Aliasing Limits on Translating C to Safe Rust

MEHMET EMRE, University of San Francisco, USA
PETER BOYLAND, University of California Santa Barbara, USA
AESHA PAREKH, University of California Santa Barbara, USA
RYAN SCHROEDER, University of California Santa Barbara, USA
KYLE DEWEY, California State University Northridge, USA
BEN HARDEKOPF, University of California Santa Barbara, USA

The Rust language was created to provide safe low-level systems programming. There is both industrial and academic interest in the problem of (semi-)automatically translating C code to Rust in order to exploit Rust's safety guarantees. We study the effectiveness and limitations of existing techniques for automatically translating unsafe raw pointers (in Rust programs translated from C) into safe Rust references via ownership and lifetime inference. Our novel evaluation methodology enables our study to extend beyond prior studies, and to discover new information contradicting the conclusions of prior studies. We find that existing translation methods are severely limited by a lack of precision in the Rust compiler's safety checker, causing many safe pointer manipulations to be labeled as potentially unsafe. Leveraging this information, we propose methods for improving translation, based on encoding the results of a more precise analysis in a manner that is understandable to an unmodified Rust compiler. We implement one of our proposed methods, increasing the number of pointers that can be translated to safe Rust references by 75% over the baseline (from 12% to 21% of all pointers).

CCS Concepts: • **Software and its engineering** → *Software evolution*; *Source code generation*; **Software maintenance tools**; **Automated static analysis**.

Additional Key Words and Phrases: Rust, C, Translation, Memory Safety, Empirical Study

ACM Reference Format:
Mehmet Emre, Peter Boyland, Aesha Parekh, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2023. Aliasing Limits on Translating C to Safe Rust. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 94 (April 2023), 29 pages. https://doi.org/10.1145/3586046

1 INTRODUCTION

The Rust programming language targets the same low-level system software domain as C, but with much stronger memory and thread safety guarantees. Rust has been used for building operating systems, web browsers, and garbage collectors [Anderson et al. 2015; Levy et al. 2015; Lin et al. 2016], and is being adopted by projects with large C/C++ codebases (e.g., Firefox [Bryant 2016], Linux [Corbet 2021; Elhage 2020], and Android [Stoep and Hines 2021]). However, most existing critical software infrastructure predates Rust and is written in C, and thus suffers from C's lack of safety guarantees; this fact motivates the goal of (semi-)automatically porting C programs to Rust.

Authors' addresses: Mehmet Emre, memre@usfca.edu, University of San Francisco, San Francisco, CA, USA; Peter Boyland, boyland@ucsb.edu, University of California Santa Barbara, Santa Barbara, CA, USA; Aesha Parekh, aeshaparekh@ucsb.edu, University of California Santa Barbara, Santa Barbara, CA, USA; Ryan Schroeder, rschroeder@ucsb.edu, University of California Santa Barbara, Santa Barbara, CA, USA; Kyle Dewey, kyle.dewey@csun.edu, California State University Northridge, Northridge, CA, USA; Ben Hardekopf, benh@cs.ucsb.edu, University of California Santa Barbara, Santa Barbara, CA, USA.

2475-1421/2023/4-ART94
https://doi.org/10.1145/3586046

Existing C to Rust translators (e.g., `c2rust` [Immunant inc. 2020]) heavily rely on `unsafe` blocks, which disable safety checks for annotated code. To maximize safety, `unsafe` usage should be kept to a minimum. However, *fully* automated translation to *completely* safe Rust is difficult, if not impossible: Rust uses an ownership-based type system [Boyapati et al. 2002, 2003] to enforce memory and thread safety, and C programs are not usually written with ownership-based semantics. A more realistic goal, then, is (1) to maximize the amount of safe Rust code that is automatically translated from C and (2) to provide developers insight into the reasons why remaining code is marked unsafe, so that they can manually rewrite unsafe parts.

In this paper, we build on prior C to Rust translation work by focusing specifically on the translation of unsafe pointers to references. We observe that this translation's succcess is highly dependent on ownership and lifetime inference. **Our goal is to understand the current limitations of ownership and lifetime inference, in the context of translating unsafe *raw pointers* to verifiably safe references.** We offer new insights into the (lack of) effectiveness of existing translation techniques, demonstrate that lack of effectiveness with empirical evidence, and based on that evidence offer suggestions for overcoming these limitations. We also implement and evaluate one of our suggestions.

In Section 3, we introduce a new program tranformation technique called *pseudo-safety* which enables this entire study. The obstacle for prior studies is that pointers may be marked unsafe for multiple reasons unrelated to ownership and lifetimes, such as pointer arithmetic or unsafe casts. These confounding factors limit prior studies to only the small percentage of pointers that are not influenced by those confounding factors. Pseudo-safety removes these confounding unsafe behaviors from Rust programs, while preserving the static aliasing and lifetime relationships relevant to ownership and lifetime inference, though not the dynamic behavior of the original program. This transformation allows us to answer questions about the effectiveness of ownership and lifetime inference independently of other factors. Moreover, even if solutions to these confounding factors are found, these solutions will still be limited without improved lifetime and ownership inference methods. Thus, pseudo-safety allows us to evaluate and discover the limits of any technique for translating unsafe code to safe code. However, pseudo-safety *deliberately does not preserve the program semantics*, so it is intended as a tool for evaluation rather than a basis for a C to Rust translation tool.

In Section 4, we study the effectiveness of ownership and lifetime inference on pseudo-safe Rust programs translated from C, based on a previous, more limited study by Emre et al. [2021]. Our results show that, contrary to prior conclusions, the vast majority of raw pointers *cannot* be automatically made into safe references. This insight was possible because, unlike Emre et al., pseudo-safety allows us to extend our study to include pointers involved in other unsafe behaviors.

In Section 5, we investigate the reasons behind our results, and show that *type equality*, and more specifically the imprecision of Emre et al. [2021]'s interpretation of Rust's typechecking algorithm, is responsible: many pointers are put into type equivalence classes by their analysis, and if any one pointer in a class cannot be made safe, then none of them can. For example, in our largest benchmark (`tmux`), having only 4 pointers marked unsafe is enough to force 85% of the 4,635 pointers in the program to be marked unsafe. In Section 6, we investigate the effect of more precise analyses on the type equality problem, specifically equality-based (the baseline), subset-based, field-sensitive, and context-sensitive analyses. We show that field sensitivity does not substantially improve over the baseline, however subset-based and context-sensitive analyses each individually improve the baseline by an order of magnitude.

In Section 7, we propose methods for incorporating our findings into an improved C to Rust translation. Rather than modifying the Rust compiler to be more precise, which has major shortcomings, our proposed methods are based on program transformations and thus compiler-agnostic.

We implement one of our proposed methods in Section 8, wherein we encode the results of a subset-based taint analysis in a Rust program. We evaluate our transformation via pseudo-safety in Section 9, and show that it increases the number of unsafe pointers for which we can infer ownership and lifetime information by 75% over the baseline (an increase from 12% to 21%). We also release our implementation of the transformation, limit studies, benchmarks, and pseudo-safety transformation as an artifact for evaluation [Emre et al. 2023]. Future work involves implementing and evaluating the remaining proposed methods to determine their effectiveness.

2 BACKGROUND AND RELATED WORK

Rust. Rust's typesystem is based on ownership and borrowing, which statically guarantees memory safety [The Rust Project 2021]. Central to borrowing are *references*, which borrow some value for a fixed duration of time. References are identical to pointers in a dynamic semantics, but are statically augmented with additional *lifetime* information, which encodes how long the value the reference refers to is live. Lifetimes statically, conservatively, and soundly encode how long the object the reference references is live. Rust's *borrow checker* computes lifetime information and enforces two safety constraints on them: (1) it prohibits dereferencing references which outlive the objects they refer to, and (2) prohibits having multiple references to the same object if at least one of these references is mutable, and if the references' lifetimes overlap. While references statically guarantee memory- and thread-safety, their conservative nature means that safe code can still be rejected, leading to an occasional lack of expressiveness. Rust additionally supports *raw pointers*, or just *pointers* in this paper. Raw pointers are more expressive, as they lack lifetime information and allow dereferencing at any point. However, this dereferencing is inherently unsafe, so the compiler forces `unsafe` to be used around associated code.

Translating C to Rust. Citrus [Lesinski 2018], Corrode [Sharp 2020], and `c2rust` [Immunant inc. 2020] translate C to Rust using a purely syntactic translation scheme, and wrap the resulting translated code in `unsafe` annotations. In particular for our study, these tools translate all C pointers to unsafe Rust pointers, and thus do not tackle the issue of translating C to *safe* Rust.

Emre et al. [2021] classify and quantify the unsafety in Rust programs translated from C by `c2rust`. They find that raw pointer dereferences account for most uses of `unsafe`. They also clasify the causes of raw pointers into the following categories, based on the Rust reference [The Rust Project 2021] and their inspection of the code emitted by `c2rust`: casts (and void pointers), use in external APIs and unions, use in mutable global variables, inline assembly, pointer arithmetic, calling `malloc`, and a lack of ownership and lifetime information. We use this classification when devising pseudo-safe transformations. They then take the output of `c2rust`, as well as some semantic rules from Oxide [Weiss et al. 2020], and leverage the Rust compiler's type and borrow checkers to automatically infer ownership and lifetime information for a subset of the raw pointers, specifically those that have no other source of unsafety (e.g., pointer arithmetic, unsafe casts, global initialization, C-style unions, or being involved in an `extern` function call). We provide a summary of how the tool they build (Laertes) works at the end of this section. They find that a majority of these raw pointers can have their ownership and lifetime information inferred, and thus can be transformed into safe references. However, this subset of raw pointers turns out to be a small percentage of the total number of raw pointers (11% on average), because their study is limited by other causes of unsafe pointers affecting most of the pointers. We show in Section 4 that this subset is not representative of the whole, and that even if all other sources of unsafety can be removed, their technique is unable to infer ownership and lifetime information for the vast majority of raw pointers. We build directly on top of Emre et al.'s work.

Ling et al. [2022] take an alternative approach, wherein 220 ad-hoc rewrite rules are applied to `c2rust`'s output. These rules are intended to make the Rust code more idiomatic, and in so doing, remove some uses of `unsafe`. However, while they mention that 22 of these rules do not preserve program semantics, these 22 rules are not disclosed, nor do they indicate which behaviors are broken. Their evaluation indicates that over 95% of functions are made safe, but with a major caveat: a function marked safe may still call functions that execute unsafe code either directly or transitively, because they push the `unsafe` keyword into function bodies. As such, it is questionable how safe this generated code actually is; unsafety may simply be being pushed around in a manner which minimizes the use of the `unsafe` annotation, but without improving the inherent unsafety in the program. They do not provide any evaluation to this effect, though it is known that transitive calls can can hide lots of unsafe behavior [Emre et al. 2021].

Enforcing Pointer Safety in C.. Necula et al. [2005] extend C to introduce *safe* pointers in CCured. If a pointer cannot statically be deemed safe by an equality-based analysis, it is instead checked dynamically. Elliott et al. [2018] introduce casts backed by possible runtime checks in Checked C, where the casts assert related information like nullability or array bounding. Machiry et al. [2022] improve on these works by using casts to enforce boundaries between safe and unsafe pointers. Checked C and works based on it do not enforce complete memory safety and are not ownership-based, thus they are not directly applicable to the goal of translating C to safe Rust; they do not compose with Rust's existing ownership-based memory model (necessitating modification to Rust itself), and they do not completely enforce memory safety. Cyclone [Jim et al. 2002] is an extension to C that guarantees memory safety by static verification of liveness conditions using region-based memory management, and by dynamic bounds checking. Cyclone's approach is similar to how Rust maintains memory safety using its ownership and lifetime system to ensure reference validity, and bounds checking to validate array accesses.

Pointer Analysis. We use pointer analyses to determine how unsafety spreads across the program. Some seminal work on flow-insensitive pointer analysis is most relevant to us, along with related improvements. Andersen [1994] presents a reduction from subset-based flow-insensitive points-to analysis to an iterative fixpoint problem based on the transitive closure of a graph. Under Andersen's analysis, the points-to set of a pointer p subsumes the points-to sets of all values assigned to p, independent of the program's control flow, hence it is flow-insensitive. Steensgaard [1996] presents a faster (in almost linear time) but imprecise pointer analysis based on type equality. As mathematical equality lacks a direction, Steensgaard's analysis interprets `p := q` and `q := p` as the same constraint, whereas Andersen's analysis distinguishes between these two by deriving $\text{ptsto}(p) \supseteq \text{ptsto}(q)$ for the former and the flipped version for the latter. Here, ptsto denotes a mapping from pointers to their points-to sets, so it is equality-based. Rust's type checker effectively performs a Steeensgaard-style analysis to propagate unsafety among pointers.

Pearce et al. [2007] present a way to encode function arguments and parameters in a subset constraint system that can express Andersen-style and Steensgaard-style analyses. In Pearce et al.'s encoding, an n-ary function f is represented with a constructor $\lambda(p_1, \ldots, p_n, r) \supseteq f$, where the variables $p_1, \ldots, p_n$ denote the parameters of the function (hence are contravariant), and r denotes the return value of the function (hence is covariant). Each call site `r = f(a1, ..., an)` also corresponds to a constructor $\lambda(a_1, \ldots, a_n, r) \subseteq f$ where $a_1, \ldots, a_n$ are the arguments at the call site, and r is the location for the return value at the call site. We make use of Pearce's encoding to handle function pointers in our pseudo-safety transformation.

Methods of Evaluating Pointer Analyses. Existing work comparing pointer analyses with different levels of precision is split between two approaches: direct measures such as points-to set sizes,

and the effectiveness of a client analysis. Here, we present representative works that use either method. Shapiro and Horwitz [1997a,b] use average and total points-to set sizes to compare an Andersen-style analysis to a Steensgaard-style analysis. Shapiro and Horwitz [1997a] use the effectiveness of several client analyses (program slicing, program dependency graph construction, liveness analysis) to evaluate pointer analysis sensitivity. Kastrinis and Smaragdakis [2013] use average points-to set sizes to measure the effect of context sensitivity. Guyer and Lin [2003] measure effectiveness in terms of the number of alarms raised by a client vulnerability analysis. Kastrinis and Smaragdakis [2013]; Lhoták and Hendren [2006]; Smaragdakis et al. [2014] use call graph construction and downcasting safety as client analyses when evaluating the effectiveness of different context-sensitive analyses.

A Brief Explanation of Laertes's Core Algorithm. Laertes [Emre et al. 2021] is a tool that infers lifetime and ownership information by iteratively querying the Rust compiler and resolving compiler errors. It operates on pointers that are unsafe only due to a lack of lifetime information (see our discussion of Emre et al. [2021] on page 3 for other causes of unsafety in pointers that they consider). Laertes starts with optimistic assumptions about all pointers it operates on: (1) all pointers can be converted to safe references, (2) all references are borrowing, and (3) Every lifetime in function and type signatures is independent (has a different type variable). Laertes rewrites the pointers to references under these assumptions. Then, it invokes the Rust compiler as an oracle to invalidate the incorrect assumptions in a loop:

$pointerInfo \leftarrow [lifetimeFacts : \emptyset, owning : \emptyset, raw : \emptyset]$
$program \leftarrow$ `rewriteProgram`$(pointerInfo, originalProgram)$
$errors \leftarrow$ `runRustCompiler`$(originalProgram)$
while $errors \neq \emptyset$ **do**
  $fixes \leftarrow$ `resolveErrors`$(errors)$
  $pointerInfo \leftarrow$ `propagateFixes`$(pointerInfo, fixes)$
  $program \leftarrow$ `rewriteProgram`$(pointerInfo, originalProgram)$
  $errors \leftarrow$ `runRustCompiler`$(program)$

`resolveErrors` takes the compiler errors, and applies heuristics to determine whether to make some pointers unsafe (raw pointers) or owning (`Box<T>`), and whether to derive new lifetime facts. `propagateFixes` performs a taint analysis to propagate the unsafety and ownership information to affected pointers; Laertes implements this by modeling the spread of unsafety using an equality-based (Steensgaard-style) analysis, and the spread of ownership using a subset-based (Andersen-style) analysis. These analyses emulate how unsafe and owned pointers spread through the type system, and are based on the semantic rules from Oxide [Weiss et al. 2020] that Emre et al. use. We give examples of the ramifications of this analysis further in Section 7. *pointerInfo* holds the pointer unsafety, ownership, and lifetime facts. `rewriteProgram` rewrites the program using these facts. Eventually, the loop terminates; worst-case, all pointers are made unsafe, yielding the original program.

3 INTRODUCING PSEUDO-SAFETY

Rust supports two mechanisms to refer to memory: safe references and unsafe raw pointers. All C pointers are translated to raw pointers by `c2rust`, and raw pointers can only be dereferenced in `unsafe` code. Laertes [Emre et al. 2021] takes `c2rust`-translated code, and attempts to automatically transform pointers into safe references via ownership and lifetime inference. However, Laertes is inherently restricted to raw pointers that are unsafe exclusively due to a lack of ownership and lifetime information, excluding those involved in pointer arithmetic, unsafe casts, and other confounding factors (we cover the list of causes Emre et al. consider in Section 2). This limits

Laertes' applicability to only around 11% of pointers. Pointers marked unsafe for other reasons are *also* unsafe due to lack of ownership and lifetime information, that is, even if those other factors are removed these raw pointers would still need something like Laertes to be transformed into safe references. It is an open question how well ownership and lifetime information can be inferred for all raw pointers, rather than just the small subset that Laertes can handle.

In order to answer this question we have developed a technique called *pseudo-safety*[1]. The idea is to rewrite a Rust program (translated from C via `c2rust`) to replace unsafe pointer behaviors with substitutes that preserve the static pointer relationships relevant to Rust's type and borrow checkers, but not the runtime behavior of the program itself. In other words, we simulate fixing all other causes of pointer unsafety in order to focus on the question of inferring ownership and lifetimes. In the rest of this section we detail the program properties that we preserve and describe the rewrites that handle each extraneous cause of unsafety. All rewrites in this section are implemented with `c2rust`'s refactoring tools and Laertes.

Properties to Preserve. Rust's safety checks hinge on object lifetimes and aliasing, and our rewrites preserve the data flow information of three related program properties: aliasing, object lifetimes, and provenance. Our rewrites remove unsafety related to pointers but not necessarily other causes of unsafety (e.g., global mutable variable access), in order to keep our rewrites minimal. These rewrites preserve the number of unsafe pointer declarations and dereferences, and so they do not fundamentally change the program from the perspective of pointer use.

More specifically, for a given expression e, we rewrite it to e' and preserve the following properties as checked by the Rust compiler:

- The lifetime of the result of e' is same as the lifetime of the result of e.
- If an object is used in the computation of e, it is also used in the computation of e', preserving the uses of objects at each program point.
- If the result of e is a pointer, it points to the same memory region as the result of e'. For this purpose, we use the borrow checker's informal notion of regions where each heap allocation, variable, and field belongs to a separate region, and members of an array belong to the same region. We consider all pointers derived from a pointer p using pointer arithmetic to point to the memory region that p points to. So, aliasing and provenance of objects are preserved.

We ensured these properties for each rewrite manually, and are planning to release our explanation for how each specific rewrite preserves these properties as part of our artifact.

Although we preserve these properties, we do not preserve the dynamic behavior of the program, that is *pseudo-safety is not expected to produce a program with the same run-time behavior*, and it is intended only as a limit study tool.

Rewriting Pointer Arithmetic. Any pointer subjected to pointer arithmetic must necessarily be raw. Pointer arithmetic is performed by `arr.offset(i)`, which is equivalent to the C expression `arr + i`. To preserve the properties in Section 3, we translate the unsafe expression `arr.offset(i)` to the safe block of expressions `{i; arr}` (i.e., compute `i` then compute `arr`, returning the value of `arr`). While the dynamic semantics are different (and computation of `i` can be optimized out by the compiler), this block still performs the computation of both `arr` and `i` so their original static lifetimes are preserved (at least up until borrow checking, which happens before any optimization stage). Similarly, the result of this expression still depends on `arr`, so aliasing and origin point information is preserved. The array offsets coalesce into a single reference after our transformation,

[1]We call our technique pseudo-safety because it replaces unsafe pointer uses for reasons other than lack of lifetimes/ownership (hence safety) with a substitute that is not semantically equivalent but preserves the data flow properties we are interested in (hence pseudo-).

and none of the analyses we consider are array index-sensitive. So, we do not need to reconstruct something like `&arr[i]`.

We also consider the related expressions of pointer difference and array-to-pointer conversion. Pointer difference is performed by `a.offset_from(b)`, equivalent to C's `a - b`. We rewrite this as the expression block `{a; b; 0}`, which maintains that both `a` and `b` are used and must be alive. We use `0` as a substitute for the actual difference between the two pointers, because the value does not affect any compile-time lifetime properties. We rewrite array-to-pointer conversion as returning a pointer to the first element of the array. For example, `arr.as_mut_ptr()` (which returns a pointer to `arr`) is rewritten as `(&mut arr[0] as * mut T)`, where `T` is the element type of the array. While this rewrite loses precision for array index-sensitive analyses, array index-insensitive analyses (such as the ones we use here) maintain their precision.

Stubs for External Functions. The Rust compiler cannot reason about external function definitions, hence pointers passed to and returned from such functions give problems for ownership and lifetime inference. We replace each external function with a function implemented in Rust. Per the behavior of the C linker, we want to preserve having a single function definition for all external functions with the same name. As such, we generate an empty stub for each uniquely-named external function declaration. The body of the stub contains an infinite loop, which has the bottom type in Rust—this fact allows us to accommodate any return type. For example, we generate the following stub for C's `memcpy` function:

```
pub unsafe extern "C" fn memcpy(* mut c_void, * mut c_void, size_t) -> *
    mut c_void { loop {} }
```

The stub has only `loop {}` in the body to allow any return type, optimistically assuming that the actual externally declared function's signature matches what the analysis derives. We preserve the `extern "C"` linkage, but it is not used by Laertes. As no code in the body links the parameters to the return type, the stub can be rewritten by Laertes to use references in a manner consistent with all of the function's uses. This scheme gives us the most optimistic possible rewrites for unsafe pointers interacting with external functions.

Rewriting Casts with Lifetime-Preserving Substitutes. Raw pointers can be freely cast between different types. However, Rust does not permit casting between references. Removing casts altogether would alter the provenance of some pointers, since casts establish new pointers. To simulate casts in safe code, we rewrite casts between unrelated types into calls to a function `pseudocast` that we define as:

```
pub fn pseudocast<'a,'b:'a,T:'b,U:'a>(_:T) -> U { loop {} }
```

The lifetime annotations of `pseudocast` specify that the lifetime of the function output is contained within the lifetime of the function input, thereby preserving the relevant properties we care about. As a caveat, if there is a cast from a non-pointer to a pointer we rewrite the cast but do not track the provenance of the non-pointer leading into the cast.

Rewriting Global Variable Initializers. Safe Rust code does not permit global variable initializers to create heap-allocated values. We rewrite unsafe global initializers into global assignments contained in newly created public functions that are never called (which does not affect the flow-insensitive analyses performed by the rewrite tools and the compiler) and instead initialize global variables with default values (all of the types in the programs translated from C can be default-initialized). This scheme is similar to rewriting global variables to be initialized with commonly used APIs such as `lazy_static` [The Rust Project 2022a] or `OnceCell` [The Rust Project 2022b], which perform

thread-safe lazy initialization. We generate new public functions instead of using these APIs in order to generate code that is simpler and easier to analyze.

Rewriting Unions to Structs. C-style unions allow type punning in an unsafe manner and also lose pointer provenance for their members. We rewrite C-style unsafe unions into structs, and rewrite each union initializer to initialize the other struct members to default values. This rewrite breaks the expected runtime behavior of programs that use unions for type punning or physical subtyping, but it preserves the properties outlined above.

Inline Assembly. C programmers use inline assembly code to implement low-level optimizations or to access hardware capabilities. In order to simulate a translation of inline assembly to safe code, we treat inline assembly regions as unique functions that take all associated variables by reference. For example, if we have an inline assembly region `llvm_asm!(... : "r" a, : "=r" b)` that has `a` as an input and `b` as an output, we create a new function `fn f(&mut a:A, &mut b:B) {}` where `A` and `B` are the types of `a` and `b` respectively, then we rewrite the inline assembly region into the function call `f(&mut a, &mut b)`. This rewrite allows us to preserve the constraint that these variables need to be accessed mutably at this point of the program while not constraining the exact semantics of the inline assembly code.

Limitations. Pseudo-safety emulates only low-level rewrites that do not change data flow facts between pointers in the program. It does not account for potential translation schemes that might perform higher-level transformation (e.g. creating shims for functions, using a different API for external functions, or eliminating global variables). Specifically, we do not consider (1) using Rust libraries with different conventions to replace external functions; (2) introducing locks or synchronization mechanisms to guard global variables; or (3) reorganizing the program to abide by the lifetime restrictions that C programmers do not care about (e.g., using a variable after moving its value to another variable). This is not a limitation in terms of making a program safe, but a limitation in terms of not modelling some potential higher-level code transformations.

Our method also has limitations around handling function pointers that hold values coming from sources with different lifetime parameters. We extend Laertes to use lambda constructors [Pearce et al. 2007] in its pointer analyses in order to support function pointers. Function pointer types in Rust do not encode lifetime constraints; e.g. we cannot have a function type with `where`, such as in `fn <'a, 'b>(&'a i8) -> &'b i8 where 'a : 'b`. As a result, our method does not handle cases where functions with different lifetime constraints flow into the same function pointer. We encountered this problem in two of the benchmarks used by Emre et al. [2021] (`optipng` and `snudown`), and so we exclude them in our evaluation.

Pseudo-safety transformation on an example. Fig. 1 shows how pseudo-safety transformations are used in a code snippet taken from one of our corpus programs (`libcsv`). The irrelevant parts of the function definition are pruned for the sake of brevity. Fig. 1a shows the code snippet before the transformations. The `csv_fwrite2` function uses 3 causes of unsafe pointers besides lack of lifetime information (each of the causes is marked in bold):

(1) It calls `fputc` (an external function declared on line 2) and passes it a pointer.
(2) The expression `csrc.offset(1)` on line 18 performs pointer arithmetic.
(3) There is an unchecked type cast from `* const c_void` to `* const u8` on line 10. This cast cannot be converted to a cast between references.

By applying the pseudo-safety transformations, we removed these causes of unsafe pointers while maintaining pointer provenance (modulo array indexing), yielding the code snippet in Fig. 1b. Each pointer use was addressed by:

```
1  extern "C" {
2    fn fputc(_: i32, _: *mut FILE)
3      -> i32;
4  }
5  unsafe fn csv_fwrite2(
6   fp: *mut FILE,
7   src: *const c_void,
8   mut src_size: usize) -> i32 {
9   let mut csrc =
10   src as *const u8;
11
12  while src_size != 0 {
13   if fputc(*csrc as i32,fp)==-1{
14    return -1;
15   }
16   src_size =
17    src_size.wrapping_sub(1);
18   csrc = csrc.offset(1);
19  }
20  return 0 as i32;
21 }
```

(a) Code before the pseudo-safety rewrites.

```
fn fputc(_: i32, _: *mut FILE)
  -> i32 { loop { } }

unsafe fn csv_fwrite2(
 fp: *mut FILE,
 src: *const c_void,
 mut src_size: usize) -> i32 {
 let mut csrc =
   pseudocast<*const c_void,
              *const u8>(src);
 while src_size != 0 {
  if fputc(*csrc as i32,fp)==-1{
   return -1;
  }
  src_size =
   src_size.wrapping_sub(1);
  csrc = { (1); csrc };
 }
 return 0 as i32;
}
```

(b) The code after the pseudo-safety rewrites.

Fig. 1. An example code snippet from the `libcsv` benchmark before and after applying the pseudo-safety transformations.

(1) Converting fputc to a function stub with a body that doesn't compute anything (line 2), so it is no longer externally-defined.
(2) Converting the pointer arithmetic to an expression that evaluates the index then returns the source pointer (line 18).
(3) Converting the cast to a call to our `pseudocast` function which can be rewritten to a call involving reference types rather than pointers (line 10).

The code after the transformations (Fig. 1b contains no causes of unsafe pointers other than lack of lifetime and ownership information. Now we can use the snippet above to evaluate lifetime inference methods such as the one in Laertes, and see if they can infer proper lifetimes for `src` under the assumption that other causes of unsafe pointers can be removed through local transformations.

4 EVALUATING LAERTES IN THE LIMIT

In this section we evaluate Laertes from Emre et al. [2021] on a set of `c2rust`-translated programs which had our pseudo-safety transformations applied. Laertes attempts to automatically transform raw pointers into references for those raw pointers whose unsafety depends solely on the lack of ownership and lifetime information, and it is the most advanced method for doing so in the current state of the art. Since pseudo-safety guarantees that these are the *only* possible reasons for pointer unsafety, Laertes can theoretically handle all raw pointers in our benchmarks (unlike the original study by Emre et al. [2021] that could only handle ~11% of raw pointers). Our research

question is: **RQ1: How many pointers can Laertes make safe when all raw pointers are made eligible for transformation via pseudo-safety?**

Experiment Setup. We use 14 of the 17 programs used by Emre et al. [2021] in our evaluation, as shown in Table 1. We omit programs `optipng` and `snudown` because of the limitation outlined in Section 3, and we omit `libxml2` because Laertes times out on `libxml2` when handling all pointers.

We apply the transformations described in Section 3 after the ResolveImports[2] step of Laertes. We use the result of this phase as our baseline. Then, we run the main step of Laertes (called ResolveLifetimes by Emre et al. [2021]) that uses the compiler as an oracle to derive ownership and lifetime information.

Experiment Results. Table 1 shows the number of eligible raw pointer declarations (i.e., those that Laertes can handle) and the number of raw pointer declarations that Laertes transforms into safe references, both before and after our pseudo-safety transformations. We also measured the number of raw pointer dereferences; the trends are similar, and so we omit them for space. Emre et al. [2021] report that eligible raw pointers (those with ownership and lifetime as the only cause of unsafety) are only 11% of the total raw pointers, and we confirm this result with our own experiment (we report a slightly different figure of 9.5% because we only use a subset of Laertes' benchmarks). Using pseudo-safety to make all raw pointers eligible increases the number of eligible raw pointer declarations by an average of 10.5×. We also see that while 93% of eligible raw pointer declarations can be made safe before pseudo-safety is applied (consistent with the results reported by Emre et al. [2021]), only 12% of eligible raw pointer declarations can be made safe afterwards. This means that the vast majority of raw pointer declarations cannot be inferred by Laertes, even if all confounding factors are removed (i.e., the cause of unsafety is limited to ownership and lifetime information). We investigate why in the next section.

5 TYPE EQUALITY AS A VECTOR FOR UNSAFETY

In this section, we investigate why Laertes is unable to transform a significant number of raw pointers into safe references. Laertes's view of the Rust typechecker is equality-based, meaning that raw pointers are placed into type equivalence classes; if any raw pointer in a given equivalence class is marked `unsafe` (e.g., because ownership and/or lifetime cannot be inferred for it), all raw pointers in the same class must also be marked `unsafe`, even if Laertes can infer ownership and lifetime information for them. We conjecture that this *unsafety tainting* effect is the culprit behind Laertes' lack of success. To test this conjecture, we measure statistics for the raw pointer equivalence classes in our pseudo-safe benchmarks.

Fig. 2a shows the relative sizes of the equivalence classes in each benchmark. Each column is a benchmark, a column contains one mark for each raw pointer type equivalence class, and the placement of a mark on the y-axis indicates the percentage of raw pointers that are in that type equivalence class. Hence, low marks are small type equivalence classes and high marks are large type equivalence classes. Having large type equivalence classes means that the unsafety of one raw pointer can easily spread to many other raw pointers.

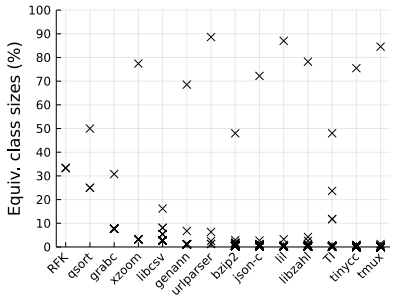
We observe that in all benchmarks except `grabc` and `xzoom` there is a single equivalence class that affects more than 45% of the raw pointers. Moreover, the four largest equivalence classes account for 85% of the raw pointers in all benchmarks; this means that having only four necessarily unsafe [3] raw pointers is enough to poison 85% of the total raw pointers in the worst case. Moreover,

[2]ResolveImports step merges duplicate struct and external function declarations in the Rust code that result from the same header being included in separate translation units in the original C code.

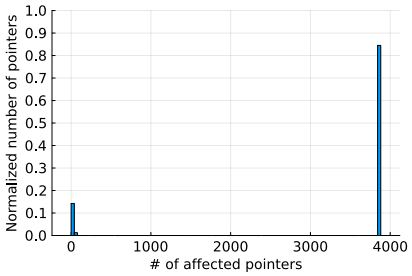
[3]That is, a pointer that is directly used in an unsafe operation, as discussed by [Emre et al. 2021]. This is in comparison to pointers that have to be unsafe (a raw pointer) because they have data flow from/to "necessarily unsafe" pointers.

Table 1. Benchmark programs ordered by Rust lines of code along with raw pointer declarations before and after our transformations. We report both the number of eligible pointer declarations, and the declarations made safe by the ResolveLifetimes pass of Laertes [Emre et al. 2021].

Benchmark	Lines of code		Before pseudo-safety			After pseudo-safety		
	C	Rust	Eligible	Made safe	%	Eligible	Made safe	%
qsort	27	39	2	2	100%	4	2	50%
libcsv	1,035	951	18	18	100%	37	25	68%
grabc	224	994	5	5	100%	13	8	62%
urlparser	440	1,114	5	5	100%	79	7	9%
RFK	838	1,415	0	0	–	2	2	100%
genann	642	2,119	0	0	–	73	12	16%
xzoom	776	2,409	0	0	–	29	3	10%
lil	3,555	5,367	23	8	35%	438	34	8%
json-c	6,933	8,430	29	23	79%	325	72	22%
libzahl	5,743	10,896	7	7	100%	457	51	11%
bzip2	5,831	14,011	37	21	57%	227	78	34%
TI	4,643	19,910	0	0	–	866	18	2%
tinycc	46,878	62,569	184	143	78%	1352	207	15%
tmux	41,425	191,964	331	232	70%	4635	468	10%
Total	118,990	322,188	641	464	72%	8537	987	12%



(a) Size of each equivalence class of declared variables as a percentage of the sum of the sizes of equivalence classes for each benchmark. The benchmarks are ordered in terms of total number of variables.



(b) Histogram of pointers in `tmux` (our largest benchmark). The x axis denotes the number of pointers affected, and the y axis denotes the number of instigating pointers in each bin divided by the total number of pointers in the program.

Fig. 2. Results of our investigation on aliasing caused by type equality.

all benchmarks have at most 77 equivalence classes containing more than 1 pointer. From these measurements we see that the underlying issue is type equality, or more precisely, the imprecision of how Laertes models the Rust typechecker using an equality-based analysis.

In the rest of the paper we investigate the effects of increasing the precision of analyses modeling the typechecker. However, this means going beyond equality-based analysis and therefore means that raw pointers will no longer be grouped into equivalence classes. We need a metric for measuring the impact of unsafety tainting that is independent of equivalence classes. The metric we will use is a histogram that conveys how easily unsafety taint can be spread among raw pointers. For each raw pointer R we count the number of other raw pointers whose safety depends on that of R (i.e.,

the number of raw pointers that will necessarily be marked unsafe if R is marked unsafe); call R the *instigating* raw pointer and call the raw pointers whose safety is dependent on R the *affected* raw pointers. We then plot a histogram where the x-axis indicates the number of affected raw pointers and the y-axis indicates how many instigating raw pointers affect that many other raw pointers. Note that "instigating" vs. "affected" are just terms of convenience: the metric looks at every raw pointer as a potential instigator and any raw pointer may be affected by some other raw pointer.

Fig. 2b demonstrates this new metric on our largest benchmark, `tmux`. We see that, for example, over 80% of instigating raw pointers affect more than 3,500 other raw pointers. That is, if any of those 80% of raw pointers are marked unsafe, then necessarily at least 3,500 other raw pointers will also be marked unsafe.

6 INVESTIGATING ANALYSIS PRECISION

In the previous section we showed that imprecision in Laertes's model of the type system is responsible for allowing necessarily unsafe raw pointers to spread unsafety to many other raw pointers. In this section, we perform a limit study to investigate the effects of adding different kinds of precision in order to determine what kinds of analyses, if any, can mitigate this problem.

The baseline Rust typechecking analysis is equivalent to an equality-based (Steensgaard-style), field-based[4], context-insensitive pointer analysis. We explore three dimensions of precision to make the analysis more precise: context sensitivity, field sensitivity, and directionality (i.e., going from an equality-based analysis to a subset-based analysis). To implement these analyses we build on the pointer analyses in the SVF [Sui and Xue 2016] framework. SVF analyzes LLVM bitcode, so we first compile Rust programs to LLVM bitcode with all optimizations and overflow checks disabled to get a program that is as close as possible to the high-level Rust IR (HIR) that Laertes analyzes. However, there are three important differences between LLVM bitcode and HIR:

- LLVM bitcode is in static single assignment (SSA) form, which creates different versions of local variables, thus enables Andersen-style (subset-based, field-insensitive) analyses to have strong updates for local variables a la flow-sensitive analysis. However, Rust HIR can be transformed to SSA form by a tool so the results still apply.
- LLVM bitcode uses offsets instead of field names when accessing struct fields, which can cause precision loss for field-based and field-sensitive analyses. To remedy this problem, we use the type and field index information computed by SVF to restore some missing type information.
- HIR code is parametric polymorphic but LLVM code is monomorphized. This is not an issue in our experiments, as the code translated from C does not use generics.

We implement a flow-insensitive dataflow analysis client that uses SVF to build a dataflow graph containing all top-level pointers (i.e., each global and local pointer variable). We build four versions of this graph using different levels of sensitivity, each building on top of the previous:

P1 A field-based, equality-based, and context-insensitive analysis, as a baseline model of the Rust typechecker. We build an undirected data flow graph and merge all nodes that access the same field (by analyzing `GetElementPointer` instructions), even on different objects.

P2 A *field-sensitive*, equality-based, context-insensitive analysis. This is similar to the prior analysis, but does not merge field access nodes.

P3 A *field-sensitive*, *subset-based*, context-insensitive analysis. This adds a directional data flow graph to the prior level of precision.

[4]A field-based analysis treats all accesses to the same struct field as accessing the same object [Heintze and Tardieu 2001]. This is in contrast to a field-sensitive analysis. See Section 7.2 for why we consider Rust's type system to be field-based.

P4 A *field-sensitive*, *subset-based*, *context-sensitive* analysis. We do the same analysis as the prior level, but we use pairs of call contexts and program locations as nodes in the data flow graph.

We use four levels of pointer analyses from SVF to build the P1–P4 client analyses. The SVF equality-based pointer analysis is Steensgaard-style, the SVF subset-based pointer analysis is Andersen-style, and the SVF context-sensitive pointer analysis is actually a demand-driven flow- and context-sensitive analysis (SVF does not allow for a flow-insensitive, context-sensitive pointer analysis; note that the P4 client analysis built on top of SVF is still flow-insensitive). The context-sensitivity strategy uses the immediate caller of the current function being analyzed as the context. As most of our benchmarks are libraries, we pick all externally visible functions (all functions marked `pub` in the Rust code) as program entry points.

Experiment Setup. We analyze 16 out of 17 benchmarks used in Emre et al. [2021]. We do not use the `libxml2` benchmark because the context-sensitive analysis times out after 48 hours[5]. All of our experiments are run on a Intel i7-6600k processor with 32 GiB of memory running Void Linux. The context-sensitive analysis on `tmux` used 27 GiB of memory and took 25 hours; the same analysis for `tinycc` used 2.8 GiB of memory and took 10 minutes. All other experiments used < 2 GiB of memory and took less than 5 minutes.

To answer how analysis precision impacts the spread of unsafety, we use the metric described in Section 4 that looks at each raw pointer as a potential *instigator* of unsafety and how many other raw pointers it would force to be unsafe if the instigator is marked unsafe. The smaller the number of the affected pointers per potential instigator, the better the analysis curbs the spread of unsafety. We are using "the number of affected pointers" as a proxy for the main metric we are interested in: the spread of unsafety to other pointers. Existing methods to measure relative precision of pointer analyses discussed in Section 2 are not helpful to us, so we do not focus on them in our evaluation.

We pose the following research question: **RQ2: How does the distribution of affected pointers change with analysis precision?** We are interested in how many more pointers can be proven as "well-contained" by increasing analysis precision. That is, how many raw pointers do not affect many other pointers in the program, when considered as potential instigators. To answer RQ2, we collect two types of data: (1) how many pointers are "well-contained" (which we define as affecting at most 1% of the program's raw pointers), and (2) summary statistics (mean and standard deviation) of the distribution of affected pointer set sizes for each benchmark and precision level. We also graphically display this distribution in the style of Fig. 2b.

The context-sensitive analysis considers instances of a pointer in different contexts as separate pointers, so its results are not directly comparable to the results obtained by context-insensitive analyses. However, in order to meaningfully compare the sizes of the affected pointer sets across different precisions we count affected pointers only once no matter how many contexts they appear in. This raises the question of how to count affected pointers that may appear in the instigator's affected set in some contexts but not in others: counting it as affected may be too conservative, but counting it as not affected may be too optimistic. We compute the metric twice, once under the conservative assumption and once under the optimistic assumption, and report both results.

Results. Table 2 presents well-containedness data for each level of analysis precision. We observe that adding field sensitivity (**P2**) to an equality-based analysis does not significantly improve precision; the highest increase observed is 4% (in `optipng`). However, adding directionality (**P3**) causes a sudden jump in precision, with the subset-based analysis having more than 90% of the pointers affect less than 1% of the pointers in 12 out of 16 benchmarks. `qsort` is an outlier because

[5]We repeated this experiment also on a computer with 128 GiB memory and an AMD EPYC 7281 processor and it timed out after a week (168 hours).

Table 2. Number of total pointers, and percentage of pointers affecting ≤ 1% of pointers. **ptrs** is the number of pointer-typed variables in the dataflow graph, and **cptrs** is the total number of clones of all raw pointers in all contexts. The remaining columns refer to instigator pointers affecting ≤ 1% pointers under each analysis precision. **some-ctx** and **all-ctx** count a pointer as affected only if it is affected under at least one or all call contexts, respectively. Results > 90% are marked in bold.

Benchmark	ptrs	cptrs	Percentage of pointers affecting <1% of the pointers				
			P1	P2	P3	P4:some-ctx	P4:all-ctx
qsort	16	109	0.00 %	0.00 %	37.50 %	55.05 %	55.96 %
grabc	1,850	5,076	35.89 %	38.76 %	**92.16 %**	**98.29 %**	**99.57 %**
libcsv	429	1,051	15.15 %	17.25 %	81.82 %	**95.91 %**	**96.96 %**
urlparser	1,067	3,767	0.84 %	1.03 %	83.32 %	**96.81 %**	**97.29 %**
xzoom	2,584	7,823	31.27 %	33.36 %	**92.92 %**	**97.95 %**	**98.65 %**
robotfindskitten	3,095	2,438	26.20 %	28.72 %	**95.32 %**	**97.70 %**	100.00 %
snudown	3,728	8,621	17.95 %	19.21 %	**98.85 %**	**99.22 %**	**99.99 %**
genann	4,108	23,299	25.46 %	27.14 %	**93.65 %**	**98.09 %**	**98.31 %**
libzahl	5,441	34,304	4.43 %	4.52 %	**94.16 %**	**95.20 %**	**95.40 %**
json-c	5,598	19,856	10.29 %	11.97 %	**92.69 %**	**96.12 %**	**96.25 %**
lil	6,144	21,544	14.71 %	16.03 %	**91.78 %**	**96.03 %**	**96.62 %**
tulipindicators	8,376	20,859	13.98 %	14.77 %	**93.04 %**	**99.85 %**	**100.00 %**
bzip2	11,909	37,037	17.03 %	17.65 %	**96.84 %**	**98.52 %**	**98.54 %**
optipng	21,984	77,638	20.37 %	24.46 %	**96.77 %**	**99.61 %**	**99.63 %**
tinycc	28,830	264,811	14.09 %	17.97 %	**93.54 %**	**95.03 %**	**98.35 %**
tmux	77,345	258,662	6.73 %	7.87 %	88.86 %	**93.05 %**	**93.05 %**

it has only 12 pointers. Adding context sensitivity (**P4**) shows that 94% (97% excluding `qsort`) of the instigator raw pointers affect less than 1% of the total raw pointers on average (geometric mean). We also observe that if an unsafe pointer can affect one clone of another pointer, the unsafe pointer can likely affect all clones of it. So, an analysis that incorporates all three aspects of precision is crucial for taming unsafety, and it can potentially help identify the remaining "lynchpin" pointers that spread unsafety even after drastic automatic program transformations.

To get a complete picture, we also look at overall changes in the distributions of affected raw pointers. We present statistics for the distribution of affected pointers in Table 3. The results of this experiment agree with the results of our previous analysis of "well-contained" pointers: Both the mean and the standard deviation of this metric shrink as sensitivity increases, and field-sensitivity improves the metrics only by a small amount. In the best-case scenario (an analysis with full precision, and assuming that another pointer is not reachable unless it is reachable in all contexts), a random pointer does not affect more than 181 other pointers, or merely 20 when excluding `tmux`.

While summary statistics give a general understanding of the distribution, they may be misleading [Anscombe 1973]. So, we also investigate the change in the shape of this distribution graphically. Fig. 3 presents this distribution for each precision level on `tmux`. For space reasons we present only the results for our largest benchmark; other benchmarks have similar distributions. Ideally, we would like to see all pointers binned on the leftmost column, indicating that instigator pointers generally do not affect many other pointers. As the precision increases, we see a trend towards the left. At one extreme, ≈ 90% of the pointers can affect almost all pointers under **P1**. With **P3** only 10% of the pointers can affect more than 80% of the pointers. Finally, with **P4** only 2% of the pointers can affect more than half of the pointers. So, our results suggest that an analysis with the sensitivity choices of **P3** is useful. The jump from **P2** to **P3** *suggests* that adding directionality is

Table 3. Summary statistics for number of affected pointers for each instigator pointer, classified by benchmark and analysis precision. μ is the mean and σ is the standard deviation. **some-ctx** and **all-ctx** count a pointer as affected only if it is affected under at least one or all call contexts, respectively.

Program	P1		P2		P3		P4:some-ctx		P4:all-ctx	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
qsort	7.4	3.9	7.4	3.9	2.2	3.3	0.7	1.2	0.7	1.0
grabc	708.3	553.3	544.8	489.8	10.8	40.6	2.0	6.7	1.3	3.6
libcsv	63.8	49.9	55.1	43.9	3.0	5.3	1.2	2.0	1.1	1.6
urlparser	1049.1	96.7	1044.1	106.5	8.0	18.1	2.0	5.2	1.6	3.2
xzoom	1144.8	810.1	958.1	758.5	9.4	37.5	2.1	7.1	1.6	5.1
RFK	1641.9	1000.5	613.5	440.4	7.7	35.0	2.0	14.2	0.5	2.2
snudown	2497.4	1269.9	2048.8	1191.9	3.9	37.2	1.7	11.0	0.8	2.2
genann	2285.8	1331.5	1871.4	1275.3	17.4	85.9	9.0	53.9	4.3	20.2
libzahl	5149.6	1107.6	4959.4	1078.1	126.2	651.8	76.5	342.9	5.5	19.7
json-c	3389.3	1669.3	3029.9	1646.3	95.7	425.3	31.6	158.1	7.7	34.3
lil	4355.0	1889.3	4062.0	1940.2	155.5	616.4	28.5	140.3	3.6	13.6
TI	5560.5	2629.1	5246.6	2676.6	37.2	189.7	1.9	19.3	1.1	3.0
bzip2	7989.3	3754.3	7660.9	3799.0	159.2	1005.7	76.4	628.8	9.3	70.9
optipng	16371.8	8268.0	12193.6	7119.8	66.9	602.6	11.5	141.9	4.6	42.8
tinycc	21997.8	8903.6	19402.5	9070.5	912.9	3552.1	172.6	996.9	20.2	122.7
tmux	69663.3	18706.5	65644.2	19188.8	5478.3	16058.1	2481.0	9068.2	181.0	656.5

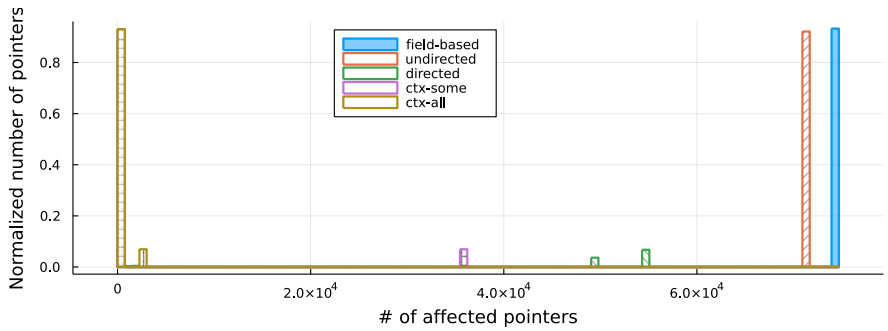


Fig. 3. Histogram of pointers in `tmux`. The x axis denotes the number of pointers affected, and the y axis denotes the normalized ratio of pointers in the program in the bin. We report two separate figures for context-sensitive analysis: **some-ctx** and **all-ctx** count a pointer as affected only if it is affected under at least one or all call contexts, respectively.

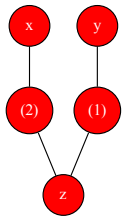
a good starting point to reduce the number of affected pointers found by an analysis. However, there are several confounding variables we need to control before making a stronger claim: our experiments work on LLVM bitcode which have far more intermediate variables that can affect the distribution, (2) the bitcode is in SSA form which makes directional analyses more precise, and (3) that we do not test on a field-based Andersen-style analysis in isolation. We conduct a second experiment controlling only directionality to answer these in Section 9.

7 CURBING THE SPREAD OF UNSAFETY

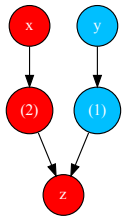
We have shown that more precise analysis can curb the spread of unsafety when attempting to transform raw pointers into safe references. However, it is not feasible to make substantial changes

```
let x: * const i8 =/*unsafe*/;
let y: * const i8 =/*safe*/;
let z: * const i8;
// ...
z = y; // (1)
// ...
z = x; // (2)
```

(a) Rust snippet wherein unsafety spreads from x to y through z.



(b) The undirected DFG for the snippet in Fig. 4a.



(c) The directed DFG for the code snippet in Fig. 4a.

Fig. 4. A short code snippet and the data flow graphs for it computed by the directional and the undirectional analyses. In both cases, we assume that x is used unsafely, so it is the root cause for all other unsafe pointers in this example. Red indicates unsafe, and blue safe.

to the Rust typechecker in order to support translating C programs into safe Rust. To put our insights into practice, we must develop methods to gain the benefits of more precise analysis without modifying the Rust compiler. We suggest several program transformations that would allow for more precise reasoning by the Rust typechecker without compiler modification; these transformations mimic the effects of field-sensitive, directional (subset-based), and context-sensitive analyses. We implement one of these transformations (inserting casts as proposed in Section 7.1), and leave implementing and evaluating the other two transformations for future work.

7.1 Casting References to Pointers to Introduce Directionality

Rust's type checker performs an equality-based analysis, meaning that information can flow in both directions on an assignment; effectively, information is propagated *backwards* as well as forwards. Fig. 4a provides an example showing how this spreads unsafety. In this example, x's right-hand side is assumed to be inherently unsafe, but y's right-hand size is considered safe. When the typechecker analyzes this code snippet, it computes the data flow graph (DFG) shown in Fig. 4b. The nodes marked 1 and 2 correspond to the expressions on the right-hand sides of lines 5 and 6 (marked (1) and (2)), respectively. The DFG is undirected, so unsafety flows both from x to z, and from z to y. In this manner, unsafety spreads to y, despite the fact that y is never used unsafely. If we were to use a directional data flow analysis, we would get the directed DFG in Fig. 4c. In this graph, unsafe and safe nodes are marked red and blue, respectively. Unsafety flows from x to z, but not backward from z to x. So, we can deduce that y is safe when using a directional analysis. If we are to rewrite the types in Fig. 4a using the results of the directional analysis (Fig. 4c), we get the code snippet in Fig. 5a. This code snippet would be well-typed if the typechecker were to use a directional analysis and automatic conversion from references to pointers, but since it is equality-based, we instead get a type error on the assignment `z = y;`.

We can address this problem by inserting a cast from a reference to a pointer, shown in Fig. 5b. Variable y is now declared as a reference, and a cast is added on line marked with (1)–the call to `as_ptr()`. The cast tells the compiler that the type of node (1) and z do not need to be the same, effectively removing the edge between these nodes. So, the new undirected DFG the typechecker effectively computes (Fig. 5c) separates the equivalence classes (1) and z belong to. The compiler derives that y and node (1) have the same type, which is distinct from the types of x, node (1) and z. In this manner, cast prevents the spread of unsafety from x to y.

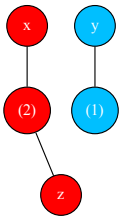
To determine where to insert casts, we use the results of the directional (subset-based) data flow analysis to find the points in the program where data flows *immediately* from a safe location to an

```
let x: *const i8 =/*unsafe*/;
let y: & i8 =/*safe*/;
let z: *const i8;
// ...
z = y; // (1) type error
// ...
z = x; // (2)
```

(a) The code snippet in Fig. 4a, using the types computed using a directional analysis (Fig. 4c).

```
let x: *const i8 =/*unsafe*/;
let y: & i8 =/*safe*/;
let z: *const i8;
// ...
z = y.as_ptr(); // (1)
// ...
z = x; // (2)
```

(b) The code snippet in Fig. 4a, but with a reference-to-pointer cast inserted to make y a safe reference.



(c) The *undirected* DFG for the snippet in Fig. 5b.

Fig. 5. Variations of the code snippet in Fig. 4a with (1) types computed using a directional analysis (Fig. 4c), and (2) a cast to make y as safe, along with the DFG as computed by the typechecker after insterting the cast. Red indicates unsafe, and blue safe. x is the only pointer used unsafely, so it is the source of all unsafety.

```
struct Foo<'a>{bar: &'a i8;}

let z1 : Foo = ...;
let z2 : Foo = ...;
z1.bar = x;
y = z2.bar;
```

```
struct Foo<'a>{bar: &'a i8;}
struct Foo2{bar: * const i8;}
let z1 : Foo2 = ...;
let z2 : Foo = ...;
z1.bar = x;
y = z2.bar;
```

Fig. 6. A code snippet before and after duplicating Foo to emulate field sensitivity.

unsafe one. In the DFG computed by our directional analysis (Fig. 4c), node (1) is now marked safe, but it immediately flows to into an unsafe location (z). At this position, we insert a cast around node (1) to encode the results of the directional analysis. This results in the code snippet in Fig. 5b. As a result, y can now be used as a safe reference until it is passed to z.

Although the directional analysis seems deceptively straightforward, we are interested in building an analysis whose results can be encoded with reference-to-pointer casts (i.e., by calling `as_ptr()`) while not introducing any undefined behavior resulting from using pointers and references in conjunction. We describe both the details of this analysis and how we use its results to transform the program in Section 8.

7.2 Duplicating Struct Definitions for Field Sensitivity

Rust's type system is field-based, with a single assigned type for each field of each struct type, thus merging the types of different instances of the same field into a single type. For example, the leftmost snippet in Fig. 6 shows that although there is no data flow between x and y, any unsafety in x forces y to become unsafe because of field-based analysis; x flows into `z1.bar`, so if x is unsafe, the bar field of Foo becomes unsafe, causing `z2.bar` to be unsafe, making y unsafe. A field-sensitive analysis could distinguish between `z1.bar` and `z2.bar`. To get the effect of a field-sensitive analysis without a field-sensitive analysis, we can define separate struct types for each abstract object or each combination of struct fields, as shown on the right of Fig. 6. Since `z1` and `z2` now have different types, Rust determines that `z1.bar` and `z2.bar` are unrelated.

7.3 Duplicating Functions to Introduce Context Sensitivity

If a single call to a given function uses a raw pointer argument, then all calls must use a raw pointer argument, potentially spreading unsafety to other call sites. The most direct way to solve this

problem is function cloning, i.e., introduce a different version of the same function for each call site, each specialized for its particular use (mimicking a context-sensitive analysis). However, there are several observations and challenges for implementing such an idea that a future implementation would need to take into account: (1) We do not need to duplicate a function for each call site, but rather for each combination of pointer, box, and reference arguments. (2) To keep program size manageable, only a small number of functions should be duplicated, but in a way that maximizes how much of the program is safe. (3) Programs with function pointers require additional bookkeeping, which is exacerbated by additional complexity from the prior item.

So, an effective method that duplicates functions requires developing heuristics on which functions to duplicate, and exploring the tension between maintainability and safety.

Another challenge left for future work is cloning function pointers for different function signatures. One possible approach is to perform defunctionalization, and to dispatch the correct function with the correct signature at each call site. Defunctionalization could also overcome the limitation of pseudo-safety regarding functions with different signatures. However, it would need a precise enough call graph construction to have only the relevant functions considered at each call site.

8 ENCODING DIRECTIONAL ANALYSIS RESULTS USING CASTS

In this section, we elaborate on our proposal to encode the results of a directional analysis by inserting casts (Section 7.1), and show how we realize this proposal. We first describe a sound analysis that captures the spread of unsafety when inserting top-level casts (calls to `as_ptr()`), and then describe how we use the results of this analysis to insert casts in general.

8.1 A Type-Safe Directional Data Flow Analysis

For our analysis, we focus on the following issues: (1) capturing immediate data flow from each expression so we can accurately insert casts, (2) spreading unsafety so that there are no type mismatches when we consider only casts between top-level pointer types, and (3) ensuring that adding casts do not introduce any new undefined behavior. For (1), we must access the locations to which an expression immediately flows, ignoring transitive flows. The initial DFG captures most of these flows, except for when the expression flows to a pointer dereference or call site. We insert placeholder nodes in these cases, so that we can use the initial DFG while consulting the final analysis results when we need to resolve pointer dereferences and call sites. For (2), we must capture all immediate data flow from each expression, and limit the precision of our analysis to capture the effect of inserting only top-level casts. For (3), we must reason about and avoid pointer–reference aliasing, and we piggyback on the Rust compiler to do so. We first explain our analysis in general with an example (Section 8.1.1), and then discuss how we resolve (3) in Section 8.1.2.

We formulate our data flow analysis in terms of set constraint-based program analysis [Aiken 1999]. In our notation for the subset (directional) constraints, a constructor $c(x_1, \ldots, x_n; y_1, \ldots, y_n)$ is covariant over the arguments $x_1, \ldots, x_n$ and it is contravariant over $y_1, \ldots, y_n$. The two kinds of constructors we are concerned with are $\text{ref}(x; x)$ (a reference to x) and $\lambda(r; p_1, \ldots, p_n)$ (denoting a function with parameters $p_1, \ldots, p_n$ and return location r). Both are from Pearce et al. [2007]. From here, we make the following modifications to a typical constraint-based analysis:

(1) We unify all pointees of a pointer, effectively switching to an equality-based analysis for inner pointer types, as our method inserts casts only at the top level. For example, we do not insert a cast from the type `* mut & mut T` to `* mut * mut T`. This change can be expressed in the language of set constraint systems [Aiken 1999] as $ref(x) \subseteq ref(y) \implies x = y$ (meaning that ref is an invariant constructor).

(2) We reason about all function pointers using an equality-based analysis, again because we cannot support casts between function types soundly; a function expecting a `& mut T` would expect exclusive access, while a function expecting a `* mut T` does not. Conversely, a function expecting `* mut T` may require transfer of ownership, while a function expecting `& mut T` does not. So, neither function's type can be cast to the other's, and there is no supertype of both that we can use. This change can be expressed by treating Pearce et al. [2007]'s λ constructors as invariant, similar to how we make points-to set constructors invariant in (1).
(3) We insert special nodes to represent (1) function parameters, (2) points-to sets, and (3) declared variables to act as intermediaries to guarantee that each DFG node that corresponds to an AST node has only one successor in the DFG. Here, (1) and (2) are pseudo-locations described later in this section. Declared variable nodes (3) were already in Laertes [Emre et al. 2021].

Once we perform the data flow analysis, we propagate the unsafety information along the data flow edges by solving the set constraint system, so that all uses of an unsafe pointer also become unsafe. We then use this unsafety information when querying unsafety of the nodes in the original graph, before solving the set constraint system.

Fig. 7 shows the types of program locations l (i.e., DFG nodes) in our analysis. These locations are a standard representation for field-based, context- and control flow-insensitive data flow analysis [Pearce et al. 2007]. They represent actual program locations, or fresh logic variables needed to fill in the inputs/outputs of public APIs. We extend the data flow analysis used in Laertes for propagating ownership to use these locations. The free locations f are used to create a structure that mimics the type structure of function and nested pointer types in the program. They are used by Laertes to represent nested type structure (e.g., nested pointers), where an explicit location for the inner locations (e.g., the pointee of a nested pointer) do not exist in the program.

Pseudo-locations p are not necessary for the analysis *per se*, but they allow us to structure the graph such that every node derived from an expression has a single successor in the data flow graph before transitive closure. We want this property to correctly identify the immediate data flows from each expression so that we can determine whether an expression's value is immediately used as an unsafe pointer. There are two cases where we do not know this before the analysis, and so we insert pseudo-locations as placeholders we can query later for these cases:

- If an expression is immediately put into a dereference, we insert a points-to pseudo-location **pointsto** p as its successor, where p is the pointer being dereferenced. E.g., when handling an assignment `*p = q`, we add the edge $q \rightarrow$ **pointsto** p to the DFG, so that q has a single successor in the initial DFG.
- If an expression is used as an argument in a function call, we insert a parameter pseudo-location **param** $f\ i$ as its successor, where f is the location of the callee (which may be unknown until the analysis is complete), and i is the position of the current expression in the argument list. E.g., when handling a function call `f(x)`, we insert the edge $x \rightarrow$ **param** $f\ 0$ indicating that x immediately flows into the first parameter of f.

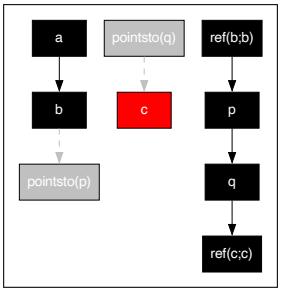
8.1.1 Example Involving Pseudo-locations. Fig. 8 shows an example where subset-based data flow analyses lack the information about *immediate data flow into a pointer dereference*, and how pseudo-locations can solve this problem. It shows a program and its directional DFGs, skipping expression nodes from Section 8 for brevity. The directional analysis completely ignores the nodes and edges in gray. These nodes correspond to pseudo-locations, and they are there for the client to query immediate flows when inserting casts. The analysis starts with the graph on Fig. 8b and computes its transitive closure, adding the edge $b \rightarrow c$ when discovering the edge $ref(b;b) \rightarrow ref(c;c)$. This continues until the fixpoint DFG in Fig. 8c is reached. Now, suppose `*q` is inherently unsafe, but `a`

$$
\begin{array}{rll}
l \in \text{Location} ::= & x \in \text{Var} & \text{Program variables} \\
& \mid e \in \text{Expr} & \text{Expressions} \\
& \mid f \in \text{Free} & \text{Free location variables} \\
p \in \text{PseudoLoc} ::= & \textbf{pointsto}\, l & \text{Points-to sets} \\
& \mid \textbf{param}\, l\, i & \text{Parameters}
\end{array}
$$

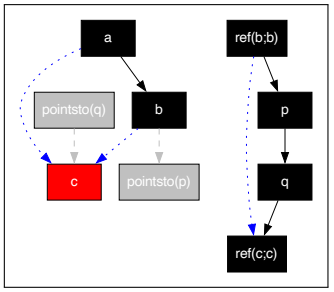
Fig. 7. Locations and pseudo-locations used in our data flow analysis. Expressions and program variables in Location come from the definition of variables in Emre et al. [2021]. Program variables correspond to variables and function names in the program, and free location variables are generated by the analysis as needed.

```
let a: *mut i8 = ...;
let b: *mut i8 = a;
let p: *mut *mut i8 = &b;
let q: *mut *mut i8 = p;
let c = *q;
// use *q unsafely
```

(a) A code snippet illustrating immediate and non-immediate flows.



(b) Initial DFG.



(c) Final DFG.

Fig. 8. A code snippet on the left, and the DFGs before and after running a directional data flow analysis. Edges added by the analysis are dotted and marked in blue. Pseudo-nodes are marked in gray, and edges to/from them are dashed and marked in gray. c is the only unsafe pointer node, so only it is marked in red.

and b are not. Our goal is to infer that b immediately flows into an unsafe location (because *q is used unsafely) so that we can correctly identify b as the location to insert a cast around. We need to determine whether to insert a cast around the use of a or b when using the method described in Section 8.2. In the initial data flow graph, there is no inherently unsafe successor of a or b (**pointsto** p is not inherently unsafe at this stage, the analysis has not solved the set constraints yet). If we are to use it, we would opt not to insert a cast around a or b, causing a type error: there needs to be a cast on this path because a is safe whereas c and *q are not. If we can query the information that the points-to set of q (or p) contains an unsafe pointer, then we could correctly insert a cast around b. The final graph has this information, though it also includes the transtive closure. So, it would be incorrect to use because it has an edge from a to c indicating that we should insert a cast around a. If we have a version of the initial graph where we have placeholders for points-to sets that we can query, we can have the information we need from each stage of the analysis: immediate data flows from the initial graph, and what the points-to sets resolve to from the final analysis result. Our DFG construction inserts the pseudo-nodes (marked in gray) to solve this problem: When we are interested in the locations a node immediately flows into, we can query its successors in the initial graph, and when we get a pseudo-node we resolve it using the final graph. With this strategy, we now notice that the successor of b (**pointsto** p) may be unsafe because it may be c.

8.1.2 Pointer–Reference Aliasing Woes. Rust has strict aliasing rules for references that the compiler relies on for memory safety and optimization, whereas pointers do not have any aliasing restrictions checked by the compiler. If we were to cast a reference to a pointer, we may not use both the pointer

```
let x : * const i8 = ...;
let y : Box<i8> = ...;
let z : * const i8;
// ...
z = y.as_ptr(); // (1)
// ...
z = x; // (2)
```

Fig. 9. The code snippet in Fig. 4a with a reference-to-pointer cast, along with the ownership constraints inserted to make y a safe owning reference (Box).

and the reference after the cast. This can violate the aliasing assumptions of the Rust compiler, and introduce new undefined behavior. We call this naive method *the unsafe transformation*. In order to resolve this problem, we need to make sure that a reference is never used if the pointer it is cast to is in use. To ensure this, we use the Rust borrow checker: whenever a location needs to become unsafe, we also mark that location to be owned. This ownership requirement does not affect the location itself, but it enforces that all safe references that flow into this location are also owned (i.e., they are of type Box<T> rather than & T). We consume the Box<T> when inserting a cast, so the cast invalidates the incoming reference, and the ownership requirement invalidates all prior references to the same object (enforced by Rust). We call the method that enforces this constraint *the safe transformation*. Fig. 9 shows the result of the safe transformation applied to the snippet in Fig. 4a.

Adding these ownership constraints comes with a cost in how many pointers we can make safe, as the ownership requirement would invalidate some uses of pointers. Moreover, Laertes cannot convert a reference expression (such as &mut x) to an owned pointer because of two reasons:

(1) Creating a Box out of &mut x would require moving x to the heap (making a heap allocation) which may negatively impact the performance of the program.
(2) When the type of x implements Copy (i.e., it is trivially copyable), the compiler may implicitly copy x instead of moving it to the heap. In this case, rewriting &mut x to Box::new(x) would be incorrect, as Box::new(x) will copy the value instead of making a reference to it.

We quantitatively evaluate the impact of adding ownership constraints in Section 9, in order to assess how their limitations affect the number of pointers and declarations made safe.

8.2 Representing Directional Flow using Casts

When rewriting an expression e with associated location l, we check if there is a data flow edge $l \rightarrow l'$ *in the initial DFG with pseudo-locations*, such that l is marked safe but l' is marked unsafe *in the final DFG*. If there is such an edge, then we insert a cast surrounding e, because e represents a reference that will be used immediately as a pointer. We query the edges in the DFG before the transitive closure (i.e., the graph before solving the set constraints) in order to capture such immediate flows. Otherwise, we would insert a cast on every expression that may eventually flow to an unsafe location. We also resolve all pseudo-locations using the final DFG, allowing us to identify immediate data flows at call sites and dereferences not present in the initial DFG.

9 EVALUATION OF OUR METHOD OF INSERTING CASTS

In this section, we evaluate Laertes with our modifications to measure the impact of both the safe and the unsafe version of inserting casts. The safe version introduces ownership with each cast (solving the issue described in Section 8.1.2), whereas the unsafe version does not (potentially

introducing undefined behavior by aliasing pointers and references). We are specifically interested in the following research questions:

- *RQ3: Does the analysis we propose reduce the set of affected pointers?*
- *RQ4: How effective is adding only top-level casts in terms of making more declarations and dereferences safe?*
- *RQ5: How much room for improvement is there between the safe and the unsafe transformations described in Section 8.1.2?*
- *RQ6: How does inserting casts affect the typechecker's view of the program (as interpreted by Laertes)?*

9.1 Experiment Setup

We use pseudo-safety (Section 3) to evaluate our method in this section, so we use the same benchmarks we used in Section 4. Table 1 lists the corpus of benchmark programs, the number of eligible pointers under pseudo-safety, and the number of pointers made safe by the baseline. We run Laertes under the following configurations after the pseudo-safety transformations:

- **Equality-based:** This is a version of Laertes with an equality-based analysis, without any of the transformations discussed in this paper. This serves as a baseline.
- **Subset-based (unsafe):** This is a version of Laertes with a directional analysis that naively inserts casts everywhere possible without guaranteeing a lack of aliasing between pointers and references. We use this version of the analysis to assess the room for improvement for future work that might design custom pointer types that can borrow a reference rather than transferring ownership.
- **Subset-based (safe):** This is a version of Laertes with a directional analysis that inserts casts only when it can guarantee the lack of aliasing between pointers and alive references by consuming the references, as described in Section 8.1.2.

In our analysis of the experiment results, we focus on the change in the number of pointer declarations, the number of pointer dereferences, and the sizes and the number of equivalence classes of pointers induced by Laertes's model of the typechecker. We do not look into the change in the number of functions because pseudo-safety replaces unsafe pointer uses with other unsafe substitutes in some cases (Section 3). In order to measure the potential effect of handling nested pointers in a directional manner, we also conduct the following experiment:

(1) Laertes computes a set of instigator pointers, i.e. the root causes of unsafety due to lifetimes for each iteration of invoking the compiler [Emre et al. 2021]. We record these for each benchmark.
(2) Then, we compute how many pointers are *not* made unsafe by these root causes using both the analysis described in Section 8.1, and an analysis that uses directionality for nested pointers. The number of pointers marked safe by the first analysis is the number of pointers made safe, whereas the number of pointers marked safe by the second analysis is an *an upper bound* on the number of pointers that could be made safe with more elaborate casts.

9.2 Results

RQ3. As discussed in Section 6, our earlier analysis precision study suggests that directionality can reduce the number of affected pointers, thus curbing the spread of unsafety. However, as discussed in the end of that section, there are some confounding variables we did not control for. We repeat the same study using the analyses we built on top of Laertes: an equality-based (Steensgaard-style) analysis that Laertes provides (corresponding to P1 in Section 6), a subset-based (Andersen-style) analysis, and the analysis described in Section 8.1.

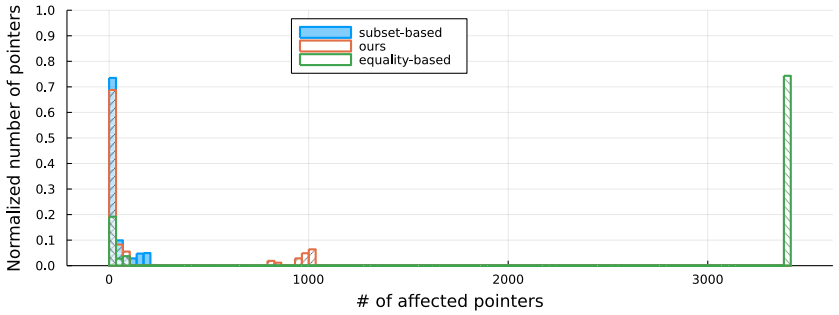


Fig. 10. Histogram of pointers in `tmux` at the HIR level. The x axis denotes the number of pointers affected, and the y axis denotes the normalized ratio of pointers in the program in the bin. **equality-based** is the result of the Steensgaard-style analysis, **subset-based** is the result of the Andersen-style analysis, and **ours** is the result of the analysis we describe in Section 8.1.

Our results on directionality match our observations in Section 6. For lack of space, we present only the histogram for `tmux`, and omit the overall statistics. Fig. 10 presents this distribution of affected pointers for a fully-directional analysis, the analysis suggested in Section 8.1, and an equality-based analysis (used by baseline Laertes). The results show that a field-based, directional analysis yields the smallest affected pointer sets, albeit not to the same extent as what we observed for P3 in Fig. 3. Fig. 10 also shows that the precision sacrificed by making the analysis effectively equality-based for nested pointers has a noticeable impact (Section 8.1), but the analysis we suggest still yields mostly small affected pointer sets.

RQ4. Tables 4 and 5 show the declarations and dereferences in the program (respectively), as well as how many declarations/dereferences are made safe by each method. When looking at declarations (Table 4), inserting top-level casts *safely* increases the effectiveness of lifetime inference by 75% (an increase from 12% to 21%). We see a similar overall picture for dereferences (Table 5). Our *safe* method increases the number of dereferences made safe by 54% (an increase from 11% to 17%). Although our method improves on Laertes relatively well (a 75% and 57% increase in the number of declarations and dereferences made safe, respectively), the overall number of pointers made safe is still low. Encoding other causes of analysis imprecision (such as a lack of context-sensitivity and field-sensitivity) is a worthwhile goal for future work on making more pointers safe.

When looking at the number of pointer declarations/dereferences made safe, introducing casts does not significantly improve Laertes' efficacy *relatively* in four benchmarks: `grabc`, `xzoom`, `libcsv`, and `TI`. The first two use an effectively global pointer unsafely (while interacting with the X graphics library), resulting in a spread of unsafety through the program. The unsafety in `libcsv` is caused directly by instigator pointers, so it is unsafety that needs to be fixed by the programmer (it is outside the scope of lifetime inference). Finally, the unsafety in `TI` spreads through function pointers, limiting the efficacy of our method; we elaborate on this in Section 9.3.

Based on our limit study using the directional analysis in Laertes, more elaborate casts could make *at most* 114 (1.3%) more pointer declarations safe total across benchmarks. We do not expect this limitation to have a large effect in the efficacy of Laertes in terms of making pointers safe.

RQ5. Comparing the number of pointers made safe by the safe and unsafe methods (Table 4), we see that the unsafe method allows a further 25% increse (an increase from 21% to 24% in the total percentage of pointers made safe). So, a more elaborate handling of the aliasing issues discussed in Section 8.1.2 may improve our method in the future. However, our method of offloading aliasing

Table 4. Pointer declarations. The "All pointers" column denotes the number of all pointers in the program (because we use pseudo-safety). Without casts = pointers made safe by the baseline (equality-based) transformation. With casts (unsafe) = pointers made safe when introducing casts while allowing unsafe aliasing. With casts (safe) = pointers made safe when introducing casts and preventing unsafe aliasing by consuming the original object in casts. All percentages are relative to the "All pointers" column.

Benchmark	All pointers	Made safe					
		Without casts		With casts (unsafe)		With casts (safe)	
robotfindskitten	2	2	(100%)	2	(100%)	2	(100%)
qsort	4	2	(50%)	3	(75%)	2	(50%)
grabc	13	8	(62%)	8	(62%)	8	(62%)
xzoom	29	3	(10%)	3	(10%)	3	(10%)
libcsv	37	25	(68%)	27	(73%)	25	(68%)
genann	73	12	(16%)	18	(25%)	15	(21%)
urlparser	79	7	(9%)	66	(84%)	46	(58%)
bzip2	227	78	(34%)	141	(62%)	135	(59%)
json-c	325	72	(22%)	151	(46%)	115	(35%)
lil	438	34	(8%)	162	(37%)	158	(36%)
libzahl	457	51	(11%)	208	(46%)	83	(18%)
tulipindicators	866	18	(2%)	36	(4%)	35	(4%)
tinycc	1352	207	(15%)	207	(15%)	387	(29%)
tmux	4635	468	(10%)	1045	(23%)	775	(17%)
TOTAL	8537	987	(12%)	2077	(24%)	1789	(21%)

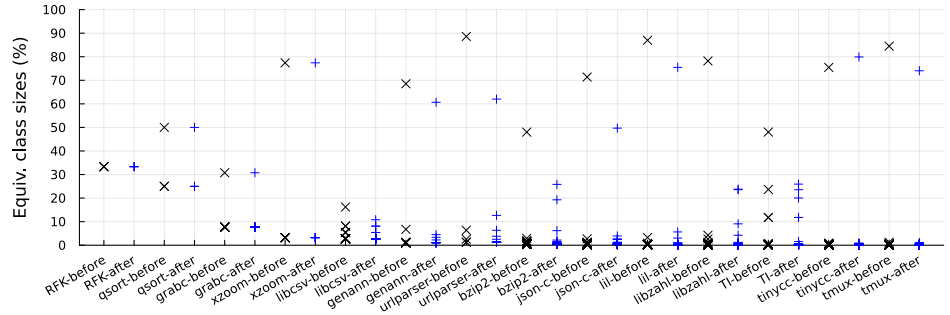


Fig. 11. Equivalence classes among pointers computed by Laertes's model of the typechecker before and after our transformations. The points denoted with × are the original equivalence classes, and the points denoted with + are the equivalence classes on the programs with casts inserted by our unsafe method. Members of an equivalence class must have the same pointer type. The benchmarks are ordered by the number of pointers.

constraints to the compiler using ownership is nevertheless an effective and simple solution. The gap between the safe and the unsafe method is larger when looking into dereferences (a 36% increase from 17% to 21% (relative to 12% baseline) in the absolute number of dereferences made safe). Most of these cases are in 3 benchmarks (`tinycc`, `json-c`, and `libzahl`), where there are pointers that are dereferenced many times but could not be made safe using the safe translation.

RQ6. Our method controls the spread of unsafety by inserting reference–pointer casts, partitioning equivalence classes of pointers, as described in Section 8.2. Fig. 11 shows the equivalence class sizes before and after our transformation. We present only the changes in equivalence classes

Table 5. Pointer dereferences. The "All dereferences" column denotes the number of all pointer dereferences in the program (because we use pseudo-safety). Without casts = dereferences made safe by the baseline transformation. With casts (unsafe) = dereferences made safe when introducing casts while allowing unsafe aliasing. With casts (safe) = dereferences made safe when introducing casts and preventing unsafe aliasing by consuming the original object in casts. All percentages are relative to the "All dereferences" column.

Benchmark	All dereferences	Made safe					
		Without casts		With casts (unsafe)		With casts (safe)	
qsort	10	4	(40%)	4	(40%)	4	(40%)
grabc	21	17	(81%)	17	(81%)	17	(81%)
robotfindskitten	24	24	(100%)	24	(100%)	24	(100%)
urlparser	60	58	(97%)	58	(97%)	58	(97%)
xzoom	172	119	(69%)	119	(69%)	119	(69%)
libcsv	174	51	(29%)	51	(29%)	51	(29%)
genann	339	5	(1%)	11	(3%)	6	(2%)
lil	1668	634	(38%)	903	(54%)	885	(53%)
tulipindicators	1847	154	(8%)	583	(32%)	582	(32%)
json-c	1878	167	(9%)	739	(39%)	234	(12%)
libzahl	2457	195	(8%)	573	(23%)	268	(11%)
bzip2	3764	319	(8%)	777	(21%)	750	(20%)
tinycc	5632	949	(17%)	949	(17%)	1321	(23%)
tmux	21641	1888	(9%)	3460	(16%)	2284	(11%)
Total	39687	4584	(12%)	8268	(21%)	6603	(17%)

after our safe method (the version that marks a reference as owned when it is cast to a pointer) for the sake of space. We also calculated the average changes in some statistics; we calculate the average change across a statistic s across by computing $GeoMean\ (s_{\text{after}}/s_{\text{before}})$ where $GeoMean$ is the geometric mean across programs, s_{before} is the value of the s before inserting casts, and s_{after} is the value of s after our unsafe method. The average size of an equivalence class decreases by 19% on average across our benchmarks (min=0%, max=50%). Our method increases the number of equivalence classes by 26% on average (min=0%, max=360%). Finally, the largest equivalence class in each benchmark decreases by 16% on average (min=0%, max=56%).

We still have large equivalence classes after our transformation because we break down equivalence classes (via cast inserting) only when doing so is beneficial; only the casts required to break "reverse data flow" from unsafe pointers to references are present in the final program, and all other data flow edges are present. So, large equivalence classes where all values are safe, or large equivalence classes where directionality does not lead to improvement are still present in the program. The latter case happens when a "necessarily unsafe" pointer flows into many pointers (e.g., a global value read from multiple locations). Additionally, the analysis presented in Section 8 degenerates into an equality-based analysis for nested pointers, which is another contributor to large equivalence classes.

Our method does not change the equivalence classes in the benchmarks `qsort`, `robotfindskitten`, `grabc`, and `xzoom`, because the data flowing from the pointers used unsafely in these programs make the rest of the pointers unsafe. As such, these have no opportunities to insert a reference–pointer cast. These are the programs with the lowest number of pointer declarations, and the first three have at most 4 pointers in an equivalence class. `xzoom` is the exception, and our method yields no improvement as 17/29 of the pointers in are already unsafe, and they become a source of unsafety

for the rest of the pointers in the program even with a directional analysis. In this case, all pointers are unsafe, so there are no places to insert casts and break equivalence classes.

9.3 Limitations

As mentioned in Section 8.1, our analysis uses equality-based reasoning for function pointers (since casting functions pointers is unsound), and for inner pointers (to avoid more complex casts). The limitations around function pointers impacts the `TI` benchmark disproportionately; `TI` is a time series analysis library that implements hundreds of analysis functions, and the functions are dispatched via a global array of function pointers. Only 4% of the pointers were made safe in this benchmark, even with directionality (Table 4). Investigating the spread of unsafety in `TI`, we observe that there is an unsafe pointer that flows into the parameters of a function pointer from this global array in one of the main drivers of the program, and this is a genuine data flow that cannot be remedied by encoding the results of a more precise analysis. This data flow results in almost all function parameters in the program to be unsafe, requiring resolution by the programmer. In the interim, approaches that encode the results of a context-sensitive analysis may allow this driver to be unsafe while making other drivers in the same program safe. Moreover, some of these data analysis functions use their parameters unsafely, and the equality-based reasoning for functions causes this unsafety to spread to other data analysis functions through the global array. Future work can introduce programmer-verified barriers and casts at call sites, similar to the work done by Machiry et al. [2022]) to help contain the spread of unsafety through function pointers.

9.4 Performance

We run Laertes with an optimization: Emre et al. [2021] generate a separate lifetime variable for each struct field, we use the same lifetime variable for all fields of a struct. In our experiments, this optimization did not cause any loss of precision (the resulting programs had the same safe pointers) while cutting the run time for our longest program by half. Both the safe and unsafe versions of our method finish in under 2 minutes for all programs except for `tinycc` and `tmux`. The unsafe version of the method takes 46 minutes for `tinycc` and 104 minutes (1h 44m) for `tmux`. The safe version finishes in 40 minutes for `tinycc` and 100 minutes (1h 40m) for `tmux`.

10 FUTURE WORK AND ALTERNATIVE APPROACHES

In Section 7, we propose three methods to curb the spread of unsafety using a pointer analysis-based approach that modifies source programs, and we implement one of these methods (encoding directionality using casts) in Section 8. The other two methods are left for future work. In the remainder of this section, we discuss two alternative approaches to curb unsafety without pointer analysis, namely (1) integrating more tightly with the Rust typechecker to exploit fine-grained control over typechecking results, and (2) modifying the Rust type system rather than the source programs. We also discuss why we settled on our approach in this paper, and we leave the implementation of these approaches for future work.

10.1 Tighter Integration with the Rust Type Checker

We follow a pointer analysis-based framework in Section 6 because (1) there is extensive research on pointer analysis (and data flow analysis in general) with different sensitivity levels, (2) existing tools such as SVF [Sui and Xue 2016] enable us to easily experiment with different sensitivity levels, and (3) Laertes [Emre et al. 2021] already uses a taint analysis to reason about the spread of unsafety through the type system, so pointer analysis-based approaches are easier to integrate with Laertes. So, a pointer analysis-based framework allows us to conduct the limit studies in this paper with great flexibility.

An alternative approach is to build a typechecker on top of Rust's typechecker, allowing for fine-grained access to typechecking decisions. Such an approach might more faithfully represent the decisions made by the typechecker, and reduce the overhead when tools like Laertes consult the typechecker to discover appropriate lifetime and ownership information from type errors. A type system-based model can also reason about the different sensitivity levels we explore in Section 6, and one could implement our proposed solutions in Sections 7 and 8 by building a more elaborate typechecker on top of the Rust typechecker. However, extending Rust's typechecker would require work comparable to what we sketched in Section 7: it would require engineering a precise, scalable, whole program analysis, performed by the typechecker. While such an approach could replicate our limit study in Section 6 within the intellectual framework of typesystems, it is unlikely that such a study would yield any additional insights.

10.2 Changing the Type System rather than the Programs

In Section 7, we focus on keeping the Rust type system the same, and instead change the programs to work with Rust. This approach is rooted in the larger goal of translating C code to idiomatic, safe Rust. Alternatively, one could change the Rust type system to accept more programs, which might provide an easier path for the future work sketched in Section 7. For example, one could use parametric polymorphism to help reason about context sensitivity [Fähndrich et al. 2000]. Such an approach could also exploit existing work on flow-sensitive type qualifiers [Foster et al. 2002] to introduce flow-sensitive directionality, which was left unexplored in Section 7. However, such type system extensions might not be adopted by the Rust community, because they are specific to a problem that does not exist in safe Rust code. Moreover, such an elaborate type system analogous to a context- and flow-sensitive analysis would be too slow for standard compilation, making adoption overall unlikely.

11 CONCLUSIONS

We have conducted a series of limit studies on the effectiveness of ownership and lifetime inference for unsafe raw pointers in `c2rust`-translated programs. Our first limit study uses a new technique called *pseudo-safety* that extends the study to all raw pointers, rather than just the small subset used in prior studies. This additional data contradicts prior results and the claim that [Emre et al. 2021]'s technique can make most pointers safe when the only source of unsafety is a lack of lifetime and ownership information. In contrast, our limit study shows that the majority of raw pointers cannot be translated to safe references via existing techniques. We empirically show that type equality is the culprit, causing unsafety to spread from only a few pointers to many others. We show that more precise pointer analysis can mitigate this problem, and suggest several program transformations that encode analysis results in manner understandable by an unmodified Rust compiler. We implement and evaluate one of these suggestions: encoding the results of a subset-based (directional) data flow analysis via introducing casts from references to pointers. We show that the analysis and transformation need to be co-designed to guarantee that the transformation captures the results of the analysis. Finally, we evaluate this transformation using pseudo-safety, and we see a 75% increase (from 12% to 21% of pointers) in the effectiveness of Laertes when we introduce casts. So, encoding the results of a subset-based analysis helps contain the spread of unsafe pointers and makes lifetime inference handle a larger part of the program. We leave investigating our other suggestions to encode the results of context-sensitive and/or field-sensitive analyses to future work, along with transformations handling subset-based analysis for function pointers. We also observe that both programmer intervention and encoding the results of more precise analyses are needed to tame unsafety further.

REFERENCES

Alexander Aiken. 1999. Introduction to set constraint-based program analysis. *Science of Computer Programming* 35, 2 (Nov. 1999), 79–111. https://doi.org/10.1016/S0167-6423(99)00007-6

Lars Ole Andersen. 1994. Program Analysis and Specialization for the C Programming Language. noURL

Brian Anderson, Lars Bergstrom, David Herman, Josh Matthews, Keegan McAllister, Manish Goregaokar, Jack Moffitt, and Simon Sapin. 2015. Experience Report: Developing the Servo Web Browser Engine using Rust. *arXiv:1505.07383 [cs]* (May 2015). http://arxiv.org/abs/1505.07383 arXiv: 1505.07383.

F. J. Anscombe. 1973. Graphs in Statistical Analysis. *The American Statistician* 27, 1 (Feb. 1973), 17–21. https://doi.org/10.1080/00031305.1973.10478966

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. ACM, New York, NY, USA, 211–230. https://doi.org/10.1145/582419.582440

Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. 2003. Ownership Types for Safe Region-based Memory Management in Real-time Java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, New York, NY, USA, 324–337. https://doi.org/10.1145/781131.781168

David Bryant. 2016. A Quantum Leap for the Web. https://medium.com/mozilla-tech/a-quantum-leap-for-the-web-a3b7174b3c12

Jonathan Corbet. 2021. Rust support hits linux-next. https://lwn.net/Articles/849849/

Nelson Elhage. 2020. Supporting Linux kernel development in Rust. https://lwn.net/Articles/829858/

Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development*. IEEE, Cambridge, MA, USA, 53–60. https://doi.org/10.1109/SecDev.2018.00015

Mehmet Emre, Peter Boyland, Ryan Schroeder, and Aesha Parekh. 2023. *Artifact for "Aliasing Limits on Translating C to Safe Rust"*. https://doi.org/10.5281/zenodo.7714175

Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating C to Safer Rust. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 121 (Oct. 2021), 29 pages. https://doi.org/10.1145/3485498

Manuel Fähndrich, Jakob Rehof, and Manuvir Das. 2000. Scalable Context-Sensitive Flow Analysis Using Instantiation Constraints. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 253–263. https://doi.org/10.1145/349299.349332

Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-Sensitive Type Qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/512529.512531

Samuel Z. Guyer and Calvin Lin. 2003. Client-Driven Pointer Analysis. In *Static Analysis (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 214–236. https://doi.org/10.1007/3-540-44898-5_12

Nevin Heintze and Olivier Tardieu. 2001. Ultra-fast Aliasing Analysis Using CLA: A Million Lines of C Code in a Second. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 254–263. https://doi.org/10.1145/378795.378855

Immunant inc. 2020. immunant/c2rust. https://github.com/immunant/c2rust original-date: 2018-04-20T00:05:50Z.

Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: a safe dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.

George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 423–434. https://doi.org/10.1145/2491956.2462191

Kornel Lesinski. 2018. Citrus / Citrus. https://gitlab.com/citrus-rs/citrus

Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership is theft: experiences building an embedded OS in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS '15)*. Association for Computing Machinery, New York, NY, USA, 21–26. https://doi.org/10.1145/2818302.2818306

Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Compiler Construction (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 47–64. https://doi.org/10.1007/11688839_5

Yi Lin, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2016. Rust as a language for high performance GC implementation. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management - ISMM 2016*. ACM Press, Santa Barbara, CA, USA, 89–98. https://doi.org/10.1145/2926697.2926707

Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R. Cordy, and Ahmed E. Hassan. 2022. In Rust We Trust – A Transpiler from Unsafe C to Safer Rust. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings*. IEEE, Pittsburgh, PA, USA, 354–355. https://doi.org/10.1109/ICSE-Companion55297.2022.9793767

Aravind Machiry, John Kastner, Matt McCutchen, Aaron Eline, Kyle Headley, and Michael Hicks. 2022. C to checked C by 3c. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 78:1–78:29. https://doi.org/10.1145/3527322

George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.* 27, 3 (May 2005), 477–526. https://doi.org/10.1145/1065887.1065892

David J. Pearce, Paul H.J. Kelly, and Chris Hankin. 2007. Efficient Field-sensitive Pointer Analysis of C. *ACM Trans. Program. Lang. Syst.* 30, 1 (Nov. 2007). https://doi.org/10.1145/1290520.1290524

Marc Shapiro and Susan Horwitz. 1997a. The effects of the precision of pointer analysis. In *Static Analysis (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 16–34. https://doi.org/10.1007/BFb0032731

Marc Shapiro and Susan Horwitz. 1997b. Fast and Accurate Flow-insensitive Points-to Analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/263699.263703

Jamey Sharp. 2020. jameysharp/corrode. https://github.com/jameysharp/corrode original-date: 2016-05-05T21:12:52Z.

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 485–495. https://doi.org/10.1145/2594291.2594320

Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

Jeff Vander Stoep and Stephen Hines. 2021. Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html

Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.

The Rust Project. 2021. The Rust Reference. https://doc.rust-lang.org/stable/reference/

The Rust Project. 2022a. lazy-static.rs. https://github.com/rust-lang-nursery/lazy-static.rs original-date: 2014-06-24T08:25:15Z.

The Rust Project. 2022b. standard lazy types - Rust RFC #2788. https://github.com/rust-lang/rfcs/pull/2788

Aaron Weiss, Daniel Patterson, Nicholas D Matsakis, and Amal Ahmed. 2020. Oxide: The Essence of Rust. , 27 pages. https://doi.org/10.48550/arXiv.1903.00982