usenix.org

# Memory Safety is Merely Table Stakes

*Authors: Leon Schuermann, Jack Toubes, Tyler Potyondy, Pat Pannuto, Mae Milano, Amit Levy*

16–20 minutes

---

Safe Interactions with Foreign Languages through Omniglot

The past few years has seen a massive success story for systems programming. Entire categories of bugs that used to plague systems programmers—like use-after-free, data races, and segmentation faults—have begun to completely disappear. The secret to this new reality is a set of systems programming languages chief among them Rust—whose powerful type systems are able to constructively eliminate these kind of bugs; if it compiles, then it's correct … or at least, will not contain use-after-free or other memory safety errors. These languages are gaining widespread adoption across industry [1, 2, 3] and academia [4, 5, 6, 7] alike, and are adopted for ambitious and critical systems, such as new high-performance compute libraries, distributed storage systems, and operating systems.

Despite these successes, the reality is a little more complicated. There is a great amount of software already written in other languages. And often, external constraints such as certification requirements or developer expertise force even new components to be written in other, less safe languages. Therefore, an important feature for any new systems programming language is its ability to easily and efficiently interact with existing foreign libraries. Developers building new systems can leverage existing native cryptography, mathematics, graphical, and other libraries immediately, without waiting for them to first be ported to new languages and without suffering a performance hit. They can incrementally migrate existing systems, replacing components in a legacy C/C++ codebase with safe alternatives [1].

Unfortunately, interacting with foreign code can result in subtle, but nonetheless devastating safety violations that re-introduce the very concerns many developers are trying to avoid by using type-safe languages. For example, foreign libraries may themselves include memory safety vulnerabilities, such as OpenSSL's infamous Heartbleed bug [8]. When foreign code is invoked through a Foreign Function Interface (FFI), it runs in the same address space and with the same privileges as the host language. Therefore, vulnerabilities in native libraries can affect the entire host program and break memory or type safety guarantees.

While we are quick to reach for tools like process isolation, a system call boundary, or a client-server model to solve this, these tools often only help uphold memory safety, which is only half the battle. Each language has specific invariants over its types (like permissible values) which its compiler relies on when producing code. Ensuring that all types are correctly inhabited goes beyond memory safety; it requires type safety. In fact, memory and type safety are intertwined: a violation of one can easily break the other. And finally, some program invariants—like whether references can be aliased—require reasoning about both type and memory safety. Interactions with untrusted code or between different languages that violate these invariants can lead to undefined behavior and, in turn, break other safety properties.

We present Omniglot [9], a new approach and framework we have developed that can maintain both memory and type safety across interactions with untrusted foreign libraries, in different settings: we implement prototypes for Linux userspace applications and a Rust-based kernel. In this article, we want to focus on illustrating the fundamental link between memory and type safety through an example of interacting with a foreign library and provide an intuition on how the Omniglot framework works.

## Breaking Memory Safety with Invalid Values

Before we can discuss how Omniglot enables safe interactions with foreign languages, we need to understand the types of invariants that programming languages require developers to uphold, and why conventional solutions such as memory isolation are insufficient. In this section, we use a simple example to illustrate how breaking one of Rust's invariants—that of valid values—can in turn violate a range of other safety properties (including memory safety) and break conventional isolation techniques.

In Listing 1 we show an example of a C function and type definition, and corresponding bindings for this type and function signature in Rust. We define a C enum `async_res_t` to signal the completion state of an asynchronous operation. In Rust, we define a corresponding type `enum AsyncRes` with identical variants. Annotating this type with `#[repr(C)]` ensures that the enum will have an identical representation to its C counterpart. Finally, we define a function `async_print` in C, and declare a corresponding foreign function binding in Rust.

C
```c
typedef enum {
  SUCCESS = 0,
  FAILURE = 1,
  PENDING = 2,
} async_res_t;

async_res_t async_print(
    const uint8_t *msg, size_t size) {
  return rand();
}
```

Rust
```rust
#[repr(C)]
enum AsyncRes {
  Success = 0,
  Failure = 1,
  Pending = 2,
}

extern "C" {
  fn async_print(msg: *const u8,
    size: usize) -> AsyncRes;
}
```

Listing 1: Rust bindings for a C enum type async_res_t and C function definition async_print.

Immediately, there is one oddity: the stubbed out `async_print` returns a value produced by `rand()`—*any* integer value—while our enum only has 3 variants. Surprisingly, this is valid C, as its enums are merely named integer constants. The rest of the code, meanwhile, seems solid: the `async_res_t` enum type has a one-to-one mapping from its C to Rust representation, and the `async_print` function binding accurately reflects the C function's signature.

Unfortunately, these bindings are nonetheless subtly incorrect and can lead to safety issues down the line. This is because C's enums work differently from Rust’s enum types: while in C enums are merely named integer constants, in Rust an enum type declaration creates a new type which is limited to a fixed set of values. This means that, while it is legal in C for an enum type to assume values that do not correspond to a declared variant of this enum, this is considered undefined behavior in Rust [10]. And interpreting the result of `rand()` as a value of the `enum AsyncRes` type may well produce such an invalid variant.

Meanwhile, avoiding undefined behavior is a worthwhile goal: Rust’s unsafe code guidelines state rather bluntly that, in the presence of undefined behavior, “the program produced by the compiler is essentially garbage” [11]. In fact, by extending the above example slightly, we can see how this violation of Rust’s restriction on valid values can result in a range of other safety issues. For this, consider the wrapper around the `async_print` foreign function of Listing 2.

```rust
enum PrintResult {
  Async(AsyncRes, &'static [u8]),
  Sync(CString),
}

fn print(msg: &'static [u8]) -> PrintResult {
  loop {
    let result = unsafe { async_print(msg.as_ptr(), msg.len()) };
    if let AsyncRes::Pending = result {
      continue; // keep polling
    } else {
      return PrintResult::Async(result, msg); // return Success or Failure
    }
  }
}

fn main() {
  println!("Print result: {:?}", print(b"Hello World!"));
}
```

Listing 2: A wrapper around the FFI bindings from Listing 1. Rust requires various invariants to be maintained for correct program behavior: for example, a type may only have a limited set of valid values. In this example we violate this type-safety invariant, which can in turn escalate into a violation of memory safety.

The wrapper function `print()` polls the `async_print()` function until it returns a value other than `PENDING`, and then returns the `AsyncResult` return value alongside the `msg` parameter in a new enum called `PrintResult`. This outer enum has two variants: it represents the result of an asynchronously printed a byte array, or synchronously printed null-terminated `CString` value.

Again, the snippet above seems unproblematic at first. Unfortunately, when paired with the violation of Rust's invariants on valid values above, it will perform an out-of-bounds memory access. This is due to an optimization called niche filling. When considering the underlying memory representation of the `enum PrintResult` type (assuming a 32 bit system), it holds the following components:

- a discriminant value indicating the enum's active variant (4 bytes),
- in case of `PrintResult::Async` being active, the `enum AsyncRes` value and a `&[u8]` slice pointer and length (12 bytes),
- and, in case of `PrintResult::Sync` being active, a pointer to a null-terminated `CString` allocated on the heap (4 bytes).

This means that the size of the PrintResult type should be 16 bytes. However, Rust knows that the only values an enum AsyncRes can assume are `0`, `1`, or `2`. Therefore, it can combine this field together with the outer enum's discriminant value and reduce the overall size of this type to 12 bytes.

Yet, when we break the assumption that the `AsyncRes` type will only contain values from `0` to `2`, the above optimization can cause the program to misbehave: for instance, assuming the call to `rand()` within the `async_print` function returned `3`, then Rust would store this value as the `PrintResult` type's discriminant. However, reading this value back, it would incorrectly assume that the `PrintResult::Sync` variant is active, and interpret the stored slice pointer as a pointer to a null-terminated C string, potentially reading other out-of-bounds data or experiencing a segmentation fault.

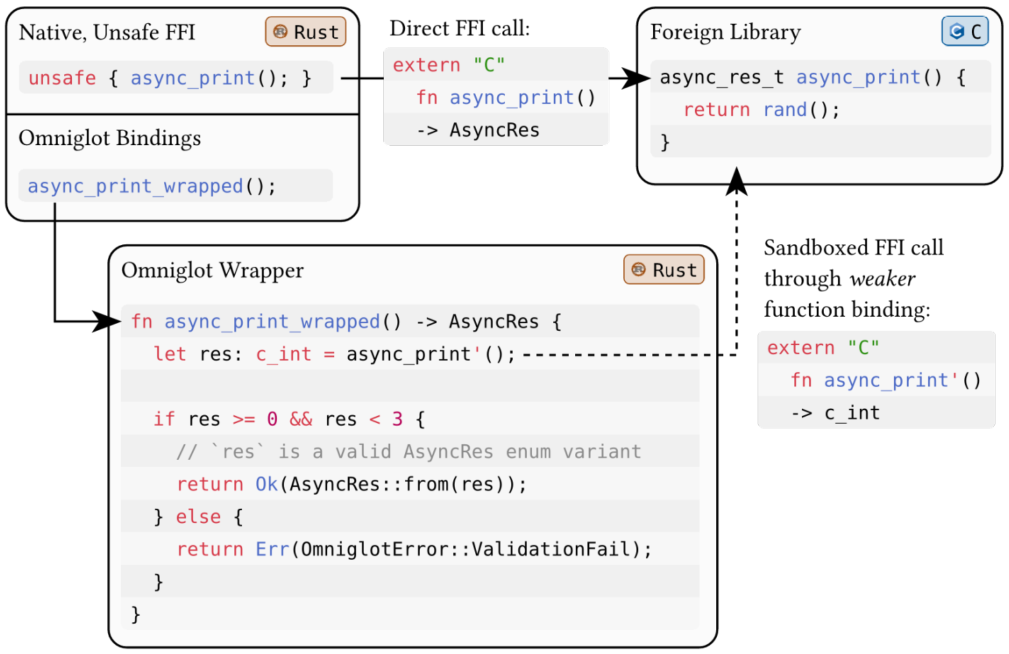
Notably, many existing approaches to safely interact with foreign or untrusted libraries would not prevent the above soundness violation: the out-of-bounds memory accesses occur from within the Rust domain itself! Even if a foreign library was prevented from accessing any of Rust's memory, the soundness violation could still occur. This is a violation of type safety, which has escalated to a violation of memory safety.

Next to valid values there are many more safety-critical invariants that Rust requires a developer to uphold: for instance, a defining feature of Rust is its concept of aliasing XOR mutability, which disallows any aliased references from being mutated. Similar to the invariants around valid values, these safety properties are difficult to reason about and maintain across complex interactions with foreign code. Thus, instead of considering all individual invariants at any point where Rust interacts with foreign code, we need a systematic approach to reason and maintain them.

When looking at the above example, the fundamental error—leading to the subsequent soundness violations and memory safety issues—was that we trusted the proclaimed type of the C function signature. And while, in this particular case, the fault lies within the bindings failing to accurately model the differences between C's and Rust's respective enum types (the appropriate type on Rust's side would have been a `c_int`), these issues are commonplace: for instance, even types as simple as booleans have different (or missing) definitions depending on the C standard used. And beyond that, there are few guarantees that a given foreign library is itself correct and adheres to all of its language's requirements.

Conventionally, developers using Rust's native, unsafe FFI will need to reason about the entire program and ensure that any Rust code, foreign code, and their composition is sound. While the Rust compiler validates soundness of safe Rust code, it cannot do so for interactions with foreign code. Omniglot does not model the entire program, and takes a different approach: instead of statically reasoning about the behavior of foreign code, it reduces the assumptions that Rust places on it, and employs runtime validations to ensure that any cross-language interactions do not break Rust's invariants. In this article, we want to provide an intuition for how this approach captures the soundness violations of the above example. Omniglot combines this method with other mechanisms to enable safe interactions with entirely untrusted foreign libraries; we encourage reading the paper for more details [9].

We illustrate Omniglot's approach in Figure 1: our goal is to provide a foreign function binding for the `async_print` function, returning an instance of the `enum AsyncRes` Rust type, but without the potential soundness violations shown in the previous sections. In this setting, we cannot prevent the C function from returning a value not part of its async_res_t enum definition. However, when this happens, it should result in an explicit error indicating that the C library violated its API contract, instead of introducing undefined behavior in the Rust host program.



```
unsafe { async_print(); }
```

```
extern "C"
  fn async_print()
  -> AsyncRes
```

```
async_res_t async_print() {
  return rand();
}
```

```
async_print_wrapped();
```

```
fn async_print_wrapped() -> AsyncRes {
  let res: c_int = async_print'();

  if res >= 0 && res < 3 {
    // `res` is a valid AsyncRes enum variant
    return Ok(AsyncRes::from(res));
  } else {
    return Err(OmniglotError::ValidationFail);
  }
}
```

```
extern "C"
  fn async_print'()
  -> c_int
```

Figure 1: Omniglot interposes on interactions between Rust and foreign code. In this example, Omniglot models a foreign function through a weaker function binding "fn async_print'() -> c_int" and restores the intended function return type through a runtime check in the wrapper "async_print_wrapped".

Instead of invoking a foreign function directly (illustrated in the top half of Figure 1), a developer uses Omniglot's modified version of the `rust-bindgen` utility to generate a set of *safe* foreign function bindings from a C header file, producing wappers like `async_print_wrapped` in Figure 1. These bindings—together with an Omniglot runtime library—internally invoke the foreign functions, surrounded by a range of runtime and static checks that catch and prevent potential violations of Rust's invariants before they can manifest in undefined behavior.

For the issue of valid values in particular, it uses a trick: instead of declaring the foreign function's signature faithfully—with the exact types that a developer would want to use—it declares another function symbol (`fn async_print'`) with a different, more permissive set of types. These types have the same size and layout constraints as their original counterparts, but carry fewer invariants that could be violated by foreign code. For instance, instead of representing C's `async_res_t` enum using the Rust `enum AsyncRes` type, we can represent it using a simple C integer (`c_int`). This type has identical size and alignment to that of the `enum AsyncRes`, but any bit-pattern represents a valid value of this type. This prevents foreign code from violating Rust's invariants by returning unexpected values.

Finally, the wrapper inspects and validates values returned by foreign code. If it corresponds to a valid value of the original return type, it converts the weaker `c_int` type back into an `enum AsyncRes`. And if it does not correspond to any `AsyncRes` variant, instead of introducing undefined behavior, the wrapper returns an error indicating that foreign code returned an unexpected value.

In addition to validating types, Omniglot also addresses Rust's invariants around memory safety and aliasing. For instance, it integrates with a memory isolation primitive (such as x86's Memory Protection Keys or the RISC-V Physical Memory Protection unit) to prevent buggy or malicious foreign libraries from writing to any of Rust's memory. And because Rust's borrow checker cannot reason about pointers passed through foreign code, we introduce a mechanism that upholds Rust's aliasing restrictions by statically reasoning about when references are allowed to read from or write to foreign memory. We explain these mechanisms in more detail in our paper [9].

Using the above approach, alongside other mechanisms that are responsible for maintaining Rust's invariants around memory safety and aliasing, Omniglot can provide developers with a safe API to interact with foreign libraries that is similar to Rust's native, unsafe FFI, without having to reason about the entire program.

In our paper, we evaluate Omniglot for use in both Linux userspace applications (making use of x86 Memory Protection Keys to isolate foreign libraries' memory), and for resource constrained embedded systems by integrating it into the Tock embedded OS kernel. We show that Omniglot works for a range of libraries libraries such as cryptography, compression, image decoding, file system and TCP/ IP networking.

While Omniglot does introduce some runtime overhead, we find that it performs comparably to existing memory isolation based approaches while delivering a stronger set of safety guarantees (adding 0% to 3.4% in overheads across our benchmarks). In addition, Omniglot can perform significantly better compared to previous approaches that utilize Inter-Process Communication (IPC) and serialization to exchange data to interact with untrusted components.

We will publish our research prototype of Omniglot in the coming weeks and will update this article once it becomes available.

Appendix

References:

[4] A. Levy et al., "Multiprogramming a 64kB Computer Safely and Efficiently," in Proceedings of the 26th Symposium on Operating Systems Principles, in SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 234–251. doi: 10.1145/3132747.3132786. Available: https://dl.acm.org/doi/10.1145/3132747.3132786

[5] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA: USENIX Association, 2016, pp. 203–216. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation...

[6] K. Boos, N. Liyanage, R. Ijaz, and L. Zhong, "Theseus: an Experiment in Operating System Structure and State Management,"

in 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), USENIX Association, 2020, pp. 1–19. [Online]. Available: https:// www.usenix.org/conference/osdi20/presentation/boos

[9] L. Schuermann, J. Toubes, T. Potyondy, P. Pannuto, M. Milano, and A. Levy, "Building Bridges: Safe Interactions with Foreign Languages through Omniglot," in 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25), Boston, MA: USENIX Association, 2025. [Online]. Available: https:// www.usenix.org/conference/osdi25/presentation/schuermann