the location where this owning variable goes out of scope which prevents double-free errors. Lifetimes tracked statically by the compiler ensure that references to a value do not outlive the owner, hence preventing Use-After-Free (UAF) errors.

Safe Pointer-Like Types. The Rust language provides a set of primitive *safe pointer-like* types which are subject to the borrow checker and lifetime analysis. These include:

- `&T` – shared, immutable reference,
- `&mut T` – exclusive, mutable reference,
- `Box<T>` – owning pointer to heap-allocated data,
- `[T; N]` – statically-sized array,
- `fn, Fn` – function pointers and closures,
- `&[T], &str` – dynamically sized slice/string,
- `&dyn T` – trait object reference, a pointer to an object with dynamic type.

Our approach particularly focuses on the first five pointer types, all of which have statically-sized pointees with sizes known at compile-time. For the remainder of the paper, we will usually refer to them summarily as `safe` pointers, in contrast to `raw` pointers. For `safe` pointers that originate in `safe` Rust code, the Rust compiler ensures that they are non-null, properly aligned, and dereferenceable for the size of their pointee-type [7]. When `safe` pointers are derived from `raw` pointers, which may happen in `unsafe` code, Rust requires the developer to guarantee those same memory safety properties.

Raw Pointers and Unsafe Code. Rust also provides another primitive pointer type, called the `raw` pointer: `*const T` and `*mut T`. The `raw` pointer type is not subject to borrow checking or lifetime tracking. It does not provide any memory safety guarantees, thus it behaves exactly like a pointer in C/C++ and also has the same layout. Raw pointers may only be dereferenced inside functions or code blocks that are marked with the `unsafe` keyword in Rust. Besides dereferencing raw pointers, `unsafe` Rust code also enables the following operations which can undermine Rust's safety guarantees: 1. calling other `unsafe` functions or foreign functions, 2. accessing mutable static variables, 3. accessing `union` fields, 4. implementing unsafe traits. Those `unsafe` operations can undermine Rust's type system. Thus, within `unsafe` blocks, it is the programmer's responsibility to uphold Rust's memory safety guarantees. Because programmers can make mistakes in `unsafe` code, Rust programs might introduce memory safety violations despite Rust's strong type system. Miri [17] is a popular tool in the Rust ecosystem for detecting misuses of `unsafe` code that undermine Rust's type system and can lead to undefined behavior including memory safety violations. Since Miri is based on an interpreter for Rust's Mid-Level Intermediate Representation (MIR) and employs heavy-weight dynamic analyses, it introduces a significant performance overhead. As an alternative for memory safety, the Rust compiler offers the possibility of using sanitizers (see §2.1). However, these memory safety sanitizers lack awareness of Rust's static guarantees and therefore redundantly check even `safe` Rust code. This motivates our approach to combine Rust's compile-time guarantees with selective run-time checks of sanitizers.

Memory Safety in Mixed-Language Applications. A typical scenario of using `unsafe` code in Rust is the invocation of foreign functions written in other languages like C/C++. Rust provides a Foreign Function Interface (FFI) for declaring or exporting a function symbol from/to a foreign library, which later is linked in by the linker. Many real-world applications combine Rust with existing C or C++ code (or vice versa) this way; we refer to these as Mixed-Language Applications (MLAs). Raw pointers are used heavily to exchange data across the FFI boundary because they have the same layout in Rust as in C/C++. Bugs in the FFI definition or in the C/C++ code parts of an MLA can undermine Rust's guarantees and thus affect the safety of the whole application [23]. Even worse, cross-language vulnerabilities can be vehicles for bypassing hardening mechanisms for C/C++ like Control-Flow Integrity (CFI) [28, 38]. Unfortunately, Miri has very limited support for mixed-language scenarios because it only interprets Rust code. Memory safety sanitizers, on the other hand, are applicable to MLAs and protect the whole application because Rust code and C/C++ code can both be compiled to LLVM IR code, which sanitizers like ASan and HWASan can instrument.

3 Threat Model

To set the scene for explaining the SafeFFI approach, we first define our threat model, i.e., the capabilities and goals of our attacker, the scope of SafeFFI's protection, and the underlying assumptions and limitations.

The attacker aims to trigger memory safety violations in a target application with the goal to corrupt or leak data or hijack control flow. In order to do so, the attacker is able to supply arbitrary inputs to the application but cannot bypass sanitizer checks (e.g., by modifying the binary itself). Memory safety violations may be triggered by targeting memory bugs in unsafe parts of the application, i.e., foreign code or Rust code marked as `unsafe`. In detecting memory bugs, SafeFFI inherits the capabilities and limitations of the underlying sanitizer. For both HWASan and ASan this includes all standard types of spatial and temporal memory bugs but excludes more specific types, such as sub-object over- and underflows [46]. Furthermore, SafeFFI inherits limitations regarding thread safety from the underlying sanitizer. For ASan and HWASan, this means that a data race in the application may lead to a data race in the sanitizer's metadata updates. This causes undefined behavior, which might lead to a detectable memory safety violation but may also go undetected [41].

To provide its protections, i.e. guarantee that a Rust application including unsafe components remains memory-safe within the defined bounds, SafeFFI relies on a few fundamental *assumptions*:

Type System Soundness. We assume that the Rust type system is sound and safe in the absence of unsafe components (i.e., C/C++ and unsafe Rust code).

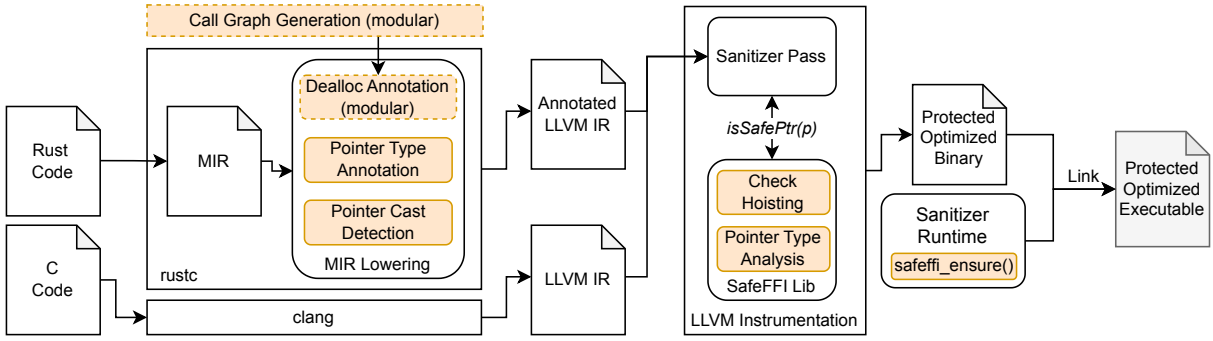


Figure 1: SafeFFI architecture overview. Rust and C/C++ are compiled to LLVM IR, a common intermediate representation for sanitizer instrumentation. The orange boxes depict our extensions to rustc and LLVM in this compilation workflow.

Toolchain Correctness. We assume that the compiler toolchain implementation is correct.

Sanitizer Correctness and Integrity. We assume that the underlying sanitizer works correctly and that its metadata is safe against the attacker.

Run-time Environment Integrity. We assume that the environment of the application process is intact and uncompromised, especially including the OS kernel and mappings.

SafeFFI only targets memory safety bugs and typical software-based attacks, which implies some basic *limitations*:

Logic and Type Safety Errors. SafeFFI only covers memory safety issues. Incorrect program behavior that does not violate memory safety cannot be detected or prevented. This includes type safety violations from `unsafe` code that do not lead to memory safety violations detectable by sanitizers.

Side-Channels and Hardware. Attacks based on side channels, speculative execution, and hardware fault injection are out of scope for the considered sanitizers and SafeFFI.

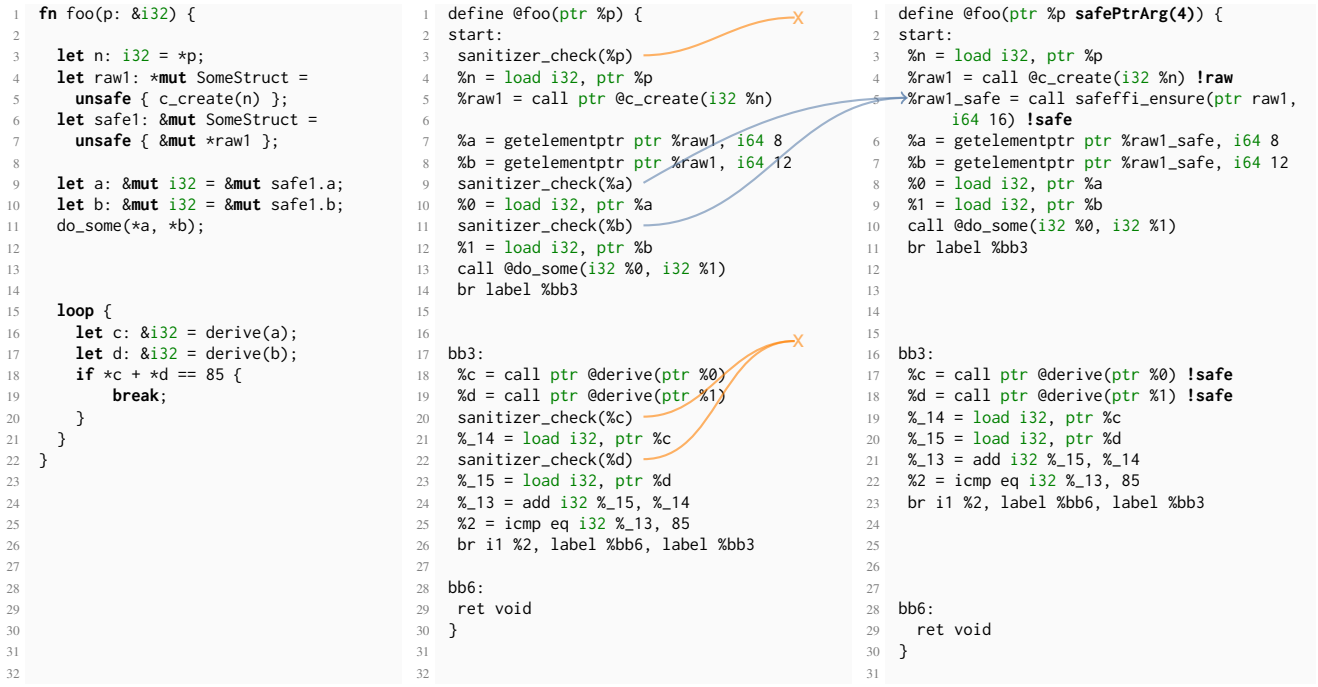
4 SafeFFI Overview

The architecture of SafeFFI, shown in Figure 1, is implemented as an extension of the Rust compiler pipeline. We extend the Rust compiler with an analysis of Rust's MIR to determine the types of all pointers, locate pointer cast locations, and generate corresponding annotations for the LLVM IR code that is lowered from the MIR. We extend LLVM with our new SafeFFI-Lib which consumes the pointer type annotation, conducts a pointer type analysis on LLVM IR level, inserts additional sanitizer checks at the boundary between safe and unsafe pointer types, and provides a simple API for existing sanitizers to elide checks for provably safe pointer types. A detailed description of the architecture is given in §6.

Example. We intuitively illustrate the effect of SafeFFI using the example in Figure 2, which contains Rust source code of a function `foo`, the corresponding LLVM code including sanitizer checks, and the same LLVM code after optimization with SafeFFI. In its normal operation, the sanitizer inserts `sanitizer_check()` for the pointer operands in every LLVM `load` or `store` instruction. By using information from Rust's type system, SafeFFI can deduce that pointers `a`, `b`, `c`, and `d` are all derived from a Rust reference `safe1`. The Rust type system is guaranteeing that all derived references only ever access memory within the bounds and lifetime of the object they are derived from. Therefore, if we can ensure that the requirements of Rust's type system (see §2.2) are met for the creation of the original reference, then all subsequent sanitizer checks are obsolete and we can elide them.

In line 6 of the Rust source code in Figure 2a, the `safe` pointer `safe1: &mut SomeStruct` (a mutable reference) is created by casting from the `raw` pointer which has been returned from calling the unsafe function `c_create()`. Later, in lines 9 and 10, two more `safe` pointers `a` and `b` are derived from `safe1` by taking the respective addresses of the fields defined in `SomeStruct`. Both are then dereferenced in line 11 to pass their pointee values to the function `do_some()`. Those dereference operations correspond to the `load` instructions in lines 10 and 12 of Figure 2b. One can see how the sanitizer inserts `sanitizer_check()` calls before each `load` instruction. Because Rust's static type system guarantees that `safe1` and its derived pointers `a` and `b` only ever access the memory object within the bounds of the `SomeStruct` object, those sanitizer checks can be elided. However, to be able to rely on the type system's soundness, we have to ensure that its requirements are upheld when casting `raw1` to `safe1` by inserting a sanitizer check. This is depicted by the blue arrows: SafeFFI removes the checks before the load instructions and instead inserts the `safeffi_ensure()` call in line 5 of Figure 2c.

The `safeffi_ensure()` function takes the `raw1` pointer as input as well as the size of the `SomeStruct` type (in this case `16`) and internally uses the existing check implementation provided by LLVM's sanitizers to validate that the memory object behind `raw1` is still allocated, has at least the expected size, and that the pointer has the correct provenance for this object. If the `safeffi_ensure()` check fails, we immediately abort the program, pointing the developer directly to the cast



```
fn foo(p: &i32) {

  let n: i32 = *p;
  let raw1: *mut SomeStruct =
    unsafe { c_create(n) };
  let safe1: &mut SomeStruct =
    unsafe { &mut *raw1 };

  let a: &mut i32 = &mut safe1.a;
  let b: &mut i32 = &mut safe1.b;
  do_some(*a, *b);




  loop {
    let c: &i32 = derive(a);
    let d: &i32 = derive(b);
    if *c + *d == 85 {
        break;
    }
  }
}
```

(a) Rust code.

```
define @foo(ptr %p) {
start:
 sanitizer_check(%p)
 %n = load i32, ptr %p
 %raw1 = call ptr @c_create(i32 %n)

 %a = getelementptr ptr %raw1, i64 8
 %b = getelementptr ptr %raw1, i64 12
 sanitizer_check(%a)
 %0 = load i32, ptr %a
 sanitizer_check(%b)
 %1 = load i32, ptr %b
 call @do_some(i32 %0, i32 %1)
 br label %bb3


bb3:
 %c = call ptr @derive(ptr %0)
 %d = call ptr @derive(ptr %1)
 sanitizer_check(%c)
 %_14 = load i32, ptr %c
 sanitizer_check(%d)
 %_15 = load i32, ptr %d
 %_13 = add i32 %_15, %_14
 %2 = icmp eq i32 %_13, 85
 br i1 %2, label %bb6, label %bb3

bb6:
 ret void
}
```

(b) LLVM IR with sanitizer checks.

```
define @foo(ptr %p safePtrArg(4)) {
start:
 %n = load i32, ptr %p
 %raw1 = call @c_create(i32 %n) !raw
 %raw1_safe = call safeffi_ensure(ptr raw1,
      i64 16) !safe
 %a = getelementptr ptr %raw1_safe, i64 8
 %b = getelementptr ptr %raw1_safe, i64 12
 %0 = load i32, ptr %a
 %1 = load i32, ptr %b
 call @do_some(i32 %0, i32 %1)
 br label %bb3


bb3:
 %c = call ptr @derive(ptr %0) !safe
 %d = call ptr @derive(ptr %1) !safe
 %_14 = load i32, ptr %c
 %_15 = load i32, ptr %d
 %_13 = add i32 %_15, %_14
 %2 = icmp eq i32 %_13, 85
 br i1 %2, label %bb6, label %bb3


bb6:
  ret void
}
```

(c) IR with sanitizer checks hoisted by SafeFFI.

Figure 2: A simplified example illustrating SafeFFI's annotations and optimizations. Blue arrows indicate hoisting of checks for locally-created `safe` pointers. For `safe` pointers received via parameters or call returns, orange arrows indicate elision of checks.

location. This is a specific advantage of SafeFFI: it can reveal a misuse of `unsafe` code or FFI code much earlier compared to normal sanitizer operation. Regular sanitizers only fail at the dereference location which might happen much later, e.g., in a subsequent function call, making the error harder to debug.

If the `safeffi_ensure()` succeeds then it returns a new LLVM value (`%raw1_safe` in the example) representing the casted `safe` version. Usually, the Rust compiler would only generate one LLVM variable to represent both the `raw` and the `safe` pointer as an optimization, because they have the same machine layout and carry the same value. SafeFFI allows us instead to differentiate `raw` and `safe` pointers on LLVM IR level and to elide sanitizer checks accordingly.

In the following section, we give a detailed explanation of our approach to determine in which cases sanitizer checks can be elided and when the boundary between `raw` and `safe` pointers needs validation at run time by inserting additional sanitizer checks.

5 Type-System-Guided Sanitizer Checks

Now we explain our concept in detail. We begin by motivating pointer casts as logical boundaries for memory safety enforcement (§5.1). We introduce our method for using function-local type information to hoist checks to cast locations while eliding sanitizer checks for safe pointer-like types with statically-sized pointees (§5.2) and discuss additional measures required for full temporal memory safety (§5.3). Finally, we show how to leverage the Rust type system to reason about memory safety across function calls without expensive whole-program analysis (§5.4). Note that this design includes full support for the presence of unsafe foreign code in mixed-language scenarios.

5.1 Memory Safety Enforcement Boundaries

Incorrect memory access through raw pointers can cause memory bugs even in safe Rust [30]. Raw pointer dereferences are dangerous because Rust does not enforce any guarantees for `raw` pointers. For safe pointer-like types, however, the Rust compiler provides the following memory safety guarantees (see §2.2):

- *Spatial safety*: every read or write through such a pointer is guaranteed to only access memory inside the bounds of its pointee type.
- *Temporal safety*: no safe pointer-like can exist outside the lifetime of its original pointed-to memory object.

SafeFFI is designed to elide sanitizer checks for safe pointer-like types with statically-sized pointees whose sizes are known at compile time, i.e., Rust references (`&T` and `&mut T`),

Box pointers (`Box<T>`), static arrays (`[T;N]`, and function pointers). For simplicity, we will refer to them as `safe` pointers, as opposed to `raw` pointers. There are multiple ways to create and derive `safe` pointers in Rust. All of them are checked by the Rust compiler, *except* for the cast from a `raw` pointer to a `safe` pointer. Casts from `raw` to `safe` pointers can only happen in `unsafe` Rust code, where the Rust compiler does not check that the `raw` pointer adheres to the requirements of the `safe` pointer type. Thus, a cast of an invalid `raw` pointer can undermine the memory safety guarantees of Rust's type system.

The key idea behind SafeFFI is that using sanitizer checks, we can dynamically guarantee validity for a `raw` pointer at the time of the cast and then continue to rely on the guarantees statically enforced by the Rust compiler for the remaining lifetime of the `safe` pointer after the cast. Hence, we consider these cast operations to form the boundary between memory safety enforcement by the sanitizer and the Rust compiler. Casts from `raw` to `safe` pointers are the central point of focus for SafeFFI, and in the following we show how to hoist the sanitizer checks for `safe` pointers by protecting this boundary with a strategically placed sanitizer check.

5.2 Local Type Analysis for Check Hoisting

The basic idea of SafeFFI's optimizations is to use function-local pointer type information provided by annotation of local variables, globals, and arguments to reduce the number of sanitizer checks for `safe` pointers. Usually, the sanitizer inserts checks at every instruction that dereferences a pointer. For `raw` pointers, we just keep all the original checks in place that the sanitizer inserted. For `safe` pointers, we hoist the checks from the dereference location up to the beginning of their scope, where they are created. Then, based on the rules enforced by Rust's type system, we can safely elide all subsequent checks.

We differentiate the following cases of how a `safe` pointer can be created in the current function's scope:

(a) Allocating a new stack object. On LLVM-IR level this corresponds to an `alloca` instruction which returns a pointer variable.
(b) Deriving a reference from a local stack object or from another safe Rust reference.
(c) Casting a `raw` pointer to a reference via a Reborrow operation (`safe_ptr: &T = &*raw_ptr`) or to a Box via `Box<T>::from_raw(raw_ptr)`. Although its syntax looks like a dereference-and-take-address operation, a Reborrow just creates a reference that points to the same memory object as pointer `a1`, without dereferencing.
(d) Loading a `safe` pointer from memory, e.g., as a field of another object that resides in memory.

In cases (a) and (b), Rust takes care of memory management through its lifetime and borrowing mechanics and no explicit `raw` pointers are involved, so we can elide all checks. In case (c), we have to ensure the validity of the `raw` pointer before we can safely cast it to a `safe` pointer and rely on the Rust compiler for memory safety. To check the validity of `safe` pointers casted from `raw` pointers, SafeFFI emits a call to `safeffi_ensure()`, a custom dynamic check function using sanitizer metadata to determine whether the object pointed to by the raw pointer is still alive and is at least of the size of the pointee type `T`. For case (d), we also need to insert a sanitizer check. Since `unsafe` Rust or foreign code can arbitrarily modify memory contents, pointers stored on either heap or stack might be corrupted, so we must check their validity when they are loaded into the current function scope.

As a result, we elide sanitizer checks at dereference locations for `safe` pointers; if the pointer is not safe by construction, we insert a sanitizer check at the creation site of the `safe` pointer, effectively combining and hoisting the checks. This is illustrated by the arrows in Figure 2. The run-time benefit of removing the sanitizer checks is most pronounced when checks can be hoisted out of loops.

Spatial Safety. The inserted sanitizer check enforces the size requirement of the `safe` pointer type at the cast site. Once the `safe` pointer is created, we can rely on the Rust type system to ensure that all subsequent accesses through the `safe` pointer in the current function are within the bounds of the type, as explained above. Thus, SafeFFI ensures spatial safety for the whole scope of the `safe` pointer (to the extent that the underlying sanitizer guarantees it).

Temporal Safety. For reasoning about temporal safety, we distinguish *Free-Before-Scope* vulnerabilities from *Free-During-Scope* vulnerabilities. So far, we combined the Rust type system with dynamic sanitizer checks to ensure that any `safe` pointer points to a memory object that is alive *at the start of the pointer's scope*. Thus, SafeFFI always reliably detects all temporal vulnerabilities where the memory object is deallocated *before* the start of the pointer's scope, i.e., Free-Before-Scope vulnerabilities. If the memory object is deallocated *during* the `safe` pointer's scope, e.g., through an alias pointer, this can cause a Free-During-Scope vulnerability. To also catch these, SafeFFI provides the option for further checks, as we explain in the next section.

5.3 Catching Free-During-Scope Violations

For Free-During-Scope violations, we need to check that the `safe` pointer remains valid until the end of its scope. SafeFFI provides the option to detect Free-During-Scope violations which allows developers to trade safety for run-time and compile-time performance.

In Rust, memory may be deallocated either by popping a stack frame at the end of a function scope or by calling a heap deallocation function. We reason about a `safe` pointer's local scope within the current function—we will extend our

Algorithm 1: Insert additional heap checks for Free-During-Scope violations.

Input: A Rust function *F* and the set *DeallocFns* of functions that may (transitively) lead to heap deallocation

```
1 foreach memInst ∈ MemoryInstructions(F) do
2     foreach callInst ∈ memInst.operands() do
3         if hasSafePointerOperand(memInst) then
4             if callInst.callee() ∈ DeallocFns then
5                 if IsReachable(memInst, callInst) then
6                     InsertCheckAt(memInst, callInst);
```

reasoning across function calls in §5.4. Because the `safe` pointer's scope is limited to the current function, this stack frame is guaranteed to not be deallocated within the entire scope. Thus, only heap deallocations remain for potential Free-During-Scope violations.

Heap deallocation requires calling `__rust_dealloc()` or `libc::free()`. Hence, a `safe` pointer can only become dangling within its scope in the current function if there is a call to one of those deallocation functions in between. To detect Free-During-Scope violations, SafeFFI inserts an additional `safeffi_ensure()` check for every dereference of the `safe` pointer that is reachable from a call to a potential deallocation function. Algorithm 1 for inserting those heap checks can be implemented efficiently within LLVM and is linear in the number of instructions in the analyzed function. To determine the set *DeallocFns* of functions that may transitively lead to a heap deallocation, we develop an efficient and sound analysis for constructing the call graph, which executes on-the-fly during compilation (see §6.4 for a detailed description).

In single-threaded applications, inserting additional `safeffi_ensure()` checks guarantees that each `safe` pointer remains valid throughout its scope, allowing SafeFFI to reliably detect Free-During-Scope violations. For Free-During-Scope violations, check hoisting is not thread-safe, however. A deallocation could occur concurrently between the check and the subsequent dereference, which could allow a Free-During-Scope violation to go undetected. Nevertheless, SafeFFI remains fully compatible with multi-threaded programs and is guaranteed to detect spatial and Free-Before-Scope violations to the extent that the underlying sanitizer does.

5.4 Interprocedural Memory Safety

Intraprocedurally, SafeFFI establishes a safety invariant for every `safe` pointer created in each function: *each* `safe` *pointer created in the current function is guaranteed to be safe to dereference for its whole scope in the current function.* To guarantee whole-program memory safety, we also need to ensure validity of `safe` pointers passed across function calls.

Pointer Arguments. SafeFFI generates annotations for the function signature during lowering from MIR to LLVM IR, emitting attributes for pointer parameters (e.g., see the `safePtrArg` attribute in line 1 of Figure 2c for the parameter `p: &i32`). When the currently analyzed Rust function is called from another Rust function, the type system guarantees that the argument types in the call and the function signature match, so functions with `safe` pointer arguments will only ever receive `safe` pointer values from the caller. Because of our established memory safety invariant, we know that `safe` pointers are indeed safe to dereference for the scope of the caller, which means they are also valid for the whole scope of the callee.

For Rust functions with external visibility, which can be called by foreign code, invalid data could be passed to a parameter of a `safe` pointer type, because linking foreign code requires only ABI compatibility and may ignore types. Therefore, SafeFFI inserts an additional `safeffi_ensure()` check in the prologue of functions with external visibility to ensure that the `safe` pointer is indeed valid. Hence, we can elide all original sanitizer checks for `safe` parameters.

Pointer Return Values. Similar to the previous case, we can rely on the Rust type system to guarantee that the return value of a function call is of the correct type. Because of our invariant, we can guarantee that a `safe` pointer returned from a function call is valid until the end of its scope in the callee. And if the pointer is valid at the end of the callee, then it is also valid at the return in the caller—with one exceptional case that needs special handling.

Each return instruction is also the deallocation of the callee's stack frame. If the `safe` pointer is derived from a `raw` pointer pointing to an object on that same stack frame, then this creates a stack Use-After-Return (UAR) violation because by the time the `safe` pointer is received in the caller function, the pointed-to memory object on the callee's stack frame is already deallocated. Figure 3 shows an example of such a stack temporal violation. The call to `derive()` returns a `safe` pointer that has been derived from a `raw` pointer pointing to an object on the stack frame. Because the developer did not choose the correct lifetime for the returned pointer in the `derive()` signature and foreign C code is involved, the Rust compiler has no chance at preventing this UAR violation. To catch such violations, SafeFFI inserts an additional `safeffi_ensure()` check after every function call that returns a `safe` pointer. Thus, we can now guarantee that the `safe` pointer is valid for the scope in the caller function.

6 SafeFFI's Implementation

In this section, we describe the architecture and implementation details of our approach. We implemented SafeFFI as modifications to the Rust compiler version 1.52.0 and LLVM

```c
// C code
void *c_derive(int *n) {
  int *p = n;
  ...
  return p;
}

// Rust code
fn derive(_: &'a i32) -> &'a i32 {
  let n = 42;
  // p points to n on the stack
  let p: *const i32 = unsafe { c_derive(&n as *const i32) };
  __safeffi_ensure(p, sizeof(i32));
  // at this point, *p is still valid, so cast check succeeds
  let p_safe: &i32 = unsafe { &*p };
  return p_safe;
  // n is deallocated here, so pointer p_safe is now dangling
}

fn foo() {
  ...
  let b: &i32 = derive(a);
  // additional check catches invalid pointer
  __safeffi_ensure(b, sizeof(i32));

}
```

Figure 3: Example of a stack temporal violation that SafeFFI catches by inserting an additional sanitizer check after the pointer returned to the caller function `foo()`.

version 12. The blue boxes in Figure 1 highlight how SafeFFI integrates into the Rust and LLVM toolchain.

First, rustc lowers Rust source code to the *Mid-Level Intermediate Representation (MIR)* which is where the Rust type system implements its static checks, also known as the *Borrow Checker*. While the MIR is then lowered to LLVM IR, SafeFFI attaches pointer type annotations and inserts sanitizer checks for pointer casts. The LLVM IR is processed by standard optimization and sanitizer passes and finally linked with the sanitizer runtime. In mixed-language applications, C code is compiled to LLVM IR and linked in the same way. Details on each component are described in the remainder of this section.

The goal for our architecture was to make integration with existing sanitizers as easy as possible by requiring only minimal changes to the sanitizer: (i) querying pointer types via the `is_safe_pointer(Value* ptr)` function provided by SafeFFI-Lib, and (ii) providing an implementation for calls to `safeffi_ensure(void* p, u64 size)`. These are simply the existing checks regulary enforced by the LLVM sanitizers ASan (`__asan_region_is_poisoned()`) and HWASan (`__hwasan_test_shadow()`).

6.1 Pointer Type Annotation in MIR

Our goal is to know the type of every pointer in LLVM IR, because for every pointer, the sanitizer can potentially request the type via the `isSafePtr()` API of SafeFFI-Lib. Thus we have to annotate local variables, function arguments, global variables, and constants. We implement this by associating LLVM metadata nodes (`MDNode`) with the corresponding generated LLVM instructions and LLVM `Attributes` for function parameter values.

For this we hook into the lowering process from MIR to LLVM IR. The lowering process is implemented in rustc via the visitor pattern: the `rustc_codegen_ssa` module visits MIR statements and calls the functions in `rustc_codegen_llvm` (the LLVM backend interface) to generate the corresponding LLVM IR. The challenge lies in the loose connection between MIR symbols and corresponding LLVM symbols because the lowering of an MIR symbol depends on the target ABI of its type. The Rust compiler differentiates between the following ABIs for any MIR type:[1]

- `Uninhabited`: a zero-sized type, that actually does not exist in memory. No LLVM code will be generated for it, so there is nothing to annotate.
- `Scalar`: a type that is represented by a single LLVM value. If this is a pointer type (also called *thin pointer*), then we annotate it accordingly. This is the case for references, arrays, and raw pointers; but it is also the case for all algebraic data types that are represented by a single pointer value in LLVM IR, e.g. the `Box<T>` or any custom arbitrarily nested struct that only has one field and that field is a pointer. A special case of this is Rust's `Union` type. If it has a `Scalar` ABI, then we can only annotate it as `safe` if all of its fields are `safe` pointers, otherwise the pointer value could be manipulated in an unsafe way through another type representation like an integer or a raw pointer. We implemented a recursive type analysis to detect such cases.
- `ScalarPair`: a type that is represented by two LLVM scalar values, mostly dynamically-sized types (e.g., slices) which are lowered to fat pointers containing a data pointer and a size value. As we cannot reason about the dynamic size of such types, SafeFFI annotates them as `raw`. We leave it to future work to find further optimizations to elide checks for such types. However, in the case of Trait objects (`&dyn Trait`), the second value is a vtable pointer. Because it is lowered to a LLVM pointer, we need to annotate it, too. Because the vtable pointer is not user-manipulated but generated and managed by the compiler, we always annotate it as `safe`.
- `Vector`: those are only used for LLVM's SIMD vector types, which are not relevant for our approach.
- `Aggregate`: a type that is represented by custom LLVM structs. Those types are not pointers in MIR; however, if a local variable of an `Aggregate` type is allocated on the stack using a LLVM `alloca` instruction, then a LLVM pointer is created to represent this variable. We annotate these with a `NOPTR` tag and treat them as safe because they cannot be user-manipulated.

[1] There is no official specification of Rust, thus we have to take the Rust compiler's behavior as reference: https://github.com/rust-lang/rust/blob/1.52.0/compiler/rustc_target/src/abi/mod.rs#L847

The ABI of the type and the calling convention also dictate how a value is passed or returned as function argument in a call. Fortunately, the Rust compiler already annotates parameters of the function signature if they are safe to dereference using LLVM's `dereferenceable(<size>)` attribute from which we inherit our annotations.

6.2 Pointer Type Analysis on LLVM IR

A further challenge is that the ABI of MIR types also controls how values are loaded from memory. For example, accessing a MIR field (`let a = safe1.a`) can generate different sequences of LLVM instructions like `GEP, BITCAST, LOAD`. Since the Rust compiler does not track all LLVM instructions generated for a MIR value, SafeFFI annotates only the final LLVM value representing the loaded MIR operand, lacking pointer type annotations for intermediate LLVM pointer values. Thus, in SafeFFI-Lib, we implement an intra-procedural pointer type analysis that forwards the types (`safe`, `raw`, and `NOPTR`) throughout each function. This analysis initializes a map of pointer types using the annotations inserted by the Rust compiler for alloca Instructions, Call/Invoke instructions, function parameters, and global variables. Then, for every remaining LLVM instruction that produces a pointer value (BITCAST, GEP, INTTOPTR, PHI), SafeFFI forwards the type from the operands of the instruction to the resulting pointer based on the semantics of the instruction. The GetElementPointer (GEP) instruction, which computes a derived pointer by offsetting a base pointer, is handled more carefully: if the base pointer is `raw`, the result is always marked as `raw`. If the base pointer is `safe` and the offset arguments of the GEP instruction are constant, we statically compute the resulting offset via LLVM's `GEP::accumulateConstantOffset()` and check whether it remains within the bounds of the original allocation. If so, SafeFFI marks the result as `safe` and otherwise conservatively downgrades it to `raw`.

As a result, *every* pointer value in LLVM IR is classified as `safe`, `raw`, or `NOPTR`, enabling the sanitizer to reliably query pointer types via the `isSafePtr()` API.

6.3 Pointer Cast Detection in MIR

The goal of this component is to detect casts from `raw` pointers to `safe` pointers in MIR. Our definition of `safe` pointers distinguishes 3 types of pointers: References, Boxes, and static arrays. Raw pointers cannot be casted to static arrays (only to references to static arrays), thus, we only have to detect the following two cases:

1. *Casting a raw pointer to a Box*. This always has to happen through a call to `Box::from_raw(p)` which is an explicit statement in MIR.
2. *Casting a raw pointer to a reference.* This too is always an explicit statement in MIR, because Rust does not allow implicit coercion in this case [8]. In MIR this pattern looks like `q = &*p`, where `&` denotes the creation of a reference and `*` denotes a dereference operation. Because the dereferenced `p` can be any kind of Rust pointer, we have to check if `p` actually has the raw pointer type to confirm this is a cast.

With both cast types, `p` can be a projection, e.g., the access of a field of a (nested) struct (`&*a.b.c`) or an array (`&*x[y][z]`). SafeFFI determines whether the inner-most element is a raw pointer by iterating over the MIR projections until we find the type of the accessed element.

We implement the pointer cast detection by hooking into the MIR visitor for `rvalues` (right-hand-side values) of MIR `Assign` statements. During lowering of an `rvalue`, the Rust compiler generates a series of LLVM IR instructions, the last of which producing an LLVM Value that corresponds to the MIR `rvalue`. For each `rvalue`, rustc keeps a mapping from the MIR `rvalue` to the corresponding LLVM value. This is where SafeFFI inserts a sanitizer check for pointer casts by extending the series of generated LLVM instructions with a call to the `safeffi_ensure()` function.

We show this effect in Figure 2. The `raw` pointer `raw1` is lowered by generating a call to `c_create()` in Figure 2b line 5 and its returned LLVM value `%raw1` is mapped for the `rvalue`. For lowering the pointer cast `safe1 = unsafe &mut *raw1` the Rust compiler usually just maps the `rvalue` to the same value as the `raw` pointer because there is no difference in the LLVM representation of a `raw` pointer and a `safe` pointer. Thus, in lines 7 and 8, the LLVM value `%raw1` is used to access the fields of the `safe1` reference. One can see how this is changed by SafeFFI in the Figure 2c lines 5-7. Our rustc modifications insert the `safeffi_ensure()` call and generate a new LLVM value `%raw1_safe`. SafeFFI then replaces the `rvalue` mapping for cast operation, now mapping to `%raw1_safe`. Thus, the subsequent accesses of the fields behind the `safe` pointer `safe1` now use the LLVM value `%raw1_safe`.

Generating a new LLVM value for the casted pointer has the advantage that we can now differentiate between the `raw` pointer and the `safe` pointer on LLVM IR level and separately reason about their safety guarantees at every subsequent location of use in LLVM IR. Moreover, our experiments showed that implementing cast detection on MIR level is the most reliable way to detect casts and insert sanitizer checks consistently. Other attempts to detect casts by finding differences in pointer type annotations at LLVM IR level have shown to be unstable because type annotations are transported via LLVM metadata nodes which can get lost or moved around during optimization passes in LLVM.

6.4 Callgraph-Based Deallocation Checking

As mentioned in §5.3, detecting Free-During-Scope temporal vulnerabilities requires checking that a dereferenced pointer has not become dangling since its creation.

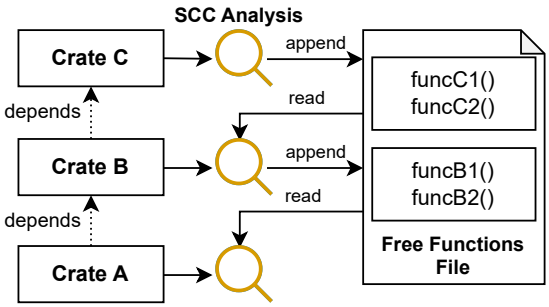


Figure 4: The NoFree SCC Analysis is performed on all dependencies to determine possibly heap-deallocating functions.

To address this, we implemented a call graph-based heap deallocation analysis in LLVM to determine whether a given function may perform a deallocation. We adopt LLVM's default call graph analysis, which resolves static function calls, and traverse the call graph bottom-up: functions without call instructions are visited first. The analysis is implemented as a LLVM SCC (strongly connected components) pass as the call graph might contain cycles. A function is annotated as *nofree* if it does *not* contain any calls to: (i) known deallocation functions (e.g., `free()` or `__rustc_dealloc()`), (ii) functions without a nofree annotation, or (iii) unknown callees. If any function within an SCC cannot proven to be nofree, then the entire SCC treated as potentially deallocating.

Another challenge is that deallocations can happen outside the current compilation unit, e.g., in a Rust dependency or external C library, and thus are not visible to the current compilation process. To address this, we serialize the nofree annotations to a persistent file after the analysis has finished for the current compilation unit. Subsequent compilation processes read in the serialized annotations and restore them before running the analysis. This compositional cross-crate analysis is illustrated in Figure 4. To support C/C++ dependencies, we provide build flags for clang to include our analysis. When C dependencies are dynamically linked or precompiled, we conservatively assume all external C functions may perform deallocations.

7 Evaluation

We implemented SafeFFI on Rust nightly-2021-02-22, which corresponds to Rust compiler version 1.52.0 and LLVM version 12. We integrated SafeFFI into two popular and well-maintained sanitizers for LLVM, ASan and HWASan. After introducing our methodology (§7.1), we present in this section our evaluation of SafeFFI answering the following research questions:

RQ1 How does SafeFFI affect the detection capabilities of the sanitizer (§7.2)?

RQ2 How many sanitizer checks can SafeFFI reduce in sanitized programs (§7.3)?

RQ3 How much can SafeFFI reduce the run-time of sanitized programs (§7.3)?

RQ4 Is SafeFFI's implementation robust in the presence of FFI interactions in MLAs (§7.4)?

RQ5 How much compile-time overhead does SafeFFI incur (§7.5)?

7.1 Methodology

We conducted experiments with SafeFFI on ASan on a x86_64 system with an AMD EPYC 9645 with 384 cores and 1.5 TB RAM running an Ubuntu 20.04 docker container. For comparison with related work, we evaluate the same crates (Rust's term for packages or libraries) as RustSan [5] and ERASan [30] and choose a common version that all three tools can compile with their respective toolchains. The crates *Rocket* and *RustPython* are missing because they contain a compilation error in the version compatible with Rust 1.52.0. Although we compare SafeFFI to ERASan and RustSan as closely as possible, note that there are differences in the baseline toolchain (Rust and LLVMcompiler version) and build process, which we account for in our measurement and discussion of the results.

Because it is essential to the soundness of RustSan and ERASan that Rust's standard library is analyzed, we rebuild it for all benchmarks. RustSan and ERASan require the whole program for identifying points-to sets containing `raw` pointers, so missing points-to information from the standard library would lead to misclassifying pointers as `safe`, thus, leading to false negatives. Therefore, both tools require the standard library to be rebuilt together with each application binary to generate complete points-to sets. SafeFFI also instruments the standard library to detect invalid casts from `raw` to `safe` pointers, because we cannot assume the standard library to be memory safe. In contrast to RustSan and ERASan, however, SafeFFI only relies on local reasoning; therefore, the standard library could be analyzed and instrumented separately only once. Still, for evaluation consistency and ensuring a fair comparison, we always rebuild the standard library, with each benchmark and for all tools and configurations. Note that this matches RustSan's methodology [5], whereas the published evaluation of ERASan omits the standard library from their evaluation [30], leading to differences in their measurements of ERASan versus our own. We make all datasets, scripts, and tools used in our evaluation available for reproduction.

7.2 Correctness

We answer **RQ1** by testing SafeFFI on known real-world memory safety vulnerabilities. We further include RustSan [5] and ERASan [30] as related work and unmodified (vanilla) HWASan and ASan as baselines. We assembled the dataset

Table 1: Vulnerability Detection

ID	INTERCEPTOR	HWASAN	SafeFFI+HWASAN	ASAN	SafeFFI+ASAN	RustSan	ERASan
CVE-2017-1000430	✓	●	●	●	●	●	●
CVE-2018-20991	✗	●	●	●	●	●	●
CVE-2018-21000	✓	●	●	●	●	●	●
CVE-2019-15551	✓	●	●†	●	●†	●	●
CVE-2019-16140	✓	●	●	●	●	●	●
CVE-2019-16882	✓	●	●	●	●	●	●
CVE-2019-25009	✓	●	●	●	●	●	●
CVE-2020-25574	✓	●	●†	●	●†	◐	●
CVE-2020-25791	✓	●	●	○	○	○	○
CVE-2020-25792	✓	●	●	○	○	○	○
CVE-2020-25795	✓	●	●†	●	●†	●	●
CVE-2020-35858	✗	●	●	●	●	●	●
CVE-2020-35860	✓	●	●	●	●	●	●
CVE-2020-35861	✓	●	●	●	●	●	●
CVE-2020-35891	✗	●	●†	●	●†	●	○
CVE-2020-35892	✓	●	●†	●	●†	●	●
CVE-2020-35893	✓	●	●†	●	●†	●	●
CVE-2020-35906	✓	●	●	●	●	●	●
CVE-2020-36434	✗	●	●	●	●	◐	○
CVE-2020-36464	✗	●	●†	●	●†	●	●
CVE-2020-36465	✓	●	●	●	●	●	●
CVE-2021-25900	✓	●	●	●	●	●	●
CVE-2021-26954	✓	●	●†	●	●†	●	○
CVE-2021-28028	✓	●	●†	●	●†	●	○
CVE-2021-28031	✓	●	●	●	●	●	●
CVE-2021-29933	✓	●	●†	●	●†	●	○
CVE-2021-30455	✓	●	●†	●	●†	●	●
CVE-2021-30457	✗	●	●†	●	●†	●	◐
CVE-2021-45694	✗	●	●†	●	●†	●	○
CVE-2021-45713	✗	●	●†	●	●	◐	○
CVE-2021-45720	✗	●	●†	●	●†	●	○
RUSTSEC-2020-0061	✗	○	◐	○	○	○	○
RUSTSEC-2020-0091	✗	●	●†	○	○	○	○
RUSTSEC-2020-0097	✗	●	●†	●	●†	◐	○
RUSTSEC-2020-0167	✓	●	●	●	●	●	●
RUSTSEC-2021-0003	✓	●	●	●	●	●	●
RUSTSEC-2021-0031	✗	●	●†	●	●	●	●
RUSTSEC-2021-0033	✓	●	●†	●	●†	●	●
RUSTSEC-2021-0039	✓	●	●†	●	●†	●	●
RUSTSEC-2021-0047	✓	●	●†	●	●†	●	◐
RUSTSEC-2021-0048	✓	●	●	●	●	●	●
RUSTSEC-2021-0049	✓	●	●	●	●	●	○
RUSTSEC-2021-0053	✓	●	●	●	●	●	●
RUSTSEC-2022-0070	✓	●	●	●	●	●	○
RUSTSEC-2022-0078	✓	●	●	●	●	●	●
RUSTSEC-2023-0005	✓	●	●†	●	●†	●	●

● Detected ◐ Different Error ○ Not Detected †SafeFFI-specific Check
INTERCEPTOR: ✓= detected by interceptor ✗: detected by elidable check

of known vulnerabilities by merging and deduplicating the datasets used by ERASan and RustSan. Note that the authors of RustSan have not provided their dataset, thus we manually reconstructed it based on the RustSec advisories.

Table 1 shows the results of our evaluation on the dataset of known vulnerabilities. ◐ indicates that the vulnerability was detected by the sanitizer, but with a different error message than the baseline. For example, CVE-2021-30457 is both a use-after-free and *later* a double-free. The former is detected by instrumentation and susceptible to check elision while the latter is caught by the sanitizer's interceptor of `free()`. Segmentation faults and other signals are classified as not detected (○), as they imply that a memory error was not caught by the sanitizer before leading to the crash. SafeFFI reports some vulnerabilities *earlier*, at the root cause, instead of at the access location, due to the hoisting of checks (cf. §5.2); these cases are classified as detected (●). RUSTSEC-2020-0061 is a NULL pointer dereference that is not detected by any of the approaches. This anomaly is caused by SafeFFI changing the memory layout and incidentally allowing HWASan to detect the vulnerability; this is not a conceptual advantage. Some vulnerabilities in the dataset are non-deterministic which makes it necessary to run each compiled test binary multiple times to observe the vulnerability reliably, especially for HWASan because its probabilistic detection mechanism adds further non-determinism on top, even for deterministic vulnerabilities. If at least one run triggered the vulnerability, we classified it as detected.

The results show that SafeFFI catches all vulnerabilities that ASan and HWASan detect, respectively. Due to hoisting checks to the memory safety boundary at the location of pointer casts, SafeFFI is able to report 21 vulnerabilities earlier than the underlying sanitizer, increasing debuggability. Additionally, we have not encountered any false positives during the evaluation of the other research questions as shown in the following sections, which further indicates that SafeFFI does not introduce false positives.

Limitations. While SafeFFI successfully detects all known vulnerabilities in our dataset, we acknowledge the following limitations. SafeFFI inserts additional checks for Free-During-Scope vulnerabilities for every `safe` pointer between a call to a potential deallocation function (see §5.3) and a subsequent use of the `safe` pointer. In single-threaded environments, SafeFFI guarantees that the object is still allocated at the point of use. In multi-threaded programs, however, there is a potential risk of missing Free-During-Scope vulnerabilities if a concurrent deallocation of the pointed-to object occurs in another thread *between* SafeFFI's additional check and a subsequent pointer dereference. This is a conceptual limitation of our solution to Free-During-Scope vulnerabilities and invites future work on multi-threaded settings.

Furthermore, SafeFFI currently has limited support for inline assembly and Rust's *transmute* operations. Programs with both features can be compiled and run with SafeFFI, but may cause false negatives (missed bugs). Inline assembly is completely hidden from Rust type system checks and sanitizer instrumentation and can arbitrarily manipulate pointers and memory. Rust's `transmute()` and `transmute_copy()` functions allow for unchecked reinterpretation of bits, effectively casting between any two types of the same size, including `raw` and `safe` pointers. We expect to lift this technical limitation in the near future by detecting transmutes resulting in `safe` pointers and inserting the necessary checks.

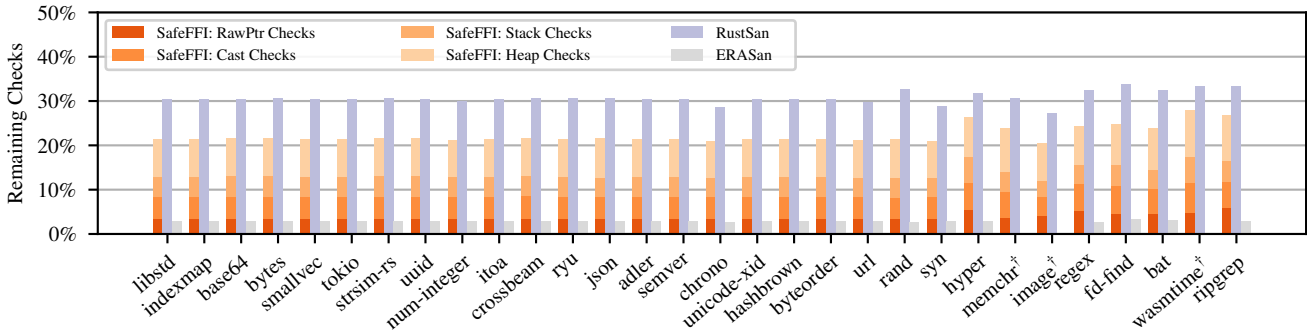


Figure 5: Sanitizer checks remaining after elision by SafeFFI, ERASan, and RustSan, relative to original sanitizer checks for individual crates. For SafeFFI, checks are subdivided into remaining `raw` pointer checks and added cast, stack and heap checks. † denotes crates that could not be compiled with ERASan.

Comparison. RustSan also detects all known vulnerabilities in our dataset that ASan detects. In contrast, ERASan performs significantly worse than SafeFFI and RustSan. We investigated further and discovered that ERASan's implementation does not properly annotate all unsafe pointers and therefore removes checks that would have been necessary to detect the vulnerabilities. Yet, some vulnerabilities are detected because ERASan keeps ASan's interceptors in place, e.g., for `free()` and `memcpy()`. We reached out to the authors of ERASan to debug our findings but have not received a response.[2] Regarding multi-threaded programs, note that ERASan's default mode also only ensures temporal safety for single-threaded programs [30].

7.3 Effectiveness

We now provide empirical evidence that SafeFFI is effective in reducing the number of sanitizer checks (**RQ2**) and, consequently, the run-time overhead of the sanitizer (**RQ3**). We further compare SafeFFI against RustSan and ERASan.

Elided Checks. Figure 5 shows SafeFFI's capability to elide ASan checks for x86_64 targets in comparison to other approaches, measured by us. The y-axis shows the remaining checks, i.e., lower is better. Note that the remaining checks of crates also include checks of all their dependencies. The three crates marked with † could not be compiled with ERASan due to an assertion failure in their analysis.[3] The amount of checks is measured at LLVM IR level.

SafeFFI (orange bars) retains on average 3.77% of the ASan checks because they vet raw pointers. Checks at cast sites (cf. §5.1) contribute an additional 5.17%. Stack checks (cf. §5.4) contribute another 4.58%, while the heap checks for Free-During-Scope vulnerabilities (cf. §5.3) contribute 8.67%. Overall, 22.22% of checks remain on average.

With RustSan (purple bars) on average 30.70% of the ASan checks remain in the program. SafeFFI consistently beats RustSan across all benchmarks. ERASan (gray bars) elides consistently more than 96% of all checks on the benchmarks, more than both SafeFFI and RustSan. However, as discussed in §7.2, ERASan introduces false negatives due to unsound removal of necessary checks, likely due to an implementation bug. The high elision rates of ERASan are therefore not comparable to the results of SafeFFI and RustSan.

For smaller crates, the number of remaining checks is dominated by the checks residing in the standard library. Excluding the standard library, SafeFFI retains 18.69% on average, with individual crates ranging from 5.89% (num-integer) to 42.75% (indexmap). Despite variations between individual crates, we observe that the average elision rate over all crates matches the elision rate including the standard library.

Run-time Performance. Not all checks are equally important for the run time, checks on hot paths through the program disproportionately impact the run time [47]. Therefore, we also evaluate the run time on several benchmarks. Figure 6 shows the run-time overhead of ASan, SafeFFI and related work on the benchmarks. Note due to the different build process and toolchains, we include ASan three times, once for each toolchain. Throughout the following section, ASan refers to the specific version on top of which SafeFFI is built. Additionally, we include an ASan Base configuration for each ASan version, which only includes the overhead of ASan's metadata maintenance and interceptors, but no instrumentation around loads and stores; i.e., it represents the run time with a perfect 100% elision rate and therefore marks the theoretical upper bound for check elision approaches.

ASan's base run-time overhead is 2.10× on average (median: 2.22×) compared to an uninstrumented binary. ASan's instrumentation increases the average run-time overhead to 2.71× (median: 2.78×). SafeFFI reduces the overhead to 2.44× on average (median: 2.45×). For crates like regex and num-integer, SafeFFI cannot improve the runtime. One rea-

[2] https://github.com/S2-Lab/ERASan/issues/4
[3] https://github.com/S2-Lab/ERASan/issues/5

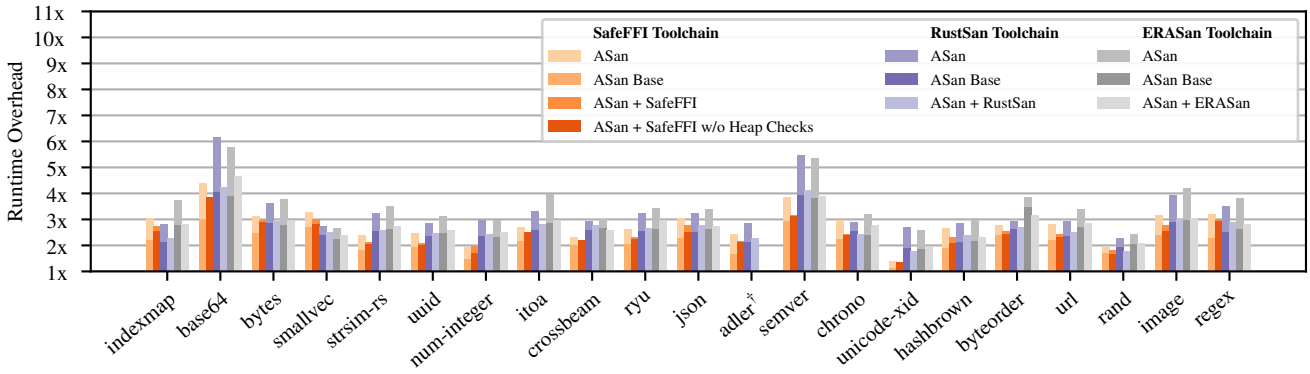


Figure 6: Run-time overhead of SafeFFI, RustSan, and ERASan relative to the baseline run-time without sanitizer. Each approach is evaluated on their respective toolchain and depicted next to their respective version of ASan, and ASan Base (all elidable checks disabled) as lower bound for check-elision approaches. SafeFFI without Heap Checks shows the additive run-time overhead incurred by our additional Free-During-Scope checks (cf. §5.3). † denotes crates that could not be compiled with ERASan.

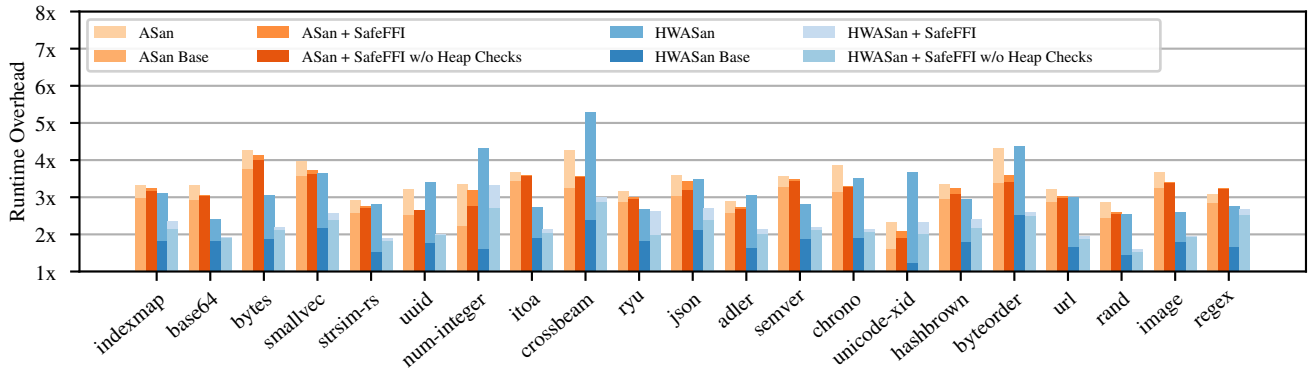


Figure 7: Comparing the run-time overhead of SafeFFI with HWASan vs ASan as underlying sanitizer on ARM64. We show the overhead of (HW)ASan Base (all elidable checks disabled) as lower bound for check-elision approaches. SafeFFI without Heap Checks shows the additive run-time overhead incurred by our additional Free-During-Scope checks (cf. §5.3).

son for this discrepancy, besides hot paths, is that cast checks inserted by SafeFFI validate ASan's shadow memory for the *entire size* of the pointed-to object, while the elided checks only validate shadow memory for the actual *access size* which might be much smaller. This makes cast checks potentially more expensive than the elided checks and is an interesting research avenue for further optimizations. In contrast, our heap checks for Free-During-Scope violations are less expensive, since SafeFFI only needs to check the first byte of the allocated object to verify that the entire object is still allocated. Therefore, our heap checks have less impact on the run time, as illustrated by the small difference between SafeFFI and SafeFFI w/o Heap Checks in Figure 6, even though they make up 39.00% of all remaining checks.

The overhead of ASan on RustSan's toolchain is 3.22× on average (median: 2.94×) compared to an uninstrumented binary, which is considerably higher than the ASan overhead measured with our toolchain and what the original ASan authors report [41]. We measured even higher numbers for ASan on ERASan's toolchain, with an average run-time overhead of 3.48× (median: 3.42×). The ASan base overheads for RustSan and ERASan are 2.52× and 2.65× on average (medians: 2.52× and 2.62×), respectively. RustSan reduces the run time down to 2.63× on average (median: 2.58×). ERASan reduces the overhead to 2.80× on average (median: 2.81×).

To compare the approaches, we consider the difference between vanilla ASan and its base overhead (ASan Base) which marks the lower bound for check elision approaches. We call this difference the *instrumentation overhead* and compare how much each approach can reduce the instrumentation overhead. ERASan gets closest to the lower bound of the instrumentation overhead (avg. 17.20%) but we stress again that ERASan's high elision rates and therefore run-time performance come at the cost of false negatives, thus the results

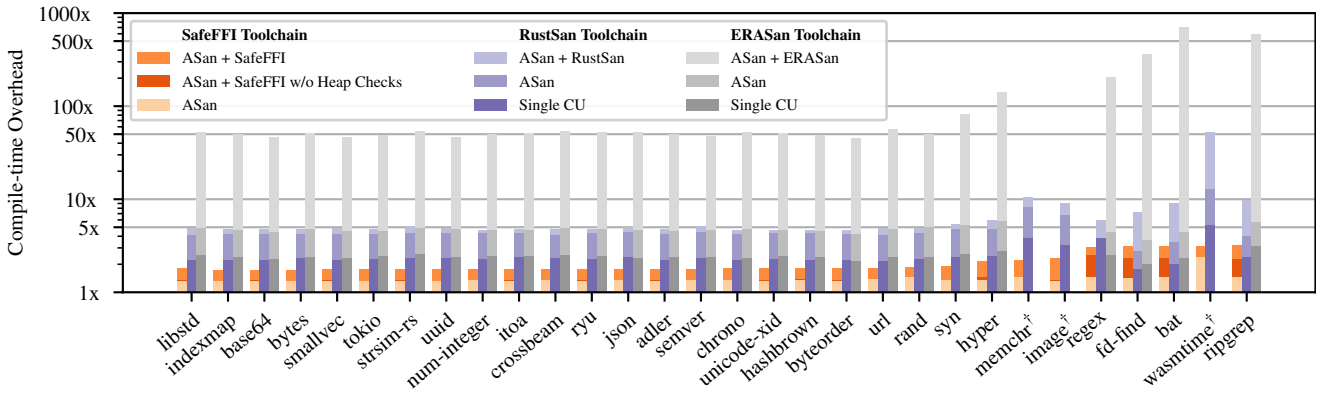


Figure 8: Compilation-time overhead of SafeFFI, RustSan, ERASan. Each approach is evaluated on their respective toolchain and compared to baseline compilation-time without sanitizer. We show each approach's overhead on top of its corresponding ASan. SafeFFI without Heap Checks is measured show the additive compilation-overhead incurred by our call graph generation method (cf. §6.4). For RustSan and ERASan, we also show the overhead of their modified single-compilation-unit build process (Single CU). Note that the scale is logarithmic to accommodate large overheads of the related work. † denotes crates that could not be compiled with ERASan.

are not directly comparable. RustSan reduces the instrumentation overhead to 19.86%, performing better than SafeFFI with 53.04%. We see two explanations for this behavior. First, the cast checks inserted by SafeFFI can be more expensive than the elided checks as explained above. Second, RustSan not only elides checks but in its implementation we discovered that it also suppresses instrumentation for the whole function if none of the pointers within the function aliases with a `raw` pointer. This optimization leads, e.g., to omitting redzone poisoning which accounts for a significant portion of ASan's overhead. Hence, theoretically, RustSan could be even faster than the ASan base overhead. This optimization is only possible because of the whole-program analysis of RustSan, which comes at the cost of impractical compile-time overheads.

ARM and HWASan. We also evaluate run-time overhead on the AARCH64 architecture. We ran benchmarks on a Mac Studio M2 Ultra with 24 ARMv8 cores and 192 GB RAM running an Ubuntu 20.04 docker container on ASAHI Linux 6.12.0. Figure 7 shows the results. The color-coding for SafeFFI on top of ASan is the same as in Figure 6. In Figure 7 we also include SafeFFI on top of HWASan and HWASan in blue color shades. We observe that on ARM SafeFFI reduces the overhead of ASan from 3.40× to 3.16× on average (medians: 3.35× to 3.24×). Comparing the vanilla sanitizers and their base overheads, we see that ASan benefits less from check elision than HWASan because ASan's base overhead is not dominated by load and store checks but mostly by metadata maintanence, especially redzone (un-)poisoning. In HWASan, load and store checks have greater impact on the run time. Thus, applying SafeFFI on top of HWASan significantly reduces its overhead from 3.18× to 2.29× on average (medians: 3.06× to 2.20×). Because RustSan's run-time overhead reduction is heavily influenced by their optimization of redzones, we conjecture that applying their approach on top of HWASan would not be as performant as SafeFFI.

7.4 Robustness in MLA Scenarios

Our experiments on correctness (§7.2) and effectiveness (§7.3) contain real-world MLAs in the benchmark sets, e.g., bat depending on libgit2, rusqlite (CVE-2021-45713) on libsqlite3, or xcb (RUSTSEC-2020-0097) on libxcb. Since we did not encounter FFI-related compilation issues nor false positives/negatives during execution, this is evidence that SafeFFI works as designed in MLA scenarios. To be confident in the robustness of SafeFFI, we created a systematic set of minimal test cases that covers all common FFI interactions between C and Rust that we could conceive of. It covers the following dimensions:

1. Allocation: Global, C stack/heap, Rust stack/heap;
2. Deallocation: Global, C stack/heap, Rust stack/heap;
3. Pointer invalidation: pointer arithmetic, deallocation, invalid pointer crafting, or no invalidation (benign test);
4. Control flow permutation: interleaving of pointer casts, invalidations, and dereferences;

We evaluated SafeFFI on all meaningful combinations of these dimensions leading to 45 distinct test cases (35 with vulnerabilities, 10 without). All tests are included in the artifact. SafeFFI is robust against all of those cases, meaning that it does not raise any false positives or false negatives.

7.5 Compile-Time Performance

Figure 8 shows the compile-time overhead of ASan, SafeFFI, ERASan and RustSan, answering **RQ5**. Note that overheads

are computed respective to the toolchain of each tool. All overheads are compared to baseline runs in a default build process with multiple compilation units. Since RustSan and ERASan both require a single compilation unit, we also measured the overhead incurred by their modified build process which consistently lies between 2.28× and 2.39× for smaller crates. For larger crates the overhead of single compilation units is up to 5.30×. SafeFFI is designed to work with the default multi-compilation-unit build process and does not incur this overhead.

For the SafeFFI toolchain, ASan induces a compile-time overhead of 1.38× on average (median: 1.34×) which SafeFFI increases to a reasonable 2.01× on average. The maximum compilation overhead is 3.25× on ripgrep. We see that the our call graph analysis (cf. §6.4) for inserting Free-During-Scope heap checks only adds a moderate overhead compared to SafeFFI without Free-During-Scope checks which lies at 1.49× on average, in a similar range as vanilla ASan. RustSan adds a compile-time overhead of avg. 5.91× (median: 4.87×), with outliers reaching an overhead of up to 53.21×, because it performs a whole-program points-to analysis to identify safe pointers that alias raw pointers, based on SVF [44]. ERASan's approach shares the high-level idea and the dependency on SVF with RustSan and therefore also incurs high compile-time overhead. However, with 73.05× on average (median: 51.69×) and outliers up to 701.83×, ERASan shows worse compile-time overheads which might be an effect of the implementation of their analysis. Regarding memory consumption during compilation, SafeFFI requires on average 1.04× (max. 1.07×) while RustSan and ERASan require an average of 5.92× (max. 54.91×) and 12.70× (max. 153.37×), respectively.

To conclude, SafeFFI outperforms the state of the art by a large margin and thus enables adoption of the proposed techniques even for larger software projects.

8 Related Work

Run-time Check Reduction for Rust. Rust for Morello [12] executes Rust programs on the CHERI architecture [48] for hardware-enforced memory safety (not commercially available). They elide software bounds checks emitted by the Rust compiler, but find little benefit due to compiler optimizations.

RustSan [5] and ERASan [30] are two recent approaches that, akin to SafeFFI, aim to reduce the overhead of ASan in Rust-only programs by eliding checks for pointers already proven to be safe by the Rust compiler. The approach of both tools is similar: ① annotate `raw` pointer types during lowering to LLVM IR, ② perform a whole-program points-to analysis using SVF [44] to identify pointers that alias with `raw` pointers and ③ only retain ASan checks for pointers that may alias `raw` pointers. Both tools perform *optimistic* static analysis and decide elision based on whether a points-to set contains a `raw` pointer, i.e., missing `raw` pointer type annotations or imprecisions in the alias analysis may lead to vulnerabilities going undetected (false negatives). Consequently, both tools require the entire LLVM IR of a program in a single compilation unit to uphold the security guarantees of ASan. In practice, this means that the LLVM IR of all dependencies, need to be merged together with the application code into a large monolithic compilation unit, analyzed, instrumented, optimized, assembled and linked for *every compilation run*. In particular, this requires the build process to always rebuild and include Rust's standard library into the analysis to soundly compute points-to sets which leads to significant compilation overhead, as shown in §7.5.

In contrast, SafeFFI is able to elide checks with efficient local reasoning and a compositional analysis for Free-During-Scope checks. Consequently, SafeFFI seamlessly integrates into the standard multi-compilation-unit build process and only introduces moderate compile-time overhead, enabling adoption even for large projects. We designed SafeFFI to be *conservative*, i.e., in case the type of a pointer is unknown to our analysis, SafeFFI treats it as `raw` and retains the sanitizer checks, enabling SafeFFI to be used for MLAs. Our concept of hoisting checks to locations of casts and other `safe` pointer creations also leads to superior debuggability because SafeFFI fails earlier and points developers directly to a violation of the Rust's requirements for `safe` pointers at the boundary to `unsafe` or foreign code.

General Sanitizer Optimizations. Prior work on C/C++ includes ASAP [47] which removes checks on hot paths, trading security for performance. Moll et al. [33], hoist bounds checks in loops. ASan-- [50] uses lightweight static analysis and SanRazor [49] a combination of static and dynamic analysis to reduce redundant checks. These are complementary to SafeFFI and could be applied to raw pointers in Rust or C/C++ parts of MLAs.

Run-time Isolation for Rust. Several proposed approaches isolate safe Rust from unsafe code, e.g., Sandcrust [21], Fidelius Charm [1], XRust [26], relying on developer annotations or hardware/OS support for process isolation. TRust [3], PRKUSafe [18] and Gülmez et al. [11] instead use Intel's Memory Protection Keys (MPK) to separate safe and unsafe objects in different memory regions. Galeed [39] and Omniglot [40] explicitly address cross-language attacks [28] by isolating Rust from C code, relying on hardware features for in-process isolation. As opposed to those isolation approaches that only inhibit the spread of memory vulnerabilities from unsafe code to safe Rust code, SafeFFI prevents memory vulnerabilities in the first place and therefore enables better debugging, without requiring source or OS changes.

Static and Dynamic Analyzers for Memory Safety in Rust. Tools like MIRChecker [24], Rudra [2], Yuga [36], and SafeDrop [6] use static analysis to detect memory bugs in Rust, while FFIChecker [23] and CRust [15] focus on FFI safety. These approaches suffer from false positives and excessive compile-time overhead, whereas SafeFFI is a dynamic analysis based on sanitizers resulting in high precision at the cost of run-time overheads which SafeFFI significantly reduces for Rust code. Miri [17] is also a dynamic analysis tool which performs an interpreter-based execution of Rust programs checking for undefined behavior. It implements precise tracking of pointer provenance which provides more precise memory safety than HWASan. However, it is not feasible to use Miri for MLAs since it only interprets Rust's MIR. MiriLLI [27] approaches this limitation by bridging Miri's interpreter with *lli*, an interpreter for LLVM IR. Since interpreters are orders of magnitude slower than native execution, Miri and MiriLLI incur a run-time overhead within three orders of magnitude higher than (HW)ASan and SafeFFI.

9 Conclusion

We presented SafeFFI, a novel approach to hoist sanitizer checks in Rust programs and MLAs to reduce the run-time overhead of sanitizers and detect memory safety violations at the boundary between safe and unsafe code. Compared to existing approaches, SafeFFI only uses function-local reasoning to hoist checks instead of depending on whole-program static points-to analysis, which can be expensive and error-prone.

Our approach for hoisting checks to locations of casts from `raw` pointers to `safe` pointers shows improved performance. We showed practicality and effectiveness in reducing the total number of sanitizer checks for popular Rust libraries by $72.02\,\% - 79.63\%$, resulting in a reduction of the average run-time overhead for ASan from $2.71\times$ to $2.44\times$ and for HWASan from $3.18\times$ to $2.29\times$. On our data set with existing real-world vulnerabilities, we were able to demonstrate that SafeFFI detects the same violations as the underlying sanitizer and even improves debuggability by failing closer to the root cause of memory safety violations in the interaction between safe Rust and unsafe Rust or foreign code. We showed that SafeFFI can be implemented in a modular way, such that it may in the future be adapted to further memory sanitizers, including those with stronger guarantees, such as SoftboundCETS [34, 35].

Acknowledgments

We would like to thank our shepherd and the reviewers for their constructive feedback that was essential in finalizing the paper. This project was partly funded by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).

Ethical Considerations

SafeFFI, as all base memory sanitizers, is generally designed as a defensive tool for software hardening and bug finding, but may be considered dual use. We have identified the following stakeholders for whom code instrumentation can lead to benefits or harm: (a) software developers benefit from detecting vulnerabilities early during development, being able to fix them to reduce risk (and costs); (b) attackers can instrument open-source software with sanitizers to detect vulnerabilities, which might guide them towards successfully exploiting vulnerable code; (c) users benefit from improved security when vulnerabilities are discovered and fixed during development. If software is deployed in production with sanitizers in enabled, exploitation is made considerably more challenging; yet, a potential risk is that benign executions could be terminated by the sanitizer, although the observed memory error would not have had negative effects. Overall we believe that the risks are clearly outweighed by the benefits.

All vulnerabilities used for evaluation in this work (see Table 1) were already publicly known, and we did not try to discover new vulnerabilities.

Open Science

We comply with the open science policy by releasing the SafeFFI prototype as open source and for artifact evaluation. This includes our modifications to rustc and LLVM, test sets, benchmarks and the corresponding scripts to build and execute them with SafeFFI, as well as our raw evaluation data. The artifact can be downloaded on Zenodo:
`https://doi.org/10.5281/zenodo.17976648`

The project repositories are hosted on Github and will be made public here: `https://github.com/SafeFFI/`

References

[1] Hussain MJ Almohri and David Evans. Fidelius Charm: Isolating Unsafe Rust Code. In *Proc. 8th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 248–255. ACM.

[2] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proc. ACM SIGOPS 28th Symp. on Operating Systems Principles (SOSP)*, pages 84–99. ACM.

[3] Inyoung Bang, Martin Kayondo, HyunGon Moon, and Yunheung Paek. TRust: A compilation framework for in-process isolation to protect safe rust against untrusted code. In *32nd USENIX Security Symposium*, pages 6947–6964. USENIX Association, 2023.

[4] Xingman Chen, Yinghao Shi, Zheyu Jiang, Yuan Li, Ruoyu Wang, Haixin Duan, Haoyu Wang, and Chao Zhang. MTSan: A feasible and practical memory sanitizer for fuzzing COTS binaries. In *Proc. 32nd USENIX Security Symposium*. USENIX Association, 2023.

[5] Kyuwon Cho, Jongyoon Kim, Kha Dinh Duy, Hajeong Lim, and Hojoon Lee. RustSan: Retrofitting AddressSanitizer for efficient sanitization of rust. In *33rd USENIX Security Symposium*, pages 3729–3746. USENIX Association, 2024.

[6] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 32(4):1–21, 2023.

[7] The Rust Project Developers. Primitive Type Reference - Safety. Section of "The Rust Standard Library" documentation. https://doc.rust-lang.org/std/primitive.reference.html#safety.

[8] The Rust Project Developers. Type Coercions. Section of "The Rust Reference". https://doc.rust-lang.org/reference/type-coercions.html.

[9] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Translating C to safer Rust. *Proc. ACM Program. Lang. (OOPSLA)*, October 2021.

[10] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. Towards translating real-world code with LLMs: A study of translating to Rust, 2025.

[11] Merve Gülmez, Thomas Nyman, Christoph Baumann, and Jan Tobias Mühlberg. Friend or foe inside? exploring in-process isolation to maintain memory safety for unsafe rust. In *2023 IEEE Secure Development Conference (SecDev)*, pages 54–66. IEEE, 2023.

[12] Sarah Harris, Simon Cooksey, Michael Vollmer, and Mark Batty. Rust for Morello: Always-On Memory Safety, Even in Unsafe Code (Artifact). *Dagstuhl Artifacts Series*, 9(2):25:1–25:2, 2023.

[13] Ben Hawkes. 0day "in the wild", 2019. https://googleprojectzero.blogspot.com/p/0day.html.

[14] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. Cryptsan: Leveraging arm pointer authentication for memory safety in c/c++. In *Proc. 38th ACM/SIGAPP Symposium on Applied Computing (SAC)*, page 1530–1539. ACM, 2023.

[15] Shuang Hu, Baojian Hua, Lei Xia, and Yang Wang. CRUST: towards a unified cross-language program analysis framework for Rust. In *22nd IEEE Int. Conf. Software Quality, Reliability and Security (QRS)*, pages 970–981. IEEE, 2022.

[16] Intel Corp. Pointer checker, 2021. https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/pointer-checker.html.

[17] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. Miri: Practical undefined behavior detection for rust. In *53rd ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*. ACM, 2026.

[18] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. PKRU-safe: Automatically locking down the heap between safe and unsafe languages. In *Proc. 17th European Conference on Computer Systems (EuroSys)*, pages 132–148. ACM.

[19] Steve Klabnik, Carol Nichols, and Chris Krycho. Understanding Ownership. Chapter 4 of "The Rust Programming Language". https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html.

[20] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: buffer overflow checks without the checks. In *Proc. 13th European Conference on Computer Systems (EuroSys)*. ACM, 2018.

[21] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in Rust. In *Proc. 9th Workshop on Programming Languages and Operating Systems (PLOS)*, pages 51–57. ACM.

[22] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. PACMem: Enforcing spatial and temporal memory safety via ARM Pointer Authentication. In *Proc. 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2022.

[23] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. Detecting Cross-language Memory Management Issues in Rust. In *Computer Security – ESORICS 2022*, Lecture Notes in Computer Science, pages 680–700. Springer Nature Switzerland.

[24] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. MirChecker: Detecting bugs in rust programs via static analysis. In *Proc. ACM SIGSAC*

Conference on Computer and Communications Security (CCS), page 2183–2196. ACM, 2021.

[25] Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing. CAMP: Compiler and allocator-based heap memory protection. In *Proc. 33rd USENIX Security Symposium*. USENIX Association, 2024.

[26] Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe Rust programs with XRust. In *Proc. ACM/IEEE 42nd Int. Conf on Software Engineering (ICSE)*, pages 234–245. ACM.

[27] Ian McCormack, Joshua Sunshine, and Jonathan Aldrich. A Study of Undefined Behavior Across Foreign Function Boundaries in Rust Libraries . In *47th Int. Conf. on Software Engineering (ICSE)*, pages 2075–2086. IEEE, May 2025.

[28] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-Language Attacks. In *Proc. 2022 Network and Distributed System Security Symposium (NDSS)*. Internet Society.

[29] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. In *BlueHat IL*. Microsoft Security Response Center, 2019.

[30] Jiun Min, Dongyeon Yu, Seongyun Jeong, Dokyung Song, and Yuseok Jeon. ERASan: Efficient Rust Address Sanitizer. In *2024 IEEE Symp. on Security and Privacy (S&P)*, pages 4053–4068. IEEE.

[31] MITRE Corp. 2024 CWE Top 10 KEV Weaknesses, 2024. https://cwe.mitre.org/top25/archive/2024/2024_kev_list.html.

[32] MITRE Corp. 2024 CWE top 25 most dangerous software weaknesses, 2024. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html.

[33] Simon Moll, Henrique Nazaré, Gustavo Vieira Machado, and Raphael Ernani Rodrigues. Bounds Check Hoisting for AddressSanitizer. In *Programming Languages*, pages 47–61. Springer International Publishing.

[34] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *Proc. ACM SIGPLAN Conference on Programming Language Desing and Implementation (PLDI)*. ACM, 2009.

[35] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. In *Proc. 2010 International Symposium on Memory Management (ISMM)*. ACM, 2010.

[36] Vikram Nitin, Anne Mulhern, Sanjay Arora, and Baishakhi Ray. Yuga: Automatically detecting lifetime annotation bugs in the Rust language. *IEEE Trans. Softw. Eng. (TSE)*, 50(10):2602–2613, October 2024.

[37] Benjamin Orthen, Oliver Braunsdorf, Philipp Zieris, and Julian Horsch. SoftBound+CETS revisited: More than a decade later. In *Proc. 17th European Workshop on Systems Security (EuroSec)*. ACM, 2024.

[38] Michalis Papaevripides and Elias Athanasopoulos. Exploiting mixed binaries. *ACM Trans. Priv. Secur. (TOPS)*, 24(2), January 2021.

[39] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. Keeping Safe Rust Safe with Galeed. In *Proc. Computer Security Applications Conference (ACSAC)*, pages 824–836. ACM.

[40] Leon Schuermann, Jack Toubes, Tyler Potyondy, Pat Pannuto, Mae Milano, and Amit Levy. Building bridges: Safe interactions with foreign languages through Omniglot. In *Proc. 19th Symp. Operating Systems Design and Implementation (OSDI)*, pages 595–613. USENIX Association, 2025.

[41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proc. 2012 USENIX Annual Technical Conference (ATC)*. USENIX Association, 2012.

[42] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves C/C++ memory safety. arxiv:1802.09517, Google LLC, 2018.

[43] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. In *Proc. 2019 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019.

[44] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *Proc. 25th Int. Conf. Compiler Construction (CC)*, pages 265–266. ACM, 2016.

[45] The Chromium Developers. Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/.

[46] Emanuel Vintila, Philipp Zieris, and Julian Horsch. Evaluating the Effectiveness of Memory Safety Sanitizers. In *Proc. 2025 IEEE Symp. on Security and Privacy (S&P)*, pages 88–88. IEEE, May 2025.

[47] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High System-Code Security with Low Overhead. In *2015 IEEE Symp. on Security and Privacy (S&P)*, pages 866–879. IEEE.

[48] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symp. on Security and Privacy (S&P)*, pages 20–37. IEEE.

[49] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. SANRAZOR: Reducing redundant sanitizer checks in C/C++ programs. In *Proc. 15th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 479–494. USENIX Association, 2021.

[50] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer. In *31st USENIX Security Symposium*, pages 4345–4363. USENIX Association, August 2022.

[51] Tianyang Zhou, Haowen Lin, Somesh Jha, Mihai Christodorescu, Kirill Levchenko, and Varun Chandrasekaran. LLM-driven multi-step translation from C to rust using static analysis. *CoRR*, abs/2503.12511, 2025.