University of California
Santa Barbara

Translating C to Safe Rust: Reasoning about Pointer Types and Lifetimes

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Mehmet Emre

Committee in charge:

Professor Ben Hardekopf, Chair
Professor Chandra Krintz
Professor Yu Feng

September 2022

The Dissertation of Mehmet Emre is approved.

Professor Chandra Krintz

Professor Yu Feng

Professor Ben Hardekopf, Committee Chair

July 2022

Acknowledgements

The Ph.D. program has been a long, winded, and satisfying journey for me. There has been many unexpected turns, and hurdles to overcome. There are many people I am thankful to for helping me throughout this process (I am bound to miss some names, and I apologize for that):

Ben Hardekopf, for teaching me how to do research, pointing me in the right direction while giving me the necessary freedom, and making me keep a healthy balance between research and recreation.

Chandra Krintz and Yu Feng, for helping me not lose the sight of the bigger picture, and present my work in a more holistic way.

Kyle Dewey, for treading and clearing the path before me, and being a tremendous resource of good advice. I am grateful for your guidance on teaching and research as well as you sharing your earnest outlook on life.

Lawton Nichols, for being a great labmate, being there to bounce ideas, and getting me started to dance. You have been a great inspiration of tenacity and curiosity during the time I worked next to you.

Michael Christensen for being a great partner in the projects we have taken on together, and for sharing the experience of going through the program at the same time. Project Neptune lives!

Miroslav Gavrilov for reminding me that we do the Ph.D. program because it is fun and interesting, also for poking through all weird corners of programming languages. I could not survive Salinas without you.

Zach Sisco for making the lab environment much less solitary and saner during the pandemic, and for being a person to talk about my research problems. It was a pleasure working in the lab together especially in the past year.

Harlan Kringen for making me revisit the fundamentals of my knowledge of programming languages with better understanding countless times. Your questions pierce through the veneer and all the complexity of so many ideas.

Ryan Schroeder for being a great person to work with, and showing me many great techniques to use. You are one of the most brilliant engineers I have met.

Peter Boyland, Aesha Parekh, Ben Darnell, and other members of the PL Lab that made it a great place to work.

İsmet Burak Kadron for being a great friend that's always there for me, and as someone I could discuss both technical and non-technical matters. I could not persevere through this without a friend like you.

My parents and siblings for their constant support and encouragement through the most difficult parts of this journey.

Gürkan Gür for getting me interested in research, and applying to a Ph.D. program.

Curriculum Vitæ
Mehmet Emre

Education

2022	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2021	M.S. in Computer Science, University of California, Santa Barbara.
2015	B.S. in Computer Engineering, Bogazici University.

Publications

Mehmet Emre, Peter Boyland, Kyle Dewey, and Ben Hardekopf. Taming the Spread of Unsafe Pointers in Rust Programs. ***Under submission***, 2022.

Mehmet Emre, Peter Boyland, Aesha Parekh, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Aliasing Limits on Translating C to Safe Rust. ***Under review***, 2022.

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

Lawton Nichols, Kyle Dewey, **Mehmet Emre**, Sitao Chen, and Ben Hardekopf. Syntax-based Improvements to Plagiarism Detectors and Their Evaluations. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, pages 555–561. ACM, 2019.

Lawton Nichols, **Mehmet Emre**, and Ben Hardekopf. Fixpoint Reuse for Incremental JavaScript Analysis. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 2–7. ACM, 2019.

Lawton Nichols, **Mehmet Emre**, and Ben Hardekopf. Structural and Nominal Cross-language Clone Detection. In *International Conference on Fundamental Approaches to Software Engineering*, pages 247–263. Springer, 2019.

Mehmet Emre, Gürkan Gür, Suzan Bayhan, and Fatih Alagöz. Cooperativeq: Energy-Efficient Channel Access Based on Cooperative Reinforcement Learning. In *2015 IEEE International Conference on Communication Workshop (ICCW)*, pages 2799–2805. IEEE, 2015.

Abstract

Translating C to Safe Rust: Reasoning about Pointer Types and Lifetimes

by

Mehmet Emre

Infrastructure software is written in low-level programming languages like C to allow precise control of resources. However, C lacks safety features to ensure memory and thread safety. These safety issues result in serious security vulnerabilities or unreliable behavior. Rust is a programming language that provides the same fine-grained control with automatically-checked safety measures. Rust is being adopted by some large C and C++ code bases such as Linux, Firefox, and Chromium. However, proving a program's safety to the Rust compiler requires non-local reasoning about ownership and lifetimes of objects in the program so translating C programs to safe Rust programs is nontrivial.

This thesis presents the challenges in translating C programs to safe Rust programs, along with a categorization of different classes of unsafety. To kick-start automated translation from C to safe Rust, I present a novel method to infer object lifetimes and ownership using the compiler as an oracle. I then develop an evaluation methodology to measure the potential impact of this method independent of other causes of unsafety in the program. With this methodology, I show that the efficacy of this method (along with any potential method to discover existing safe uses of objects in the program) is limited by the precision of the type system when propagating unsafety information. Then, I investigate the impact of more sensitive data flow analyses to curb the spread of unsafety, and show that they can be encoded by transforming the program without changing the type system. Overall, the findings of this thesis are that (1) the causes of

unsafety are intertwinend, (2) using the compiler errors to derive lifetime information is effective for discovering existing safe uses of objects, (3) imprecision of the type system leads to excessive spread of unsafety (as lack of provable safety according to the compiler), and (4) this can be mitigated by transforming the program to make the results of a more precise analysis available to the type checker.

Contents

Chapter 1

Introduction

[C has] the power of assembly language and the convenience of ... assembly language.

– Attributed to Dennis Ritchie [33]

Rust is a programming language is intended to address the safety issues of C, without compromising on C's advantages. Like C, Rust is designed for writing efficient low-level software with precise control of memory. However, unlike C, Rust provides strong static guarantees about memory and thread safety through a more elaborate type and ownership system. However, not all usage patterns (such as shared mutable data, and cyclic data structures) are expressible in pure safe Rust. So, Rust also contains an *unsafe* fragment that allows expressing arbitrary data access patterns where the programmer can control how objects are accessed. This fragment of language is gated behind the `unsafe` keyword, and the programmer has to maintain the invariants the Rust compiler expects when using unsafe Rust. As such, programmers should avoid `unsafe` as much as possible, as Rust can degenerate into C with excessive `unsafe` usage. Rust has been used for building operating systems, web browsers, and garbage collectors [5, 28, 29] and it is being adopted into complex software projects with large C/C++ code-bases such as Firefox [10], the Linux kernel [17, 30], and Android [42].

So, Rust provides a safer alternative to C, as well as a gradual path to adopt it through `unsafe`. There are also extensions to C with safety guarantees (such as CCured [35] and Checked C [18]) that may seem attractive as safer alternatives to C, given that their type systems build on top of C's type system. However, they do not guarantee memory and thread safety at the language level, and they do not prevent use-after-free errors *at compile time*. Thus, we focus on Rust as a language that can give compile-time memory safety guarantees as well as precise control of resources needed in systems software.

A large amount of critical systems software predates Rust, and is written in low-level languages without language-level memory safety guarantees such as C and C++ because of requiring explicit and fine-grained control of resources such as memory. Lack of memory and thread safety has led to numerous critical bugs and security flaws [1, 2, 16] costing both money and human lives [16, 27]. In light of Rust's recent development and promise of safety, a natural question arises about the possible benefits of porting software from these unsafe languages to Rust, eliminating a large class of potential errors. In fact, there has been some informal investigation into the question of how effective Rust would be at fixing critical errors in existing C code (after all, not all bugs and security flaws are due to memory or thread unsafety). An anecdotal study done on cURL, a popular data transfer utility written in C, conservatively estimates that using Rust would eliminate 53 of the 95 known cURL security flaws as of 2021 [21].

Porting existing software to Rust requires a lot of effort, as all such code rewrites do. Moreover, this problem is exacerbated by the impedance mismatch between the sophisticated type system of Rust and that of C when it comes to reasoning about thread and memory safety. Automated translation and refactoring tools would make such a porting effort feasible. Such an automated translation tool would need to analyze the relevant properties of C code and reason about potential concurrent uses

of objects, then create a suitable well-typed Rust program that correctly expresses those properties. Moreover, the original C program *may not be completely "safe" (in the Rust sense)*, as the C programmer most likely did not take into Rust's notion of safety when authoring the program.

Although push-button translation from C to *safe* Rust is an open problem, there are existing tools that translate C code to *unsafe* Rust code as a first step (e.g., C2Rust [22], Citrus [11], Corrode [40]). These translations are purely syntactic in nature, and they preserve the semantics at a very low-level in order to emit Rust code that emulates the original C code using unsafe constructs. As a result, the translated Rust programs heavily use *unsafe*, hence these programs may have memory- or thread-safety issues that would not exist in an idiomatic, safe Rust program. The expectation in these tools' workflow is for the programmer to iteratively refactor and rewrite the resulting Rust program to arrive at a safe, idiomatic Rust program. The state of the art among these translation tools, C2Rust, also comes with a refactoring tool to help automate this process.

There are seven interleaving causes of unsafety that need to be disentangled and resolved in order to automatically translate as much of the program to safe Rust as possible, and that is beyond a single dissertation. So, in this dissertation, we (1) analyze unsafety in Rust programs that mimic C programs, and (2) focus on a core cause of unsafety tied to the difference between the type systems of C and Rust.

Rust's memory and thread safety guarantees are based on the soundness of its type system, specifically its reasoning about types and lifetimes to check for use-after-free and data races at compile-time if possible, and inserting run time checks for things that are not feasible to be verified at compile-time (array bounds). So, on a conceptual level, unsafe features are manifestations of:

1. Using objects without any lifetime, aliasing, or ownership information for the compiler to check,

2. Deliberately circumventing the type system (via casts, unions, forging invalid pointers via pointer arithmetic), and

3. Calling into code that the compiler does not have access to (e.g., inline assembly, external function calls).

Only the first reason to need unsafety relates to what Rust's type system brings, whereas the remaining two relate to how the type system's soundness can be compromised. Also, all three of these causes ultimately stem from lack of information in the original C program that the Rust compiler needs to prove safety. So, we focus on using the safety features in the type system correctly, rather than patching various ways that the programs circumvent the type system.

The core mechanism Rust's type system provides to verify memory safety is the borrow checker, which uses lifetime, borrowing, and ownership information in the program to verify memory safety. So, we need to infer lifetime and ownership information in order to make these programs[1] safe. Thus, the aim of this dissertation is to answer the following question:

> **Thesis question:** *What are the limits of automatically inferring lifetime and ownership information for pointers that are already used safely?*

Although investigating this question does not sufficiently give a way to automatically translate C to safe Rust, it is a *necessary* problem because we need to understand how the core language construct Rust brings to solve the memory safety issues, and how the programs translated from C can be transformed to incorporate this language

[1] Rust programs translated from C.

construct. Besides addressing this problem, we also quantitatively and qualitatively investigate how other causes of unsafety are used in programs translated from C in order to inform future research projects that would complement this work.

Also, our goal is *discovering existing safe usage of memory, and exposing it to the Rust compiler* rather than rewriting the programs at a high-level (as in, not preserving statement-level semantics) in order to make them safe. Expressing complex memory usage patterns in safe Rust may require introducing memory allocation abstractions (such as arenas), opt-in run-time support for some of the objects (such as reference collection), or novel data structures with internally unsafe implementations. The choice of the most appropriate representation in is highly application-specific with different design trade-offs (for example, edges in a graph data structure can be represented with reference counted, run-time borrow-checked pointers, or by integer indices into an arena of graph nodes; and each representation has its own trade-offs). Rather than delving into application-specific workarounds for these cases, we focus on a general approach to discovering lifetime and ownership information that can be used at compile time to guarantee safety.

Overall, inferring lifetimes and ownership is an important first step because:

1. All other safe reference type in the language require reasoning about this information.

2. It frees the programmer from the general case of reasoning about lifetimes, and lets them focus on code that is likely to require application-specific workarounds or higher-level changes to the algorithms and data structures used. Such cases are likely to arise and may point to actual safety bugs (as observed by Bryant [10] when rewriting part of Firefox in Rust).

3. Finally, such a method can help isolate memory usage patterns encountered in

practice yet not deemed safe by Rust to inform future language or library design to support these use cases.

As we are interested in the potential for inferring lifetimes (and ownership) *in the limit*, we need to quantify how much of the program is unsafe because of lack of lifetime information, as well as how much of it is unsafe *only* because of lack of lifetime information. We perform such a limit study. In this study, we precisely define unsafe language features that appear programs translated from C, and analyze their uses. Although there are similar studies in literature for Rust programs as a whole [8, 20, 38], we perform a study to inform the process of translation from C to Rust with the following differences: (1) we quantify unsafety in programs translated from C, and (2) we track uses of unsafe features through function calls to understand the spread of unsafety in this context. Overall, we find that there is a large overlap between uses of different language features, that is functions tend to use more than one unsafe language feature. We also come up with an overall ordering to tackle these features that correlates with how often they appear in programs. The two most common features we list (unsafe pointers, and mutable global variables) are unsafe because they can violate the lifetime and aliasing model of Rust. Global variables cause this because of concurrent mutable access to same value (so, they can be guarded behind synchronization mechanisms to ensure safe concurrent access). Unsafe pointers (a.k.a. raw pointers) as they are unsafe because of lack of lifetime information. Note that we are using "unsafe" to mean "not provably safe from the compiler's perspective", the actual usage patterns may be safe in practice. We further investigate how each unsafe language feature is used. We observe that unsafe pointers are involved in a multitude of unsafe behaviors, although lack of lifetimes is the underlying cause that needs to be resolved for all pointers.

We then approach inferring lifetime and ownership *using the compiler as an oracle*. The key insight of our approach here is that we can start from an optimistic program and send the program to the compiler, get the compiler errors, and derive the necessary lifetime and ownership information from these errors, and repeat this process until we reach an error-free program. In some cases, the program uses pointers in an manner where the derived lifetime information shows a safety issue. For example, two mutable references to an object may be alive at the same time, which can potentially cause a data race (and Rust programs have to be thread-safe), so it is unsafe. In such cases, our method promotes the relevant pointers to be unsafe, as there is no safe behavior we can infer. We implement this method as a tool called Laertes. Overall, Laertes makes 87% of the eligible pointers (pointers that are unsafe *only* because of lack of lifetime information) in our benchmarks. So, this is a promising first step to solve the lifetime inference.

Although Laertes is effective on eligible pointers, only 11% of the pointers are eligible—the remaining 89% of the pointers are unsafe because of other causes as well. So, the evaluation of this method is subject to sampling bias. We are interested in its efficacy *in the limit*, so that we can understand whether pointers are used in a different way when other unsafe uses of pointers are also involved. As we are interested in measuring the effectiveness of lifetime inference in isolation (without any interference from other unsafe uses of pointers) for the whole program, we cannot use any partially effective method to handle other uses of unsafe pointers (as not all pointers would be eligible). We introduce an evaluation methodology dubbed *pseudo-safety* to make all pointers in the program eligible for lifetime inference. The core idea behind pseudo-safety is replacing causes of unsafety that are irrelevant to lifetime inference with counterparts that preserve lifetime and ownership information. We carefully generate these counterparts to mimic a local rewrite that would replace only the uses of these

language features while leaving the rest of the program intact. For example, we replace all declarations of the same external function with a single stub that serves to connect all call sites as calling the same function hence ensuring that the lifetime, ownership, and safety information of the arguments in all call sites match. This stub represents as an ideal replacement of the external function. We don't need to fill in this stub because we preserve only the compile-time behavior related to safe pointer use, but not the run-time behavior, hence the name pseudo-safety. With pseudo-safety all pointers in the program are eligible for lifetime inference, and LAERTES is effective on only a small number of pointers (only 12%) in this case. The naïve conclusion is that pointers used unsafely in more than one way potentially have a very different lifetime and ownership behavior. However, we empirically show that the underlying issue here is type equality (more specifically, the lack of precision in the type system): the type system spreads unsafe pointers (just like any other type) through an imprecise (equality-based, field-based, context- and flow-insensitive) data flow analysis (e.g., whenever we have an assignment like `x = y;`, we need to make sure that `x` and `y` have the same type, so we end up with an equality-based analysis). Under this data flow analysis, for each of our benchmarks, making 4 pointers unsafe is enough for unsafety to spread to half of the pointers.

So, we find that unsafety of pointers in the program depends not only on specific unsafe uses, but also how the unsafety "taint" spreads via the type system. This spread puts a hard limit on our approach of incrementally inferring lifetimes, as a large part of the pointers become unsafe as a result of imprecision of the type system. We define two notions of unsafe pointers: if a pointer is unsafe because it is directly involved in an unsafe use (e.g., its use is not valid according to the borrow checker rules), then we call it an *instigator*; if a pointer p is unsafe only because there is a data flow from an instigating pointer to p, then we call p an *affected* pointer. This terminology allows

us to discuss how unsafety spreads from instigators to affected pointers. We can now investigate two questions to understand and solve the spread of unsafety:

1. If we can use a more precise analysis to calculate the spread of unsafety, would that reduce the number of affected pointers in the program?

2. Can we use the results of a more precise analysis *without changing Rust's type system?*

In order to answer the first question, we build data flow analyses of increasing levels of precision along different axes (adding field-sensitivity, context-sensitivity, and subset-based analysis), and measure the pointers affected for each pointer in the program. Our findings show that adding context-sensitivity and switching to subset-based analysis are effective for eliminating a lot of spurious data flow between pointers.

The second question is crucial for building a method without proposing changes to Rust, as our ultimate goal is to translate programs from C to Rust, and any feature proposals that complicate reasoning about unsafety are not likely to be adopted. We approach the second question by proposing ways to transform the program to encode the results of a more precise analysis in each dimension we evaluated:

- Emulating field-sensitivity by duplicating type definitions (so each type will have multiple versions with safe and unsafe pointers for each field).

- Emulating context-sensitivity using a similar trick: duplicating function definitions for different signatures they may have (e.g., the first argument may be a safe reference or an unsafe pointer), and applying defunctionalization to allow using function pointers in this setting.

- Emulating a subset-based analysis by explicitly inserting casts from references to pointers (so that unsafety taint does not flow against the direction data flow).

The first two cases create a tension between safety and maintainability, as cloning all type definitions or all functions would not be feasible for generating a program that is going to be maintained. We leave the problem of identifying crucial types and functions that need to be cloned to future work.

Finally, we implement our suggestion of inserting casts to emulate a subset-based analysis, and evaluate it using pseudo-safety in order to show that the result a more precise analysis can be encoded into the program to curb the spread of unsafety. Although this transformation seems straightforward at a first look (just inserting a call to `.as_ptr()` whenever we have a reference but a pointer is expected), we show that one needs to be careful to ensure soundness of such a transformation:

- The original reference and the result of the cast alias, and using them both at the same time is undefined behavior (UB) *that did not exist in the original program*.
- We consider only casts at a top-level, and a subset-based analysis not sound for computing the correct types in the presence of nested pointers, and function types.

In order to solve the first problem, we piggyback on Rust's type system, and force casts to consume the original object (the pointee) so no object can be access through both safe and unsafe pointers. We fall back to an equality-based analysis for function types and inner pointer types in nested pointers. Even with these limitations, we see an 75% increase (from 12% to 21% of pointers) in the effectiveness of Laertes when we introduce casts. So, increasing analysis precision is a way to contain the spread of unsafe pointers and make lifetime inference handle a larger part of the program, and just introducing casts is not enough to tame unsafety, and encoding results of more precise analyses is needed.

In summary, we answer the question of discovering the limits of lifetime inference

with the following research contributions: We identify and quantify the prevalence of different unsafe language features, and how different causes of unsafe pointers co-occur to understand the potential of lifetime inference by itself. Then, we build an iterative method to discover lifetime and ownership constraints from compiler errors, and show that this method is effective on pointers that do not contain causes of unsafety besides lack of lifetime information. Next, we build an evaluation methodology that hides other causes of unsafety while maintaining the lifetime constraints we care about, and evaluate the effectiveness of lifetime inference on all pointers in the program. This evaluation shows that lifetime inference does not scale well when considering all pointers, and we show that the underlying cause here is the spread of "accidental" unsafety through the type system. We then conduct a limit study evaluating potential impact of making the type system more precise (by keeping track of data flows more precisely), and we propose methods to encode the results of a more precise analysis by program transformation. Finally, we implement one of our proposals to show that such an encoding is feasible and it can double the effectiveness of lifetime inference.

The remainder of this dissertation is structured as follows:

- A short background in Rust's ownership system and related work (Chapter 2).
- A categorization of sources of unsafety that occur in programs translated from C, and a qualitative and quantitative evaluation of these sources of unsafety to chart which sub-problems about safety to focus on (Chapter 3).
- A method to derive lifetime, ownership, and aliasing information using the compiler as an oracle, and an evaluation of this method (Chapter 4).
- An evaluation methodology based on automatically transforming the programs to ignore certain classes of unsafety, to allow focusing on a single cause of un-

safety, and to evaluate methods that work on a single cause independent of other causes, and an investigation into the sensitivity of the data flow analysis induced by the Rust type checker to show that the lack of precision in type checking causes unsafety to spread like wildfire (Chapter 5).

- A method for inserting data flow barriers to the program in order to encode the directionality of data flow, and to improve on the effective sensitivity of the Rust type checker without changing the type checker (Chapter 6).
- Finally, concluding remarks along with suggestions for future work to complement and continue this research (Chapter 7).

1.1 Permissions and Attributions

1. The contents of Chapters 3 and 4 are the result of a collaboration with Ryan Schroeder, Kyle Dewey and Ben Hardekopf, and has previously appeared as our OOPSLA 2021 paper [19]. It is reproduced here with updates and a more in-depth analysis, abiding by the license of that work (Creative Commons Attribution 4.0 International).

2. The contents of Section 2.1 is the result of a collaboration with Ryan Schroeder, Kyle Dewey and Ben Hardekopf, and has previously appeared as part of the the supplementary material for our OOPSLA 2021 paper [19]. It is reproduced here with updates and a more in-depth analysis, abiding by the license of that work (Creative Commons Attribution 4.0 International).

Chapter 2

Background and Related Work

2.1 Rust's Ownership System

This section serves a short primer to how Rust handles *ownership* and *borrowing*. Both of these features are central to Rust's memory model, and enable it to statically ensure memory safety in safe code without resorting to garbage collection at runtime. Given that our work must work with Rust's memory model closely, it is necessary to have some understanding of Rust's memory model in order to understand the significance of our own work. That said, this section is intended only as a quick introduction; readers curious for more details are directed to the online Rust book for basics [26], as well as as a more formal alias-based formulation at [32].

2.1.1 Motivation

Rust's memory model ensures memory safety statically, without resorting to potentially expensive runtime memory management techniques like garbage collection. In Rust, well-typed programs are memory-safe by construction. As with a garbage collected language, users explicitly perform memory allocation, but do not explicitly

perform deallocation. Unlike with garbage collection, the Rust compiler statically inserts routines to deallocate heap-allocated memory when it is no longer needed. The type system of Rust is designed in such a manner that the compiler statically knows exactly where these memory deallocations need to be performed. This knowledge of when to perform deallocation is based around *ownership*.

2.1.2 Ownership

By default, data is said to be *owned* in Rust. For example, consider the following function definition `f`, which uses type `Vec` from the Rust standard library (representing a vector):

```
1 fn f(v: Vec<i32>) {}
```

`f` is said to take ownership of `v`. This is indicated by the fact that `v` is directly of type `Vec<i32>`. Whoever owns the data is ultimately responsible for deallocating any heap-allocated data held. Deallocation implicitly occurs whenever the variable bound to the data falls out of scope. With this in mind, any heap-allocated data held in `v` is deallocated immediately after the call to `f`, as `v` will no longer be accessible.

Within a scope, ownership can be transferred from one variable to another. For example, consider the following code snippet:

```
1 fn example() {
2   let v1 = vec![1, 2, 3]; // creates a vector holding 1, 2,
      3
3   let v2 = v1;
4 }
```

In this case, `v1` initially holds the underlying vector. Ownership is then transferred to variable `v2`. Because ownership is never transferred away from `v2`, `v2` will have all

heap-allocated memory deallocated at `example`'s termination. Because ownership was transferred away from `v1`, there is no similar deallocation performed for `v1`, beyond typical stack deallocation of `v1`.

Ownership can also be transferred between scopes. For example, consider the following:

```
1 fn identity(v: Vec<i32>) -> Vec<i32> { return v; }
```

In this case, like the prior `f` example, `identity` takes ownership over `v`. However, because `identity` later returns `v`, it transfers ownership to `identity`'s caller. Any heap-allocated memory bound to `v` then becomes the concern of `identity`'s caller.

2.1.3 Borrowing and Lifetimes

While the ownership model unambiguously allows the compiler to safely statically deallocate all heap-allocated memory, it is nonetheless very restrictive. For example, if you wanted to define a function that merely printed the contents of a vector, it would need to transfer ownership back to the caller. This would mean having an unintuitive type signature like:

```
1 fn print_all(v: Vec<i32>) -> Vec<i32> { ... }
```

With this in mind, the more data a function needs to do its job, the more data the very same function needs to return. There are also negative performance implications of ownership transfer, since barring compiler optimizations, it entails copying any stack-allocated memory behind a variable.

To address these issues around ownership transfer, Rust also has a concept known as *borrowing*. As the name suggests, data can be temporarily borrowed without changing ownership. Data is borrowed through a reference, which bear similarity to references in other languages. Borrowed data can be used like owned data, with some restrictions.

One important restriction is that borrowed data cannot outlive the actual data being borrowed. Using C/C++ terminology, Rust must ensure that there are no dangling pointers to any allocated data.

To ensure that the underlying data being borrowed is always valid, Rust introduces the concept of a *lifetime*. Lifetimes are type-level variables which abstractly define how long the underlying data being borrowed will be in memory. For example, consider the following code:

```
1 fn has_lifetime<'a>(v: &'a Vec<i32>) { ... }
```

Instead of having ownership of `v` transferred to `has_lifetime`, this instead borrows the underlying `Vec<i32>` for lifetime `'a`. Rust will ensure that the underlying `Vec<i32>` is in memory for the duration of the call to `has_lifetime`. Because `has_lifetime` merely borrows the `Vec<i32>`, there is no memory deallocation of `v` performed; `has_lifetime` does not own the vector, and so it is not `has_lifetime`'s responsibility to deallocate the vector.

Like regular type variables, data structure definitions themselves can take lifetimes, as with:

```
1 struct SomeData<'a, 'b> {
2   first: &'a i32,
3   second: &'b i32
4 }
```

With the above code in mind, Rust will make sure that no allocated instance of `SomeData` will outlive anything it borrows. That is, the data referred to by `first` and `second` will always be in memory at least as long as the `SomeData` data structure itself.

To show this in practice, consider the following example, which is rejected by the Rust compiler:

```
1 fn rejected() {
2   let the_data;
3   let first_int = 1;
4   {
5     let second_int = 2;
6     the_data = SomeData { first: &first_int, second: &
     second_int };
7   }
8   print!("{}'', *the_data.second);
9 }
```

The above code is rejected by the Rust compiler, with an error message stating that `second_int` does not live long enough. To understand why, first understand that each block in Rust corresponds to a separate lifetime variable. That is, an enclosing scope maps directly to object lifetimes. For speaking purposes, the outer scope of rejected will be called `'a`, and the inner scope (where `second_int` is declared) will be called `'b`. With this in mind, `the_data` has type `SomeData<'a, 'b>`, and it itself has lifetime `'a`. However, `'b` does not live as long as `'a`. As such, we have attempted to create a data structure with a lifetime longer than its constituents, which is not permitted. As such, Rust rejects the program. Thinking in terms of C/C++, this rejection makes sense - `second_int` is allocated on the stack and subsequently deallocated after `the_data` is initialized, so `the_data.second` would be a dangling pointer.

2.1.3.1 Restrictions

All borrows seen so far are immutable borrows, meaning that the underlying object cannot be changed through these borrows. Furthermore, the underlying object may

not be changed at all while any immutable borrows are active. Similarly, Rust disallows ownership transfers while any borrows are active. This can be statically checked at compile time, as shown in the code below:

```
1 struct MyStruct {
2     first: i32
3 }
4
5 fn involves_borrows<'a>(datum: &'a MyStruct) -> &'a
    MyStruct {
6   return datum;
7 }
8 fn performs_transfer(x: MyStruct) {}
9
10 fn main() {
11     let x = MyStruct { first: 42 };
12     let r = involves_borrows(&x);
13     performs_transfer(x);
14     print!("{}", r.first)
15 }
```

The above code fails to compile, as the the transfer performed by `performs_transfer` is disallowed because reference `r` still refers to the same data structure. Specifically, Rust tracks that `x` has an active borrow at the call to `performs_transfer`, disallowing the call. As an aside, the subsequent use of `r.first` is required to get this code to compile, as this forces the compiler to internally keep the borrow of `x` around after the call to `performs_transfer`; effectively, Rust will permit the existence of a dangling

pointer, but not the access of a dangling pointer.

2.1.3.2 Immutable and Mutable Borrows

All prior borrow examples are based on immutable borrows, meaning the underlying object cannot be changed through the borrow. Rust also supports mutable borrows, which use the `mut` reserved word, like so:

```
&'a mut Vec<i32>
```

The above snippet refers to a mutable borrow of a `Vec<i32>`, where the underlying vector is in memory for at least `'a` lifetime.

Mutable borrows work similarly to mutable borrows, with the following twists. With immutable borrows, the same data may be borrowed multiple times in the same context, as none of the borrows can change the underlying object. However, with mutable borrows, only one such mutable borrow may be active at any time. Furthermore, if a mutable borrow is active, all mutation must be done through the mutable borrow, and no immutable borrows or ownership transfers are permitted. While restrictive, these requirements prevent data races from occurring - all mutation is very carefully tracked and made explicit in the types; it is not possible for data to be modified "out from under you", as it is in most languages.

2.2 Translating C to Rust

Citrus [11], Corrode [40], and C2Rust [22] all translate C code to Rust, albeit with frequent usage of `unsafe`. C2Rust lacks any formal translation guarantees, so it is additionally packaged with a *cross-check tool*, which compares the execution trace of the original C program with the translated Rust program under the same input. The cross-check tool is intended to test whether or not the translator produced a truly

equivalent Rust program. We [19] use the output of C2Rust, as well as some semantic rules from Oxide [46], in order to generate Rust code which uses `unsafe` less frequently than C2Rust alone. As a result, our work is also dependent on C2Rust's cross-check tool to ensure the correctness of the translation.

2.3 Referring to Memory in Rust

Rust supports two mechanisms to refer to memory: references and raw pointers. References are carefully restricted so that their dereference is guaranteed safe. In contrast, raw pointers are far less restrictive, but they can only be dereferenced within code marked `unsafe`. Raw pointers semantically correspond to C pointers, and C2Rust unconditionally translates C pointers to raw pointers. Our work, in contrast, translates C pointers to Rust references, where possible.

2.4 Characterizing Unsafe Code in Rust

We [19] classify and quantify unsafety in Rust programs translated from C by C2Rust on a program corpus. They find that raw pointer dereferences account for most uses of `unsafe`. They divide raw pointers into four overlapping categories: 1.) pointers used in pointer arithmetic; 2.) void pointers; 3.) pointers used in external APIs; and 4.) single-object pointers not involved in the other categories. They find that most pointers fall into the fourth category, and so they focus on translating these pointers into safe references. However, they do not safely handle pointer arithmetic, which was the second-largest category of unsafe pointers. Handling pointer arithmetic safely is the focus of our work.

Besides Rust programs translated from C, there have been several studies of un-

safety in the Rust ecosystem at large (e.g., [8, 20, 38]). The next most common causes of unsafety according to both Evans et al. [20] and Astrauskas et al. [8] are raw pointer dereferences and global mutable variable usage. Qin et al. [8] also report that the most common (42%) purpose of `unsafe` usage is to reuse existing C code with minimal modification, including code that performs pointer arithmetic or calls into external libraries.

2.5 Reasoning about Rust's Type and Ownership Systems

There are several Rust formalizations in the literature (e.g., [9, 25, 39, 46]). Among these, the RustBelt project [25] describes a mechanized formal semantics for Rust's mid-level intermediate representation (MIR) called λ_{Rust}. λ_{Rust} has been used to derive the verification conditions for safety of widely-used standard library abstractions using `unsafe`, and to formally prove that the API they expose is a safe extension of the language. λ_{Rust} includes a complete Rust specification.

Prusti [7] is a verification tool built on top of Viper verification framework [34]. It allows the user to specify verification conditions as annotations in Rust, and leverages Rust's type system to simplify the verification process.

MIRAI [12] is an industry-backed abstract interpreter for MIR that tries to verify absence of panics, and custom verification conditions in an annotated Rust program.

2.6 Pointer Analysis

Pointer analysis is a static program analysis that determines the information about which pointers may point to which objects, as well as whether certain pointers may

alias (point to the same object). It is a core program analysis used by many data flow analyses including ours to reason about pointer values, building a call graph in the presence of function pointers, etc. In our work, we use pointer analyses to determine how unsafety spreads across the program, specifically to handle the whole program at once with function pointers, and to derive unsafety signatures for nested pointers.

There has been a plethora of work in pointer analysis. The work that is most related to this dissertation is some of the seminal work on flow-insensitive pointer analysis, along with specific improvements.

Andersen [4] presents a reduction from subset-based (also called *directional* in this dissertation) flow-insensitive points-to analysis to an iterative fixpoint problem based on transitive closure on a graph. Under Andersen's analysis, the points-to set of a pointer p subsumes the poins-to sets of all values assigned to p, without regard to the control flow of the program, hence it is flow-insensitive. Andersen's pointer analysis runs in $\mathrm{O}(n^3)$ time in terms of number of pointers in the program with standard optimizations.

Steensgaard [41] presents a faster (in almost linear time) but imprecise pointer analysis based on type equality. Under Steensgaard's analysis, an assignment `p := q` in the program is interpreted as an equality constraint $\mathrm{ptsto}(p) = \mathrm{ptsto}(q)$ where ptsto denotes a mapping from pointers to their points-to sets, so it is equality-based. As mathematical equality does not have 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 $\mathrm{ptsto}(p) \subseteq \mathrm{ptsto}(q)$ for the former and the flipped version of it for the latter.

Both of these analyses have been extended to be context-sensitive to reason about the context (the call sites) a pointer occurs in, rather than . However, context-sensitive whole program points-to analysis is not feasible in practice for large programs as it

slows down exponentially in terms of the length of the context.

Another parameter of interest for pointer analysis is field-sensitivity: whether fields of a struct should be distinguished from each other. There are three levels of field-sensitivity releveant to this work, in the order of increasing analysis sensitivity:

1. *Field-based analysis*: Under this mode, all accesses to the same struct field fld (e.g. `x.fld`, `(*y[0]).fld`, ...) are mapped to a single variable for the purposes of program analysis. In this mode, different fields of the same struct (or different structs) are distinguished from each other but the same field of different object are indistinguishable. This is the sensitivity mode induced by Rust's type system.

2. *Field-insensitive analysis*: Under this mode, objects are merged with their fields (so, `x` and `x.fld` are indistinguishable), but field accesses to different objects are distinguished. In a sense, this sensitivity level is the opposite of the sensitivity level above. This sensitivity level is a common optimization in pointer analyses as the next level can be too expensive, and the previous level is too imprecise. However, field-insensitive analysis usually needs some refining in order to be useful as it mixes pointers of different types as it merges fields.

3. *Field-sensitive analysis*: Under this sensitivity level, an object of a structure type is distinguished both from its fields and all other structures in the program. Under subset constraint-based systems, this is achieved through a constructor encoding similar to the encoding presented below [3].

Pearce et al. [36] 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. Pearce et al. also discuss existing work using this framework to encode precise field-sensitivity within this framework by representing each struct value with a constructor.

2.7 Inferring Pointer Safety in C

CCured [35] is an extension of C with run time-checked (run time-safe) pointers. It uses a undirectional (equality-based) analysis to classify pointers as checked vs. wild. For checked pointers, it inserts dynamic nullability checks as well as bounds checks for pointers involved in pointer arithmetic. Checked C [18] introduces casts backed by possible runtime checks, where the casts assert related information like nullability or array bounding. Machiry et al. [31] improve on CCured's type inference algorithm by identifying equality-based analysis as the cause for the spread of unsafety and introducing casts at call sites to enforce boundaries between safe and unsafe pointers. These casts are either enforced at run time, or manually verified by the programmer to eliminate the runtime overhead. They implement their method on top of Checked C. These works do not enforce complete memory safety and are not ownership-based (so their safety guarantee is limited to only nullability and array bounds checking at runtime), thus they are not directly applicable to the goal of translating C to safe Rust.

Chapter 3

Classifying and Understanding Unsafety

[In] The Celestial Emporium of Benevolent Knowledge [...] the animals are divided into: (a) belonging to the Emperor, (b) embalmed, (c) trained, (d) piglets, (e) sirens, (f) fabulous, (g) stray dogs, (h) included in this classification, (i) trembling like crazy, (j) innumerables, (k) drawn with a very fine camelhair brush, (l) et cetera, (m) just broke the vase, (n) from a distance look like flies.

– Jorge Luis Borges, "The Analytical Language of John Wilkins"

Before tackling the challenge of translating C programs to *safe* Rust programs, we need to define a notion of unsafety, understand what kind of unsafe behaviors Rust programs translated from C, why the programmers use the unsafe language features, their prevalence in the program, and finally what information we need to derive to resolve each unsafe behavior. In this chapter, we investigate the various sources of unsafety in Rust programs that have been translated from C using C2Rust.

In order to define unsafety in the context of translation from C, we fix a corpus of C programs to analyze in Section 3.1. Then, we concretize our notion unsafety, and

investigate the prevalence of each cause of unsafety, along with why they are used in Section 3.2. Our categorization of unsafety follows the Rust language specification while incorporating possible behavior in programs translated from C. Finally, we sum up our discussions in this chapter in Section 3.4.

Our overall findings indicate that there is a large overlap between different causes of unsafety, and we suggest the following order for tackling unsafety based on their prevalence in our corpus: unsafe pointers, memory allocation (manual memory management), mutable global variables, external functions, unsafe casts, untagged unions, and inline assembly. We postulate that unsafe pointers are a good starting point because they are common, they don't involve any non-Rust code (unlike external functions), and reasoning about them would also build up to reasoning about allocations (the second item we suggest).

While there are existing studies of unsafe code in the native Rust ecosystem [8, 38] our investigation is specifically about automatically translated Rust programs, which may have a different distribution of unsafe code than Rust programs written by developers.

3.1 C Program Corpus

Previous studies of unsafe Rust code have taken advantage of large repositories of native Rust programs such as `crates.io`. There does not exist a large repository of Rust code that has been translated from C, and so we must create our own corpus of C programs. While there are many existing C programs to choose from, each translation requires a fair amount of manual labor to correctly insert C2Rust in that C program's particular build process, and also C2Rust itself does not work on all C programs and build environments.

Table 3.1: The corpus of C programs, ordered by Rust lines of code. Programs coming from the C2Rust manual are marked with **bold**. LoC = lines of code, not counting comments or blank lines. The tulipindicators and robotfindskitten are abbreviated as TI and RFK, respectively.

Program	Domain	C LoC	Rust LoC	Functions	`unsafe` Functions
qsort	Algorithms	27	39	3	3
libcsv	Text I/O	1,035	951	23	23
grabc	GUI Tool	224	994	7	6
urlparser	Parsing	440	1,114	22	21
RFK	Video games	838	1,415	18	17
genann	Neural nets	642	2,119	32	27
xzoom	GUI tool	776	2,409	11	10
lil	Interpreters	3,555	5,367	160	159
snudown	Markdown parser	5,002	6,088	92	92
json-c	Parsing	6,933	8,430	178	178
libzahl	Big integers	5,743	10,896	230	230
bzip2	Compression	5,831	14,011	128	126
TI	Time series	4,643	19,910	234	229
tinycc	Compilers	46,878	62,569	662	625
optipng	Image processing	87,768	93,194	576	572
tmux	Terminal I/O	41,425	191,964	1,371	1,370
libxml2	Parsing	201,695	430,243	3,029	3,009
Total	—	413,428	851,674	6,773	6,694

We have collected 17 open source C programs of various sizes and application domains, as shown in Table 3.1. 11 of the programs came from the C2Rust manual [23] (marked with **bold** in the table); the remaining six came from GitHub. We picked programs from a variety of application domains, as described in the table. Table 3.1 shows that, on average, the translated Rust programs are 1.8× larger than their C counterparts. Decreases in translated LoC arise because C2Rust removes obviously dead or unreachable code. Increases in translated LoC come from macro expansion, adding function declarations for functions included from the headers, translation of increment and decrement operators[1], and annotations such as `#[no_mangle]` and `#[repr(C)]` to make the Rust code compatible with the C ecosystem.

Table 3.1 also shows that the vast majority of functions in the translated code are marked `unsafe`. Specifically, all translated functions directly from the original C program are marked `unsafe`, and only auxiliary functions generated and introduced during the translation itself are marked safe. Although all functions directly coming from C are conservatively marked `unsafe` by the translation, we observe that some do not actually require the `unsafe` tag. In Section 3.2 we quantify how many functions are unnecessarily marked `unsafe` by the translation. Furthermore, we characterize different sources of `unsafe` and quantify how prevalent they are in the program.

3.1.0.1 Threats to Validity

Our corpus of C programs is limited in number because of the manual effort required to: (1) convert each C program to a corresponding Rust program with necessary adjustments to their respective build processes; and (2) reorganize the code (such as unit tests) in a way that Cargo, the de-facto standard build system for Rust, can build

[1]Rust does not have increment-and-return operators like `++x` and assignments do not return the left-hand side, so these operators are translated into multiple statements in Rust.

the resulting Rust project reliably. The size of the corpus means that the percentages we report may not reflect the percentages of a larger pool of C programs. We have selected different C programs from a variety of domains to help increase the validity of our corpus and to try to generalize results.

3.2 Provenance of Unsafety

The Rust Reference [45] defines the following sources of unsafety:

1. Dereferencing a raw pointer
2. Reading from or writing to a mutable global (i.e., `static`) or external variable
3. Reading from a field of a C-style untagged union
4. Calling a function marked `unsafe` (including external functions and compiler intrinsics)
5. Implementing a trait that is marked `unsafe`

These categories are too coarse-grained for our purposes. In particular, Category 4 includes almost all calls to the other functions in the program, as nearly all functions in the program are initially marked `unsafe`. Category 4 also includes the use of inline assembly and unsafe casting, which we would like to separate from other sources of unsafety for our study.

We have refined the official categories above into distinct *features*, where each feature reflects a particular unsafe feature in Rust. These features give us a clearer picture of programs translated from C. Since none of the programs in our corpus implement any unsafe traits (they only implement traits that can be derived by the compiler, which are all safe), we do not consider Category 5 further. Programs in our corpus call external

functions extensively (e.g., `malloc`), making external function calls (Category 4) a major source of unsafe function calls. We count calls to `malloc` and `free` separately from other external function calls, as we conjecture that most of the allocation-related external calls can be converted to safe memory allocation mechanisms in Rust such as `Box::new`. In our corpus, the only unsafe Rust standard library function called is `std::mem::transmute`, used for reinterpreting/casting a value. We exclude calls to `std::mem::transmute` when it is used for casting byte arrays to C-style character arrays (which is safe under the assumption made by C2Rust that a character is 8 bits). The resulting features that we measure for our corpus are as follows, where the text in `bold` indicates the column names in our tables:

- **RawDeref**: dereferencing a raw pointer;
- **Global**: reading from, writing to, or making a reference to a mutable global (`static`) or external variable;
- **Union**: reading from a field of a C-style untagged union;
- **Allocation**: direct external function calls to `malloc` and `free`;
- **Extern**: calling an external function other than a function defined in another module in the same program,[2] `malloc`, or `free`; or making an indirect call via a function pointer;[3]
- **Cast**: unsafe casting using `std::mem::transmute`;
- **InlineAsm**: using inline assembly.

[2] C2Rust uses `extern` declarations to import functions from other modules in the same program. These functions can be imported directly as non-external functions after the changes described in Section 4.1, so we do not count these functions as external functions in our study.

[3] An indirect call could be calling an external function, and just like an extern call the compiler can only see the function signature of the callee but not the body.

We collect our data on a function-level because (1) C2Rust marks functions `unsafe` rather than inserting `unsafe` blocks,[4] and (2) existing work on quantifying unsafe behavior of Rust programs in general [8] aggregates the relevant information on a function level because different developers may prefer to use different granularities for `unsafe` blocks.

An important omission in our categories of unsafety is that of direct calls to `unsafe` functions (i.e., the original Category 4 above). As previously mentioned, this category is not useful for our translated corpus because almost all function calls are to `unsafe` functions, and what we are interested in is *why* the functions are unsafe. For this reason, we count sources of unsafety differently from any existing work: a function is unsafe in relation to some category above not only if it directly contains unsafe code relevant to that category, but also if it directly or transitively calls a function that is unsafe due to that category. In other words, we count a function as unsafe for a category if executing that function can exhibit unsafe behavior relevant to that category. To calculate this, we build a call graph and propagate unsafe behavior from callees to their transitive set of callers. We use a transitive metric since our ultime goal is to see how many functions the compiler could prove safe if a specific cause of unsafety is fixed.

For each unsafe feature, we collect the following information for our study: 1. How many unsafe functions in the program use the `unsafe` feature, directly or transitively (i.e., how many functions need the unsafe feature), 2. How many unsafe functions in the program use *only* this unsafe feature, 3. How many times a use of the unsafe feature appears in the program text, and 4. The total size (in lines of code) of unsafe functions that directly or transitively use the unsafe feature

To get the feature counts for item 3 in the above list, we first convert the translated

[4] Except when generating shims for the `main` function, which cannot be marked `unsafe`. These shims extract the program arguments then immediately call the `main` function from the C program.

Rust programs to Rust High-level IR (HIR)[5], an AST-based representation. From there, we count individual features in the HIR in the following ways: for pointer dereferences, we count the number of raw pointer dereference nodes[6]; for inline assembly, we count the number of inline assembly nodes; for interaction with mutable or external globals, we count how many times these variables are used (read from, written to, or taken a reference of) in the source code.; for reading from a union, we count each field access involving a union, *unless* it is immediately on the left-hand side of an assignment; and for memory allocation, external functions, and unsafe casting, we count the number of static call sites to the relevant functions.

Table 3.2 lists how many times each source of unsafety statically appears for each program in our corpus. We observe that there are two sources which do not appear across many programs, namely C-style unions (which appear only in larger programs) and inline assembly (which is only used in one program). Table 3.2 shows that the most common source of unsafety is raw pointer dereferencing, which is eight times more common than the next most common source (globals), followed closely by external function calls. The number of direct calls to `malloc` and `free` (Alloc) was occasionally surprisingly low, as with libxml2; upon observation of libxml2's codebase, it uses custom memory allocation functions almost everywhere, limiting the number of static allocation sites our analysis could find.

Table 3.3, in contrast to Table 3.2, takes a function-level approach, counting the number of functions directly or transitively affected by each category of unsafety. We record functions that are uniquely affected by a single category of unsafety (under the

[5]HIR is used internally in the Rust compiler, and is close to initial AST obtained after expanding macros, type checking, and normalizing loops and conditionals. We chose to use HIR because it provides type information needed by our analyses and it is close to the source code.

[6]At the minimum, a static analysis must consider all dereferences to ensure the safety of raw pointers. As such, analysis cost is expected to increase with the number of dereference nodes, making this an interesting feature to track.

Table 3.2: How many times different categories of unsafety appear in each corpus program. The meaning of each column is explained in Section 3.2.

Benchmark	Union	Global	InlineAsm	Extern	RawDeref	Cast	Alloc
qsort	0	0	0	0	10	0	0
grabc	6	15	0	31	21	0	0
libcsv	0	2	0	35	174	4	0
RFK	0	127	0	87	24	0	2
urlparser	0	1	0	122	60	43	55
genann	0	164	0	188	339	3	5
xzoom	15	455	0	76	172	0	2
lil	0	10	0	149	1668	11	62
snudown	0	19	0	104	842	0	9
json-c	101	93	0	208	1843	17	30
bzip2	0	700	0	424	3764	1	14
TI	0	108	0	352	1847	84	9
libzahl	0	430	29	63	2457	0	43
tinycc	613	2552	0	465	5632	31	2
optipng	82	1361	0	816	6062	37	43
tmux	74	769	0	2707	21658	161	599
libxml2	499	3571	0	4593	52546	15	15
Total	1390	10377	29	10420	99119	407	890

$\exists!$ columns) and those that are affected by multiple categories of unsafety including this one (under the $\exists_{\geq 2}$ columns). The $\exists_{\geq 2}$ columns will count a function multiple times, once for each category it is affected by. Functions which were marked `unsafe` by the translation but nonetheless are devoid of unsafe behavior are totalled in the false positives (FP) column; we observe that 6% of functions fall into this category. Tables 3.2, and 3.3 show RawDeref, Global, and Extern to be the biggest sources of unsafe behavior, typically in that order. However, while RawDeref is heavily overrepresented in terms of sheer usage (Table 3.2), at the function level it compares much more closely to Global and Extern (Table 3.3). From the standpoint of trying to make more functions safe, this is an important observation to make, as it shows that RawDeref is not much more important than Global or Extern.

3.3 Underlying Causes of Unsafety

We now investigate the behaviors in the original C programs that lead to each category of unsafety. Some categories of causes are obvious and uninteresting: mutable globals (Global) and dynamic memory allocation (Allocation) are needed in C programs for creating long-lived objects that are accessible from different parts of the program; inline assembly (InlineAsm) is used in only one of the programs in our corpus (`libzahl`) for architecture-specific optimizations. We examine the remaining categories in more detail below.

3.3.1 Raw Pointers

We inspected the translated corpus programs and how they use raw pointers in detail. We recognize five distinct reasons that a program might have for using a raw pointer:

Table 3.3: Number of functions affected by each category of unsafety (a function may be counted multiple times if affected by multiple categories). FP denotes false positives: functions that do not contain any unsafe behavior but are marked `unsafe` by C2Rust. The column labels are explained in Section 3.2.

Program	Union		Global		InlineAsm		Extern		RawDeref		Cast		Alloc		FP
	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	
qsort	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0
grabc	0	3	1	4	0	0	0	4	1	5	0	0	0	0	0
libcsv	0	0	1	1	0	0	0	9	13	22	0	4	0	0	0
urlparser	0	0	0	14	0	0	0	20	0	17	0	1	0	19	0
RFK	0	0	0	15	0	0	1	15	0	7	0	0	0	2	1
genann	0	0	0	14	0	0	1	24	0	21	0	13	1	18	2
xzoom	0	1	1	10	0	0	0	9	0	8	0	0	0	4	0
lil	0	0	2	73	0	0	1	134	14	148	0	52	1	100	2
snudown	0	0	1	37	0	0	0	63	19	90	0	0	0	34	1
json-c	0	62	10	49	0	0	4	114	24	144	0	49	1	51	11
bzip2	0	0	3	79	0	0	7	85	23	82	0	3	2	26	6
libzahl	0	0	0	115	0	111	0	114	90	230	0	0	0	110	0
TI	0	0	0	13	0	0	1	104	74	175	0	73	1	16	49
tinycc	0	286	5	492	0	0	8	498	54	577	0	244	1	358	30
optipng	0	57	4	297	0	0	14	371	126	487	0	57	7	141	29
tmux	0	569	9	710	0	0	9	1030	244	1328	0	489	1	653	5
libxml2	0	198	28	2220	0	0	39	2359	369	2740	0	1156	0	1268	183
Total	0	1176	65	4143	0	111	85	4953	1054	6081	0	2141	15	2800	319

- The raw pointer appears as part of the public signature of an API implemented by the program. This is a common occurrence in the programs in our corpus because most of them (except `lil` and `RFK`) are either libraries or contain libraries.
- The raw pointer is obtained via custom memory allocation (i.e., calling `malloc`). These raw pointers could be converted to safe references if we replace `malloc` with Rust's safe memory allocation and compute suitable lifetime information for them.
- The raw pointer is obtained via a cast to or from `void*`. In all cases this reason turns out to be the result of an idiomatic C method for overcoming C's lack of generics and implementing polymorphism. These raw pointers could be converted to safe references by introducing generics or traits to implement polymorphism.
- The raw pointer is passed as an argument to, or returned from, an external function call. These raw pointers can only be converted into safe references by replacing the external call.
- The raw pointer is used in pointer arithmetic. Because arrays in C decay to pointers, this reason captures most array accesses (unless the array has a fixed size known at compile time). Rust does not allow pointer arithmetic on safe references, but these raw pointers could be converted to safe references if we can convert the pointer arithmetic into safe array slices.

In our data collection we group the first two categories above into a single category named `Lifetime` because converting these raw pointers into safe references requires computing the same information for both categories and does not involve much invasive code transformation beyond changing the pointer declarations and inserting

lifetime information. Note that deriving the lifetime information is needed for making pointers safe in all categories, so `Lifetime` specifically denotes pointers that do not fall into any other category. The remaining categories are named `VoidPtr`, `ExternPtr`, and `PtrArith` respectively. For each category of raw pointer we collect the following information, using the same methodology as for Section 3.2:

1. Number of declared pointers involved in that category (Table 3.4);
2. Number of dereferences of pointers in that category that appear in the code (Table 3.5);
3. Number of unsafe functions that use pointers from that category (Table 3.6).

A pointer may be contained in multiple categories (e.g., a pointer returned by `malloc` that undergoes pointer arithmetic and is then passed to an external function). As in Table 3.3, we split our counts into pointers that uniquely belong to a particular category ($\exists!$) and those that belong to that category but also others ($\exists_{\geq 2}$). A raw pointer may be involved in multiple overlapping causes, so the sum of the other columns is greater than the Total column for all three table. Because the `Lifetime` category contains only pointers not involved in other categories we only give the $\exists!$ column for it. For counting the number of unsafe functions in Table 3.6 we only consider those functions for which raw pointers are the only reason for their unsafety; that is, we do not consider functions that use global variables, unsafe cast, inline assembly, or read from a C-style union. "Using" a pointer means any one of declaring (as a parameter or in the function body) or dereferencing the pointer. As a reminder, we consider a function to use a pointer either if the function does so directly, or calls (directly or transitively) a function that uses the pointer.

To determine how the pointers are being used we implemented and executed a flow-insensitive, field-based taint analysis based on Steensgaard-style pointer analysis

Table 3.4: Raw pointer declarations, grouped by category. $\exists!$ and $\exists_{\geq 2}$ are explained in Section 3.2. Lifetime category contains only unique ($\exists!$) causes by definition.

Program	VoidPtr		PtrArith		ExternPtr		Lifetime	Total
	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	
qsort	0	0	2	0	0	0	2	4
grabc	0	0	1	0	5	0	7	13
libcsv	7	10	0	7	2	3	18	37
urlparser	0	70	0	70	4	70	5	79
RFK	0	0	1	0	1	0	0	2
genann	0	61	0	62	6	62	5	73
xzoom	0	24	1	25	3	25	0	29
lil	1	314	60	316	10	317	50	438
snudown	0	159	2	161	47	156	31	244
json-c	13	227	1	227	9	211	41	297
bzip2	43	89	47	70	9	89	37	227
libzahl	9	324	114	322	3	319	7	457
TI	15	41	724	41	4	41	82	866
tinycc	18	1,100	16	1,094	24	1,084	191	1,352
optipng	12	1,016	121	987	51	1,013	207	1,407
tmux	265	3,550	17	3,311	177	3,554	622	4,645
libxml2	451	8,332	171	7,729	152	8,336	839	9,950
Total	834	15,317	1,276	14,422	507	15,280	2,142	20,116

[41] and Rust's type system [45]. We chose a flow-insensitive, equality-based analysis because all values that flow into a variable and from the variable are necessarily of the same type, and if any one of those values is used for a reason on our list then that reason forces that variable and all of the places its value flows to be a raw pointer. We consider a pointer to belong to a particular category (`Lifetime`, `VoidPtr`, `ExternPtr`, or `PtrArith`) if the pointer may contain a value that is potentially obtained from a source relevant to that category (e.g., the result of a pointer arithmetic operation, the return value of an external call, a value of type `* const void` or `* mut void`) or if its value may flow into a sink relevant to that category (e.g., pointer arithmetic, or an argument to an external call, or a value that is cast to a `void` pointer).

Tables 3.4 to 3.6 contain the results of our analysis. Tables 3.4 and 3.5 can be used

Table 3.5: Raw pointer dereferences, grouped by category. $\exists!$ and $\exists_{\geq 2}$ are explained in Section 3.2. Lifetime category contains only unique ($\exists!$) causes by definition.

Program	VoidPtr		PtrArith		ExternPtr		Lifetime	Total
	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	
qsort	0	0	6	0	0	0	4	10
grabc	0	0	2	0	4	0	15	21
libcsv	0	26	0	26	0	17	148	174
urlparser	0	2	0	2	0	2	58	60
RFK	0	0	24	0	0	0	0	24
genann	0	312	22	313	4	313	0	339
xzoom	0	37	23	114	12	105	23	172
lil	0	895	127	897	8	897	636	1,668
snudown	0	489	35	493	185	474	129	842
json-c	9	1,639	39	1,646	56	1,433	93	1,843
bzip2	1,704	1,192	173	627	11	1,195	679	3,764
libzahl	1	1,220	1,183	1,220	22	1,191	31	2,457
TI	426	184	1,237	184	0	184	0	1,847
tinycc	28	4,525	122	4,522	9	4,491	946	5,632
optipng	5	5,212	203	5,043	36	5,208	606	6,062
tmux	1,002	17,687	131	16,449	345	17,694	2,486	21,658
libxml2	986	45,764	372	41,475	235	45,771	5,175	52,546
Total	4,161	79,184	3,693	73,011	927	78,975	11,025	99,109

Table 3.6: Functions using raw pointers in a given category. $\exists!$ and $\exists_{\geq 2}$ are explained in Section 3.2. Lifetime category contains only unique ($\exists!$) causes by definition.

Program	VoidPtr		PtrArith		ExternPtr		Lifetime	Total
	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	
qsort	0	0	2	0	0	0	1	3
grabc	0	0	0	0	1	0	1	2
libcsv	1	5	0	3	0	4	12	18
urlparser	0	5	0	5	0	5	2	7
robotfindskitten	0	0	0	0	0	0	2	2
genann	0	6	0	6	0	6	3	9
xzoom	0	9	0	5	0	8	0	9
lil	1	62	9	61	0	62	6	78
snudown	0	47	0	47	2	47	6	55
json-c	6	50	0	50	1	47	20	77
bzip2	7	28	2	19	0	28	4	41
libzahl	0	79	9	79	0	79	2	90
tulipindicators	2	3	96	3	0	3	45	146
tinycc	3	76	5	75	0	65	35	119
optipng	4	175	12	171	5	177	62	260
tmux	26	483	2	467	7	482	31	549
libxml2	20	532	6	492	7	535	210	779
Total	70	1,560	141	1483	23	1,548	441	2,241

to show that 77% of raw pointer declarations, and 80% of raw pointer dereferences use pointers that are (sometimes indirectly) involved in multiple causes (these percentages are obtained by subtracting all unique causes from the total in each table). The highest unique cause of raw pointer declarations and dereferences is the `Lifetime` category (9.5% and 10.0% respectively). However, the most prominent cause may depend on the program. For example, the highest contributing categories (in all 3 metrics) are `VoidPtr` in bzip2 which uses `void *` for polymorphism in order to share code between encoding and decoding stages, and `PtrArith` in TI which is a time series analysis library using and passing around dynamically allocated arrays. Finally, 70% of the functions use raw pointers for more than one reason, and 20% of these functions use pointers stemming from only `Lifetime`.

3.3.2 External Function Calls

We break this investigation down into two questions: (1) How prevalent are calls to specific external functions? (2) Which external functions have the highest impact on safety? To answer these questions, we focus on the external functions and the internal functions that are, directly or transitively, made unsafe due to calls to those external functions. These internal functions may be unsafe for other reasons as well, but for this investigation we ignore other causes of unsafety.

3.3.2.1 How prevalent are the external functions across benchmarks?

It would be useful to know if there are a small set of external functions that appear across many benchmarks, making them an attractive target for replacement. There were 409 external functions used in total across all of our benchmarks. We observe that 73% of the external functions are unique to a particular benchmark,

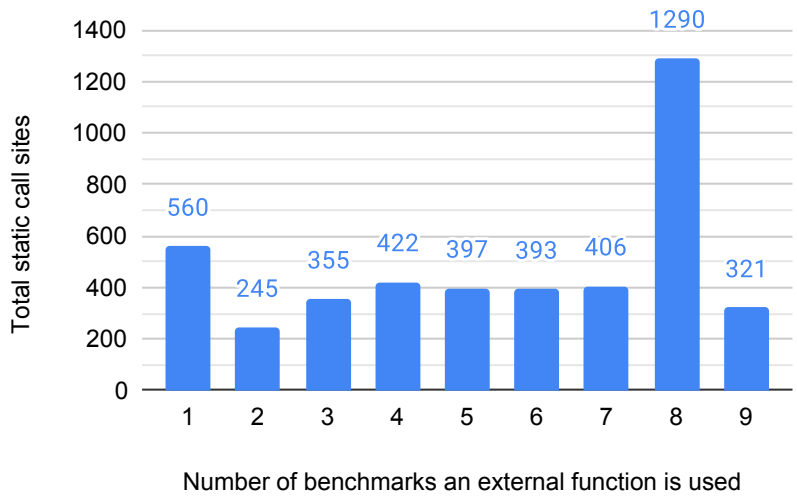


Figure 3.1: Number of calls to external functions based on how many benchmarks use them

and that no external function is used by more than 9 (out of 12) of our benchmarks. Only 11 functions (namely `fprintf`, `strcmp`, `memset`, `printf`, `strlen`, `strncmp`, `exit`, `memcpy`, `realloc`, `fopen`, and `fclose`) are used in more than half of the programs. These 11 functions account for 43% of all external function calls, indicating that looking at the functions used across many programs might be a useful heuristic for picking which functions to replace with safe alternatives first.

Most of these functions deal with string manipulation or I/O. Figure 3.1 divides the external calls into bins based on how many benchmarks use them, then counts for each bin how many static call sites to a function in that bin appear across the benchmarks. For example, the column labeled '2' shows that there were 245 static call sites to an external function that appears in exactly two benchmarks. The total calls to functions used in exactly 8 benchmarks is much higher due to `fprintf`, which is called from 858 places across the benchmarks. We can see that the external functions listed above as common across 7 or more benchmarks account for 43% of all external function calls.

3.3.2.2 What are the external functions with the highest impact on unsafety?

Another useful statistic for prioritizing external functions is their relative impact on unsafety. In order to measure this factor, we investigate:

- The number of static call sites for each external function
- In how many functions an external function is called (directly or transitively)

Figure 3.2a shows an optimal ordering of external functions that maximizes cumulative static call sites. Overall, only seven external functions need to be replaced by safe alternatives to eliminate more than half (52%) of the external function calls. The most common functions in this ordering are similar to the most common functions reported above. The ten most commonly called functions we encounter in order are: `fprintf`, `strcmp`, `memset`, `printf`, `memcpy`, `strlen`, `snprintf`, `__assert_rtn`, `__ctype_toupper_loc`. Here, `__assert_rtn` is the C library function used in implementing the `assert` macro which can be replaced by Rust's safe `assert!` macro, and `__ctype_toupper_loc` is an implementation detail of the `toupper` function in C which has a safe counterpart in the Rust standard library.

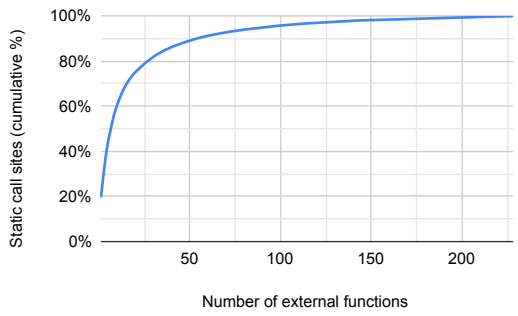
The other statistic we focus on is the external functions that make the highest number of functions unsafe (that is, the external functions that are called from the most functions, directly or transitively). Figure 3.2b shows how many transitive callers each external function has, both as absolute value and percentage. The ten extern functions that have most transitive callers in our benchmarks in order are: `memset`, `memcpy`, `__xmlRaiseError`, `strlen`, `snprintf`, `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_mutex_init`. Each of these functions contribute to the unsafety of 32.5–54.6% of the extern-calling functions. Here, `__xmlRaiseError` and the `pthreads`-related functions are used only by our largest benchmark, `libxml2`. `__xmlRaiseError` is

an external function because of how libxml2 is linked: some features such as error reporting are linked from support modules that are compiled separately from the main program. This fact shows that an effort to make the whole benchmark project link in an idiomatic way for a Rust program can reduce pervasive external calls.

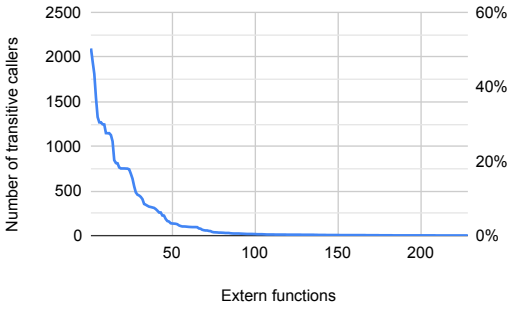
Some of the most-called external functions above are specific to a single benchmark. To assess the impact of external functions that are not specific to one benchmark, we applied the same analysis restricted to external functions used in more than one benchmark. Figure 3.2c shows how many transitive callers each external function used in more than one benchmark has. The 10 most called external functions with this restriction in order are: `memset`, `memcpy`, `strlen`, `snprintf`, `fprintf`, `memmove`, `__errno_location`, `memcmp`, `strchr`, `strcmp`. These functions contribute to the unsafety of 19.9–54.6% of the functions. Each function in this list is called in at least 4 benchmarks, except `__errno_location`, which is called in 3 benchmarks and it comes from accessing the `errno` variable in the C standard library. This list is similar to the previous list for the prevalence question in that it consists mainly of string manipulation, copying/initializing arrays in memory, and I/O.

3.3.3 C-style Unions

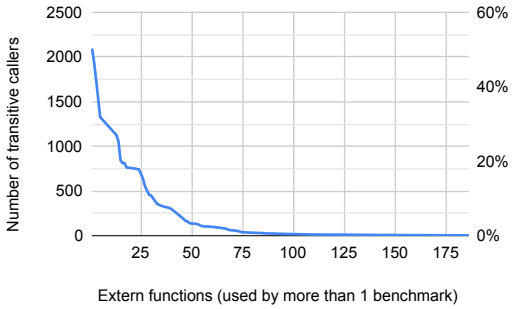
We manually inspected all C-style unions declared in the programs in our corpus. Most of these were defined by the C developers with accompanying tag data in order to manually implement a tagged union. In some programs, the tag information was not stored with the union data but rather inferred from invariants that hold at a given program point. `libxml2` contains declarations for pthreads-related unions used in external calls; however, these unions are used only by pthreads functions and never read directly by the Rust program so they do not contribute to unsafety. Apart from



(a) Cumulative number of calls in program text for external functions, x-axis is ordered from the functions with most call sites to least



(b) Number of transitive callers each extern function has, ordered by most to least.



(c) Number of transitive callers each extern function used by more than one benchmark has, ordered by most to least.

Figure 3.2: Impact of external function calls on unsafety.

these, none of the unions in our programs are passed to or obtained from external functions, and we conjecture that they can be replaced with safe tagged unions (Rust `enums`) to reduce the use of C-style unions in the program. However, this transformation would yield highly un-idiomatic Rust code which would check the type of the union twice in the cases where there is an explicit tag that the C program checks.

3.3.4 Unsafe Casting

We inspected the calls to `mem::transmute` generated by C2Rust. There are two uses of unsafe casting in the translated corpus programs: (1) converting 8-bit byte arrays to C character arrays (different from Rust strings) which corresponds to 79% (456 out of 576) instances of unsafe casting, and (2) converting between function pointers[7] and `void *` which corresponds to the remaining 21% (120) unsafe casts. The first option is safe on architectures using 8-bit unsigned characters (most modern architectures), and can be put behind a wrapper function.

3.4 Observations and Discussion

From Table 3.3, we can see that most functions are affected by multiple categories of unsafety: for each category, the number of functions uniquely affected by that category is 0–1% of the total number of functions affected by that category, with RawDeref being an outlier at 16%. Unions, inline assembly, and casts never appear by themselves at all. These numbers indicate that making translated Rust programs safer is a multi-faceted problem, in that fixing a single category of unsafety will not make a large impact on the number of unsafe functions. Only by fixing multiple categories can we hope to make a

[7] Function pointers are represented in Rust as optional references rather than raw pointers, so casting them directly to and from raw pointers is unsafe.

significant difference. It is also possible that division into finer categories would yield a categorization which is less inter-dependent, though we believe these categories are sound, given that they are rooted in the sources of unsafety defined directly by the Rust developers.

Because an effective method for making translated Rust programs safe needs to handle multiple categories of unsafety, an interesting question is how to prioritize which categories to handle. To answer this question, we graph the cumulative impact of fixing categories highest-to-lowest according to the following heuristic order of impact: *raw pointer dereference*, *memory allocation*, *extern calls*, *access to globals*, *unsafe casts*, *access to unions*, *inline assembly*. This ordering was selected by searching through all possible orderings and finding the one where each added cause had the highest added impact in terms of the cumulative number of functions made safe. We then adjusted this ordering by moving allocations to the second place from the fourth place, as allocations are a source of raw pointers with a simple fix (i.e., rewriting them to create a new `Vec` or `Box`).

To assess the potential of solving these problems in this order, we calculate the cumulative impact of how many unsafe functions become safe as each of these categories of unsafety are eliminated. Figure 3.3 shows the results of this calculation. We include both the results for all functions in the programs in our corpus, and the result for the four largest programs in order to demonstrate the variability of the results. In this graph we include the functions unnecessarily marked unsafe. The results on the graph indicate that, in order to make more than half of the functions safe, we need to handle the four most common sources in our list. Also, the the graph (along with the tables) shows that the impact of unsafe casts and C-style unions vary considerably depending on the program.

An important issue for translating C to safer Rust is how the translation can derive

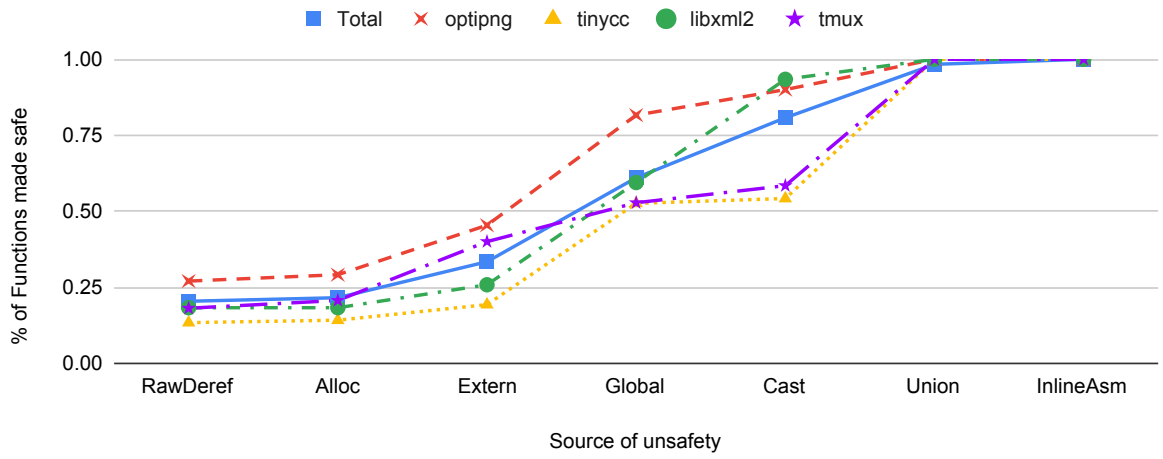


Figure 3.3: Cumulative percentage of functions made safe by fixing the given unsafety category. The "Total" line shows this number for all functions across all programs.

the necessary information required to produce verifiably safe code. Ultimately, unsafety stems from the fact that the compiler does not have enough information about a piece of code (e.g., the underlying types in the case of `void` pointers, or the code being executed in the case of external functions). While some unsafety is likely unavoidable (e.g., unsafety needed to implement a memory allocator), other unsafety is rooted in C's lack of language features. For example, C's untagged unions could be safety replaced with tagged unions, and certain uses of `void` pointers could be safety replaced with generics. In all cases except external functions and internal assembly, the translator would have access to the code being executed and thus at least in theory could use static analysis to derive the information required to make the translated program safe. However, some C programs would require a fairly deep analysis and rewriting strategy that operates at a higher level than just translating the direct operational semantics of the program.

For example, a C program that uses a pointer-based graph data structure cannot be trivially translated to Rust because Rust's affine type system cannot represent such

a data structure. Rust does have mechanisms for representing graphs like this (using alternative data structures or using hand-verified `unsafe` code in the Rust standard libraries), but translating the C program to use those mechanisms requires a more holistic view of the code. Similarly, we have to account for the different abstractions that the two languages use. Idiomatic safe Rust code and safe Rust code translated directly from C can look very different because of the abstractions that Rust and the Rust standard library provide. For example, safe code translated from C may use `while` loops and indexing to go over arrays and other data structures whereas an idiomatic Rust program would use mutable and immutable iterators and higher-order functions such as `map` for the same purpose.

So, "low-level" translation approaches may leave some of the program unsafe because the original program may contain inherently unsafe parts, as it is not written with Rust's notion of safety in mind. So, another interesting research direction is exploring how unsafety spreads throughout the program in general (as opposed to focusing on specific causes of unsafety as we did here) to also answer questions about effectiveness of such incomplete methods. We are going to investigate this problem later in Chapter 5.

Nevertheless, translating C to safer Rust is a worthwhile goal and, we believe, a reasonable endeavour to undertake. We show in Chapter 4 a first attempt at doing so that targets one particular source of unsafety with good success. Addressing the remaining sources of unsafety, and the issues discussed above, will provide a rich vein of research problems for some time to come.

Chapter 4

Deriving Lifetime and Ownership Using the Compiler as an Oracle

In this chapter we describe a first attempt to automatically translate a C program into a safer Rust program, building on top of the C2Rust syntactic translation from C to completely unsafe Rust. While our study in Chapter 3 shows that addressing only a single category of unsafety is insufficient for removing the majority of unsafety in translated programs, we nonetheless need a starting point. Since we observe that raw pointer dereferences are the biggest sole contributor to unsafety, and furthermore that rewriting raw pointers to safe references requires resolving ownership and lifetime information, we decided to start with addressing the `Lifetime` category. `Lifetime` will have an immediate impact on some programs, and provide much-needed lifetime information to reason about other forms of unsafety. Thus, our goal is to translate a subset of the raw pointers in the `Lifetime` category to safe references.

A raw pointer can be converted into a safe reference if, in the resulting rewritten program, the Rust compiler can prove that the reference guarantees a single owner and can also derive the appropriate lifetime for the object being referred to. One possible

approach would be to implement a static analysis for either the original C program or the translated unsafe Rust program to compute this information; however, the drawbacks of such an approach are: (1) designing and implementing an efficient, useful analysis that can reason about aliasing and lifetime information in conformance with Rust's sophisticated type system is highly non-trivial; and (2) even if the implemented analysis can prove safety, that doesn't matter unless the Rust compiler can *also* prove safety, i.e., the analysis must be tuned to be no more precise than the Rust compiler.

Our key insight is that *we can piggy-back on top of the Rust compiler* and allow it to derive the information we need to infer which `Lifetime` raw pointers can be made safe. To do so, we first optimistically rewrite the unsafe Rust program to convert all of the relevant raw pointers into safe references, making optimistic assumptions about mutability, aliasing, and lifetimes. This optimistic version is very unlikely to compile—but the errors that the Rust compiler derives while attempting to compile it allow us to refine our initial optimistic program into a more realistic version. By iterating this process in a loop, we essentially use the Rust compiler as an oracle to continually refine the program until it passes the compiler. For this first attempt we do not try to introduce any additional memory management mechanisms (e.g., reference counting) that might allow more raw pointers to become safe, focusing purely on converting raw pointers into safe references with the same memory representation and performance characteristics; future work will investigate these other possibilities.

During our translation, we assume that any pointers passed to an API are valid pointers (`null` or a valid reference to an object) if the program dereferences them, because dereferencing an invalid pointer would result in undefined behavior in both C and Rust. Therefore, these raw pointers could be converted to safe references without changing the *defined* program behavior, if their use does not invalidate Rust's borrow checker rules.

Our method consists of two stages after the C2Rust translation of the original C program:

1. **ResolveImports**. Connect the definitions and uses of types and functions across modules, and remove unnecessary unsafety and mutability markers (Section 4.1). This stage effectively emulates what a C linker does to merge definitions, and remove unnecessary `extern` declarations. C2Rust also provides a similar method based on its knowledge of C header files to merge all declarations that come from the same header to a module.

2. **ResolveLifetimes** Determine initial lifetimes to convert unsafe raw pointers into safe references then iteratively rewrite the program to resolve lifetime inference and borrow checking errors (Section 4.2). This stage uses our key insight of using the Rust compiler as an oracle to discover lifetime and ownership constraints.

In the rest of this chapter, we use a running example to demonstrate each step of our technique. Figure 4.1 shows the initial C program which implements a binary search tree. Figure 4.2 shows the result of running C2Rust on the C program. Our technique uses this translated Rust program as an input.

The rest of this chapter is structured as follows:

- We give a description of ResolveImports along with the aforementioned running example in Section 4.1.

- We give a description of ResolveLifetimes along with the same running example, and the formal rewrite rules we use in Section 4.1.

- We evaluate these methods on the same set of benchmarks we used in Chapter 3 in Section 4.3.

- Finally, we conclude with a summary of our evaluation results and potential directions and challenges for future work in Section 4.4

4.1 Connecting Function and Data Structure Definitions across Modules

The original C program may consist of multiple compilation units (e.g., Figure 4.1 has two: `bst.c` and `main.c`). C2Rust translates each compilation unit separately into its own Rust module (e.g., Figure 4.2 has `bst.rs` and `main.rs`). However, unlike C, all Rust modules in a program are compiled together in the *same* compilation unit. Because C2Rust translates each C compilation unit separately, the translated modules contain (1) duplicate data structure declarations from shared header files; and (2) functions declared as `extern` because they are defined in a different module, even though the definitions are actually available during compilation. In Figure 4.2, note that `main.rs` contains a duplicate declaration of `node_t` and declares both `find` and `insert` as extern functions. All calls to declared extern functions must be marked `unsafe`, regardless of the fact that the functions are not truly extern. The result is that, even if we manage to make `find` and `insert` safe in the `bst.rs` module, `main_0` must remain unsafe because it contains calls to those functions and they were declared extern in the `main.rs` module.

The immediate solution is to remove the extern declarations and replace them with imports from the modules in which those functions are defined. However, doing so can cause a type error if the functions use a data structure that has been duplicated across modules. Rust's type system is nominal, and these duplicated definitions are treated as separate types. In Figure 4.2 the type `bst::node_t` and the type `main::node_t` are

```
1 // bst.h: BST node definition
2 typedef struct Node {
3   Node* left;
4   Node* right;
5   int value;
6 };
7
8 Node* find(int value, Node* node);
9 void insert(int value, Node* node);
10
11 // bst.c: BST implementation
12 #include "bst.h"
13
14 Node* find(int value, Node* node) {
15   if (value < node->value && node->left) {
16     return find(value, node->left);
17   } else if (value > node->value && node->right) {
18     return find(value, node->right);
19   } else if (value == node->value) {
20     return node;
21   }
22   return NULL;
23 }
24
25 void insert(int value, *Node n) {
26   // Implementation omitted for brevity.
27 }
28
29 // main.c: program entry point
30 #include "bst.h"
31
32 int main() {
33   Node* tree = malloc(sizeof(Node));
34   tree->value = malloc(sizeof(int));
35   *(tree->value) = 3;
36   insert(1, tree);
37   insert(2, tree);
38   *(find(3, tree)->value) = 4;
39   return 0;
40 }
```

Figure 4.1: A C program implementing a binary search tree. We omit the implementation of insert for brevity.

two different types; because the formerly extern functions are now imported and use the duplicated type, there is now a type error in the example program. In order to fix this issue, we need to detect and deduplicate these data structure declarations. After

```
// bst.rs
use std::os::raw::c_int;

#[derive(Copy,Clone)]
pub struct Node {
  pub left: *mut Node,
  pub right: *mut Node,
  pub value: c_int,
}

pub unsafe fn find(mut value: c_int, mut node:
    *mut Node) -> *mut Node {
  /* ... */
}

pub unsafe fn insert(mut value: c_int, mut node:
    *mut Node) { /* ... */ }

// main.rs
use std::os::raw::c_int;
extern "C" {
  fn find(mut value: c_int, mut node:
      *mut Node) -> *mut Node;
  fn insert(mut value: c_int, mut node:
      *mut Node);
}

// duplicate definition of Node
#[derive(Copy,Clone)]
pub struct Node {
  pub left: *mut Node,
  pub right: *mut Node,
  pub value: c_int,
}

pub unsafe fn main_0() -> int { /* ... */ }
```

Figure 4.2: The Rust program produced from Figure 4.1. Function bodies, `main`, and `main_0` are omitted for brevity, as are compiler directives for C compatibility (e.g. for disabling name mangling, ensuring C ABI, and structure field alignment).

this step, we remove unnecessary `mut` markers and `unsafe` markers. For our example, the only unnecessary `mut` markers are in the arguments of `find` and `insert`. All the `unsafe` markers in the example code are still necessary due to raw pointer dereferences. Figure 4.3 shows our example after this process.

```
// bst.rs
use std::os::raw::c_int;

#[derive(Copy,Clone)]
pub struct Node {
  pub left: *mut Node,
  pub right: *mut Node,
  pub value: c_int,
}

pub unsafe fn find(value: c_int, node: *mut Node) -> *mut Node {
  if value < (*node).value && !(*node).left.is_null() {
    return find(value, (*node).left)
  } else {
    if value > (*node).value && !(*node).right.is_null() {
      return find(value, (*node).right)
    } else if value == (*node).value {
      return node
    }
  }
  return 0 as *mut Node;
}
pub unsafe fn insert(value: c_int, node: *mut Node) { /*...*/ }

// main.rs
use std::os::raw::c_int;
use bst::{Node, insert, find};

pub unsafe fn main_0() -> int {
    let mut tree = malloc(::std::mem::size_of::<Node>()) as * mut Node;
    (*tree).value = malloc(::std::mem::size_of::<c_int>()) as * mut
    c_int;
    *(*tree).value = 3;
    insert(1, tree);
    insert(2, tree);
    *(*find(3, tree)).value = 4;
    return 0;
}
```

Figure 4.3: The Rust program from Figure 4.2 after deduplicating struct definitions and converting extern functions to imports. The unnecessary mutability annotations have been removed from the function arguments.

```
cfg ← ⊥
ComputeTaintAnalysis(cfg)
ComputeStructLifetimes(cfg)
loop
    RewriteProgram(cfg)
    errors ← RunRustCompiler()
    if errors = ∅ then Halt.
    fixes ← ResolveErrors(errors)
    cfg ← cfg ⊔ fixes
    if fixes promotes a location to owned or raw then
        ComputeTaintAnalysis(cfg)
        ComputeStructLifetimes(cfg)
```

Figure 4.4: Our algorithm for ResolveLifetimes, the parts of our method after merging struct definitions and resolving extern functions.

4.2 Computing Lifetime Information Iteratively

The core idea behind how we discover lifetime constraints in ResolveLifetimes (Figure 4.4) is starting with an optimistic version of the program, and discovering the lifetime constraints iteratively. The core data structure we use to represent the information gained from the compiler errors is *configurations*. We discuss configurations at the beginning of this section, then we are going to explain how we generate the initial program (Section 4.2.2); demonstrate how we iteratively resolve the compiler errors and realize the changes in configurations as rewrites (4.2.3); and finally analyze the complexity of our algorithm, give a termination guarantee, and discuss potential optimizations 4.2.4.

Our algorithm uses the lattice of configurations described in Figure 4.5. A configuration is a mapping from program locations to the kinds of pointers they are converted to (borrowed, owned, or raw), along with corresponding lifetime constraints. A configuration maps program locations (i.e., variables, parameters, return values, struct fields, and expressions) to the kinds of pointers they should have. In our representa-

$$\begin{aligned}
\text{Configuration} &= (\text{Location} \to \text{PtrKind}) \times (\text{Function} \times \text{LifetimeVar} \to P(\text{LifetimeVar})) \\
\text{PtrKind} &= \{\mathsf{borrowed}, \mathsf{owned}, \mathsf{raw}\} \text{ where } \mathsf{borrowed} \sqsubset \mathsf{owned} \sqsubset \mathsf{raw} \\
\text{Location} &::= x \mid e \mid \textbf{param}\ f\ n \mid \textbf{return}\ f \mid \textbf{access}\ t\ \mathit{fld} \\
& f \in \text{Function},\ t \in \text{TypeName},\ n \in \mathbb{N},\ x \in \text{Variable},\ e \in \text{Expr}
\end{aligned}$$

Figure 4.5: The lattice of configurations representing the fixes we apply based on compiler errors. LifetimeVar denotes lifetime variables. Variable and Expr denote the variables and the expressions in the program. TypeName represents a struct name, and **access** t *fld* denotes the field-based location for accessing the field *fld* of values of type t. P is the powerset operation.

tion of locations, we use HIR IDs used by the Rust compiler to represent expressions (the set Expr) using unique identifiers. This allows us to keep track of any arbitrary expression involved in a borrow conflict, and promote its pointer kind to owned or raw in a lightweight manner. The lattice of configurations are ordered lexicographically (first, according to how they map locations to pointer kinds, then if that mapping is the same, according to the set of lifetime constraints they have). Each of the maps in the configurations are defined in the structurally in the classical way: $f \sqsubset g$ if $f(x) \sqsubset g(x)$ for all elements x in the domain of f and g. We use the subset lattice when ordering sets of lifetime variables. Because there are finitely many locations, pointer kinds, and lifetime variables in a given program, the lattice of configurations is finite.

Using the locations described in Figure 4.5, we implement field-based, context-insensitive taint analyses. The data flow constraints for our taint analyses are derived from the typing, type equivalence and subtyping constraints in Oxide (Figures 4, 5, and 6 in [46]). We use the type equivalence constraints to propagate which locations in the program should have a raw pointer using a Steensgaard-style alias analysis [41], and use the subtyping constraints to propagate which locations should be owned using an Andersen-style analysis [4].

We implement the initial optimistic rewrite (Section 4.2.2), and the iterative rewrite (Section 4.2.3) as a single algorithm, known as ResolveLifetimes (Figure 4.4). The subprocedure ComputeTaintAnalysis computes the taint analysis from Section 3.3.1 and a subset-based variant of it. The rest of the functions in the algorithm are described in Sections 4.1, 4.2.2 and 4.2.3. $\perp$ is used as the initial configuration, wherein all locations are mapped to borrowed pointers, and there are no lifetime constraints. Then, based on the current configuration and analysis results, we rewrite the program using the information on which pointers are borrowed, owned, or raw, and similarly add any inferred lifetime constraints to function signatures. This rewrite process is described further in Sections 4.2.2 and 4.2.3, and we give the precise rewrite rules in Section 4.2.3.2. We then run the compiler on the rewritten program. If there are any compiler errors, we compute a set of fixes (represented as a configuration) based on found borrow conflicts and unproven constraints. We then update our configuration, re-run the analyses if any pointer kinds have changed, and iteratively repeat the process until no compiler errors result. As an optimization, if no pointer was promoted then we re-use the old analysis results because we do not need to propagate any new rawness or ownership information.

4.2.1 Computing Fixes from Compiler Errors

We get three kinds of errors from the compiler: (1) lifetime constraints that could not be derived, (2) object references which outlive the object they reference (use after move), and (3) concurrent access involving mutable borrows (borrow conflicts). The compiler infers types and lifetimes locally in the type checking stage, and if it can successfully infer the lifetimes (there are no errors of the first kind) then it runs the borrow checker which reports errors of the latter two kinds. The specific errors we get,

and the fixes we apply, are detailed in this section. We apply the fixes by adding them to the new configuration we compute (called $fixes$ in Figure 4.4). The error numbers refer to the ones in Rust Compiler Error Index [13].

The following list details the case where the compiler cannot infer or prove a lifetime constraint. We resolve these errors by adding the constraint in question to the new configuration.

- Lifetimes inside two types mismatch (E0308). The compiler tries to type check $\tau_1 <: \tau_2$ but it cannot prove the constraint because some lifetimes in τ_1 and τ_2 need to have an outlives relationship. The specific missing outlives relationships are reported as lifetime constraints of the form `'a: 'b`.

- Compiler cannot infer an appropriate lifetime because of unsatisfied constraints (E0495). This is similar to E0308, where the lifetime `'a` of an object does not match the expected lifetime `'b`. So, we resolve it by adding `'a: 'b`.

- Lifetime mismatch (E0623). Similar to E0308, but the compiler reports it when comparing lifetimes during borrow checking instead of comparing two types during type checking.

- Given value needs to live as long as `'static` (E0759). We add `'a: 'static` for each lifetime `'a` that appears in the type.

The following errors indicate that a reference outlives the object it borrows. In these cases, we mark the reference as owned.

- A reference to a local variable is returned (E0515). Here we make the return value of the associated function an owned pointer.

- A reference is used after the referred variable is dropped (E0716), i.e. use-after-free. We make the reference an owned object so that the referred value is moved

and lives long enough.

The following errors indicate borrow conflicts. To address them, we promote the relevant reference to be a raw pointer.

- Two mutable references to an object are alive at the same time (E0499).
- A mutable and an immutable reference to an object are alive at the same time (E0502).

In the cases E0716, E0499, E0502 above, we find the HIR Id of the relevant expression e, and add $e \mapsto$ raw or $e \mapsto$ owned to $fixes$, depending on the error.

4.2.2 Initial Optimistic Rewrite

The next stage is to rewrite the program into a version with no `unsafe` annotations due to `Lifetime` raw pointers (`unsafe` annotations due to other categories of unsafety will remain). Henceforth we will just refer to "raw pointers"; this term should be taken as `Lifetime` raw pointers. The rewriting process is optimistic in the sense that it will likely result in a non-compilable program. The first step of this stage is to rewrite raw pointer declarations (e.g., data structure fields and function parameters) into reference declarations. Specifically, we convert the raw pointers into optional references in order to account for null pointer values: `Option<&T>`, `Option<&mut T>` and `Option<Box<T>>` represent immutably borrowed, mutably borrowed, and owner pointers, respectively. We assume for this stage that all declarations are borrowed; the third, iterative stage may later convert them into owners instead.

When declaring a reference in function signatures or data type definitions, we must provide its lifetime information. This information includes the lifetime of the reference itself and also the information for any referenced types that are themselves

parameterized by lifetime. Our goal for this stage is to generate lifetime information that minimally constrains the declarations, in order to start with the most optimistic lifetime assumptions.

For each raw pointer data structure field we provide a lifetime based on its type, using a different lifetime variable for each type.[1] We also fill in lifetime type parameters, using the same lifetime variables for all instances of the same type. Mutably borrowed references are not copyable or cloneable, so we remove the `#[derive(Copy,Clone)]` annotation from any affected data structures. For our example program, the end result of rewriting the `node_t` data structure is:

```
1 pub struct Node<'a1, 'a2> {
2     pub left: Option<&'a1 mut Node<'a1, 'a2>>,
3     pub right: Option<&'a1 mut Node<'a1, 'a2>>,
4     pub value: Option<&'a2 mut c_int>,
5 }
```

One case this example does not show is propagating the parameters of field types to the struct itself (e.g., if we had a `struct Tree { inner: Node }`, then `Tree` would also need two lifetime parameters in order to represent the rewritten version of `Node`. However, this creates a problem for recursive types: if we continuously derived new lifetimes for the instances of `Node` inside `Node`, we would get an infinite list of lifetimes, we detect self-referential loops and generate only one lifetime for each self-referential loop. We discuss rewriting struct definitions in a way that handles recursive data types in Section 4.2.2.1.

Once the data structures are rewritten, we rewrite the function signatures in accordance with the new declarations, again making all raw pointers into borrows. Unlike data structure fields, for function signatures we use a unique lifetime for each param-

[1] We could also give each field a unique lifetime, but this type-based heuristic works well empirically and makes it easy to handle recursive type declarations.

eter. For our example, the rewritten function signatures for `find` and `insert` are:

```
1 fn find<'a1, 'a2, 'a3, 'a4, 'a5, 'a6>(value: c_int, mut node: Option<&'
      a1 mut Node<'a2, 'a3>>) ->
2     Option<&'a4 mut Node<'a5, 'a6>>;
3 fn insert<'a1, 'a2, 'a3>(value: c_int, mut node: Option<&'a1 mut Node<'
      a2, 'a3>>);
```

The signature of `main_0` does not change, since it does not involve any pointers. Next we rewrite function bodies, which entails four types of rewrites:

1. We rewrite any call to `malloc` that allocates a single object (as opposed to an array) into a call to `Box::new`, a standard Rust function for safe heap allocation. We determine which `malloc` calls to rewrite by checking for calls that are translated from `malloc(sizeof(T))` in the C program.

2. We delete any call to `free` if we can replace all pointers that are freed at that call site with safe references. If we cannot replace all such pointers, then we need to keep the call to `free` so we roll back any pointers reaching this `free` that were previously rewritten.

3. We rewrite any equality comparisons between references, which by default are value equality checks in Rust (i.e., checking equality of the objects being referenced), into a reference equality check (i.e., checking whether two references refer to the same object). This rewrite preserves the intended semantics of the original program.

4. Dereferences must be rewritten to unwrap the optional part of the reference (recall that we replaced the raw pointer with an *optional* reference). Unwrapping the option consumes the original `Option` object because `Option<T>`, unlike raw pointers, is not automatically copyable. Therefore, we do the following to

```
1 pub fn borrow<'b, 'a: 'b, T>(p: &'b Option<&'a mut T>) -> Option<&'b T>
      {
2     p.as_ref().map(|x| &**x)
3 }
4 pub fn borrow_mut<'b, 'a: 'b, T>(p: &'b mut Option<&'a mut T>)
5   -> Option<&'b mut T> {
6     p.as_mut().map(|x| &mut **x)
7 }
```

Figure 4.6: Helper functions which assist in rewriting pointers to references. They allow borrowing an optional reference for a *shorter* lifetime, where `'a` is the original object's lifetime and `'b` is the borrowed object's lifetime.

avoid consuming the original object in the contexts that it is not assigned to or deliberately consumed:

- When using an immutable reference, we clone it so the original object is not destroyed.[2]
- When using a mutable reference, we make a mutable or immutable borrow depending on the context it is used in. We describe how we create these borrows below.

To help with re-borrowing mutable references, we use the helper functions `borrow` and `borrow_mut` defined in Figure 4.6. For each pointer `p` in the original program that we converted to a mutable reference, we perform the following rewrites:

- If `p` is passed to a mutable context (a context requiring a `&mut T`), we rewrite `p` to `borrow_mut(p)`.
- if `p` is passed to an immutable context (a context requiring a `&T`), we rewrite `p` to `borrow(p)`.

[2]We could immutably borrow the reference. However, cloning an immutable reference is a trivial operation (as `Option<&T>` implements `Copy`), and the resulting reference has the same lifetime. Cloning avoids needing more helpers like `borrow`.

- if `p` is dereferenced, we rewrite `*p` as `**p.as_mut().unwrap()` to get a mutable reference and immediately dereference it. If it is dereferenced in an immutable context, we use `as_ref` instead of `as_mut`. Note that `unwrap`, `as_mut`, and `as_ref` all come from the Rust standard library.

We rewrite null pointers into `None`, i.e., the `Option` value that does not contain anything. We similarly rewrite the null pointer check `p.is_null()` into `p.is_none()`. Figure 4.7 shows our example program after all of these transformations.

4.2.2.1 Rewriting struct definitions: field lifetimes and default values

When generating an initial set of lifetimes for a data structure definition, our goal is to make the data structure be as unconstrained as possible. As such, we initially start with all references being mutably borrowed instead of owned. We may convert them on a per-field basis in when fixing the errors from Rust's borrow checker (Section 4.2.3). Each field is generally given a distinct lifetime. However, we observe that fields of the same type in the same struct generally come from the same source, and therefore we heuristically give these the same lifetime.

Another place where we need to introduce lifetime variables in a struct body is the lifetime parameters of other structs in the definition. For example, consider these two structs:

```
1  struct Foo {
2    value: * const i32, // 32-bit signed integer
3  };
4
5  struct Bar {
6    foo: * const Foo,
7  };
```

```
// bst.rs
use std::os::raw::c_int;

pub struct Node<'a1, 'a2> {
    pub left: Option<&'a1 mut Node<'a1, 'a2>>,
    pub right: Option<&'a1 mut Node<'a1, 'a2>>,
    pub value: Option<&'a2 mut c_int>,
}
impl<'a1, 'a2> std::default::Default for Node<'a1, 'a2> {  /* ... */ }
pub fn insert<'a1, 'a2, 'a3>(mut value: c_int,
                             mut n: Option<&'a1 mut Node<'a2, 'a3>>) {
    /* ... */ }

pub fn find<'a1, 'a2, 'a3, 'a4, 'a5, 'a6>(mut value: c_int, mut node:
    Option<&'a1 mut Node<'a2, 'a3>>)
 -> Option<&'a4 mut Node<'a5, 'a6>> {
    if value < **(**node.as_ref().unwrap()).value.as_ref().unwrap() &&
    !(**node.as_ref().unwrap()).left.is_none() {
        return find(value, borrow_mut(&mut (*node.unwrap()).left))
    } else {
        if value > **(**node.as_ref().unwrap()).value.as_ref().unwrap()
     && !(**node.as_ref().unwrap()).right.is_none() {
            return find(value, borrow_mut(&mut (*node.unwrap()).right))

        } else { if value == **(**node.as_mut().unwrap()).value.as_mut
    ().unwrap() { return node } }
    }
    return None;
}

// main.rs
use std::os::raw::c_int;
use bst::{Node, insert, find};

pub fn main_0() -> int {
    let mut tree = Some(Box::new(Node::default()).as_mut());
    **(**tree.as_mut().unwrap()).value.as_mut().unwrap() = 3;
    insert(1, borrow_mut(&mut tree));
    insert(2, borrow_mut(&mut tree));
    **(**find(3, borrow_mut(&mut tree)).as_mut().unwrap()).value.as_mut
    ().unwrap() = 4;
    return 0;
}
```

Figure 4.7: The Rust program from Figure 4.3 after converting raw pointers into references.

When rewriting the struct `Foo`, we introduce a lifetime parameter for it, we also need an extra lifetime parameter in `Bar` to pass to `Foo`:

```
1  struct Foo<'a> {
2    value: & 'a i32,
3  };
4
5  struct Bar<'a, 'b> {
6    foo: & 'b Foo<'a>,
7  };
```

A naive approach to handle this case is to just add extra lifetime parameters for each struct in the definition that takes lifetime parameters. However, this would not work for recursive data types such as `Node` from our running example (Figure 4.2). We propose a general algorithm to generate lifetimes for arbitrary mutually recursive definitions with the following constraints on the generated lifetimes:

- All fields of the same type (or more generally, all fields of the types involved in the same mutually recursive type definition) in a definition share the same lifetime variable.
- All instances of the same struct in a definition share the same lifetime parameters. For example, in the definition of `node_t`, both instances of `node_t` for the left and the right subtree use the same lifetime parameters `'a1` and `'a2`.
- All references in a nested pointer are assigned the same lifetime. For example, `* const * const T` is converted to `Option<&'a Option<&'a T>>`.

Our algorithm is shown in Figure 4.8. The first step we take is building a points-to graph of all structs in the program and to label the edges with fields. A struct `Foo` points to another struct `Bar` if it has a (possibly nested) pointer field that points to a

`Foo` object. In our algorithm, once we compute the strongly-connected components (SCCs), Each SCC corresponds to a set of mutually recursive struct definitions. After building the SCC points-to graph, we aggregate the labels from the original graph, and also add the source nodes for each label. For each struct definition, we collect the lifetime names on all edges reachable from that struct's SCC to determine its lifetime parameters. When rewriting a field f of a struct S to a borrowing reference, we use the assigned lifetime name from the edge that contains the label $S.f$.

Build a points-to graph of all structs in the program.
Label all edges in this graph with their fields.
Compute the SCCs of this graph.
Build the points-to graph between SCCs.

for all edge $SCC_1 \to SCC_2$ in the SCC graph **do**
Let $S_1, S_2, \ldots \in SCC_1$ be the structs that point to SCC_2.
Let $f_1, f_2, \ldots$ be the fields of $S_1, \S_2, \ldots$ that point to a struct in SCC_2.
Label the edge with $(S_1.f_1, S_2.f_2, \ldots)$.
Assign a unique lifetime variable to each edge in the SCC graph that correspond to a borrowing reference.

Figure 4.8: Our algorithm for computing lifetime parameters of structs as part of ResolveLifetimes.

When allocating a struct on the heap (i.e., when we rewrite a `malloc` call), Rust requires initializing it with a default value to prevent reading from uninitialized memory. All fields (rather than the data they point to) in the structs we rewrite fall into one of these categories: raw pointers, optional references, primitive types, or other structs. We implement the `Default` trait for all of our structs, and the `Default` trait is implemented for `Option` and the primitive types so we use `Default::default` to generate the default values for the fields of these types. We initialize all raw pointers inside a struct (that is all the fields that are not converted to safe references) with null pointers.

We do not rewrite a struct's definition in the following cases:

- The struct is contained in (directly or through pointers) a type that is part of an external API. In this case, we cannot have lifetime guarantees because the external API may hold references to the value, keep a copy of it, or be responsible for its allocation/deallocation.
- The struct contains (not points to, but immediately contains) a (C-style) union. In this case, we cannot generate a default value for the struct, because there are not any well-defined default values for unions. One may opt into picking one of the variants and generating the default value for that field. In general, we keep unions out of the scope of our method, and this is an orthogonal issue.

In both cases, we mark the pointer-typed fields in the struct as raw pointers. We cover how this marking works in Section 4.2.

We do not rewrite unions, as they are out of the scope of our method; getting a value from a union is unsafe behavior, and it may allow forging invalid references. The Rust programs translated from C do not use Rust enums (sum types), and the C-style enums are just integers which do not contain any references, so the only flavor of algebraic data type we need to handle is structs. Our technique in this section can be extended to enums by considering the points-to edges from all possible variants.

4.2.3 Iteratively Rewriting the Program until It Compiles

The initial, optimistic rewrite may have resulted in a non-compilable program, i.e., one for which the Rust compiler cannot prove safety. The last stage of our technique iteratively attempts to compile the program with the Rust compiler; for each failed attempt we take information from the compiler errors to selectively rewrite our optimistic

changes until we reach a version that compiles. These rewrites in some cases provide the compiler with more refined lifetime information or modify reference types, while in other cases we are forced to walk back on the changes and leave some raw pointers as unsafe. When a version of the program fails to compile, we track the following information:

- Any additional lifetime constraints the compiler reports. For example, when compiling the program in Figure 4.7 the compiler reports that for `find` there is an additional constraint `'a1 : 'a2`, meaning `'a1` needs to outlive `'a2` because of the return statement on line 20. For the next iteration we rewrite the program to explicitly include this constraint and any additional constraints learned from similar errors.

- The references involved when a reference outlives an object. If the original object is on the heap, we promote the reference to an owned object on the heap and move the object instead of borrowing it, i.e., converting from `Option<&T>` to `Option<Box<T>>`. If the original object was on the stack, then we demote these references to raw pointers.

- If any rewritten `malloc` and `free` calls were involved in the failure. Rewritten calls can fail to compile when the original C program uses magic numbers or a custom allocation pattern. In subsequent iterations we do not attempt to rewrite any values that come from these particular calls to `malloc`, or that flow into these particular calls to `free`.

- The references involved in either use-after-move errors or multiple mutable borrow errors. We rewrite these references back to raw pointers.

When we demote a reference back to a raw pointer, we need to make all other

references that interact with that demoted reference into raw pointers as well. We use the taint analysis from Section 3.3.1 to propagate the required information about any references we decide to convert back to raw pointers because of borrow errors. Similarly, if we decide to make a reference owned, all the values that flow into it must also be owned. We propagate these facts by performing a subset-based version of the taint analysis we used in Section 3.3.1 and marking the references promoted to owned references as sinks.

We demonstrate these steps on the example program in Figure 4.7. For this example we do not encounter issues involving the last two cases above.

The first compilation attempt fails with a compiler error stating that the following lifetime constraints are not satisfied: `'a1 : 'a4`, `'a5 : 'a2`, and `'a6 : 'a3`. All of these constraints come from the `return node;` statement on line 20, and they are all rooted in the fact that the reference `find` returns cannot outlive its argument. Specifically, `'a1 : 'a4` comes directly from the references, and the other two constraints come from the fact that the data structures are covariant on their lifetime arguments and the functions are contravariant on their lifetime arguments. To resolve the errors we add these constraints to the signature of `find` and continue iterating.

The second compilation attempt also fails, this time with a compiler error stating that recursive calls to `find` require the additional constraints `'a2 : 'a5` and `'a3 : 'a6`. We add these constraints as well, and continue iterating.

The third compilation attempt fails again, with a compiler error stating that we return a value that cannot outlive borrowing `node` in lines 16 and 19. To resolve the error we rewrite the borrows in these dereferences `**node.as_mut().unwrap()` as `*node.unwrap()`, ultimately consuming the reference `node`. This heuristic works for many of the cases in our corpus programs, but it might create use-after-move errors later on, in which case we would walk the rewrites back and make the `node` parameter

of `find` a raw pointer again. In addition we get another lifetime error indicating that the variable `tree` in `main` function outlives the object it references (line 30), the temporary boxed object. To fix this error we we convert `tree` to be an owned object (`Option<Box<Node>>`).[3] Now that `tree` is an owned reference, we rewrite the places it is borrowed as `tree.as_mut().map(|b| b.as_mut())` to get a mutable reference inside the `Option` without consuming `tree`. We need to propagate the fact that `tree` is now an owned reference to all the values that flow into `tree`. After using our taint analysis to propagate this fact, we discover that the box at line 30 should be an owning reference, so we make that expression own the allocated object by removing the call to `as_mut()` on that line.

After these rewrites, the program compiles and all raw pointers have been converted into safe references. Note that we omitted the implementation of `insert` in this example to keep the number of steps shorter. Figure 4.9 shows the final fixed program.

Rather than relying on only a taint analysis and compiler errors, we could augment our method also by region inference, however the final program still needs to be verifiable by the compiler. To guarantee this, we use the compiler as an oracle to direct the choices our algorithm makes. From this perspective, we use the taint analysis as an optimization: technically we could do away with the taint analysis and let type errors guide us in regards to which other types to rewrite, e.g., when a raw pointer flows into a borrowed pointer. This would result in many more calls to the compiler (an intractable number in practice). To reduce the number of calls and to simplify the part of the method that processes compiler errors, we use the taint analysis. Also, we are interested in investigating how much we can do using only the compiler and simple analyses rather than more sophisticated and complicated custom-implemented

[3] We could potentially make it a `Box<Node>` without the `Option` part because it is never assigned to a value containing `None`, however we apply the same strategy independent of the position (including struct fields) and we need the optional types when creating default values for struct fields.

```
// bst.rs
use std::os::raw::c_int;
// BST node
pub struct Node<'a1, 'a2> {
    pub left: Option<&'a1 mut Node<'a1, 'a2>>,
    pub right: Option<&'a1 mut Node<'a1, 'a2>>,
    pub value: Option<&'a2 mut c_int>,
}
impl<'a1, 'a2> std::default::Default for Node<'a1, 'a2> {
  // ...
}
pub fn insert<'a1, 'a2, 'a3>(mut value: c_int,
                             mut n: Option<&'a1 mut Node<'a2, 'a3>>) {
    /* ... */ }
pub fn find<'a1, 'a2, 'a3, 'a4, 'a5, 'a6>(mut value: c_int, mut node:
    Option<&'a1 mut Node<'a2, 'a3>>)
 -> Option<&'a4 mut Node<'a5, 'a6>>
where 'a1: 'a4, 'a5: 'a2, 'a6: 'a3, 'a3: 'a6, 'a2: 'a5
{
    if value < **(**node.as_ref().unwrap()).value.as_ref().unwrap() &&
    !(**node.as_ref().unwrap()).left.is_none() {
        return find(value, borrow_mut(&mut (*node.unwrap()).left))
    } else {
        if value > **(**node.as_ref().unwrap()).value.as_ref().unwrap()
     && !(**node.as_ref().unwrap()).right.is_none() {
            return find(value, borrow_mut(&mut (*node.unwrap()).right))
        } else { if value == **(**node.as_mut().unwrap()).value.as_mut
    ().unwrap() { return node } }
    }
    return None;
}
// main.rs
use std::os::raw::c_int;
use bst::{Node, insert, find};
pub fn main_0() -> int {
    let mut tree = Some(Box::new(crate::Node::default()));
    **(**tree.as_mut().unwrap()).value.as_mut().unwrap() = 3;
    insert(1, tree.as_mut().map(|b| b.as_mut()));
    insert(2, tree.as_mut().map(|b| b.as_mut()));
    **(**find(3, tree.as_mut().map(|b| b.as_mut())).as_mut().unwrap()).
    value.as_mut().unwrap() = 4;
    return 0;
}
```

Figure 4.9: The safe Rust program with no raw pointers after applying all steps of our method.

analyses. Basing our initial method on simple analyses lets us gauge if and when more complicated analyses would be necessary.

4.2.3.1 Propagating the Information in the Derived Configuration

Section 4.2.3 shows how we propagate the information in the derived configuration on an example. There are two steps in this process: (1) propagating the information in the derived configuration, and (2) using this information to apply the rewrite rules in Section 4.2.3.2.

In the first step, we propagate pointer kinds using the two variants of the analysis discussed in Section 3.3.1. Specifically, we need to maintain the distinction between reference and pointer types in the output program to produce a well-typed program. So, we use two flow-insensitive, field-based data flow analyses to propagate the information in the configurations:

- We use an equality-based analysis to propagate raw pointers (pointers whose locations are marked as Raw in the configuration). As we need to guarantee type equality in every use site, we cannot mix reference and pointer types without unsafe casts or creating cases with unsafe aliasing between a pointer and an active reference[4] (which is not allowed under most memory models for Rust), so we need this analysis.

- We use a subset-based analysis to propagate ownership information (pointers whose locations are marked as Owned in the configuration). We need to do this for ownership because all values flowing into an owned value need to be owned. Also, a subset-based analysis works here because we can borrow an owned

[4]By an *active reference*, we mean a reference that can be used and is not already borrowed by another reference.

reference to create a borrowing reference, and get it checked by the borrow checker.

4.2.3.2 The Rewrite Rules used by Lifetime Derivation

In this section, we present the detailed rewrite rules used by the lifetime derivation algorithm described in Chapter 4.

For the sake of presentation, we consider a core language based on Rust HIR shown in Figure 4.10, and we rewrite other constructs like unary/binary operations and method calls to function calls. We similarly rewrite fused assignment operators (e.g., +=) to equivalent unfused code. The notation $\overrightarrow{a}$ denotes a sequence of as. For example, a function call contains a sequence of expressions representing arguments. e_{guard} denotes the guard expression in pattern matches. Similar to Oxide, we maintain a context denoting whether an expression is used in a place where it is borrowed mutably or immutably, or owned; these contexts are defined in Figure 4.11. The context assignee denotes that the expression is on the left-hand side of an assignment-like expression or being borrowed mutably. move denotes that the expression's value should be moved, as it is used in a context that should own its value. The notation $c[d \mapsto e]$ is used for conditionally updating the context: if $c = d$ then $c[d \mapsto e]$ produces e, otherwise it produces c. AdjustedType is a function provided by the Rust compiler that gives the type of the expression in the context it is used (after coercions, and converting &mut to & if necessary).

RewriteProgram adds lifetimes to each struct according to Section 4.2.2, and assigns unique lifetimes to each lifetime variable needed in a function signature. It adds the lifetime constraints to each function signature directly from the configuration. The function bodies are rewritten using the rewrite rules given in Figures 4.12 to 4.14. The helper κ returns the kind (owned, borrowed, raw) of an expression as computed by

the taint analyses from the current configuration. Other helper functions are defined in Figure 4.15. The rewrite rule $c \vdash e_1 \to e_2$ denotes that e_1 is rewritten into e_2 under mutability context c. Similarly, $c \vdash s_1 \to s_2$ denotes that the statement s_1 is rewritten into the statement s_2 under the context c as described in Figure 4.14. After rewriting an expression according to the rules given in Figures 4.12 and 4.13, we check the context of the expression and the pointer kind to decide whether to re-borrow the expression according to Section 4.2.2 . Let c be the current context, $p = \kappa(e)$, and $c \vdash e \to e'$. Then, if e has a pointer type, we choose whether to re-borrow e' according to the following conditionally-applied rules (attempted from first to last):

- $p =$ owned and $c \neq$ move. We borrow the box by rewriting e' to e'`.as_ref().map(|x| x.as_ref())` (we use `as_mut` instead of `as_ref` if the current context is mut).
- $p =$ owned and $c =$ move. We do not re-borrow the expression's value, as it should be moved.
- $p =$ raw. We do not perform any re-borrowing, as p is a raw pointer.
- $p =$ borrowed and $c =$ mut. We rewrite e' into `borrow_mut(&mut` e'`)`.
- $p =$ borrowed, $c =$ not, and e has a mutable pointer type. We rewrite e' into `borrow(&` e'`)`.
- $p =$ borrowed, $c =$ not, and e has an immutable pointer type. We rewrite e' into e'`.clone()`.
- Otherwise, we do not re-borrow the value of the expression.

$$
\begin{array}{rll}
e \in \text{Expression} ::= & [\overrightarrow{e}] & \text{array literals} \\
e \in \text{Expression} ::= & T\{\overrightarrow{\mathit{fld} : e}\} \mid & \text{struct construction} \\
& f(\overrightarrow{e}) \mid & \text{function call} \\
& (\overrightarrow{e}) \mid & \text{tuples} \\
& \&\mu\, e \mid & \text{address-of operator} \\
& *\, e \mid & \text{dereference} \\
& l \in \text{Literal} \mid & \\
& e\ \textbf{as}\ \tau \mid & \\
& x \in Variable \mid & \\
& \textbf{loop}\, e \mid & \\
& \textbf{match}\, e_{scrutinee}\, \overrightarrow{pe_{guard} \Rightarrow e} \mid & \text{pattern matching} \\
& e_1 = e_2 \mid & \text{assignment} \\
& e.\mathit{fld} \mid & \text{field access} \\
& e_1[e_2] \mid & \text{array access} \\
& \overrightarrow{s}\, e \mid & \text{blocks} \\
& \textbf{break} \mid \textbf{continue} \mid \textbf{return}\ e & \text{control flow redirection} \\
s \in \text{Statement} ::= & \textbf{let}\ \mu x = e_1; e_2 \mid e; & \\
\mu \in \text{Mutability} ::= & \textbf{mut} \mid \textbf{not} & \\
\tau \in \text{Type} ::= & *\,\mu\tau \mid & \text{raw pointers} \\
& \&\mu\tau \mid & \text{borrowing references} \\
& \text{Option} < \text{Box} < \tau >> \mid & \text{owned references} \\
& T \in \text{StructName} \mid & \text{structs} \\
& \ldots & \text{other types} \\
p \in \text{Pattern} & &
\end{array}
$$

Figure 4.10: Abstract syntax for the fragment of Rust HIR that is relevant to our rewrite rules for expressions. Because we rely on the compiler for lifetime inference, the lifetimes inside types are elided.

4.2.4 Algorithmic Complexity

We analyze the complexity of our algorithm in terms of the number of iterations it performs, as well as the number of times the taint analysis for propagating inferred

$$c \in \text{UseCtx} ::= \mathsf{mut} \mid \mathsf{not} \mid \mathsf{move} \mid \mathsf{assignee}$$

Figure 4.11: The contexts for determining how a variable is used.

pointer kinds is invoked. The initial step of resolving external types and functions (ResolveImports) has only one iteration, and uses only a call graph analysis so we do not count it in the analysis here. The algorithm described in Figure 4.4 climbs the configuration lattice in each iteration, and it reinvokes the analysis when the pointer kinds in the configuration change. In the worst case, each location would be promoted in a separate iteration. The Steensgaard-style taint analysis propagates rawness to all locations in the same equivalence class according to type equality, so there can be at most c iterations that promote a pointer to be raw, where c is the number of equivalence classes. However, each borrowed location may be promoted separately to an owned pointer in the worst case, so there can be at most l iterations that promote a reference to be owned. So, in the worst case, we climb the lattice in $O(c+l)$ iterations that invoke the analysis.

Between two iterations that promote a pointer, we may infer lifetime constraints. In the worst case, we would infer each lifetime constraint separately. Let $r = |f_1|+\ldots+|f_n|$ be the total number of lifetime variables that appear in function signatures, and $|f|$ denote the number of lifetime variables that occur in the signature of a function f, where $f_1, \ldots, f_n$ are the functions in the program. Each lifetime may be bounded by other lifetimes defined in the same function[5] or `'static`. As such, there are $|f_1|^2+|f_2|^2+\ldots+|f_n|^2 \leq (|f_1|+|f_2|+\ldots+|f_n|)\max_i |f_i| = r \max_i |f_i|$ lifetime constraints we may add. Here, $\max_i |f_i|$ is the largest number of lifetime variables that occurs in a function signature. Because, we use $\sqcup$ to merge the configurations, and because

[5]We do not process lifetimes in nested functions

the lattice is lexicographically ordered, we discard all lifetime constraints when we promote a pointer; in total ResolveLifetimes may have $O\left((c+l)r\max_i|f_i|\right)$ iterations in the worst case. Note that l is the number locations that we may initially assign to a lifetime, so it is the number of locations that are raw-pointer-typed because of Lifetime. Empirically, the number of iterations is much lower in our benchmarks, except for the case of libxml2 which contains large structs with many distinct lifetime parameters which in turn makes the $\max_i|f_i|$ term large.

4.2.4.1 Potential Optimizations

There are several optimizations to our algorithm that can reduce the number of iterations. In our prototype, we generate fresh names for lifetimes, and we use $\sqcup$ to merge configurations so we discard the lifetime constraints when promoting a pointer kind, even if the pointer kind is irrelevant. A potential improvement is choosing a naming scheme for lifetime parameters that is stable under promoting pointers, and keeping the inferred lifetime constraints if the references involving them are not promoted (this can be checked by querying the taint analysis for the relevant function parameter, points-to set, or access location). This would allow inferring each lifetime constraint only once, hence climing the lattice faster. In turn, this would reduce the number of iterations to $O\left(c+l+r\max_i|f_i|\right)$. We are planning to investigate the effects of this optimization empirically in the future.

A potential optimization is reducing $\max_i|f_i|$ and r by using fewer lifetime variables. One option that may miss data structures with complex lifetime constraints is using only one lifetime variable for all fields of a struct, hence having at most one lifetime parameter in a struct (rather than our current heuristic of generating one lifetime parameter per type). This would reduce the number of lifetime variables in a function signature to be bounded by the sum of the number of references and the number of

structs in the function signature. We investigate the impact of this optimization in terms of efficacy and performance in Section 4.3.3.

4.2.4.2 Termination Guarantee

At each stage, either there are no compiler errors (the algorithm terminates), or the compiler reports one of the errors listed in Section 4.2, meaning the next iteration will use a larger configuration. There are finitely many configurations, so termination is guaranteed: it will either yield a safer Rust program, or the original Rust program (wherein all references are marked raw).

4.3 Evaluation

We implement our tool on top of the `nightly-2020-10-15` nightly Rust compiler build version because the compiler API for Rust is not stable. We ran C2Rust using an even older version of the compiler (the newest version that the C2Rust supports due to the compiler API instability) `nightly-2019-12-05`. We run our experiments on a computer with a 4 GHz Intel Core i7-4790 CPU, with 4 physical cores (8 hyper-threaded). The computer has 32 GB RAM and runs Ubuntu 18.04.

4.3.1 Evaluation Setup

We evaluate the two stages of our method (ResolveImports and ResolveLifetimes) separately. A quick summary of these two stages is as follows:

- **ResolveImports:** This is the first step of our technique, described in Section 4.1, which resolves externally declared types and functions and removes unnecessary `unsafe` and mutability markers. Note that this step can make functions marked

unsafe into safe functions even though it does not convert any raw pointers into safe references; this effect comes from removing unsafe annotations that C2Rust adds naively when it did not need to.

- **ResolveLifetimes:** This is the remainder of our technique, described in Section 4.2, which converts `Lifetime` raw pointers (as described in Section 3.3.1) into safe references. As we did in Section 4.2, we will use the term "raw pointers" throughout to mean specifically `Lifetime` raw pointers.

We collect the following statistics, similar to Section 3.3.1, to measure the impact of our method: the number of unsafe functions that use raw pointers; the number of raw pointer declarations; and the number of raw pointer dereferences.

4.3.2 Results

Table 4.1 shows the change in the number of unsafe functions in the scope of our method, i.e., those that are unsafe due solely to the use of `Lifetime` raw pointers as described in Section 3.3.1. Our method makes 76% of these functions safe over all of the corpus programs.

We see that ResolveImports reduces the number of unsafe functions using raw pointers by 54% even though it does not remove any raw pointers. Some of these functions did not have any underlying cause of unsafety because they use raw pointers as values (e.g., assigning them to certain fields of a struct in an initializer), which is not unsafe behavior. These cases were categorized as false positives by our definition, but making them safe requires resolving imports. ResolveLifetimes makes 63% of the remaining functions safe. The functions that are not made safe by either method were involved in the following behavior (directly or indirectly):

- Calling `free` on raw pointers that our method could not rewrite.

Table 4.1: Number of unsafe functions due uniquely to using raw pointers. ResolveImports and ResolveLifetimes are the two phases of our method explained in Section 4.3.1; the corresponding columns show how many formerly unsafe functions were made safe by each phase (remembering that ResolveLifetimes is executed on the result of ResolveImports).

Program	Orig.	ResolveImports	ResolveLifetimes	Remaining	Made safe (%)
qsort	1	0	1	0	100%
grabc	1	0	0	1	0%
libcsv	12	0	11	1	92%
RFK	2	1	0	1	50%
urlparser	2	0	2	0	100%
genann	3	2	0	1	67%
xzoom	0	0	0	0	–
lil	6	2	1	3	50%
snudown	6	1	1	4	33%
json-c	20	9	8	3	85%
bzip2	8	4	4	0	100%
libzahl	2	0	2	0	100%
TI	45	44	0	1	98%
optipng	62	29	27	6	90%
tinycc	35	28	3	4	89%
tmux	31	7	9	15	52%
libxml2	210	174	30	6	97%
Total	236	127	69	40	83%

Table 4.2: Number of raw pointer declarations and dereferences. Orig. = The number from the original program. Fixed = The number of raw pointer declarations or dereferences removed by our method.

Program	Raw Ptr. Declarations				Raw Ptr. Dereferences			
	Orig.	Remaining	Fixed	(%)	Orig.	Remaining	Fixed	(%)
qsort	2	0	2	100%	4	0	4	100%
grabc	7	2	5	71%	15	6	9	60%
libcsv	18	0	18	100%	148	0	148	100%
RFK	0	0	0	–	0	0	0	–
urlparser	5	0	5	100%	58	0	58	100%
genann	5	5	0	0%	0	0	0	–
xzoom	0	0	0	–	23	23	0	0%
lil	50	27	23	46%	636	22	614	97%
snudown	31	8	23	74%	129	36	93	72%
json-c	41	11	30	73%	93	24	69	74%
bzip2	37	0	37	100%	679	0	679	100%
libzahl	7	0	7	100%	31	0	31	100%
tinycc	191	4	187	98%	946	79	867	92%
optipng	207	9	198	96%	606	10	596	98%
tmux	622	210	412	66%	2486	633	1853	75%
TI	82	82	0	0%	0	0	0	–
libxml2	839	156	683	81%	5175	565	4610	89%
Total	2144	514	1630	76%	11029	1398	9631	87%

- Dereferencing raw pointers that our method could not rewrite.

The impact of ResolveLifetimes on making functions *completely safe* is limited because to mark a function as safe we must convert *all* dereferences of raw pointers contained in the function into dereferences of safe references. However, making half of the relevant functions safe is a good step in the right direction.

Table 4.2 shows the change in the declarations and dereferences of raw pointers. Overall, our method removes 76% and 89% of `Lifetime` raw pointer declarations and dereferences, respectively, over all the corpus programs. These declarations and dereferences make up 8.1% and 9.7% of the *total* number of raw pointer declarations and dereferences including all categories of unsafety, because of the multi-faceted

nature of how raw pointers are used. Three of our programs (RFK, genann, and TI) do not dereference any `Lifetime` raw pointers, so they do not get much improvement from our method. We investigated the declarations and dereferences that our method fails to remove. They fall under the following categories:

- The pointer is not used safely according to the borrow checker rules. This is the case for the pointers in `libxml2`, `optipng`, and `bzip2` that we fail to remove, and one declaration in `json-c` and `tmux`. An example of this in `bzip2` is where a pointer is borrowed mutably as a field of a struct, then used mutably while this borrow is alive.

- The pointer is used in the signature of a function that is used as a function pointer. This is the case for the pointers in `json-c` (on all but one declaration we failed to remove), `lil`, and `TI` that we fail to remove.

The other reason for failing to convert some raw pointers is a limitation of our method in that we do not rewrite function pointer types, so we cannot change the signature of the functions passed as function pointers. We also inspected the intermediate steps of our tool to look into the root causes related to the pointers that remain raw due to borrow checker violations. In the `bzip2` and `optipng` programs, violating the borrow checker for one pointer (in `bzip2`) and two pointers (in `optipng`) are the reason for all of the raw pointers that remain after our technique; in both programs, the pointer value with illegal borrowing flows into a struct field, so any use of that struct field also becomes a raw pointer.

4.3.2.1 Limitations of ResolveImports

The core assumption of our heuristics for ResolveImports is that the structs with the same name and the same fields represent the same data type, so their definitions

can be merged to allow importing functions from other modules in the same program. This assumption is violated in `tinycc` for four anonymous structs, because the C2Rust-generated names of those structs did not match across modules because of how C2Rust generates names for anonymous structs. Because of this problem we get an error from the Rust compiler after the ResolveImports phase, and fixing the issue involved importing the four structs from where they are defined, removing the duplicate definition, and changing the four lines of code that use them. The fix was a 38-line patch, and it took one of the authors 10 minutes to investigate and fix the issue. If the anonymous structs are renamed appropriately before ResolveImports then this limitation no longer exists. Doing such a renaming reliably requires reasoning about the source of the anonymous structs (so being done at the time of translation from C to Rust).

4.3.3 Performance

Running our method is a one-time effort when translating the C program to a Rust program. In all of our corpus programs except `libxml2` and `optipng` our method finishes under a minute. In all programs, ResolveLifetimes takes the majority of the time (harmonic mean: 71%). In all programs except `libxml2` ResolveLifetimes takes at most 3 iterations to resolve all borrow checker conflicts, and our method terminates under 2 minutes. On `libxml2`, our method takes 125 minutes to finish, and ResolveLifetimes takes 81 iterations. Although our method takes a long time to run on a code base with 400k LoC, it needs to be run only once in the software evolution lifecycle, when translating the code base from C to Rust. 63% of this time is due to the taint analyses we perform to propagate the information on which locations need to be owned references or raw pointers as described in Section 4.2.3; 78 of the 81 iterations

are due to discovering lifetime constraints. `libxml2` contains lifetime constraints to discover because it defines structs with as many as 31 pointers.

We implemented one of the optimizations suggested in Section 4.2.4: using only one lifetime parameter per struct, and tested it on `libxml2`. It reduced the total number of iterations to 25, and the total runtime to 49 minutes. Also, this optimization did not miss making any pointers the original method also makes safe. So, this is a viable optimization in practice, as it creates programs with fewer lifetime variables (hence making them easier to reason about), and it is faster. However, we did not restrict our original method to support potential complex lifetime relationships between fields of structs.

4.4 Conclusion

As our analysis in Chapter 3 has shown earlier, there is a relatively small set of well-defined categories for these causes; however, the majority of unsafety in a translated program is caused by more than one category. This means that fixing any one category will have only a small impact, and that fixing a majority of unsafety will require addressing multiple categories.

Here, we have described and evaluated a novel technique for automatically removing a particular category of unsafety: the `Lifetime` raw pointers. Our technique piggy-backs on the Rust compiler, and our evaluation shows that it removes 87% of `Lifetime` raw pointer declarations and 89% of raw pointer dereferences of this category. Overall, we show that pointers that are not involved in any other unsafe behavior are rather well-behaving in that most of them can be made safe.

The method presented here focuses on removing specific instances of unnecessary unsafe behavior to make parts of the program safe. In Chapter 5, we are going to

investigate the dual problem of understanding and containing the spread of unsafety from sources a method such as ResolveLifetimes cannot fix.

We expect future reseatch to address other causes of unsafety in a similar way to ours, and extend our analysis beyond single-threaded C programs to cover more ground in widely-used software that performs unchecked resource management.

$$\frac{}{c \vdash x \to x}\ \textsc{Var} \qquad \frac{}{c \vdash l \to l}\ \textsc{Lit} \qquad \frac{}{c \vdash \textbf{break} \to \textbf{break}}\ \textsc{Break}$$

$$\frac{}{c \vdash \textbf{continue} \to \textbf{continue}}\ \textsc{Cont} \qquad \frac{c \vdash e_i \to e_i'}{c \vdash [\overrightarrow{e_i}] \to [\overrightarrow{e_i'}]}\ \textsc{Array}$$

$$\frac{c \vdash e_i \to e_i'}{c \vdash T\{\overrightarrow{fld_i : e_i}\} \to T\{\overrightarrow{fld_i : e_i'}\}}\ \textsc{Struct} \qquad \frac{c \vdash e_i \to e_i'}{c \vdash (\overrightarrow{e_i}) \to (\overrightarrow{e_i'})}\ \textsc{Tuple}$$

$$\frac{\mathsf{assignee} \vdash e \to e'}{c \vdash \textbf{return}\ e \to \textbf{return}\ e'}\ \textsc{Return} \qquad \frac{\mathsf{not} \vdash e \to e'}{c \vdash \textbf{loop}\ e \to \textbf{loop}\ e'}\ \textsc{Loop}$$

$$\frac{\textsc{Ctx}(e_{scrutinee}) \vdash e_{scrutinee} \to e'_{scrutinee} \quad \mathsf{not} \vdash e_{guard} \to e'_{guard} \quad c \vdash e \to e'}{c \vdash \textbf{match}\ e_{scrutinee}\ \overrightarrow{pe_{guard} \Rightarrow e} \to \textbf{match}\ e'_{scrutinee}\ \overrightarrow{pe'_{guard} \Rightarrow e'}}\ \textsc{Match}$$

$$\frac{(f \neq \texttt{malloc} \vee \kappa(f(\overrightarrow{e_i})) = \mathsf{raw}) \quad \textbf{move} \vdash e_i \to e_i' \quad \kappa(\textbf{param}\ f\ i) = \mathsf{owned}}{c \vdash f(\overrightarrow{e_i}) \to f(\overrightarrow{e_i'})}\ \textsc{Call-Mv}$$

$$\frac{(f \neq \texttt{malloc} \vee \kappa(f(\overrightarrow{e_i})) = \mathsf{raw}) \quad \textsc{Ctx}(e_i) \vdash e_i \to e_i' \quad \kappa(\textbf{param}\ f\ i) \neq \mathsf{owned}}{c \vdash f(\overrightarrow{e_i}) \to f(\overrightarrow{e_i'})}\ \textsc{Call-Br}$$

$$\frac{\kappa(\texttt{malloc}(l)) \neq \mathsf{raw}}{c \vdash \texttt{malloc}(l)\ \textbf{as}\ *\mu T \to \texttt{Box::new}(T\texttt{::default()})}\ \textsc{Malloc}$$

$$\frac{c' = c[\mathsf{assignee} \mapsto \mathsf{mut}] \quad \kappa(e) = \mathsf{raw} \quad c' \vdash e \to e'}{c \vdash *e \to *e'}\ \textsc{Deref-Raw}$$

$$\frac{c' = c[\mathsf{assignee} \mapsto \mathsf{mut}] \quad \kappa(e) \neq \mathsf{raw} \quad c' \vdash e \to e'}{c \vdash *e \to *(e'\texttt{.unwrap()})}\ \textsc{Deref-Safe}$$

$$\frac{\kappa(e) = \mathsf{raw} \quad c \vdash e \to e' \quad \tau \mapsto_{\{e\}} \tau'}{c \vdash e\ \textbf{as}\ \tau \to e'\ \textbf{as}\ \tau'}\ \textsc{Cast}_1 \qquad \frac{\kappa(e) \neq \mathsf{raw} \quad c \vdash e \to e'}{c \vdash e\ \textbf{as}\ *\mu\tau \to e'}\ \textsc{Cast}_2$$

$$\frac{\kappa(e) = \mathsf{raw} \quad \mathsf{not} \vdash e \to e'}{c \vdash \&\textbf{not}\ e \to \&\textbf{not}\ e'}\ \&\textsc{-Raw} \qquad \frac{\kappa(e) = \mathsf{borrowed} \quad \textbf{not} \vdash e \to e'}{c \vdash \&e \to \mathrm{Some}(\&\textbf{not}\ e')}\ \&\textsc{-Safe}$$

$$\frac{\kappa(e) = \mathsf{raw} \quad \textbf{mut} \vdash e \to e'}{c \vdash \&\textbf{mut}\ e \to \textbf{mut}\mu e'}\ \&\textsc{Mut-Raw}$$

$$\frac{\kappa(e) = \mathsf{borrowed} \quad \textbf{assignee} \vdash e \to e'}{c \vdash \&\textbf{mut}e \to \mathrm{Some}(\&\textbf{mut}\ e')}\ \&\textsc{Mut-Safe}$$

Figure 4.12: Rules for rewriting expressions, part I.

$$\frac{c \vdash e \to e'}{c \vdash e.fld \to e'.fld}\ \textsc{Field} \qquad \frac{c \vdash e_1 \to e_1' \quad \mathsf{not} \vdash e_2 \to e_2'}{c \vdash e_1[e_2] \to e_1'[e_2']}\ \textsc{Index}$$

$$\frac{c \vdash e \to e' \quad \mathsf{not} \vdash s \to s'}{c \vdash \overrightarrow{s}\, e \to \overrightarrow{s'}\, e'}\ \textsc{Block}$$

$$\frac{\mathsf{assignee} \vdash e_1 \to e_1' \quad \kappa(x) = \mathsf{owned} \quad \mathbf{move} \vdash e_2 \to e_2'}{c \vdash e_1 = e_2 \to e_1' = e_2'}\ \textsc{Assign-Move}$$

$$\frac{\mathsf{assignee} \vdash e_1 \to e_1' \quad \kappa(x) \neq \mathsf{owned} \quad \textsc{AdjustedType}(e_1) \neq *mut\,\tau \quad \mathbf{not} \vdash e_2 \to e_2'}{c \vdash e_1 = e_2 \to e_1' = e_2'}\ \textsc{Assign-Not}$$

$$\frac{\mathsf{assignee} \vdash e_1 \to e_1' \quad \kappa(x) \neq \mathsf{owned} \quad \textsc{AdjustedType}(e_1) = *mut\,\tau \quad \mathbf{mut} \vdash e_2 \to e_2'}{c \vdash e_1 = e_2 \to e_1' = e_2'}\ \textsc{Assign-Mut}$$

Figure 4.13: Rules for rewriting expressions, part II.

$$\frac{c \vdash e_2 \to e_2' \quad \kappa(x) = \mathsf{owned} \quad \mathbf{move} \vdash e_1 \to e_1'}{c \vdash \mathbf{let}\ \mu x = e_1; e_2 \to \mathbf{let}\ x = e_1'; e_2'}\ \textsc{S-Let-Move}$$

$$\frac{c \vdash e_2 \to e_2' \quad \kappa(x) \neq \mathsf{owned} \quad \mathbf{mut} \vdash e_1 \to e_1'}{c \vdash \mathbf{let\ mut} x = e_1; e_2 \to \mathbf{let}\ x = e_1'; e_2'}\ \textsc{S-Let-Mut}$$

$$\frac{c \vdash e_2 \to e_2' \quad \kappa(x) \neq \mathsf{owned} \quad \mathbf{not} \vdash e_1 \to e_1'}{c \vdash \mathbf{let\ not} x = e_1; e_2 \to \mathbf{let}\ x = e_1'; e_2'}\ \textsc{S-Let-Not}$$

$$\frac{\mathsf{not} \vdash e \to e'}{c \vdash e; \to e';}\ \textsc{S-Semicolon}$$

Figure 4.14: Rules for rewriting statements.

$$\textsc{Ctx}(e) = \textsc{Mutability}(\textsc{AdjustedType}(e))$$

$$\textsc{Mutability}(\tau) = \begin{cases} \mathsf{mut} & \tau = *\mathbf{mut}\tau' \\ \mathsf{not} & \text{otherwise} \end{cases}$$

$$\frac{\tau \neq *\mu\tau''}{\tau \mapsto_{loc} \tau'} \text{ T-NonPtr}$$

$$\frac{\tau \mapsto_{\textsc{PtsTo}(loc)} \tau' \quad \kappa(loc) = raw}{*\mu\tau \mapsto_{loc} *\mu\tau'} \text{ T-RawPtr}$$

$$\frac{\tau \mapsto_{\textsc{PtsTo}(loc)} \tau' \quad \kappa(loc) = owned}{*\mu\tau \mapsto_{loc} \texttt{Option<Box<}\tau'\texttt{>>}} \text{ T-OwnedPtr}$$

$$\frac{\tau \mapsto_{\textsc{PtsTo}(loc)} \tau' \quad \kappa(loc) = borrowed}{*\mu\tau \mapsto_{loc} \texttt{Option<\&}\mu\tau'\texttt{>}} \text{ T-RawPtr}$$

Figure 4.15: Helper functions for rewriting expressions and nested types. $\tau \mapsto_{loc} \tau'$ rewrites a type that is associated with the set of locations loc. PtsTo returns the points-to set of given set of locations.

Chapter 5

Type Equality and Unsafety

The purpose of computing is insight, not numbers.

– Richard Hamming

Existing C to Rust translators (e.g., C2Rust [22]) depend entirely on Rust's `unsafe` annotation that disables the compiler's safety checks: all of the translated code is marked `unsafe`. To increase verified safety, at least some of the translated code should be deemed safe by the Rust compiler. *Fully* automated translation to *completely* safe Rust is difficult, if not impossible: Rust enforces safety using an ownership-based type system, and C programs are not usually written with ownership-based semantics. The 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 that the remaining code is marked unsafe, so that they can manually rewrite those unsafe parts.

In this chapter we build on prior work in C to Rust translation by focusing specifically on pointers: **our goal is to understand the current limitations in inferring ownership and lifetime information for automatically translating unsafe *raw pointers* into verifiably safe references.** We offer new insights into the (lack of) effectiveness

of existing techniques, explain that lack of effectiveness with empirical evidence, and based on that evidence offer suggestions for how these limitations could be overcome.

In Section 5.1 we introduce a new technique called *pseudo-safety* that makes this entire study possible. The obstacle for previous studies is that pointers may be marked as unsafe for multiple reasons that have nothing to do with ownership and lifetime, such as pointer arithmetic, unsafe casts, etc. These confounding factors limit previous studies to only the small percentage of pointers that are not influenced by those factors. Pseudo-safety transforms Rust programs (translated from C) containing these confounding unsafe behaviors into Rust programs that do not, 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 from these other factors.

In Section 5.2, we conduct a study on the effectiveness of ownership and lifetime inference on pseudo-safe Rust programs translated from C, based on our previous work on described in Chapter 4 [19]. Our results show that, contrary to previous conclusions, the vast majority of raw pointers *cannot* be automatically made into safe references. This insight was possible because, unlike that previous work, pseudo-safety allows us to extend our study to include pointers involved in other unsafe behaviors. This result shows that examining only a subset of program pointers (as done in Chapter 4, which looks at pointers that have no unsafety issues besides lack of lifetime information) provides misleading results, and that our pseudo-safety transformation is necessary to get a complete picture.

In Section 5.3 we look deeper into the reasons behind our results, and show that *type equality*, and more specifically the imprecision of the Rust typechecking algorithm, is responsible: many pointers are put into type equivalence classes, which means that

if any one pointer cannot be made safe then no other pointer in its equivalence class can be made safe either. For example, in our largest benchmark (tmux) having only 29 pointers marked as unsafe is sufficient to taint the safety of over 4,600 pointers. We also see that having only 4 pointers marked as unsafe is sufficient to taint the safety of more than half of the pointers in each of our benchmarks.

In Section 5.4 we investigate whether (and what kinds of) more precise pointer analyses feeding into the Rust typechecker can mitigate the type equality problem. We study the effects of equality-based (the baseline), subset-based, field-sensitive, and context-sensitive pointer analyses on type equality. We show that field sensitivity does not substantially improve over the baseline, however both subset-based analysis and context-sensitive analysis each individually improve over the baseline by an order of magnitude.

In Section 5.5 we discuss possible strategies for incorporating our findings into an improved C to Rust translation. Rather than modifying the Rust compiler to be more precise, which has obvious shortcomings, our proposed methods are based on program transformations and thus compiler-agnostic. Future work involves implementing and evaluating these methods to determine their effectiveness.

5.1 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. We had an initial attempt to infer ownership and lifetimes with Laertes (Chapter 4). However, Laertes is inherently restricted to those raw pointers that are not marked unsafe for any reasons other than the lack of ownership and lifetime information, which turns out to be only a small percentage of

the total number of raw pointers (an average of 11%). It is important to note that those 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*. 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.

5.1.1 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: (1) existing aliasing relationships, from Rust's borrow checker's perspective (e.g., whether two pointers *may* alias); (2) the lifetime of each object as specified in the original program; and (3) the provenance of each pointer, i.e., where it originates from. 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.

5.1.2 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 5.1.1, 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, this block still performs the computation of both `arr` and `i` so their original static lifetimes are preserved. Similarly, the result of this expression still depends on `arr`, so aliasing and origin point information is preserved.

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 do not encode any type equality requirement between `a` and `b`, though it could be encoded with a slightly different rewrite like `{let mut fresh = a; fresh = b; 0}`). 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 (including the ones we use here) maintain their precision.

5.1.3 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 {}
}
```

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.

5.1.4 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.

As an example of our transformation, consider the snippet `foo(x as * mut c_void)`, which casts `x` as a void pointer. Assuming `x` is of type `*mut i32`, we rewrite this as `foo(pseudocast<&mut i32, &mut c_void>(x))` by extending Laertes. Here, our definition of `cast` enforces that the casted `x` does not outlive the original `x`, and the provenance of the casted expression `x` is preserved. Here, our definition of `pseudocast` makes the compiler enforce that the result of the cast does not outlive `x` (the compiler automatically computes the necessary lifetimes for `'a` and `'b`), so it allows us to present the lifetime information as well as existing borrows to the compiler. So, the technique for using the compiler as an oracle works in the presence of casts, and the changes to Laertes to support this are minimal.

Overall, we change Laertes to rewrite casts as calls to `pseudocast` to take ownership and taint information into account, such that there are no casts that explicitly switch between borrowing, raw, or owned pointers. For example, consider the cast expression `x as * const i32` in the program, where `x` has the type `* mut i8`. This cast will be rewritten as follows:

- If `x` is borrowed (that is, if `x` is going to be rewritten as as value of type `&mut i8`), then the expression will be rewritten as `pseudocast::<&mut i8, &const i32>(x)`.
- If `x` is owned (that is, if `x` is going to be rewritten as as value of type `Box<i8>`), then

the expression will be rewritten as `pseudocast::<Box<i8>, Box<i32>>(x)`.

- if `x` is raw (it is marked as unsafe) then the expression will be rewritten as `pseudocast::<*mut i8, *const i32>(x)`. We support this last case to rewrite casts from/to nested pointers correctly according to the current configuration of Laertes (Section 4.2).

Here, our changes are part of Laertes, so we determine the pointer kind of `x` and do a single rewrite to add the call to `pseudocast` with the correct types. Also note that `x` cannot be a vector or slice, as all pointer arithmetic has been removed from the program before this step.

5.1.5 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` [14] or `OnceCell` [15], 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.

5.1.6 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.

For example, consider the following code, which allows for treating the same 16 bits as either a single integer or as two bytes:

```
union Bits { word: u16, bytes: [u8; 2] }
let a = Bits { word: 8 };
```

We rewrite the union to a struct, and add the default initializers to the missing fields to each initialization of a `Bits` object (`bytes` is default-initialized):

```
struct Bits { word: u16, bytes: [u8; 2] }
let a = Bits { word: 8, bytes: [u8; 2]::default() };
```

Other alternatives we considered are (1) extending Laertes' semantics to support unions, and (2) converting unions into Rust `enum`s (sum types) and adding support for `enum`s in Laertes. The first alternative would have preserved the program's semantics in terms of reinterpreting bits across fields, and the second alternative would detect writing to a field then reading from another field, which *might* cause undefined behavior [45]. We chose our approach as it does not require any additional modifications to Laertes, and we deliberately choose to ignore any provenance information that would be carried between union fields as Rust does not reason about any data flow relationship between them.

5.1.7 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.

5.1.8 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).

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 [36] 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 2 of the benchmarks we have previously used in Chapter 3 (`optipng` and `snudown`), and so we exclude them in our evaluation of pseudo-safety.

5.2 Evaluating Laertes in the Limit

In this section we evaluate Laertes on a set of Rust programs that have been translated from C via C2Rust and then 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; 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 we conducted earlier in Section 4.3 that could only handle a small percentage of raw pointers). Our research question is: **RQ1: How many pointers can Laertes make safe when all raw pointers are made eligible via pseudo-safety?**

5.2.1 Experiment Setup

We use 14 of the 17 programs we previously used for evaluation in [19] in our evaluation, as shown in Table 5.1. We omit programs `optipng` and `snudown` because of the limitation outlined in Section 5.1, and we omit `libxml2` because Laertes times out on `libxml2` when handling all pointers.

We apply the transformations described in Section 5.1 after the ResolveImports[1] step of Laertes. We use the result of this phase as our baseline. Then, we run the

[1]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. See Section 4.1.

main step of Laertes (called ResolveLifetimes in Chapter 4) that uses the compiler as an oracle to derive ownership and lifetime information.

5.2.2 Experiment Results

Table 5.1 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 used number of raw pointer dereferences as a metric, but the trends are similar to those for raw pointer declarations and we omit the dereference metric for space. We reported earlier in Section 4.3 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 of using only a subset of the benchmarks). Using pseudo-safety to make all raw pointers eligible increases the number of eligible raw pointer declarations by an average of $10.5\times$. We also see that while 93% of eligible raw pointer declarations can be made safe before pseudo-safety is applied, only 9% of eligible raw pointer declarations can be made safe afterwards.[2] That is, the vast majority of raw pointer declarations that were previously rendered ineligible by other unsafety factors cannot have their ownership and lifetimes inferred by Laertes after those other factors are removed. In the next section we investigate the reasons behind this result.

[2]The findings before our modifications are consistent with the results we report in [19] and Section 4.3. The overall percentage of pointers made safe is higher because we consider a subset of benchmarks, and we have applied bugfixes and improvements to the tool by fine-tuning how Laertes handles compiler errors since that paper.

Table 5.1: Benchmark programs ordered by Rust lines of code along with raw pointer declarations both 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 (the **Fixed** column).

Program	**Lines of code**		**Before pseudo-safety**			**After pseudo-safety**		
	C	**Rust**	**Eligible**	**Fixed**	**%**	**Eligible**	**Fixed**	**%**
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	9	11%
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	23	100%	438	34	8%
json-c	6,933	8,430	29	28	97%	297	42	14%
libzahl	5,743	10,896	7	7	100%	457	64	14%
bzip2	5,831	14,011	37	29	78%	227	84	37%
TI	4,643	19,910	166	160	96%	866	18	2%
tinycc	46,878	62,569	187	146	78%	1,352	177	13%
tmux	41,425	191,964	331	329	99%	4,635	470	10%
Total	118,990	322,188	810	752	93%	8,509	773	9%

5.2.2.1 How Pseudo-safety Affects Number of Lifetime Errors Found

As discussed in Section 4.2,when Laertes encounters a compiler error, it stores each pointer (specifically, each program location) that needs to be raw because of that error in a configuration. As a follow-up to our previous result, we are also interested in whether the increase in the number of pointers Laertes could not make safe correlates with the increase in the number of pointers it derived as necessarily unsafe (as a direct result of a compiler error). In other words, we want to see whether the cause of the increase in pointers that cannot be made unsafe is (1) because they are reported to be used unsafely directly by the compiler, or (2) they are derived to be unsafe from the initial configuration through the analysis in Section 4.2.3.1.

In order to answer this follow-up question, we compare the total number of such inherently unsafe pointers stored in the configuration by Laertes before and after our pseudo-safety transformations. Pseudo-safety enables finding 134 pointers that are inherently unsafe as opposed to 11 inherently unsafe pointers before applying it. The number of issues found increases to $12.2\times$ (similarly to the increase to $10.5\times$ in declared pointers we reported earlier in Section 5.2) however the total number of pointers that cannot be made safe has increased $130\times$ (derived by comparing the difference between the eligible and fixed columns in Table 5.1). So, the increase in the number of inherently unsafe pointers alone does not explain why many more pointers are still unsafe: 134 pointers are inherently unsafe but over 7,000 additional pointers are still unsafe because they interact with inherently unsafe pointers. In Section 5.3, we investigate this increase in the ratio of pointers that are unsafe because of other pointers.

5.3 Type Equality as a Vector for Unsafety

In Section 5.2 we have shown that the efficacy of Laertes drops a lot when considering pointers that used to have other causes of unsafety. In this section we investigate why Laertes is unable to transform a significant number of raw pointers into safe references. The Rust typechecker is equality-based (as discussed in Section 4.2.3.1), 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.

Figure 5.1 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 90% of the raw pointers in all benchmarks; this means that having only four necessarily unsafe raw pointers is enough to poison 90% of the total raw pointers in the worst case. Moreover, all benchmarks have at most 29 equivalence classes. From these measurements we see that the underlying issue is type

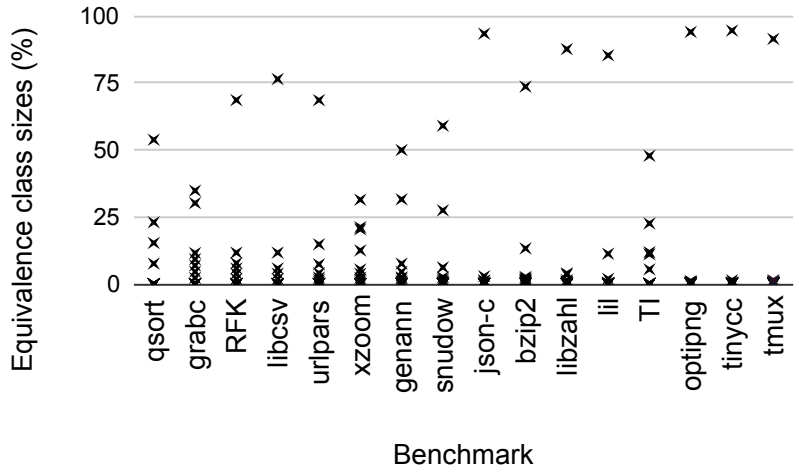


Figure 5.1: Size of each equivalence class of declared variables as a percentage of the sum of the sizes of equivalence classes for each benchmark. The benchmarks are ordered in terms of total number of variables.

equality, or more precisely, the imprecision of the Rust typechecker.

In the rest of this chapter, we investigate the effects of applying more precision for 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 p we count the number of other raw pointers whose safety depends on that of p (i.e., the number of raw pointers that will necessarily be marked unsafe if p is marked unsafe); call p the *instigating* raw pointer and call the raw pointers whose safety is dependent on p the *affected* raw pointers. We then plot a histogram where the x-axis indicated 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

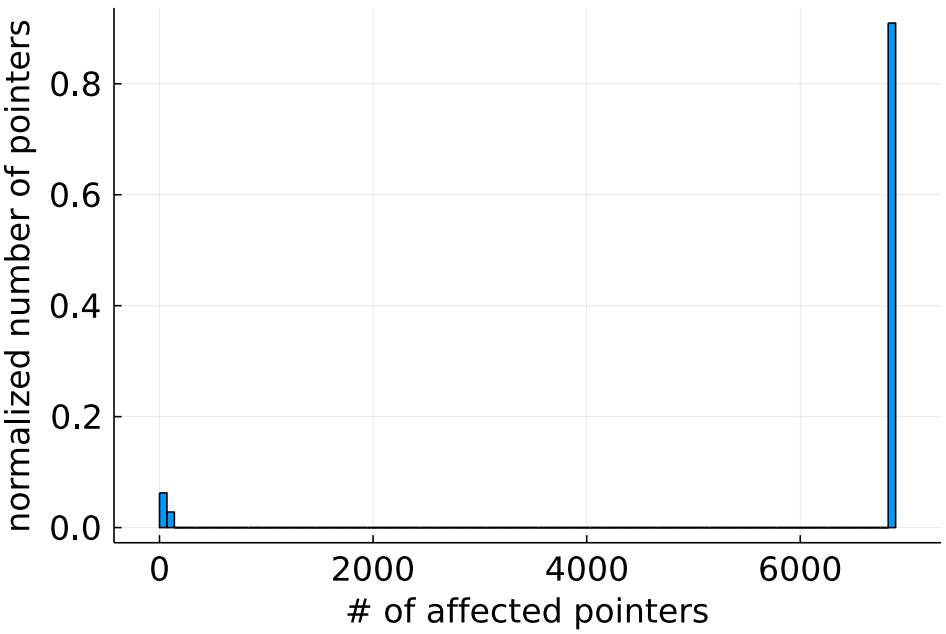


Figure 5.2: Histogram of pointers in `tmux` (our largest benchmark). The x axis denotes the number of pointers affected, and the y axis denotes the normalized ratio of pointers in the program in the bin.

raw pointer may be affected by some other raw pointer.

Figure 5.2 demonstrates this new metric on our largest benchmark, `tmux`. We see that, for example, over 90% of instigating raw pointers affect more than 6,000 other raw pointers. That is, if any of those 90% of raw pointers are marked unsafe, then necessarily at least 6,000 other raw pointers will also be marked unsafe.

5.4 Investigating Analysis Precision

In the previous section we showed that imprecision in the Rust typechecker is responsible for allowing necessarily unsafe raw pointers to spread their "unsafety taintedness" to many other raw pointers. In this section we perform a limits study to investigate the effects of adding different kinds of precision in order to determine what kinds of analysis, if any, can mitigate this problem.

The baseline Rust typechecking analysis is equivalent to an equality-based (i.e., Steensgaard-style [41]), field-based, 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 SVF [43] framework and its various pointer analyses. SVF analyzes LLVM bitcode, so we first compile the 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 Rust HIR code:

- LLVM bitcode is in static single assignment (SSA) form, which causes flow-insensitive analyses to effectively have strong updates for local variables a la flow-sensitive analysis. However, SSA form can be applied to Rust HIR if needed 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. We use the type information and the field index information computed by SVF to restore some of the missing type information in order to remedy this problem.
- HIR code is 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, context-insensitive analysis reflecting the baseline 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.

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 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 we used in Section 4.3. We do not use the `libxml2` benchmark because the context-sensitive analysis times out after 48 hours. All of our experiments are run on a Intel i7-6600k processor with 32

GiB of memory running Void Linux. We used GNU Parallel [44] to run the analyses in parallel. 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.

Results To answer how analysis precision impacts the spread of unsafety, we use the metric described in Section 5.2 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 average size of the affected raw pointers per potential instigator, the better the analysis curbs the spread of unsafety. We pose two research questions:

1. **RQ2: How many raw pointers are "well-contained"?** Specifically, we are interested in the number of raw pointers that, when considered as potential instigators, do not affect many other pointers in the program. For this study we define well-contained pointers as those that affect at most 1% of the program's raw pointers.

2. **RQ3: How does the overall distribution of affected pointers change with analysis precision?** As RQ2's 1% threshold is arbitrary, we also investigate the overall distribution of affected pointers. To answer RQ3 we collect summary statistics (mean and standard deviation) for each benchmark and precision level. We also graphically display the distribution of raw pointers in terms of how many other raw pointers in the program they affect.

The context-sensitive level of precision raises a difficulty with the metric we're using: namely, how to count pointers that are duplicated due to being in different

contexts. In order to leverage the additional precision of context-sensitive analysis we consider potential intigating raw pointers to be *(pointer, context)* pairs, that is, if a pointer appears in n contexts then we consider it as a potential instigator pointer n different times. 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 additional 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.

RQ2: How many pointers are "well-contained"? Table 5.2 presents the total number and percentage of instigator raw pointers that affect less than 1% of the raw pointers in the program for each level of analysis precision. We observe that adding field sensitivity to an equality-based analysis does not significantly improve precision: the highest increase observed is 4% (in `optipng`). However, adding directionality 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 the outlier because it has only 12 pointers, and the analysis can prove the same fact for >80% of the pointers for the remaining three benchmarks. Adding context sensitivity 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. This shows that an analysis that incorporates all three aspects of precision is crucial for taming unsafety, and that such analyses can also help identify

Table 5.2: Number of total pointers, and percentage of pointers affecting $\leq 1\%$ of pointers. **ptrs** is the number of pointer-typed variables in the dataflow graph, **cptrs** is the total number of clones of all raw pointers in all contexts. The remaining columns refer to instigator pointers affecting $\leq 1\%$ pointers under each analysis precision. **P4:all** counts a pointer as affected only if it is affected under all call contexts. **P4:some** counts a pointer as affected only if it is affected under at least one call context. Results $> 90\%$ are marked in bold.

Benchmark	ptrs	cptrs	# of pointers affecting <1% of the pointers (%)				
			P1	P2	P3	P4:some	P4:all
qsort	16	109	0.00%	0.00%	37.50%	55.05%	55.96%
grabc	108	112	52.78%	52.78%	**91.67%**	**91.07%**	**98.21%**
libcsv	429	1,048	15.15%	17.25%	81.82%	**95.91%**	**96.96%**
urlparser	1,079	2,293	2.41%	2.59%	83.60%	**96.21%**	**96.42%**
xzoom	2,465	1,808	31.32%	34.32%	**92.33%**	**96.46%**	**96.57%**
RFK	3,095	2,438	26.20%	28.72%	**95.32%**	**97.70%**	**100.00%**
snudown	3,728	8,467	17.95%	19.21%	**98.85%**	**99.22%**	**99.99%**
genann	4,108	23,298	25.46%	27.14%	**93.65%**	**98.09%**	**98.31%**
libzahl	5,441	34,092	4.43%	4.52%	**94.16%**	**95.20%**	**95.40%**
json	5,598	19,671	10.29%	11.97%	**92.69%**	**96.12%**	**96.25%**
lil	6,144	21,465	14.71%	16.03%	**91.78%**	**96.03%**	**96.62%**
TI	8,376	20,859	13.98%	14.77%	**93.04%**	**99.85%**	**100.00%**
bzip2	11,909	36,905	17.03%	17.65%	**96.84%**	**98.52%**	**98.54%**
optipng	21,984	76,709	20.37%	24.46%	**96.77%**	**99.61%**	**99.63%**
tinycc	28,830	257,853	14.09%	17.97%	**93.54%**	**95.03%**	**98.35%**
tmux	75,869	246,398	6.90%	8.07%	89.00%	**93.14%**	**93.14%**

the remaining "lynchpin" pointers that spread unsafety even after drastic automatic program transformations. Such an analysis can similarly help in identifying the cases when some automatic transformations will not control unsafety, which can be useful for building interactive methods for making programs safe.

RQ3: How does the overall distribution of affected pointers change with analysis precision? Our evaluation for RQ2 requires defining an arbitrary threshold for "well-contained". 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 5.3. The average affected pointer set size increases as the total number

of pointers increases, as there could be several lynchpin pointers flowing into pointers from different parts of the program. We observe that both the mean and the standard deviation of this metric shrink as sensitivity increases. Once we add directionality the mean shrinks by an order of magnitude, though the standard deviation shrinks only by half on average (geo. mean). This means that although directionality reduces affected pointer set sizes for most of the program, there are still many pointers with large affected pointer sets. Finally we consider a best-case scenario, i.e., an analysis with full precision, and assuming that another pointer is not reachable unless it is reachable in all contexts. In such a scenario, we observe that a random pointer does not affect more than 179 other pointers, or merely 20 when excluding `tmux`. Similar to the analysis of Table 5.2, we observe that having only field-sensitivity added to an equality-based analysis improves the mean only by a small amount.

While summary statistics give a general understanding of the distribution they may be misleading, as many differently-shaped distributions can have the same summary statistics [6]. So, we also investigate the change in the shape of this distribution graphically. Figure 5.3 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, $\approx 90\%$ 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.

Table 5.3: Summary statistics for number of affected pointers for each instigator pointer (rounded to the nearest integer for space concerns), classified by benchmark and analysis precision. μ is the mean and σ is the standard deviation. **P4:all** is counting a pointer as affected only if it is affected under all call contexts; **P4:some** is counting a pointer as affected only if it is affected under at least one call context.

Program	**P1**		**P2**		**P3**		**P4:some**		**P4:all**	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
qsort	7	4	7	4	2	3	1	1	1	1
grabc	5	7	5	7	1	1	1	1	0	0
libcsv	64	50	55	44	3	5	1	2	1	2
urlparser	1028	161	1023	167	8	17	2	5	2	3
xzoom	996	749	900	720	10	38	2	7	2	6
RFK	1642	1001	613	440	8	35	2	14	0	2
snudown	2497	1270	2049	1192	4	37	2	11	1	2
genann	2286	1332	1871	1275	17	86	9	54	4	20
libzahl	5150	1108	4959	1078	126	652	77	343	6	20
json	3389	1669	3030	1646	96	425	32	158	8	34
lil	4355	1889	4062	1940	155	616	29	140	4	14
TI	5560	2629	5247	2677	37	190	2	19	1	3
bzip2	7989	3754	7661	3799	159	1006	76	629	9	71
optipng	16372	8268	12194	7120	67	603	12	142	5	43
tinycc	21998	8904	19402	9071	913	3553	173	997	20	123
tmux	68091	18541	64113	18997	5391	15936	2441	8986	179	652

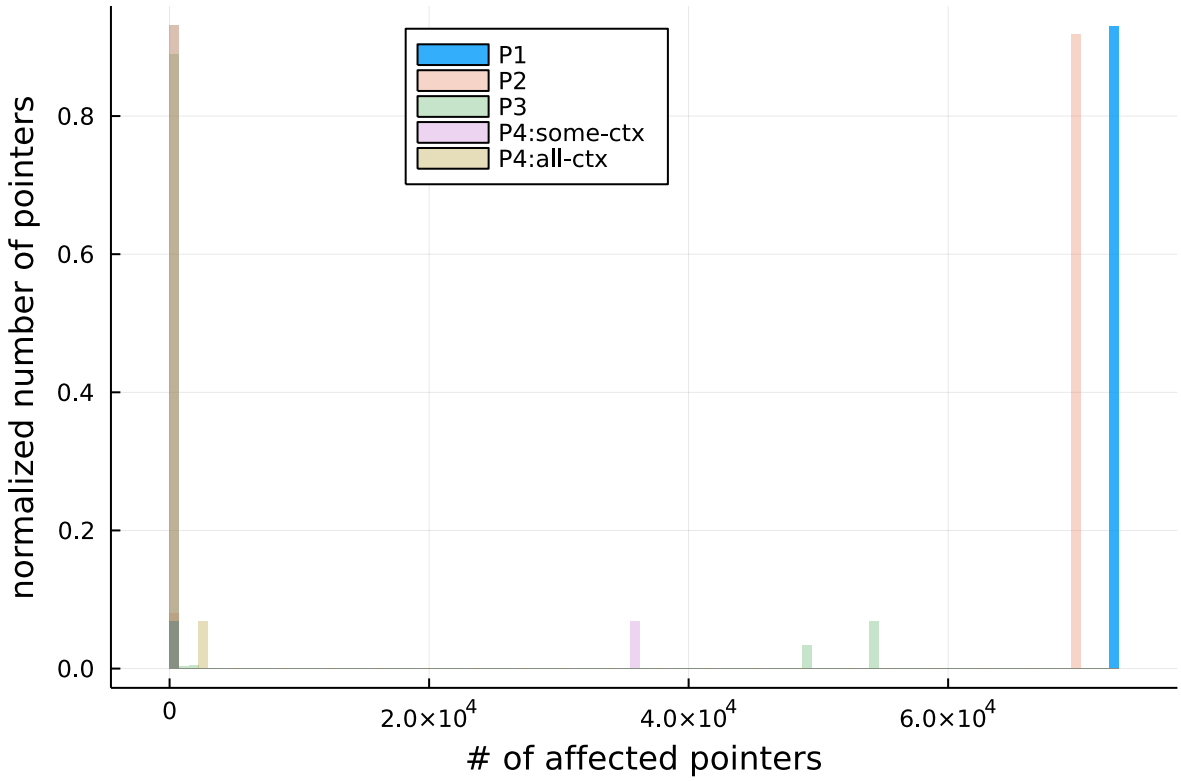


Figure 5.3: Histogram of pointers in tmux. The x axis denotes the number of pointers affected, and the y axis denotes the normalized ratio of pointers in the program in the bin. We report two separate figures for context-sensitive analysis: **some-ctx** counts a pointer as affected if it is affected in at least one context; **all-ctx** counts a pointer as affected only if it is affected in all contexts.

5.5 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 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 any compiler modification; these transformations mimic the effects of field-sensitive analysis, directional (subset-based) analysis, and context-sensitive analysis. We implement the transformations for directionality in Chapter 6. Implementing and evaluating the remaining transformations is left for future work.

An axis for pointer analysis precision that we do not focus on is control flow sensitivity. The result of a flow-sensitive analysis could be different for each program point because of strong updates, so there is no straightforward rewrite to achieve flow-sensitivity overall without changing the underlying type system. However, at least for stack objects, flow sensitivity can be gained by renaming variables to be in SSA form.

5.5.1 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 it merges the types of different instances of the same field into a single type. For example, the snippet on the left in Section 5.5.1 shows that although there is no actual data flow between `x` and `y`, any unsafety in `x` would also force `y` to become unsafe because of field-based analysis: `x` flows into `z1.bar` so if `x` is unsafe, it makes the `bar` field of `Foo` unsafe, which causes `z2.bar` to be unsafe hence it makes `y` unsafe.

```
1 struct Foo<'a>{bar: &'a i8;}
2
3 let z1 : Foo = ...;
4 let z2 : Foo = ...;
5 z1.bar = x;
6 y = z2.bar;
```

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

Figure 5.4: A code snippet before and after duplicating `Foo` to emulate field sensitivity.

A field-sensitive analysis could distinguish between `z1.bar` and `z2.bar`. To get the effect of a field-sensitive analysis while still using a field-based analysis, we can define separate struct types for each object or each combination of struct fields' safety. The code snippet on the right of Section 5.5.1 shows the result of such a transformation. The important change is that `z1` and `z2` now have different types, so the compiler can reason that `z1.bar` and `z2.bar` are unrelated.

5.5.2 Casting References to Pointers to Introduce Directionality

Rust's type checker performs an equality-based analysis, which means that information flows not just from the right-hand side of an assignment to the left-hand side, but also vice-versa—effectively, information is propagated *backwards* as well as forwards. This imprecision forces values with no shared data flow to have the same type. In Section 5.5.2, `x` and `y` are aliased because they both flow into `z`, though there is no data flow between `x` and `y`:

Suppose `x` must become a raw pointer due to some unsafe usage not shown. Due to the assignments between `x`, `y`, and `z`, variables `y` and `z` are transitively forced to become raw pointers, as they must all share the same type.

A directional (i.e., subset-based) analysis distinguishes between information flowing *into* a value and flowing *from* that value. We can mimic this feature of a directional analysis using an equality-based analysis by looking for places where a raw pointer

```
1 let x : & i8 = ...;
2 let y : & i8 = ...;
3 let z : & i8;
4 // ...
5 z = y;
6 // ...
7 z = x;
```

```
1 let x : * const i8 = ...;
2 let y : & i8 = ...;
3 let z : * const i8;
4 // ...
5 z = y.as_ptr();
6 // ...
7 z = x;
```

Figure 5.5: A code snippet before and after inserting a reference-to-pointer conversion to emulate directionality.

that otherwise could be made safe flows into a raw pointer that is marked unsafe and breaking the backwards flow (that forces the first raw pointer to also be made unsafe) by introducing a cast from a safe references to a raw pointer at that point, and turning the raw pointer flowing *into* an unsafe raw pointer into references while keeping the raw pointers flowing *from* an unsafe pointer intact.

In the example above, we can cast `y` to a raw pointer only in the assignment to `z`, keeping `y` as a safe reference as there is no unsafe pointer that flows into `y`. Thus, by allowing references to decay into pointers, we can make `z` unsafe but still leave `y` safe.

As shown, one can insert casts when appropriate to make pointer unsafety spread only in the direction of the data-flow, in order to communicate the results of a directional (Andersen-style) analysis to the Rust compiler. However, we cannot apply this arbitrarily in the program, as mixing references and pointers can cause undefined behavior in Rust. For example, the object may not live long enough for the pointers referring to it to access it, or the compiler may assume that an object pointed to by an alive immutable reference cannot be modified, even when there is a mutable pointer to that object. The undefined behavior is introduced here because the compiler can assume that the object pointed to by an alive reference cannot be changed by another reference or pointer (the borrow checker already checks this for references), and the programmer (or, in this case the translation tool) is responsible for using unsafe point-

ers while maintaining this invariant. We investigate methods to enforce these aliasing constraints for the pointers we handle by leveraging the Rust type system in Chapter 6. In that chapter, we also implement and evaluate the method described here.

5.5.3 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:

- We do not need to duplicate a function for each call site, but rather for each combination of pointer, box, and reference arguments.
- 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. This requires introducing heuristics on which functions to duplicate.
- Programs with function pointers require additional bookkeeping, which is exacerbated by additional complexity from the prior item. This creates another tension between maintainability and safety.

Potentially, one can inspect the call graph to find functions with many call sites, and inspect the context-sensitive points-to graph to discover if any of those functions take unsafe arguments at only some call sites. However, this approach would require computing a full context-sensitive analysis to discover candidate functions. Future

research is needed to investigate function duplication heuristics without such expense, perhaps via demand-driven context-sensitive analyses.

Another challenge left for future work is cloning function pointers for different function signatures. One possible representation is to convert function pointers to structs with one member for each possible function signature. Upon an update to a new function, all members of the struct would be updated appropriately. Upon calls, the relevant member of the struct would be accessed and called. Another possible approach is to perform defunctionalization, and to maintain separate maps for each function signature.

5.6 Conclusions

In this chapter we have conducted a series of limit studies on the effectiveness of ownership and lifetime inference for unsafe raw pointers in Rust programs translated from C. Our first limit study uses a new technique called *pseudo-safety* that extends the study to all raw pointers rather than the small subset used in previous studies, and contradicts previous studies by showing that the majority of raw pointers cannot be translated to safe references using existing techniques. We show empirically that type equality is the culprit, causing the unsafety of only a few raw pointers to taint the safety of many others. We then show that more precise pointer analysis can mitigate this problem by analyzing pointer analysis precision in three axes: field-sensitivity, context-sensitivity, and directionality (using a subset-based analysis rather than an equality-based analysis). We finally suggest several program transformations that could potentially mimic the effect of more precise pointer analysis without requiring any Rust compiler modifications. We implement and evaluate one of these suggestions (encoding the results of a subset-based analysis) in Chapter 6.The remaining sugges-

tions require resolving the tension between maintainability and safety in automatic transformations Implementing and evaluating them are left for future work.

Chapter 6

Directionality to Tame Unsafety

About the use of language: it is impossible to sharpen a pencil with a blunt axe. It is equally vain to try to do it with ten blunt axes instead.

– Edsger W. Dijkstra (EWD498)

In Section 5.5 we have shown that the imprecise analysis done by the type checker causes unsafety to spread in the program like wildfire, and showed that encoding the results of a more precise analysis is a viable way forward for reducing the impact of *an average pointer in the program*. Machiry et al. [31] also make a similar observation about the spread of unsafety by an equality-based analysis in the context of keeping track of nullability and array indexing in Checked C programs, and they propose a solution around containing unsafety at function call boundaries by adding run-time checks (which would potentially be validated by the programmer). In this chapter, we develop a set of methods to encode the results of a subset-based analysis throughout the program, explore how such a rewrite interacts with the aliasing rules in Rust, and evaluate the efficacy of these methods in terms of how well they contain unsafety, and what borrow errors they enable discovering to help the programmer make the rest of the program safe.

Similar to Chapter 5, we use the term *directionality* to describe the difference between a subset-based (directional) data flow analysis, and an equality-based (undirectional) one. The only program transformation we perform is inserting casts from references to pointers, as described in Section 6.1.

The pointer-to-reference casts regard only making the outermost pointer type safe, so they cannot be used for conversion between inner pointer types (e.g., they do not convert from `* mut & mut T` to `* mut * mut T`). Thus, we do not use directionality for inner pointers. Moreover, we cannot safely cast between types of functions that accept a reference vs. a parameter, so we use an undirectional (equality-based) analysis to reason about function pointers. Section 6.2 describes the specifics of the data flow analysis we use to compute where to insert casts to reduce the spread of unsafety.

As briefly mentioned in Section 5.5.2, inserting casts everywhere possible can cause undefined behavior because of aliasing an alive reference (a reference that can be used according to the borrow checker), and a pointer. We prevent this by invalidating references during a cast, and we use the Rust type checker to enforce this restriction. Section 6.3 explains this undefined behavior in detail as well as how we solve it.

We then evaluate this method (along with the version that can cause undefined behavior) using pseudo-safety in Section 6.4 and discuss the impact of the method's limitations empirically.

Finally, we conclude in Section 6.5 and discuss potential complementary methods for future work to overcome the limitations of the method we present in this chapter.

6.1 Representing Directional Flow using Casts

In this section, we are going to demonstrate the issue with undirectional analysis. Then, we show how we solve this by inserting casts as well as how we compute where

to insert casts based on the results of a directional data flow analysis.

The example we used in Section 5.5.2, along with the data flow graph computed by an undirectional analysis[1] is reproduced in Figure 6.1. The nodes marked 1 and 2 correspond to the expressions (not variables) `y` and `x` on lines marked with (1) and (2) respectively. Note that the graph is undirected, so unsafety computed from this graph can flow both in the direction of data flow (from `x` to `z`), and against it (from `z` to `y`). In this example, we consider `x` to be inherently unsafe (it is directly used unsafely), so all other pointers are marked `unsafe` as a result of the data flow analysis.

```
1 let x : * const i8 = ...;
2 let y : * const i8 = ...;
3 let z : * const i8;
4 // ...
5 z = y; // (1)
6 // ...
7 z = x; // (2)
```

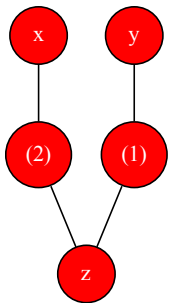


Figure 6.1: A short code snippet with the associated data flow graph computed . The red nodes are marked unsafe. We assume that `x` is used unsafely, so it is the instigator for all other unsafe pointers in this snippet.

Figure 6.2 shows the data flow graph (DFG) computed by a directional analysis (note that the graph is now directed). We observe that the location (node) corresponding to *expression* `y` on the line marked (1) is safe, but it immediately flows into an unsafe location (namely, `z`). So, we insert a cast around the expression (1) to encode the results of the directional analysis to obtain the code snippet in the same figure. As a result, we can now use `y` as a safe reference up until it is passed to `z`.

So, we i when rewriting the current expression e with associated location l, we check if thre is a data flow edge $l \rightarrow l'$ *in the original data flow graph* such that l is not

[1]We use flow-insensitive, context-insensitive, and field-based analyses throughout this chapter, as they mimic the type checker's behavior modulo directionality.

```
1 let x : * const i8 = ...;
2 let y : & i8 = ...;
3 let z : * const i8;
4 // ...
5 z = y.as_ptr(); // (1)
6 // ...
7 z = x; // (2)
```

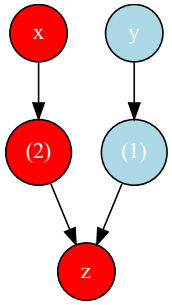


Figure 6.2: The code snippet after inserting a reference-to-pointer conversion to emulate directionality. The graph on the right is the DFG computed by the directional analysis. The nodes in red are marked unsafe, whereas the ones in blue are safe.

poisoned but l' is poisoned (where the poisons are computed by the analysis described in Section 6.2). If there is such an edge, then we insert a cast surrounding the current expression, because e represents a reference that is going to be used as a raw pointer immediately. We implement this logic in the part of Laertes that inserts borrows and re-borrows. Also, note that we use the DFG before transitive closure in our analysis (otherwise, we would insert the cast immediately to every source that may eventually flow to an unsafe location).

One issue that arises is that *"What happens if e immediately flows into more than one place?"*, as one of the uses of e might be safe while another use is not. We construct the DFG such that this does not happen. The next section outlines how we construct the DFG, and how we propagate information through the DFG.

6.2 A Type-safe Directional Data Flow Analysis

Figure 6.3 shows the program locations (i.e., nodes in the data flow graph) in our analysis besides the ones already used in Figure 4.5. This is mostly a fairly standard representation for a field-based, context- and control flow-insensitive data flow analysis. The locations represent actual program locations (or fresh logic variables

needed to fill in the inputs/outputs of public APIs), and constructors are constructors in the sense of set constraint-based program analysis [3].

The fresh locations f are used to create a structure that mimics type structure for function and nested pointer types are used as APIs for client programs to use. There are no a actual uses of these APIs and we use fresh locations to create dummy sources for the pointers in these APIs.

Pseudo-locations are not necessary for the analysis *per se*, but they allow us to structure the graph such that every node that is derived from an expression has a single successor in the data flow graph before transitive closure. For example, consider an assignment `*p = q`. In a classical subset-based data flow analysis, this assignment would generate a constraint of the form $ref(q, \bar{q}) \subseteq p$. There are no edges from q to any location until the constraint system is solved. And, by the time the constraint system is solved, we would get nodes that transitive successors of q. This is a situation want to avoid as it would make the analysis degrade to an equality-based one because we would insert a cast right after creating a source if it eventually flows into an unsafe node. We resolve this conundrum by introducing pseudo-locations that are not used for propagating unsafety, but used to resolve cases where a node may flow immediately into an indeterminate number of locations. In the example with the points-to set, we add the edge $q \rightarrow$ **pointsto** p to the data flow graph, so q has a single successor in the original data flow graph. When determining whether **pointsto** p is unsafe, we check if *any* of the nodes in the points-to set of p is unsafe. The other case where an expression may flow into multiple locations immediately is function calls (in cases where the callee is unknown) where an argument may flow into parameters of many functions, and the return value of a function may flow into many call sites. We, similarly, create pseudo-locations to denote the parameters and the return value of the functions that flow into a location.

Overall, the data flow analysis we perform is the subset-based data flow analysis with function pointers described by Pearce et al. [36] with the following modifications:

1. We unify all pointees of a pointer, effectively switching to an equality-based analysis for inner pointer types because our method inserts casts only at the top level (e.g., we do not insert a cast from the type `* mut & mut T` to `* mut * mut T`).

2. We create a dummy node to connect the two sides of a comparison operation to guarantee that type equality is maintained when two pointers are compared.

3. We reason about all function pointers using an equality-based analysis, again because we cannot support casts between function types soundly. We also create dummy nodes in the DFG for each variable, field, and parameter to forward taint information to cases where a function pointer is never explicitly initialized but used in calls where its parameters need to become raw. This case arises when rewriting higher-order functions library APIs that take a callback but the callback is never used in the program itself.

4. 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. The declared variable nodes were already in Laertes, so it is not an addition we made with pseudo-locations.

Once we perform the data flow analysis, we propagate the unsafety information along the data flow edges, 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 (including pseudo-locations).

$$
\begin{array}{lll}
l \in \text{Location} ::= \ldots \mid\mid f \in \text{Free} & & \text{Free location variables} \\
\\
p \in \text{PseudoLocation} ::= & \textbf{ptsto}\, l & \text{Points-to sets} \\
& \mid \textbf{param}\, l\, i & \text{Parameters} \\
& \mid \textbf{ret}\, l & \text{Return values}
\end{array}
$$

Figure 6.3: Pseudo-locations and additional locations used in our data flow analysis. Location also includes the locations described in Figure 4.5

6.3 Pointer–Reference Aliasing Woes

At the beginning of this chapter, we alluded to emergence of undefined behavior due to pointer-reference aliasing. Figure 6.4 shows a code snippet that exhibits this behavior. Without any optimizations, this code snippet would return `2`. A mutable reference in Rust is assumed to not alias with any other mutator, so the compiler can assume that `*y` can only be modified through `y` while `y` is not borrowed. So, a constant propagation pass may rewrite `return *y;` to `return 1;` because `y` is not borrowed between the assignment `*y = 1;` and the return statement. As a result, this program has undefined behavior. If the program used unsafe pointers everywhere, then this unsafe behavior would not occur (because there are no aliasing guarantees among pointers). So, this undefined behavior is introduced by adding casts, and it does not exist in the original unsafe Rust program.

In order to resolve this problem, we need to make sure that a reference is never used the pointer it is cast to is in use. One may approach encoding this using lifetime constraints, however there are two issues with such an approach: (1) pointers can be cloned, so we need to keep track of all clones of a pointer throughout the program, (2) the reference cast to the pointer may be derived from another reference, so we need to exclude the uses of all sources that flow to that reference. Lifetimes in Rust's type

```
1 let x : * mut i8 = ...;
2 let y : &mut i8 = ...;
3 let z : * mut i8;
4 z = x;
5 // ...
6 *y = 1;
7 z = y.as_mut_ptr();
8 *z = 2;
9 return *y;
```

Figure 6.4: A program with pointer–reference aliasing causing undefined behavior.

system allow us to encode the second property. We can try encoding the first property using phantom types (types, in our case lifetimes, that are used only for safety but not associated with actual data types) as well, similar to how `RefCell` works to lift aliasing checks to run-time. However, such an encoding does not compose well with array and field accesses, as we cannot overload these operations to generate raw pointers with lifetime tracking. So, we are left to do the lifetime analysis outside the Rust borrow checker, which loses the benefits of off-loading lifetime inference to the compiler.

We follow a different route to still use the Rust borrow checker: whenever a location needs to become unsafe (i.e., its pointer kind is promoted to raw), 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`). Now, we can consume the reference (i.e, the `Box`) when inserting a cast, so the cast invalidates the incoming reference, and the ownership requirement invalidates all previous references to the same object (as it already does in Laertes). This change comes with a cost efficacy, as the ownership requirement would invalidate some uses of pointers. We evaluate the impact of this change quantitatively in Section 6.4.

6.4 Evaluation

In this section we evaluate Laertes with our modifications to measure the impact of both the safe and the unsafe version of inserting casts. We are specifically interested in the following questions:

- How effective is adding only top-level casts in terms of making more declarations and dereferences safe?
- How much headroom is there between the safe transformation that introduces ownership, and the unsafe one that may introduce undefined behavior?

6.4.1 Experiment Setup

We use pseudo-safety to evaluate our method in this section, so we use the same experiment setup we used in Section 5.2, and use the same 14 benchmarks.

We run Laertes under the following configurations after the pseudo-safety transformations:

- **Equality-based:** This is the version of Laertes we used in Chapter 5, and it does not contain any of the transformations discussed in this chapter. It serves as a baseline for our experiments.
- **Subset-based (unsafe):** This is the version of Laertes we used in Chapter 5 with casts inserted everywhere possible without guaranteeing lack of aliasing between pointers and references.
- **Subset-based (safe):** This is the version of Laertes we used in Chapter 5 with casts inserted while guaranteeing lack of aliasing between pointers and alive references by consuming the references as described in Section 6.3.

6.4.2 Results

Tables 6.1 and 6.2 show the declarations and the dereferences in the program (respectively), as well as how many declarations/dereferences are made safe by each method. We see a similar overall picture between dereferences and declarations. So, our analysis focuses on *declarations*. Inserting top-level casts *safely* increases the effectiveness of lifetime inference by 75% (an increase from 12% to 21%), and a more elaborate handling of the aliasing issues discussed in Section 6.3 may allow a further 25% increase (an increase from 12% to 24%). However, our method of offloading aliasing constraints to the compiler using ownership is nevertheless an effective and simple solution. Also, the overall number of pointers made safe is still low, indicating that handling other causes of imprecision discussed in Section 5.5 is a worthwhile goal for future work.

Introducing casts does not improve the efficacy of Laertes much *relatively* in four benchmarks: grabc, xzoom, libcsv, and TI. The first two use an effectively global pointer unsafely (as part of interaction with X graphics library), which results in spread of unsafety to the rest of 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, which limits the efficacy of our method. We discuss how the spread of unsafety manifests in TI in the Limitations section in more detail.

6.4.3 Limitations

The method presented here handles function pointers in an equality-based manner, and it does not implement a way to handle other ways to increase analysis precision discussed in Section 5.5. In order to measure the potential effect of handling nested

Table 6.1: The pointer declarations, the Eligible column denotes the number of all pointers in the program (because we use pseudo-safety). w/o casts = pointers made safe by the baseline (equality-based) transformation. w/ casts (unsafe) = pointers made safe when introducing casts while allowing unsafe aliasing. w/ 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 Eligible column.

Benchmark	Eligible	w/o casts		w/ casts (unsafe)		w/ casts (safe)	
RFK	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	26	(70%)	27	(73%)	26	(70%)
genann	73	12	(16%)	18	(25%)	15	(21%)
urlparser	79	9	(11%)	61	(77%)	48	(61%)
bzip2	227	84	(37%)	139	(61%)	133	(59%)
json-c	325	70	(22%)	150	(46%)	115	(35%)
lil	438	35	(8%)	162	(37%)	156	(36%)
libzahl	457	64	(14%)	208	(46%)	83	(18%)
TI	866	18	(2%)	37	(4%)	35	(4%)
tinycc	1,352	207	(15%)	381	(28%)	387	(29%)
tmux	4,635	468	(10%)	836	(18%)	762	(16%)
TOTAL	8,537	1,008	(12%)	2,035	(24%)	1,775	(21%)

pointers in a directional manner, we conducted 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. 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 this chapter, and an analysis that still uses directionality for nested pointers. The value calculated by the first analysis is the number of pointers made safe, whereas the second number is *an upper bound* on the number of pointers that could be made safe with more elaborate casts.

We observe that more elaborate casts could make *at most* 114 more pointer declarations

Table 6.2: The pointer dereferences, the Eligible column denotes the number of all pointer dereferences in the program (because we use pseudo-safety). w/o casts = dereferences made safe by the baseline transformation. w/ casts (unsafe) = dereferences made safe when introducing casts while allowing unsafe aliasing. w/ 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 Eligible column.

Benchmark	Eligible	w/o casts		w/ casts (unsafe)		w/ casts (safe)	
qsort	10	4	(40%)	4	(40%)	4	(40%)
grabc	21	17	(81%)	17	(81%)	17	(81%)
RFK	24	24	(100%)	24	(100%)	24	(100%)
urlparser	60	58	(97%)	58	(97%)	58	(97%)
xzoom	132	81	(61%)	79	(60%)	79	(60%)
libcsv	174	51	(29%)	51	(29%)	51	(29%)
genann	339	5	(1%)	11	(3%)	6	(2%)
lil	1,668	634	(38%)	903	(54%)	880	(53%)
TI	1,778	85	(5%)	513	(29%)	510	(29%)
json-c	1,843	130	(7%)	702	(38%)	197	(11%)
libzahl	2,400	188	(8%)	516	(22%)	211	(9%)
bzip2	3,720	317	(9%)	730	(20%)	705	(19%)
tinycc	5,362	679	(13%)	1,388	(26%)	1,051	(20%)
tmux	21,608	1,869	(9%)	3,428	(16%)	3,116	(14%)
TOTAL	39,139	4,142	(11%)	8,424	(22%)	6,909	(18%)

safe. This is a small percentage (1.3%) of the total number of eligible pointers, so we do not expect this limitation to have a large effect in the efficacy of Laertes. This number is an upper bound because we may discover more root causes using a more precise analysis, so some of the 114 pointers marked as affected by our analysis could be instigators.

The limitations around function pointers affects the TI benchmark disproportionately: TI is a time series analysis library that implements hundreds of analysis functions, then the functions requested by the user are dispatched using a single global array of function pointers. We see that only 4% of the pointers were made safe in this benchmark even with directionality. We investigated how unsafety spreads for

this benchmark in detail by investigating the instigators and the affected pointers. We observe that:

- There is an unsafe pointer that flows into the parameters of a function pointer from this global array. This results in almost all function parameters in the program to be unsafe. Function duplication as suggested in Section 5.5.3 can help overcome this issue by maintaining safe and unsafe versions of these functions, but ultimately this unsafety issue needs to be resolved by the programmer.
- Moreover, some of these data analysis functions use their parameters unsafely, and the equality-based reasoning causes this unsafety to spread to other data analysis functions through the global array.

6.5 Conclusions

In this chapter, we presented a method to encode the results of a subset-based (directional) data flow analysis by introducing casts from references to pointers. We considered inserting only top-level casts from pointer types. So, the analysis we used is subset-based only for top-level pointers and it switches to an equality-based analysis for function pointers and inner pointer types in nested pointers. We also show that just inserting casts may cause undefined behavior by aliasing alive references and pointers. Our method overcomes this issue by consuming the original object (hence invalidating all safe references to it) on a cast. We then evaluate this method using the evaluation methodology in Section 5.2. Even with these limitations, we see an 75% increase (from 12% to 21% of pointers) in the effectiveness of Laertes when we introduce casts. We also evaluate casts without consuming the original references (which may introduce undefined behavior), and observe a further increase to 24% of the pointers, so there is

a small gain in using a more sophisticated solution than consuming the pointee object. So, encoding results of a subset-based analysis helps contain the spread of unsafe pointers and make lifetime inference handle a larger part of the program. We leave investigating more elaborate program transformations that would also allow using subset-based analysis for function types to future work. We also observe that other methods of encoding analysis precision (such as function duplication) are needed to further tame unsafety.

Chapter 7

Conclusion and Future Work

The end is never the end is never the end is never ...

– The loading screen in Stanley Parable

Translating C to safe Rust is a multifaceted open problem with sub-problems that interact with each other. In this dissertation, we have shown that the causes of unsafety in a Rust program translated from C are varied, and interleaved. We then focused on the "core" cause of unsafety that is due to lack of information that the Rust compiler needs to prove memory safety: lack of lifetime information in pointers. So, we built an iterative method to discover lifetime and ownership constraints from compiler errors, and showed that this method is effective on pointers that do not contain causes of unsafety besides lack of lifetime information. However, evaluation of such a method is incomplete because of interactions with other causes of unsafety (so, only 11% of the pointers are eligible for this method). In order to get more accurate results, we built an evaluation methodology that hides other causes of unsafety while maintaining the lifetime constraints we care about, and we evaluated the effectiveness of lifetime inference on all pointers in the program. This evaluation shows that lifetime inference does not scale well when considering all pointers, and we show that the underlying

cause here is the spread of "accidental" unsafety through the type system. We then conducted a limit study evaluating potential impact of making the type system more precise (by keeping track of data flows more precisely), and we propose methods to encode the results of a more precise analysis by program transformation. Finally, we implemented one of our proposals to show that such an encoding is feasible and it can double the effectiveness of lifetime inference.

Overall, we observe that lifetime inference has a potential to be effective to extract safe uses from most of the program, however it is hindered by the imprecision of the type system. Encoding the results of other more sensitive analysis using the methods we proposed can improve on our solution in the future.

There are two avenues for future work to explore to complement the work presented in this dissertation:

- As we reach the limits of automatic lifetime inference, future work needs to investigate (1) higher-level rewrites to transform unsafe uses of pointers to safe uses, and (2) interactive methods to point to the programmer the "pain points" where automatic translation falls short, along with potential impact of each point.
- Eventually, other causes of unsafety also need to be handled (semi-)automatically to help scale the effort of translating C to safe Rust.

Bibliography

[1] NVD - CVE-2021-21148, 2021. URL https://nvd.nist.gov/vuln/detail/CVE-2021-21148.

[2] NVD - CVE-2021-3156, 2021. URL https://nvd.nist.gov/vuln/detail/CVE-2021-3156.

[3] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2):79–111, November 1999. ISSN 0167-6423. doi: 10.1016/S0167-6423(99)00007-6. URL http://www.sciencedirect.com/science/article/pii/S0167642399000076.

[4] Lars Ole Andersen. Program Analysis and Specialization for the C Programming Language. Technical report, Dept. of Computer Science, University of Copenhagen, 1994.

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

[6] F. J. Anscombe. Graphs in Statistical Analysis. *The American Statistician*, 27(1):17–21, February 1973. ISSN 0003-1305. doi: 10.1080/00031305.1973.10478966. URL https://www.tandfonline.com/doi/abs/10.1080/00031305.1973.10478966.

[7] V. Astrauskas, C. Matheja, P. Müller, F. Poli, and A. J. Summers. How do programmers use unsafe rust? In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume OOPSLA, New York, NY, USA, 2020. ACM. doi: 10.1145/3428204.

[8] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe rust? *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi: 10.1145/3428204. URL https://doi.org/10.1145/3428204.

[9] Sergio Benitez. Short Paper: Rusty Types for Solid Safety. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS

'16, pages 69–75, New York, NY, USA, October 2016. Association for Computing Machinery. ISBN 978-1-4503-4574-3. doi: 10.1145/2993600.2993604. URL https://doi.org/10.1145/2993600.2993604.

[10] David Bryant. A Quantum Leap for the Web, October 2016. URL https://medium.com/mozilla-tech/a-quantum-leap-for-the-web-a3b7174b3c12.

[11] Citrus Developers. Citrus / Citrus, 2018. URL https://gitlab.com/citrus-rs/citrus.

[12] Mirai Contributors. facebookexperimental/MIRAI, July 2021. URL https://github.com/facebookexperimental/MIRAI. original-date: 2018-11-06T20:56:35Z.

[13] The Rust Project Developers. Rust compiler error index, 2021. URL https://doc.rust-lang.org/error-index.html.

[14] The Rust Project Developers. lazy-static.rs, May 2022. URL https://github.com/rust-lang-nursery/lazy-static.rs. original-date: 2014-06-24T08:25:15Z.

[15] The Rust Project Developers. standard lazy types - Rust RFC #2788, 2022. URL https://github.com/rust-lang/rfcs/pull/2788.

[16] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, November 2014. Association for Computing Machinery. ISBN 978-1-4503-3213-2. doi: 10.1145/2663716.2663755. URL https://doi.org/10.1145/2663716.2663755.

[17] Nelson Elhage. Supporting linux kernel development in rust, 2020. URL https://lwn.net/Articles/829858/.

[18] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 53–60, September 2018. doi: 10.1109/SecDev.2018.00015.

[19] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Translating c to safer rust. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi: 10.1145/3485498. URL https://doi.org/10.1145/3485498.

[20] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is rust used safely by software developers? In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 246–257, 2020.

[21] Tim Hutt. Would Rust secure cURL?, January 2021. URL https://timmmm.github.io/curl-vulnerabilities-rust/.

[22] Immunant inc. immunant/c2rust, February 2020. URL https://github.com/immunant/c2rust. original-date: 2018-04-20T00:05:50Z.

[23] Immunant inc. c2rust manual examples, 2020. URL https://c2rust.com/manual/examples/index.html.

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

[25] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158154. URL https://doi.org/10.1145/3158154.

[26] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, 2018. ISBN 978-1-59327-851-9. URL https://doc.rust-lang.org/book/.

[27] N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993. doi: 10.1109/MC.1993.274940.

[28] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: experiences building an embedded OS in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, PLOS '15, pages 21–26, New York, NY, USA, October 2015. Association for Computing Machinery. ISBN 978-1-4503-3942-1. doi: 10.1145/2818302.2818306. URL https://doi.org/10.1145/2818302.2818306.

[29] Yi Lin, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. Rust as a language for high performance GC implementation. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management - ISMM 2016*, pages 89–98, Santa Barbara, CA, USA, 2016. ACM Press. ISBN 978-1-4503-4317-6. doi: 10.1145/2926697.2926707. URL http://dl.acm.org/citation.cfm?doid=2926697.2926707.

[30] Linux Weekly News. Rust support hits linux-next, 2021. URL https://lwn.net/Articles/849849/.

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

[32] Nicholas D Matsakis. An alias-based formulation of the borrow checker, April 2018. URL https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/.

[33] Cade Metz. Dennis Ritchie: The Shoulders Steve Jobs Stood On. *Wired*, October 2011. ISSN 1059-1028. URL https://www.wired.com/2011/10/thedennisritchieeffect/. Section: tags.

[34] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[35] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005. ISSN 0164-0925. doi: 10.1145/1065887.1065892. URL https://doi.org/10.1145/1065887.1065892.

[36] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient Field-sensitive Pointer Analysis of C. *ACM Trans. Program. Lang. Syst.*, 30(1), November 2007. ISSN 0164-0925. doi: 10.1145/1290520.1290524. URL http://doi.acm.org/10.1145/1290520.1290524.

[37] Natalie Popescu, Ziyang Xu, Sotiris Apostolakis, David I. August, and Amit Levy. Safer at any speed: Automatic context-aware safety enhancement for rust. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi: 10.1145/3485480. URL https://doi.org/10.1145/3485480.

[38] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 763–779, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386036. URL https://doi.org/10.1145/3385412.3386036.

[39] Eric Reed. Patina: A formalization of the Rust programming language. Master's thesis, University of Washington Department of Computer Science and Engineering, 2015.

[40] Jamey Sharp. jameysharp/corrode, February 2020. URL https://github.com/jameysharp/corrode. original-date: 2016-05-05T21:12:52Z.

[41] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*,

POPL '96, pages 32–41, New York, NY, USA, 1996. ACM. ISBN 978-0-89791-769-8. doi: 10.1145/237721.237727. URL http://doi.acm.org/10.1145/237721.237727.

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

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

[44] Ole Tange. Gnu parallel 20220422, April 2021. URL https://doi.org/10.5281/zenodo.6479152. GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them.

[45] The Rust developers. The Rust Reference, 2021. URL https://doc.rust-lang.org/stable/reference/.

[46] Aaron Weiss, Daniel Patterson, Nicholas D Matsakis, and Amal Ahmed. Oxide: The Essence of Rust, August 2020.