How to achieve safety when linking separately compiled code

Version of August 7, 2025

Timen Zandbergen

How to achieve safety when linking separately compiled code

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Timen Zandbergen
born in Leuven, Belgium

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

How to achieve safety when linking separately compiled code

Author: Timen Zandbergen
Student id: 5002680
Email: t.zandbergen-2@student.tudelft.nl

Abstract

Abstract here.

Thesis Committee:

Chair:	Dr. S. Dumančić,	Faculty EEMCS, TU Delft
Daily Supervisor:	Dr. M. A. Costea,	Faculty EEMCS, TU Delft
Company Supervisor:	Ir. J. Dönszelmann	Hexcat
Company Supervisor:	M. Bos	Hexcat

Preface

Preface here.

Timen Zandbergen
Delft, the Netherlands
August 7, 2025

Brief Contents

Complete Contents

List of Figures

List of Tables

Introduction

Problem motivation; explain SOTA.

Rust offers a safe programming language with minimal run-time overhead. Safe programming languages have historically relied upon custom runtimes to uphold their safety guarantees. In contrast, Rust shifts the burden of enforcing safety to compile-time, benefiting both reliability and performance. By making memory management a compiler concern, Rust has made it easy to use libraries and modules written by others. The safety of combining the programs does, however, rely on compile-time analysis of the library code. This presents a problem for employing such components at run-time, when the compiler is no longer available to verify the program, and most information required for such verification has been erased. Previous work on safely linking separately compiled programs has relied upon a custom run-time verifier and rich annotations of the to-be-linked code. Such an approach is not acceptable to the Rust programming language, which desires to minimize the run-time burden and integrate with native system facilities. This thesis will explore this gap, and aims to provide, if not recommendations, an overview of tradeoffs within the design space of safely linking separately compiled code with minimal overhead.

Chapter 1

Background: A Rust primer

This chapter will explain the concepts of the Rust language that are germane to linking safety. Readers familiar with Rust are advised to skim through this chapter, as it disambiguates homographs used throughout this thesis. todo: Although this section introduces many aspects of Rust, it is not intended to be an introduction to the language; readers with a desire to learn Rust are referred to many of the excellent available resources.

link resources

Rust is both a safe and unsafe language; it adopts an extensible model of safety [JJKD18]. As a safe language, the compiler rejects programs for which it cannot guarantee that they do not exhibit undefined behavior. Because static analysis cannot admit every correct program [OHe20], an escape hatch is provided. For code covered by an `unsafe` keyword, the programmer is permitted some operations that may cause undefined behavior. The unsafe blocks can then be encapsulated such that the use of the abstraction is can be verified by the compiler [Jun16].

1.1 Types

Rust has an advanced type system, with various aspects that constitute a type. The Rust types can be discussed at three levels, syntactic, adorned, and semantic. A type at the syntactic level is the local definition. When a type is given additional decoration, such as trait implementations, we will call it equipped. Lastly, the meaning of a type beyond what the compiler can observe is its semantic form.

1.1.1 Syntactic types

Rust types can take many forms, and are constituted from parts [RRefe]. Some aspects of a type are only relevant during compilation, while others persist to run-time. The run-time relevant aspects of types must all be captured by the linker, but only some compile-time aspects are required to be preserved.

Rust has both nominal and structural types. Both of these terms have multiple meanings in the programming language field, so we will define them here for the rest of the thesis. A nominal type is a type defined by its "name"—the location and name given to the type. In this document, "nominal" will always refer to nominal typing, and not to the colloquial definition of "nominal". A structural type is a type defined by its "structure"—the names and types of its fields.[1] Both approaches have benefits and tradeoffs [MAV08], these considerations will be discussed where relevant.

[1] This use of structural is unrelated to a structural–substructural type system, which refers to the behavior of the type context with relation to exchange, weakening, and contraction.

Most Rust types are statically sized, and many uses of a type require a known size[RRefa]. Some types are dynamically sized (DST), where the total size is only known at run-time. A pointer to a DST is sized, and contains extra information on the pointee.

This section will go over every type, traits and generics are dealt with in a later part. The types are discussed in their relation to ABI and linking, as of writing only the C ABI is considered stable, but the stability guarantees of a potential `crabi` are also discussed. The rest of this section goes over all the Rust types in the structure that they are found in the Rust Reference[RRefe].

Primitive types

The primitive types are easy to deal with, as they will not change between different versions. Although some primitive types are ladened with soundness requirements, those will not change between versions of the language.

The four primitive type families are as follows:

Boolean: `bool`, only `0x00` and `0x01` are valid bit patterns.

Numeric: The integer (`i*`/`u*`) and float (`f32`/`f64`) types. All bit patterns are valid. The `isize` and `usize` types are always pointer sized on the current architecture, and can be considered ABI equivalent to their `i*`/`u*` equivalents.[2]

Textual: The `char` and `str` types. The instantiation of a `char` must be a valid Unicode scalar, but does not need to be a currently assigned codepoint, making it future-proof against newer Unicode versions. The `str` type is a slice, and has the same requirements as discussed later.

Never: The `!` type has no values and cannot be instantiated and indicates that a function will never return. As it cannot be inhabited, it does not exist at run time, a function that promises `!` should still not return when linked dynamically. The requirements are the same as an exhaustively matchable, empty enum.

Sequence types

Sequence types are structural. The layout of these types is not specified in the C ABI.[3] The safety requirements of sequence types are the same as their constituent types, given a stable layout.

Tuple: A heterogeneous list of arbitrary length. The empty tuple `()` (called unit) is the structural equivalent to a struct without fields.

Array: A homogenous list with a compile-time length.

Slice: A homogenous list with a dynamic length, see also the discussion on DSTs.

User-defined types

User-defined types have the same requirements as their constituent types, with the addition of potential user-specified restrictions placed upon private or `unsafe` fields.[4]

[2] The interesting case is the `x32` Linux ABI[rustc], but I suspect that a simple `i32`/`u32` is still sufficient here. TODO

[3] Although C has arrays, a C array corresponds neither to a Rust array (with a compile time fixed-length), nor a slice (which carries its length information at run-time).

[4] As unsafe fields are a recent (future) addition to the language, most types with user-defined safety requirements do not mark their fields as `unsafe`.

TODO: put user-defined restrictions here or in their own section to combine with unsafe functions?

Struct: The nominal product type. "Tuple structs" and structs with field names are equivalent for our purposes. A Rust struct is semantically analogous to a C struct, the `repr(c)` attribute makes them compatible.

Enum: The nominal sum type. An enum with no variants is a zero sized type (ZST) and equivalent to the never type. A Rust enum bears only a slight similarity to a C enum.

Union: The nominal undiscriminated union type. Accessing variants of this type is unsafe, as neither the compiler nor runtime can verify which variant is valid. The `repr(c)` attribute makes this ABI compatible with a C union.

Function types

TODO: write this
TODO: Include `unsafe` functions here or in a separate part?

Functions:

Closures:

Pointer types

Pointer types generally have the same safety considerations as their pointee type. A raw pointer does not guarantee that the pointee is valid, therefore dereferencing a raw pointer is an unsafe operation.

References: A reference points to an instantiated value according to the aliasing rules, see Section 1.2. References are guaranteed validity and liveness, and the pointee must be valid for its type.

Raw pointers: A raw pointer has no safety or liveness guarantees, but the use of a raw pointer has the same requirements as the existence of a reference.

Function pointers: See the discussion on functions above. A function pointer is equivalent to either a (non-async) non-capturing closure or a function item. A function pointer is not allowed to be null [nomicon].

Trait types

There are two ways that a trait becomes part of the layout of the struct. See Section 1.4 for an exploration of traits as a whole.

Trait objects: A `dyn` trait object is the only instance of dynamic typing in Rust. The trait object itself is a DST that contains only the `T` that implements the trait. Each instance of a trait object pointer includes: the pointer to the `T` and a pointer to the virtual method table (vtable) containing the trait methods. The vtable must be the vtable matching the pointee (dynamic) type and the (static) trait.

Impl trait: The `impl` keyword is syntactic sugar for generics (see Section 1.3). In a return position, it hides single a concrete type.[5] In a parameter position, it is a shorter form of specifying a generic bound.

[5] It does not permit dynamic typing, as every return path of the function must return the same concrete type.

1.1.2 Equipped types (traits)

Beyond its local definition, a type can be equipped with additional structure. A type can have functions nominally associated to it, often called methods. Another method of adding functionality is to implement a trait for a type. Although in most cases these can be equivocated to a bag of functions in a namespace, traits can carry important semantic information.

better term?

A trait is an abstract interface that can be implemented by a type [RRefb], analogous to Haskell typeclasses [Bra21]. Traits with no associated items to implement are called marker traits.

unsafe traits

1.1.3 Semantic types

1.2 Aliasing

Although Rust has not adopted an official aliasing model, the tree borrows[VHDJ25] model is considered authoritative. The tree borrows model supersedes the older stacked borrows model[JDKD20], as it allows for more real-world behaviors.

1.3 Generics

1.4 Traits

1.5 Soundness and Safety

1.5.1 Undefined Behavior / soundness

1.5.2 Compiler guaranteed soundness (safety)

1.5.3 User-defined safety requirements

Chapter 2

Background: The system linker

linkers are common

2.1 Object Files

2.2 Symbols

2.3 ?Loading and relocation

2.4 Prior Art

2.4.1 Unsafe: C/C++

2.4.2 Dynamic: Java/C sharp

2.4.3 Proof Carrying Code

2.4.4 Current Rust Solutions

Chapter 3

Soundness of Rust

Although there is neither a formal model nor an exhaustive enumeration of soundness in Rust, the Project does provide guidance on such matters for the purposes of writing unsafe code. The terms “unsoundness” and “undefined behavior” are both used to refer to behaviors explicitly disclaimed by the language. For the remainder of this thesis, the term “undefined behavior” will be avoided, and “unsound” will be used to refer to programs exhibiting such. An unsound program has no restrictions on the behaviors that can be exhibited. This should not be confused with “implementation-defined” or “unspecified” behavior, where the observable behavior may differ between implementations or platforms, but will not affect other parts of the program.

3.1 Why have unsoundness?

- hardware
- compiler optimizations
- formal logic
- behavior assumed to not occur

3.2 The `unsafe` keyword

The `unsafe` keyword allows for more operations, but not more behavior. A block of code enclosed by `unsafe` should never exhibit unsoundness. The permitted behavior is not altered if every statement were wrapped in its own `unsafe` rather than within one `unsafe` block. find citation

3.2.1 Permitted operations

The operations that are unsafe, and as such may cause unsoundness are as follows[nomicon].

- Dereference raw pointers
- Call unsafe functions (including C functions, compiler intrinsics, and the raw allocator)
- Implement unsafe traits
- Access or modify mutable statics

- Access fields of unions

When using these operations, the programmer must manually verify that the program is sound.

3.3 Unsoundness for Rust

The Rust Reference[RRefc] has a different list:

undefined.invalid Producing an invalid value. "Producing" a value happens any time a value is assigned to or read from a place, passed to a function/primitive operation or returned from a function/primitive operation.

undefined.runtime Violating assumptions of the Rust runtime. Most assumptions of the Rust runtime are currently not explicitly documented.

- For assumptions specifically related to unwinding, see the panic documentation.
- The runtime assumes that a Rust stack frame is not deallocated without executing destructors for local variables owned by the stack frame. This assumption can be violated by C functions like longjmp.

Justify use of Reference and Nomicon by arguing inadiquacy of existing academic resources.

Both the Rust Reference [RRefc] and the Rustonomicon [nomicon] have slightly different lists of sound behaviors. This section will discuss the behaviors found in either list that are relevant to linking.[1]

Dereferencing dangling or unaligned pointers. Storing into or reading from a pointer using $\star$ must manually ensure the pointers validity. This also prohibits invalid place-projections—*i.e.* field access, tuple indexing, and array/slice indexing. Unlike references, a raw pointer has no compiler verified guarantees. Unlike C, the pointer does not need to be valid, only its dereferencing. Relevant to linking because of unloading.

Find citation in C standard.

Breaking the pointer aliasing rules. Although Rust has not adopted an official aliasing model, the tree borrows[VHDJ25] model is considered authoritative. More information on aliasing can be found in Section 1.2. The linker may alias a global by resolving the symbol; see Section 5.1.

Calling a function with the wrong call ABI or unwinding from a function with the wrong unwind ABI. The Rust compiler speaks various **ABI!**s (**ABI!**s), these must match between the caller and callee. Moreover, not all **ABI!**s permit unwinding. Unwinding past a non-unwinding stack frame is unsound.

Having a data race. A data race is two or more operations, at least one of which is a write, and at least one of which is non-atomic. Fortunately, the system linker (on all considered systems) is thread-safe. The linking process must preserve the type-level infrastructure representing thread information. Rust uses the `send` and `sync` marker traits (see Section 1.4) and semantic types (see Section 1.1.3) to facilitate safe multi-threading. Global variables must also be shielded from data races; see also Section 5.1.

[1] Out of respect for the readers time, the behaviors have been pruned. The full lists can be found in the links in the citations.

Usually: unsupported `target_feature`. Unless the platform specifically documents otherwise, executing code compiled with features not supported by the current platform is unsound.

Producing an invalid value. Some primitive types have restrictions on their inhabitants; see Section 1.1.1. Uninitialized memory is considered an invalid inhabitant, even if the type permits any bit combination. The unstable `rustc_attrs` feature enables informing the compiler about custom inhabitation rules for the purposes of (niche) optimizations[unst]. Although only the previous is considered unsound by the compiler, Rust's extensible safety model[JJKD18] enables module authors to establish custom requirements not captured by the compiler, see Section 1.1.3.

Chapter 4

Unsound linker behavior

The system linker is dependent on the system on which the program is running. Despite such variation, most common systems have rather similar linkers; a fuller explanation is given in {background}. This section will only discuss the linkers of Windows, MacOS, and Unix variants, which covers all tier 1 and most tier 2 targets.[1]

The semantics of the system linker are modeled on the semantics of the C programming language, owing to their coevolution[bwk20, §4.7]. The limitations of the PDP-11 minicomputer required both the compilation and runtime to be as lightweight as possible. MS-DOS endured similar limitations, with a design partially based on Unix and engineered to run on the Intel 8086 microprocessor. Both systems later grew facilities for dynamic linking, which mostly mirrors the behavior of static linking. Another historical shift is that the system administration of the original Unix boxes was performed by the original Unix authors. Nowadays, most "system administration" is performed by end-users who have no desire to learn the intricacies of their system and simply want a working computer. As such, a programmer cannot make assumptions about the environment their program will execute in and must ensure that the program will execute reasonably on any system.

The simplicity of the system linker is both a boon and a bane. It is easy to predict the behavior of the system linker. Symbols are only matched by name (Windows dynamic linking also knows what binary the symbol originates from). The linker makes no distinction between function and variable symbols. The system guarantees that the linking itself is thread-safe. However, there is no way[2] to interpose on the linking process, preventing us from adding custom verification steps.

TODO: rest, mostly copied from extant notes

[1] Specifically, this selection includes all tier 2 targets with host tools and almost every tier 2 target with a system linker. [rustc]

[2] At least not in a widely supported manner; see {background} for an exploration of the topic.

Chapter 5

Sound linking of Rust

How and why can(’t) soundness be achieved using only the system linker?

In order to move linking of separately compiled programs into the domain of safe Rust, we must establish the origins and remedies of undefined behavior (UB). Such an assurance is twofold: the soundness-related assertions must persist, and the linking itself must not introduce new UB. Whenever multiple behaviors are safe/sound, the tradeoffs are explicated and the prior art is discussed.

This section will go over the causes of UB relevant to linking separately compiled code.

5.1 Namespace overlap

On Unix-derived systems, but not Windows, the dynamic linker is not aware of the expected provenance of a specific symbol. This forgetfullness is an intentional design decision [GLDW87], it aligns the behavior with static linking and enables library interposition. Although the order in which modules are searched is deternministic, there is no guarantee an earlier library does not export an overlapping symbol.

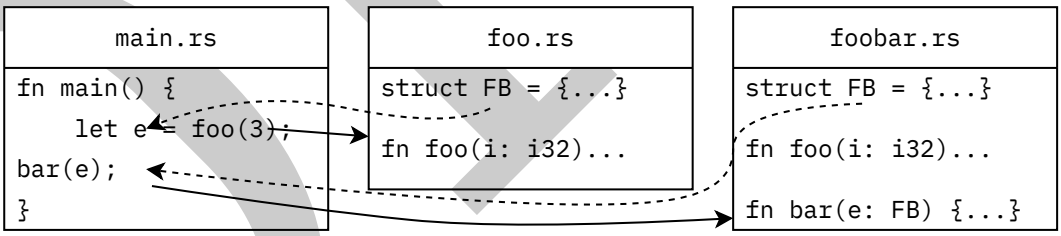

Figure 5.1: An example of a function being shadowed. In this case, the `bar` function will receive a FB from a different module, possibly causing unsoundness.

The `dlopen` facility allows modifying the search behavior, and the search is done into the handle.

5.2 Unloading

TODO

5.3 What must be ensured

5.3.1 The types of types

5.3.2 User-defined soundess

The discussion on so called “unsafe” types, like structs with private fields.

5.3.3 Traits

5.3.4 Generics

5.3.5 Lifetimes

5.3.6 Globals and thread locals

5.3.7 Unsafe contracts

Chapter 6

Use Cases

6.0.1 Possible using only symbol matching

6.0.2 Require extra features

Chapter 7

Extra features

The usecases that cannot be solved using only symbol matching.

7.1 ...

Chapter 8

Possibilities in sound linking

Solutions for sound linking that I would not recommend to the Rust Project.

8.1 Compiler

8.2 Random symbols

8.3 WASM

8.4 ...

Chapter 9

Recommended Solution(s)

The solution(s) that I would be willing to recommend to the Rust Project.

Chapter 10

Conclusion and Future Work

A one page conclusion of the thesis, and some future work recommendations (such as unloading)

Chapter 11

Conclusion

11.1 Recommendations to linkers

11.2 Recommendations to Rust libraries

11.3 Recommendations to plugin systems

Conclusion here.

References

[Bra21] Vitaly Bragilevsky. *Haskell in Depth*. Shelter Island, New York, United States: Manning Publications, 2021. 664 pp. isbn: 9781617295409. url: https://www.manning.com/books/haskell-in-depth (cit. on p. 6).

[bwk20] Brian W. Kernighan. *Unix: A History and a Memoir*. en. Independently Published, 2020. isbn: 978-1-6959-7855-3 (cit. on p. 13).

[GLDW87] Robert A Gingell, Meng Lee, Xuong T Dang, and Mary S Weeks. "Shared Libraries in SunOS". In: *The Australian UNIX* systems User Group Newsletter (AUUGN)* 8.5 (1987), pp. 112–127. url: https://books.google.nl/books?id=9TZvjK1cYJsC&lpg=PA112&ots=7QMUGGXglF&dq=Shared%20Libraries%20in%20SunOS&lr&pg=PA112#v=onepage&q&f=false (cit. on p. 15).

[JDKD20] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. "Stacked borrows: an aliasing model for Rust". en. In: 4 (POPL 2020), pp. 1–32. issn: 2475-1421. doi: 10.1145/3371109 (cit. on p. 6).

[JJKD18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: securing the foundations of the Rust programming language". In: 2 (POPL Jan. 2018), 66:1–66:34. doi: 10.1145/3158154 (cit. on pp. 3, 11).

[Jun16] Ralf Jung. *The Scope of Unsafe*. Jan. 9, 2016. url: https://www.ralfj.de/blog/2016/01/09/the-scope-of-unsafe.html (cit. on p. 3).

[MAV08] Donna Malayeri, Jonathan Aldrich, and Jan Vitek. "Integrating Nominal and Structural Subtyping". en. In: *ECOOP 2008 – Object-Oriented Programming* (2008), pp. 260–284. doi: 10.1007/978-3-540-70592-5_12 (cit. on p. 3).

[nomicon] The Rust Project. In: *The Rustonomicon*. 2025. Chap. What Unsafe Rust Can Do. url: https://doc.rust-lang.org/nightly/nomicon/what-unsafe-does.html (cit. on pp. 5, 9, 10).

[OHe20] Peter W. O'Hearn. "Incorrectness logic". In: *Proc. ACM Program. Lang.* 4.POPL (Jan. 2020). doi: 10.1145/3371078 (cit. on p. 3).

[RRefa] The Rust Project. In: *The Rust Reference*. 2025. Chap. Dynamically Sized Types. url: https://doc.rust-lang.org/reference/dynamically-sized-types.html (cit. on p. 4).

[RRefb] The Rust Project. In: *The Rust Reference*. 2025. Chap. Traits. url: https://doc.rust-lang.org/reference/items/traits.html (cit. on p. 6).

[RRefc] The Rust Project. In: *The Rust Reference*. 2025. Chap. Behavior considered undefined. url: https://doc.rust-lang.org/nightly/reference/behavior-considered-undefined.html (cit. on p. 10).

[RRefd] The Rust Project. *The Rust Reference*. 2025. url: https://doc.rust-lang.org/reference/introduction.html.

[RRefe] The Rust Project. “Types”. In: *The Rust Reference*. 2025. Chap. Types. url: https://doc.rust-lang.org/reference/types.html (cit. on pp. 3, 4).

[rustc] The Rust Project. In: *The rustc book*. 2025. Chap. Platform Support. url: https://doc.rust-lang.org/nightly/rustc/platform-support.html (cit. on pp. 4, 13).

[unst] The Rust Project. In: *The Rust Unstable Bookj*. For an example, see the attributes on the NonNull implementation https://doc.rust-lang.org/nightly/src/core/ptr/non_null.rs.html#75. 2025. Chap. rustc_attrs. url: https://doc.rust-lang.org/unstable-book/language-features/rustc-attrs.html (cit. on p. 11).

[VHDJ25] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. “Tree Borrows”. en. In: 9 (PLDI 2025), pp. 1019–1042. issn: 2475-1421. doi: 10.1145/3735592 (cit. on pp. 6, 10).

Acronyms

UB undefined behavior

LD linkage editor, linker, loader

DST dynamically sized type

ZST zero sized type

vtable virtual method table

Appendix A

Appendix here.