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]LMU Munich, 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 runtime, 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 and adds much less compile-time overhead (2.64× compared to over 8.83×). On a collection of popular Rust crates and known vulnerable Rust code, SafeFFI achieves superior performance compared to state-of-the-art systems, reducing sanitizer checks by up to 98%, while maintaining correctness and flagging all spatial and temporal memory safety violations.

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 [28] and known exploited weaknesses [27], and studies by Google [11, 41] and Microsoft [25] 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 [39, 47], 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), where the Rust application is 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 may contain *unsafe code*, marked via the `unsafe` keyword, allowed to violate the strict typing rules for implementing 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 [26].

Both C/C++ and unsafe Rust code can be protected from exploitation by applying memory-safety *sanitizers* [40, 42] to the whole code base, which transparently introduce run-time checks for memory operations. While sanitizers such as AddressSanitizer (ASan) [37] and Hardware-assisted AddressSanitizer (HWASan) [38] do not require any source code changes and can achieve thorough memory safety guarantees [42], they introduce a significant run-time overhead by inserting checks for *every* pointer dereference in the code. Many of these checks are unnecessary, however: 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, 26] 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 statically through the type system. This way, we free the memory sanitizer from checking *every* pointer dereference, including from checking dereferences of potentially unsafe pointers. Therefore, for most patterns of Rust programs, we can elide more sanitizer checks than previous approaches, leading to better run-time performance.

Because our check placement enforces memory-safety across function calls, we only require local reasoning to ensure memory-safety across the whole application. 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 (§4).
- An architecture 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 (§5).
- A systematic evaluation using LLVM's widely-used HWASan on popular Rust crates, known vulnerable Rust code, and a set of microbenchmarks. Compared to the state of the art, we demonstrate that SafeFFI significantly reduces the compile time (2.64× compared to over 8.83×), increases the rate of elided checks (up to 98%), and consequently reduces the run-time overhead (from 3.22× to 2.51× on average) (§6).

2 Background

We now introduce the necessary background on memory safety in C/C++ and existing methods for runtime 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 [40, 42] typically instrument code at compile time with additional check logic to detect violations at runtime. 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 3rd-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, 12, 19, 22, 37, 38] or pointer-based [14, 17, 30, 33]. Object-based metadata, such as redzones [37] or memory tags [38], is associated with memory allocations and mark memory regions as valid or invalid. In contrast, pointer-based metadata augments pointers with additional information, such as bounds [30, 33] and references to temporal metadata [31]. Since they are associated with pointers, they can track the validity of memory accesses based on the pointer's state, e.g., enabling also sub-object bounds tracking [33].

Two widely used memory-safety sanitizers are ASan [37] and HWASan [38], available in LLVM and GCC. ASan instruments code to place red zones in shadow memory, catching out-of-bounds and use-after-free errors; it also poisons freed memory and delays reuse, improving detection 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. Retagging freed memory improves temporal detection, while spatial detection remains probabilistic and depends on tag size. On ARM, HWASan commonly leverages top-byte-ignore (TBI) to store tags in the pointer's top byte, improving compatibility with non-instrumented code and avoiding address-translation issues.

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 [16] 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 runtime 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.

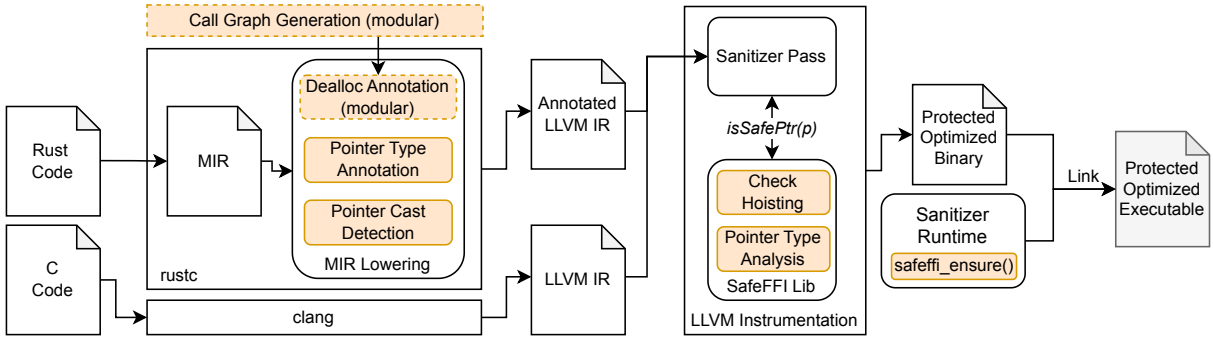


Figure 1: SafeFFI Architecture Overview. Rust and C/C++ are compiled to LLVM IR, a common intermediate representation for sanitizer instrumentation. The orange boxes depict our extensions to rustc and LLVM in this compilation workflow.

Safe Pointer-Like Types. The Rust language provides a set of *safe pointer-like* types that 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
- `&[T]`, `&str` – dynamically sized slice/string
- `&dyn Trait` – trait object reference, a pointer to an object whose type is only known at runtime
- Function pointers and closures

In the remainder of the paper, we will usually refer to these types summarily as `safe` pointers, in contrast to `raw` pointers.

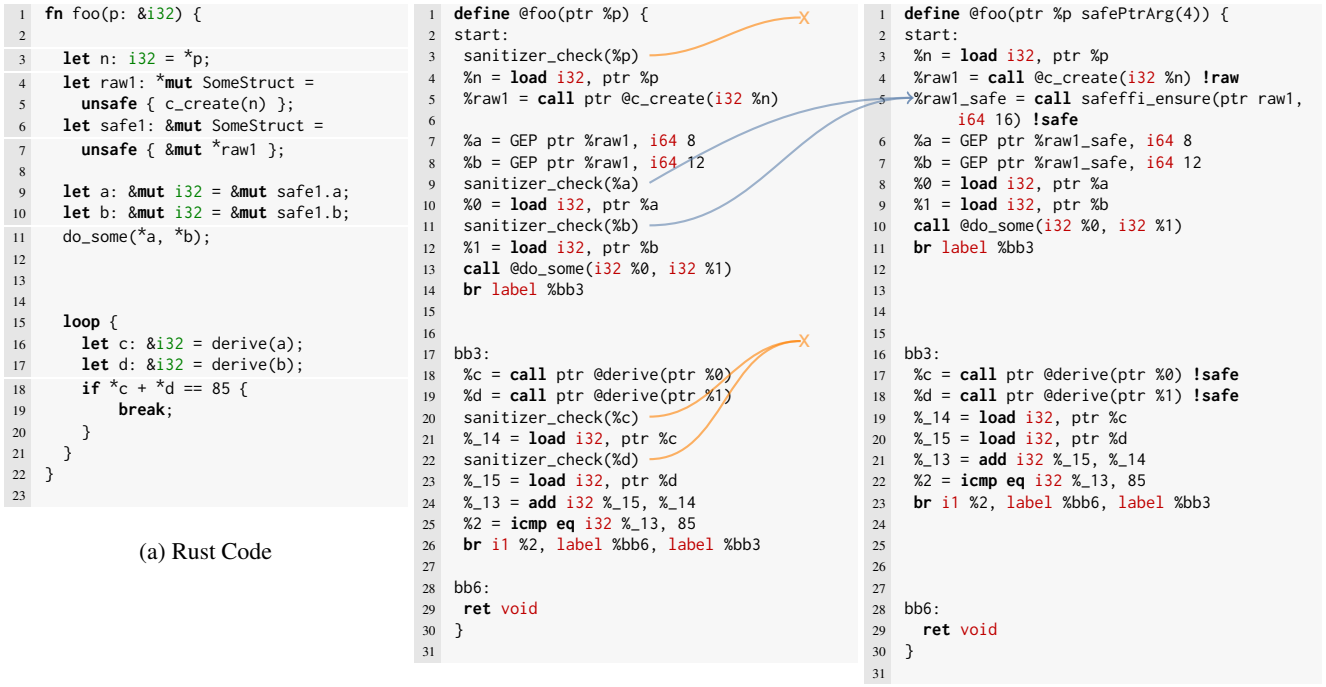
For pointers of these types that originate in `safe` Rust code, the Rust compiler ensures that they are non-null, properly aligned, and dereferenceable for the size of their pointee-type [7]. When `safe` pointers are derived from `raw` pointers, which may happen in `unsafe` code, Rust requires the developer 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 the 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 typesystem. 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. For such scenarios, the Rust compiler offers the possibility of using memory-safety sanitizers (see §2.1). However, these 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 feature called Foreign Function Interface (FFI) which allows 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 – they are called 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 [20]. Even worse, cross-language vulnerabilities can be vehicles for bypassing hardening mechanisms for C/C++ like Control-Flow Integrity (CFI) [24, 34]. For MLAs, memory-safety sanitizers are applicable to 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 Overview

Figure 1 shows an overview of SafeFFI's architecture. It is implemented as an extension of the Rust compiler pipeline. We extend the Rust compiler by an analysis of Rust's Mid-Level Intermediate Representation (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 by our new SafeFFI-Lib which consumes the pointer type annotation,



```
1  fn foo(p: &i32) {
2
3    let n: i32 = *p;
4    let raw1: *mut SomeStruct =
5      unsafe { c_create(n) };
6    let safe1: &mut SomeStruct =
7      unsafe { &mut *raw1 };
8
9    let a: &mut i32 = &mut safe1.a;
10   let b: &mut i32 = &mut safe1.b;
11   do_some(*a, *b);
12
13
14
15   loop {
16     let c: &i32 = derive(a);
17     let d: &i32 = derive(b);
18     if *c + *d == 85 {
19         break;
20     }
21   }
22 }
23
```

(a) Rust Code

```
1  define @foo(ptr %p) {
2  start:
3   sanitizer_check(%p)
4   %n = load i32, ptr %p
5   %raw1 = call ptr @c_create(i32 %n)
6
7   %a = GEP ptr %raw1, i64 8
8   %b = GEP ptr %raw1, i64 12
9   sanitizer_check(%a)
10  %0 = load i32, ptr %a
11  sanitizer_check(%b)
12  %1 = load i32, ptr %b
13  call @do_some(i32 %0, i32 %1)
14  br label %bb3
15
16
17 bb3:
18  %c = call ptr @derive(ptr %0)
19  %d = call ptr @derive(ptr %1)
20  sanitizer_check(%c)
21  %_14 = load i32, ptr %c
22  sanitizer_check(%d)
23  %_15 = load i32, ptr %d
24  %_13 = add i32 %_15, %_14
25  %2 = icmp eq i32 %_13, 85
26  br i1 %2, label %bb6, label %bb3
27
28 bb6:
29  ret void
30 }
31
```

(b) LLVM IR with sanitizer checks

```
1  define @foo(ptr %p safePtrArg(4)) {
2  start:
3   %n = load i32, ptr %p
4   %raw1 = call @c_create(i32 %n) !raw
5   %raw1_safe = call safeffi_ensure(ptr raw1,
        i64 16) !safe
6   %a = GEP ptr %raw1_safe, i64 8
7   %b = GEP ptr %raw1_safe, i64 12
8   %0 = load i32, ptr %a
9   %1 = load i32, ptr %b
10  call @do_some(i32 %0, i32 %1)
11  br label %bb3
12
13
14
15
16 bb3:
17  %c = call ptr @derive(ptr %0) !safe
18  %d = call ptr @derive(ptr %1) !safe
19  %_14 = load i32, ptr %c
20  %_15 = load i32, ptr %d
21  %_13 = add i32 %_15, %_14
22  %2 = icmp eq i32 %_13, 85
23  br i1 %2, label %bb6, label %bb3
24
25
26
27
28 bb6:
29   ret void
30 }
31
```

(c) Modified LLVM IR with sanitizer checks being hoisted by SafeFFI

Figure 2: A simplified example that shows which sanitizer checks can be elided using our SafeFFI optimization

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 more detailed description of the architecture is given in §5.

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` instructions. By using information from Rust's typesystem, 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 safe pointer-like, then all subsequent sanitizer checks are obsolete and we can elide them.

In source line 7 of 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 a sanitizer check 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 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.

4 Typesystem-Guided Sanitizer Checks

Now we explain our concept in detail. We begin by motivating pointer casts as logical boundaries for memory safety enforcement (§4.1). We introduce our method for using function-local type information to hoist checks to casts while eliding sanitizer checks for `safe` pointers (§4.2) and discuss additional measures required for full temporal memory safety (§4.3). Finally, we show how to leverage the Rust type system to maintain memory-safety across function calls without expensive whole-program analysis (§4.4). Note that this design includes full support for the presence of unsafe foreign code in mixed-language scenarios.

4.1 Memory Safety Enforcement Boundaries

Incorrect memory access through raw pointers can cause memory bugs even in safe Rust [26]. Raw pointer dereferences are dangerous because Rust does not enforce any guarantees for `raw` pointers. For `safe` pointers, 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 can exist outside the lifetime of its original pointed-to memory object.

There are multiple ways to create and derive `safe` pointers. All of them are checked by the Rust compiler, *except* for the cast from `raw` pointer to `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 specific sanitizer check.

4.2 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 those checks in place. 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 construc-

tion, we insert a sanitizer check at the creation site of the `safe` pointer, effectively combining and hosting 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.

4.3 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 reasoning across function calls in §4.4. 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 §5.4 for a detailed description).

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);
```

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.

4.4 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.

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 check in the prologue of the callee to ensure that the `safe` pointer is indeed valid. Because of our established memory safety invariant, we guarantee that `safe` pointers are indeed safe to dereference for the remainder of the scope of the caller function, which means they are also valid for the whole scope of the callee. Hence, we can elide all 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 function `derive()` returns a `safe` pointer which 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 because 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.

```
// 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 the p_safe pointer 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 in the caller function `foo()`.

5 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 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, aka. the *Borrow Checker*. While 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. Further details on each component are described in the following subsections.

The goal for our architecture was to make integration with existing sanitizers as easy as possible by requiring only minimal changes to the sanitizer: 1. querying pointer types via `is_safe_pointer(Value* ptr)` in SafeFFI-Lib, and 2. providing a run-time check implementation `safeffi_ensure(void* p, u64 size)`. This way, our architecture increases the chance of adoption.

5.1 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]:

[1]There is no official specification of Rust, thus we have to take the

- `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 and arrays, and raw pointers but it is also the case for all ADT 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 which contain a data pointer and a dynamic 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 actually no 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 are annotating them with a `NOPTR` tag and treat them as always safe because they cannot be user-manipulated because they are compiler-generated pointers.

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.

5.2 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`, we attempt to statically compute the offset 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.

5.3 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

Rust compiler's behaviour as reference: https://github.com/rust-lang/rust/blob/1.52.0/compiler/rustc_target/src/abi/mod.rs#L847

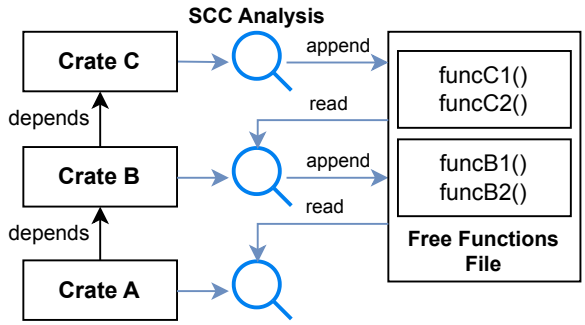


Figure 4: The NoFree SCC Analysis is performed on all dependencies to determine possibly heap-deallocating functions.

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.

5.4 Callgraph-Based Deallocation Checking

As mentioned in §4.3, detecting Free-During-Scope temporal vulnerabilities requires checking that a dereferenced pointer has not become dangling since its creation.

To address this, we implemented a call-graph-based heap deallocation analysis in LLVM to determine whether a given function may perform a deallocation. It analyzes bottom-up over the call graph: functions without call instructions are visited first. The analysis is implemented as a LLVM SCC (strongly connected components) pass because 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. bin

6 Evaluation

We implemented SafeFFI on top of Rust nightly-2021-02-22, which corresponds to Rust compiler version 1.52.0 and LLVM version 12. We integrated SafeFFI into HWASan because it is the sanitizer with the best detection capabilities under the maintained sanitizers [42]. Because HWASan uses ARM-specifc hardware features, we ran all experiments with SafeFFI on a MacStudio M2 Ultra with 24 ARMv8 cores and 192 GB RAM running an Ubuntu 20.04 docker container on ASAHI Linux 6.12.0. To reproduce results of related work as closely as possible and compare to them, we conducted experiments with ERASan [26] and RustSan [5] on a x86_64 system—the architecture on which they were developed—running Ubuntu 24.04. We make all datasets, scripts, and tools used in our evaluation available for reproduction.

In this section, we present our evaluation of SafeFFI answering the following research questions:

RQ1 How does SafeFFI affect the detection capabilities of the sanitizer (§6.1)?

RQ2 How many sanitizer checks can SafeFFI reduce in sanitized programs (§6.2)?

RQ3 How much can SafeFFI reduce the run-time of sanitized programs (§6.2)?

RQ4 Is SafeFFI's implementation robust in the presence of FFI interactions in MLAs (§6.3)?

RQ5 How much compile-time overhead does SafeFFI incur (§6.4)?

6.1 Correctness

We answer **RQ1** by testing SafeFFI on known real-world memory safety vulnerabilities. We further include RustSan [5] and ERASan [26] as related work and ASan as a baseline for those two approaches. We assembled the dataset 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. §4.2), these cases are classified as detected (●).

The results show that SafeFFI catches all vulnerabilities that HWASan detects. Due to hoisting checks to the memory safety boundary at the location of pointer casts, SafeFFI is able to report 21 vulnerabilities earlier than HWASan, increasing debuggability, and it detects one vulnerability where every other approach fails. 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.

RustSan retains nearly all capabilities of ASan, performing slightly worse than HWASan and SafeFFI in a few cases. 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 necess ary 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]

Table 1: Vulnerability Detection

ID	INTERCEPT	HWASAN	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 Error
INTERCEPT = Interception-based Detection

6.2 Effectiveness

In this section, we provide empirical evidence that SafeFFI is effective in reducing the number of HWASan checks (**RQ2**) and, consequently, the run-time overhead of the sanitizer (**RQ3**). We further compare SafeFFI against RustSan. As mentioned in §6.1, ERASan incorrectly removes necessary checks, leading to a skewed number of elided checks. Thus, we exclude ERASan from further comparisons. We use the same crates (Rust's term for packages or libraries) as RustSan for the evaluation. Note that for RustSan, neither the evaluation scripts nor the tested versions of the crates are available, thus we chose a common version that both SafeFFI and RustSan can compile with their respective tool chains. The crates Rocket and RustPython are missing because they contain a compilation error in the version compatible with Rust 1.52.0 and we bundled the libstd tests into a single crate. Elided checks are measured at LLVM IR level. To prevent the standard library from overshadowing the results of smaller crates, we only count checks outside the standard library and list the standard library results separately.

[2] Github link redacted for review, anonymized screenshot in Appendix B

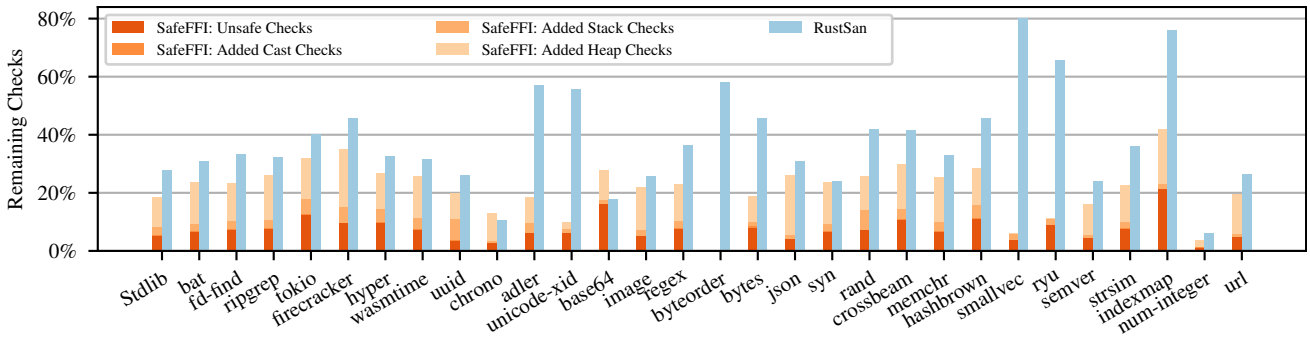


Figure 5: Sanitizer checks after elision by SafeFFI and RustSan, respectively, as fractions of original checks for individual packages. For SafeFFI, checks are subdivided into remaining unsafe checks and added cast, stack and heap checks.

Elided Checks. Figure 5 shows SafeFFI's capability to elide sanitizer checks. SafeFFI retains on average 7.38% of the HWASan checks because they vet an unsafe pointer. Then, cast checks (cf. §4.1) and stack checks (cf. §4.4) are added to ensure spatial and stack-temporal memory safety for safe pointers. After these checks, on average 10.22% of the checks remain. Adding the optional heap checks for Free-During-Scope vulnerabilities (cf. §4.3) increases the average number of remaining checks to 21.43%.

Compared to RustSan (blue bars in Figure 5), SafeFFI with full temporal safety elides on weighted average 7.34% *more* checks, providing a significant improvement over the current state of the art. On closer examination, SafeFFI performs better on 28 benchmarks and worse on 2 benchmarks, exceeding 5% difference on 24 of 28 benchmarks and 1 of 2 benchmarks respectively. We observe especially large difference ($> 20\%$) on bytes, ryu, indexmap (in favor of SafeFFI) and base64 (in favor of RustSan). bytes and indexmap are data structures with unsafe pointers that alias with safe views returned by the API. Similarly, in ryu crucial internal buffers make use of unsafe. In these cases, SafeFFI already performs better in isolation and the advantage should even increase in dependents of these libraries. Furthermore, as ryu contains no heap allocations, SafeFFI does not need to add any heap checks, which further increases the savings. In contrast, on base64 SafeFFI performs significantly worse. The reason is that base64 is lowered into many dynamic GEP statements, which we conservatively overapproximate (cf. §5.2). Note that SafeFFI and RustSan cannot operate in the exact same environment—however, it is unlikely that the differences affect elision rates.

Run-time Performance. We have shown that SafeFFI is superior in eliding checks. However, not all checks are equally important for the run-time, checks on hot paths through the program disproportionately impact the run-time [43]. Therefore, we also evaluate the run-time on several benchmarks. Figure 6 shows the run-time overhead of HWASan and SafeFFI on the benchmarks. HWASan imposes an average runtime overhead of 3.22× (median: 3.03×) compared to an uninstrumented binary. SafeFFI without heap-temporal-safety reduces this overhead to 2.12× (median: 2.06×) and SafeFFI with full temporal-safety to 2.51× (median: 2.24×). Considering each benchmark individually, SafeFFI improves the run-time by at least 10% across the board, except for four crates. Base64 contains a lot of dynamic GEP instructions, which SafeFFI conservatively overapproximates, as explained above. For json and hashbrown, the additional heap checks required to detect Free-During-Scope vulnerabilities (see §4.3) limit the achievable performance improvement, without added heap checks SafeFFI significantly reduces the run-time overhead. For regex, none of the added checks are proportionally bigger compared to other crates, therefore, we assume that some inserted checks are on hot paths in the benchmark execution.

Only base64 $1/20$ for SafeFFI None and $4/20$ for SafeFFI not exceeding a 10% improvement in overhead. Unfortunately, we could not include benchmarking data for RustSan because the cargo benchmark harness compiled with RustSan consistently raises segmentation faults during startup. As target versions and benchmarking scripts are not included in RustSan's published code, we also cannot perform a like-for-like comparioson against the numbers reported in their paper. Deducing from our comparison of elided checks with their reported results, we suspect that they benchmarked without compiling an instrumented version of the Rust standard library which could introduce false negatives and thus distorts the comparison.

6.3 Robustness in MLA Scenarios

Our experiments on correctness (§6.1) and effectiveness (§6.2) 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

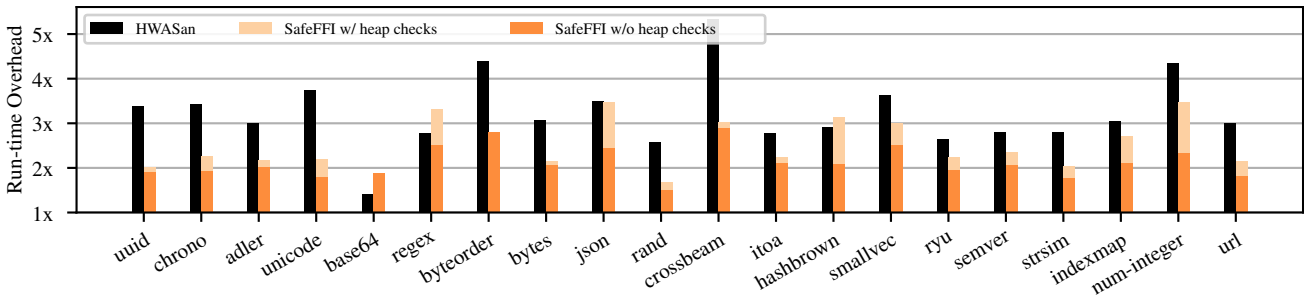


Figure 6: Run-time overhead of vanilla HWAsan, SafeFFI (including cast and stack checks), and SafeFFI with added heap checks compared to baseline run-time without sanitizer.

test set of minimal test cases that cover 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.

6.4 Compile-Time Performance

Figure 7 shows the compile-time overhead of HWASan, SafeFFI and RustSan, answering **RQ5**. Note that we chose the respective toolchain versions as a baseline for a fair comparison. On average, HWASan induces a compile-time overhead of 1.69× compared to the baseline. SafeFFI without heap-temporal-safety increases the overhead to 2.02×. SafeFFI with full temporal-safety further increases the compile-time overhead to 2.64×. The compile-time memory overhead of SafeFFI is reasonable, with an average of 1.14× compared to the baseline and within measurement deviation compared to HWASan. RustSan adds a compile-time overhead of 8.83× on average, with outliers reaching an overhead of up to 43.03×, because it performs a whole-program points-to analysis to identify safe pointers that alias raw pointers. For the same reason, RustSan also induces a significant memory overhead during compilation of 9.95× on average because bigger crates like wasmtime consuming up to 59.03× more RAM.

7 Related Work

Run-time Check Reduction for Rust. Rust for Morello [10] executes Rust programs on the CHERI-Architecture [44] 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.

All three approaches annotate pointers during MIR lowering so that safe and `raw` pointers remain distinguishable at the LLVM IR level, with the goal to elide checks for `safe` pointers. The key difference is that RustSan and ERASan rely on whole-program static value-flow analysis to compute points-to sets, while SafeFFI annotates function parameters and return values with pointer types, allowing for efficient local reasoning. This design significantly reduces compile-time overhead and scales better to large code bases.

Moreover, this difference makes SafeFFI significantly more effective at eliding checks. This happens not only because the static analysis tends to overapproximate, but also for a fundamental reason: Rust code that interacts with `unsafe` or foreign code often uses patterns where `safe` pointers are derived from `raw` pointers *or vice versa*. RustSan and ERASan need to insert checks for every pointer which shares the same points-to set as the `raw` pointer, inhibiting the elision of checks for `safe` pointers in this case. Our approach, on the other hand, benefits from the chance that the actual number of `raw` pointer interactions might be small or just unidirectional, because SafeFFI just inserts a check at casts from `raw` pointers to `safe` pointers. This makes SafeFFI especially suitable for scenarios involving `unsafe` code, e.g., fuzzing Rust crates that implement efficient datastructures or protecting MLAs. While not explicitly stated, we deem RustSan's and ERASan's concept also to be applicable to 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. Theoretically, our check hoisting concept introduces a risk of missing Free-During-Scope temporal vulnerabilities (see §4.3), an issue for which we provide a solution for single-threaded programs by optionally inserting additional sanitizer checks at potential heap deallocation operations. SafeFFI still outperforms RustSan and ERASan on the evaluated metrics in most cases.

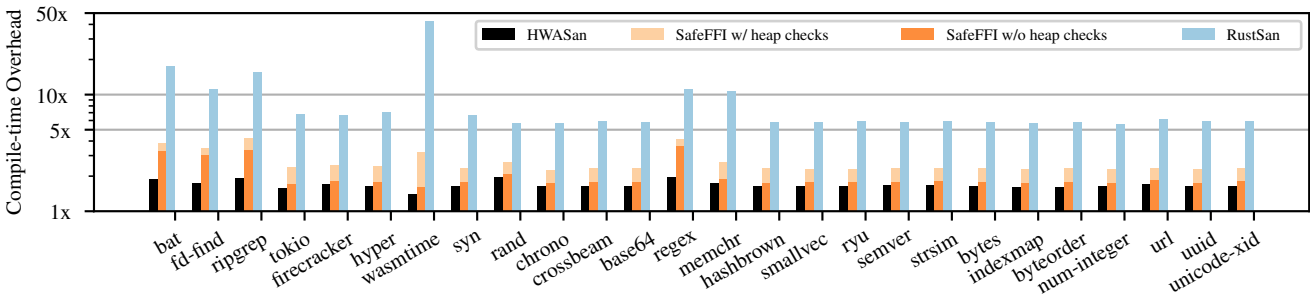


Figure 7: Compilation-time overhead of vanilla HWAsan, SafeFFI (including cast and stack checks), and SafeFFI with added heap checks compared to baseline compilation-time without sanitizer. Note that the scale is logarithmic to accommodate large overheads of RustSan.

Those two approaches conceptually avoid this risk by retaining all sanitizer checks at pointer dereference locations if the pointer might alias with a `raw` pointer.

However, interestingly, RustSan and ERASan show worse ability to retain the sanitizer's detection capabilities on our tests with real-world vulnerabilities (Table 1). ERASan in particular suffers from implementation bugs that cause false negatives. Both implement a blacklist approach for identifying `raw` pointers, potentially leading to false negatives if the type annotation is incomplete. SafeFFI uses a whitelist approach instead, validating that *every* pointer has a type and overapproximating `raw` pointers where the implementation is incomplete—which should be accounted for because the Rust compiler's internals for lowering MIR to LLVM IR are very complex. Overall, our implementation is more tightly integrated with the Rust compilation toolchain without the need to link and analyze the program externally. This should increase the chances of adoption.

General Sanitizer Optimizations. Prior work on C/C++ includes ASAP [43] which removes checks on hot paths, trading security for performance. Moll et al. [29], hoist bounds checks in loops. ASan-- [46] uses lightweight static analysis and SanRazor [45] 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.

Runtime Isolation for Rust. Several proposed approaches isolate safe Rust from unsafe code, e.g., Sandcrust [18], Fidelius Charm [1], XRust [23], relying on developer annotations or hardware/OS support for process isolation. TRust [3], PRKUSafe [15] and Gülmez et al. [9] instead use Intel's Memory Protection Keys (MPK) to separate safe and unsafe objects in different memory regions. Galeed [35] and Omniglot [36] explicitly address cross-language attacks [24] 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 Analyzers for Memory Safety in Rust. Tools like MIRChecker [21], Rudra [2], Yuga [32], and SafeDrop [6] use static analysis to detect memory bugs in Rust, while FFIChecker [20] and CRust [13] 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.

8 Conclusion

We presented SafeFFI, a novel approach to hoist sanitizer checks in Rust programs and MLAs to reduce the run-time overhead of sanitizers. Compared to existing approaches, SafeFFI only uses funcion-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 compared to existing methods, especially for Rust programs that frequently interact with unsafe code or FFI code. We showed practicality and effectiveness in reducing the total number of sanitizer checks for popular Rust libraries by $58\% - 98\%$, resulting in a reduction of the run-time overhead of HWASan from $3.22\times$ to $2.51\times$ on average. Our experiments with existing real-world vulnerabilities show that SafeFFI not only maintains the sanitizer's detection capabilities but also 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 very modular way such that in the future it can be adapted for different memory sanitizers that choose different tradeoffs like SoftboundCETS [30, 31] for more security.

Ethical Considerations

SafeFFI is a purely defensive tool to improve the efficiency of memory sanitizers in Rust programs. Enabling more software to run with memory sanitization should have a net benefit in reducing vulnerabilities. A potential risk is that benign executions could be terminated although the observed memory error would not have had negative effects, but we believe that this risk is 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. An anonymized version for review can be found on Zenodo:
https://tinyurl.com/safeffi-usenix

The final version will be published on Github upon acceptance.

References

[1] Hussain MJ Almohri and David Evans. Fidelius Charm: Isolating Unsafe Rust Code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 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 *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 84–99. Association for Computing Machinery.

[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 (USENIX Security 23)*, pages 6947–6964, 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 *USENIX Security*. 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 (USENIX Security 24)*, pages 3729–3746, 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*, 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] 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, 2023. https://arxiv.org/abs/2306.08127.

[10] Sarah Harris, Simon Cooksey, Michael Vollmer, and Mark Batty. Rust for Morello: Always-On Memory Safety, Even in Unsafe Code (Experience Paper). In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ECOOP.2023.39*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik.

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

[12] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. CryptSan: Leveraging ARM Pointer Authentication for memory safety in C/C++. In *SAC*. ACM, 2023.

[13] Shuang Hu, Baojian Hua, Lei Xia, and Yang Wang. CRUST: towards a unified cross-language program analysis framework for rust. In *22nd IEEE International Conference on Software Quality, Reliability and Security, QRS 2022, Guangzhou, China, December 5-9, 2022*, pages 970–981. IEEE, 2022.

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

[15] 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 *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 132–148. ACM.

[16] 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.

[17] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *EuroSys*. ACM, 2018.

[18] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, pages 51–57. ACM.

[19] 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 *CCS*. ACM, 2022.

[20] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. Detecting Cross-language Memory Management Issues in Rust. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng, editors, *Computer Security – ESORICS 2022*, Lecture Notes in Computer Science, pages 680–700. Springer Nature Switzerland.

[21] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Mirchecker: detecting bugs in rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 2183–2196, 2021.

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

[23] Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe rust programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 234–245. ACM.

[24] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-Language Attacks. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society.

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

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

[27] MITRE Corp. 2024 CWE top 10 kev weaknesses, 2024. https://cwe.mitre.org/top25/archive/2024/2024_kev_list.html.

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

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

[30] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for C. In *PLDI*. ACM, 2009.

[31] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. In *ISMM*. ACM, 2010.

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

[33] Benjamin Orthen, Oliver Braunsdorf, Philipp Zieris, and Julian Horsch. SoftBound+CETS revisited: More than a decade later. In *EuroSec*. ACM, 2024.

[34] Michalis Papaevripides and Elias Athanasopoulos. Exploiting mixed binaries. *ACM Transactions on Privacy and Security (TOPS)*, 24(2):1–29, 2021.

[35] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. Keeping Safe Rust Safe with Galeed. In *Annual Computer Security Applications Conference*, ACSAC, pages 824–836. Association for Computing Machinery.

[36] Leon Schuermann, Jack Toubes, Tyler Potyondy, Pat Pannuto, Mae Milano, and Amit Levy. Building bridges: Safe interactions with foreign languages through omniglot. In Lidong Zhou and Yuanyuan Zhou, editors, *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, pages 595–613. USENIX Association, 2025.

[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC*. USENIX Association, 2012.

[38] 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.

[39] Melih Sirlanci, Carter Yagemann, and Zhiqiang Lin. C2RUST-BENCH: A minimized, representative dataset for c-to-rust transpilation evaluation. *CoRR*, abs/2504.15144, 2025.

[40] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. In *S&P*. IEEE, 2019.

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

[42] Emanuel Vintila, Philipp Zieris, and Julian Horsch. Evaluating the Effectiveness of Memory Safety Sanitizers. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 88–88, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[43] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High System-Code Security with Low Overhead. In *2015 IEEE Symposium on Security and Privacy*, pages 866–879.

[44] 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 Symposium on Security and Privacy*, pages 20–37.

[45] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. {SANRAZOR}: Reducing redundant sanitizer checks in {C/C++} programs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 479–494, 2021.

[46] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4345–4363, 2022.

[47] 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.

A Full Evaluation Results

A.1 Check Elision

		SafeFFI						RustSan
Benchmark	# Sanitizer Checks	Elided Checks	Added Cast Checks	Added Stack Checks	Savings w/o Heap Checks	Added Heap Checks	Savings w/ Heap Checks	Savings
Stdlib	182923	94.68%	446	5,085	91.66%	18,523	81.53%	72.40%
bat	492479	93.41%	1,150	12,561	90.62%	69,765	76.46%	69.02%
fd-find	294081	92.73%	690	8,536	89.59%	37,919	76.70%	66.69%
ripgrep	389881	92.44%	897	11,761	89.20%	60,142	73.77%	67.80%
tokio	88971	87.73%	597	4,400	82.11%	12,369	68.21%	59.92%
firecracker	68271	90.47%	134	3,734	84.81%	13,497	65.04%	54.30%
hyper	150280	90.31%	543	6,495	85.63%	18,419	73.37%	67.44%
wasmtime	1084685	92.62%	1,969	40,667	88.69%	153,911	74.50%	68.48%
uuid	1207	96.44%	2	86	89.15%	108	80.20%	74.12%
chrono	18286	97.27%	1	108	96.67%	1,741	87.15%	89.58%
adler	193	93.78%	0	7	90.16%	17	81.35%	42.86%
unicode-xid	93	93.55%	0	1	92.47%	2	90.32%	44.28%
base64	649	83.67%	0	9	82.28%	65	72.27%	82.21%
image	241051	94.83%	433	4,397	92.82%	35,548	78.08%	74.34%
regex	214812	92.28%	359	5,569	89.52%	27,261	76.83%	63.84%
byteorder	4	100.00%	0	0	100.00%	0	100.00%	41.95%
bytes	2473	91.95%	19	29	90.01%	218	81.20%	54.46%
json	10119	95.67%	6	105	94.57%	2,088	73.94%	69.12%
syn-1.0.48	49678	93.38%	144	1,164	90.75%	7,125	76.40%	76.12%
rand	10106	92.72%	6	686	85.87%	1,165	74.34%	58.03%
crossbeam	3179	89.34%	17	103	85.56%	490	70.15%	58.67%
memchr	215707	93.60%	738	6,567	90.21%	33,424	74.71%	67.17%
hashbrown	150	88.67%	0	7	84.00%	19	71.33%	54.39%
smallvec	51	96.08%	0	1	94.12%	0	94.12%	20.00%
ryu	837	90.80%	0	15	89.01%	0	89.01%	34.52%
semver	3251	95.54%	2	31	94.52%	343	83.97%	76.09%
strsim	4805	92.20%	7	103	89.91%	608	77.25%	63.88%
indexmap	243	78.60%	0	4	76.95%	46	58.02%	24.10%
num-integer	2926	98.70%	0	9	98.39%	64	96.21%	93.96%
url	25365	95.23%	12	264	94.14%	3,414	80.68%	73.62%

A.2 Compile-time Overhead

	Baseline		HWASan		SafeFFI w/o heap checks		SafeFFI w/ heap checks		RustSan	
Benchmark	Time (s)	Memory (MB)	Time OH	Mem OH	Time OH	Mem OH	Time OH	Mem OH	Time OH	Mem OH
bat	45.32	584.27	1.90×	1.31×	3.27×	1.25×	3.86×	1.27×	17.43×	23.56×
fd-find	44.78	612.48	1.75×	1.35×	3.02×	1.33×	3.51×	1.30×	11.22×	17.10×
ripgrep	40.52	584.31	1.91×	1.34×	3.35×	1.36×	4.21×	1.36×	15.69×	21.03×
tokio	29.21	597.66	1.57×	1.29×	1.70×	1.26×	2.37×	1.12×	6.80×	7.84×
firecracker-0.23.0	27.68	588.69	1.71×	1.31×	1.80×	1.33×	2.49×	1.28×	6.70×	8.48×
hyper	38.07	612.47	1.63×	1.24×	1.78×	1.18×	2.44×	1.07×	7.02×	9.50×
wasmtime	104.99	1,012.59	1.41×	1.75×	1.63×	1.69×	3.19×	1.04×	43.03×	59.03×
syn-1.0.48	25.79	616.94	1.63×	1.23×	1.76×	1.20×	2.34×	1.08×	6.61×	6.78×
rand	22.97	618.52	1.94×	1.24×	2.07×	1.21×	2.62×	1.07×	5.73×	5.23×
chrono	22.71	589.92	1.63×	1.32×	1.73×	1.28×	2.27×	1.11×	5.65×	5.36×
crossbeam-0.8.0	20.75	615.67	1.65×	1.23×	1.79×	1.23×	2.32×	1.07×	5.95×	4.92×
base64	20.40	583.89	1.64×	1.35×	1.77×	1.24×	2.36×	1.14×	5.86×	4.84×
regex-1.4.2	33.46	611.59	1.95×	1.73×	3.61×	1.71×	4.12×	1.70×	11.08×	13.14×
memchr-2.3.4	33.31	613.22	1.75×	1.23×	1.89×	1.18×	2.64×	1.07×	10.65×	12.76×
hashbrown-0.9.1	20.50	617.00	1.64×	1.23×	1.74×	1.22×	2.32×	1.07×	5.78×	4.87×
smallvec-1.5.0	20.39	616.17	1.63×	1.23×	1.76×	1.20×	2.31×	1.07×	5.76×	4.80×
ryu	20.44	596.11	1.65×	1.29×	1.77×	1.21×	2.31×	1.12×	5.93×	4.88×
semver	20.55	606.83	1.67×	1.23×	1.79×	1.21×	2.32×	1.10×	5.79×	4.92×
strsim	20.56	610.05	1.67×	1.28×	1.80×	1.22×	2.35×	1.07×	5.91×	4.95×
bytes	20.66	616.00	1.64×	1.21×	1.78×	1.21×	2.33×	1.07×	5.82×	4.95×
indexmap-1.6.0	20.76	616.83	1.62×	1.23×	1.73×	1.22×	2.29×	1.07×	5.75×	4.80×
byteorder	20.40	616.66	1.63×	1.26×	1.77×	1.18×	2.34×	1.06×	5.86×	4.83×
num-integer	21.39	616.98	1.63×	1.23×	1.74×	1.17×	2.30×	1.06×	5.61×	4.88×
url-2.2.0	23.91	591.22	1.72×	1.24×	1.84×	1.25×	2.36×	1.12×	6.22×	5.61×
uuid	20.64	591.81	1.66×	1.25×	1.75×	1.27×	2.31×	1.11×	5.88×	4.85×
unicode-xid-0.2.1	20.24	611.31	1.63×	1.22×	1.79×	1.27×	2.33×	1.08×	5.87×	4.81×

A.3 Run-time Overhead

Benchmark	Baseline (ns)	HWASan	SafeFFI w/o heap checks	SafeFFI w/ heap checks
uuid	881.02	3.37×	1.90×	2.01×
chrono	1,726.16	3.42×	1.93×	2.25×
adler	451.26	3.00×	2.01×	2.17×
unicode	$5.82 \cdot 10^5$	3.73×	1.79×	2.18×
base64	$6.01 \cdot 10^5$	1.41×	1.87×	1.83×
regex	6,430.57	2.76×	2.51×	3.31×
byteorder	$1.33 \cdot 10^6$	4.40×	2.79×	2.77×
bytes	1,941.19	3.06×	2.06×	2.14×
json	14,969.95	3.49×	2.44×	3.47×
rand	11,795.64	2.56×	1.49×	1.67×
crossbeam	267.60	5.34×	2.89×	3.02×
itoa	165.65	2.77×	2.11×	2.23×
hashbrown	$1.42 \cdot 10^5$	2.92×	2.07×	3.12×
smallvec	606.39	3.63×	2.51×	3.00×
ryu	264.85	2.64×	1.94×	2.23×
semver	1,225.26	2.79×	2.06×	2.35×
strsim	70,290.11	2.79×	1.76×	2.02×
indexmap	$1.59 \cdot 10^6$	3.05×	2.10×	2.70×
num-integer	15,029.28	4.34×	2.31×	3.46×
url	9,972.00	3.00×	1.81×	2.15×

B ERASan Issue