Translating C to Safer Rust

MEHMET EMRE, University of California Santa Barbara, USA
RYAN SCHROEDER, University of California Santa Barbara, USA
KYLE DEWEY, California State University Northridge, USA
BEN HARDEKOPF, University of California Santa Barbara, USA

Rust is a relatively new programming language that targets efficient and safe systems-level applications. It includes a sophisticated type system that allows for provable memory- and thread-safety, and is explicitly designed to take the place of unsafe languages such as C and C++ in the coding ecosystem. There is a large existing C and C++ codebase (many of which have been affected by bugs and security vulnerabilities due to unsafety) that would benefit from being rewritten in Rust to remove an entire class of potential bugs. However, porting these applications to Rust manually is a daunting task.

In this paper we investigate the problem of automatically translating C programs into *safer* Rust programs—that is, Rust programs that improve on the safety guarantees of the original C programs. We conduct an in-depth study into the underlying causes of unsafety in translated programs and the relative impact of fixing each cause. We also describe a novel technique for automatically removing a particular cause of unsafety and evaluate its effectiveness and impact. This paper presents the first empirical study of unsafety in *translated* Rust programs (as opposed to programs originally written in Rust) and also the first technique for automatically removing causes of unsafety in translated Rust programs.

CCS Concepts: • **Software and its engineering** → *Software evolution*; **Maintaining software**; **Source code generation**; **Software maintenance tools**.

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

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

1 INTRODUCTION

Rust is a relatively recent programming language designed for building safe and efficient low-level software [Klabnik and Nichols 2018]. It provides strong static guarantees about memory and thread safety while avoiding the need for garbage collection, and allows for low-level data manipulations often required by system-level software. Rust has been used for building operating systems, web browsers, and garbage collectors [Anderson et al. 2015; Levy et al. 2015; Lin et al. 2016] and it is being adopted into complex software projects with large C/C++ code-bases such as Firefox [Bryant 2016], the Linux kernel [rus [n.d.]a,n], and Android [Stoep and Hines 2021].

An alarming amount of critical systems software (much of which predates the development of Rust) is instead written in unsafe languages such as C and C++. Those languages' lack of memory

Authors' addresses: Mehmet Emre, emre@cs.ucsb.edu, University of California Santa Barbara, Santa Barbara, CA, USA; Ryan Schroeder, rschroeder@ucsb.edu, University of California Santa Barbara, Santa Barbara, CA, USA; Kyle Dewey, kyle.dewey@csun.edu, California State University Northridge, Northridge, CA, USA; Ben Hardekopf, benh@cs.ucsb.edu, University of California Santa Barbara, Santa Barbara, CA, USA.

2475-1421/2021/10-ART121
https://doi.org/10.1145/3485498

and thread safety has led to numerous critical bugs and security flaws [noa 2021a,b; Durumeric et al. 2014] with attendant costs in terms of both money and human lives [Durumeric et al. 2014; Leveson and Turner 1993]. 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). As an example, one indicative (though unscientific) 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 [Hutt 2021].

One obvious objection to porting existing software into Rust is the sheer effort required to rewrite the code in a new language. An automated, rather than manual, translation would make that effort much more practical. The primary barrier to such an automated translation is Rust's sophisticated type system which it uses to provide the desired memory and thread safety guarantees. To produce verifiably safe Rust code from unsafe C code, for example, requires the translator to analyze the relevant properties of the C code and create a suitable well-typed Rust program that correctly expresses those properties.

There have been several industry-backed attempts to automatically translate C programs to Rust [Citrus Developers [n.d.]; Immunant inc. 2020b; Sharp 2020]. These translations are purely syntactic in nature, producing memory- and thread-unsafe Rust code that closely mimics the original C code and explicitly bypasses the safety checks of the Rust compiler (by marking all translated code with Rust's built-in `unsafe` annotation). While these tools provide a good starting point for automated translation, they leave the hard work of manually reasoning about the safety properties of the translated program and rewriting the code to enable the Rust compiler to verify those properties to the developer. To our knowledge there has been no academic or industry investigation into the question of whether and how unsafe languages can be automatically translated into *safe* Rust programs. This paper makes two major contributions towards the goal of automatically translating sequential C programs (for this stage of the work) to *safer* Rust programs, i.e., the goal for now is not complete safety but simply more safety than the existing naive syntactic translations.

Our first contribution is a quantitative study on the sources and causes of unsafety present in Rust programs that have been syntactically translated from C programs. While there have been studies on unsafe code in native, hand-written Rust programs [Astrauskas et al. 2020; Qin et al. 2020], this is the first study that examines automatically translated Rust programs. We focus on Rust code translated from C using the existing `c2rust` translator [Immunant inc. 2020b]. Our findings indicate that unsafety in automatically translated Rust code differs in various significant ways from unsafety in natively written Rust code. For example, a prevalent source of unsafety in automatically translated code, unlike native code, is due to the use of *raw pointers*: the translation to Rust converts all C pointers into raw pointers, and any dereference of a raw pointer must be marked as unsafe. We break down all of the sources of unsafety present in our corpus of programs, quantify how often they occur, explain what causes these sources of unsafety in the original C programs, and quantify the impact of addressing each source of unsafety on the overall safety of the translated Rust programs.

Our second contribution, informed by our study, is a technique for automatically generating safer Rust code by addressing one common cause of unsafety. We focus specifically on the use of raw pointers in the translated programs. Idiomatic Rust code instead uses safe *references* with explicitly annotated lifetime information that allows the Rust compiler to safely deallocate the associated memory when it is no longer needed. Rust uses an ownership-based model for statically reasoning about references, shared references, mutability, lifetimes, and overall memory- and thread-safety

[Boyapati et al. 2002, 2003]. Using this model, valid Rust programs are automatically proven safe via Rust's *borrow checker*. Invalid Rust programs, i.e., those unable to be statically proven as safe, are rejected by the borrow checker (which ignores code explicitly marked as unsafe). We make the key insight that we can piggyback on Rust's borrow checker in order to extract the lifetime, sharing, and mutability information we need to turn a subset of raw pointers into safe references. We introduce and implement a translation technique based on this insight which takes naively translated, completely unsafe Rust programs and generates *safer* Rust programs (specifically, in this case, one with fewer raw pointers). We evaluate our implementation on a corpus of C programs and report on its effectiveness.

The specific contributions of this paper are as follows:

- A study of the sources of unsafety in Rust code that has been produced by `c2rust` (Section 2);
- A technique to rewrite a particular source of unsafety in translated programs (a specific kind of raw pointer) that hooks into the Rust compiler to extract type- and borrow-checker results and uses them to generate verifiably safe code (Section 3);
- An implementation of this technique[1] with a corresponding evaluation of its effectiveness (Section 4).

We end with a discussion of related work (Section 5) and conclusion (Section 6).

2 UNSAFETY IN TRANSLATED RUST PROGRAMS

We investigate the various sources of unsafety in Rust programs that have been translated from C using `c2rust`. While there are existing studies of unsafe code in the native Rust ecosystem [Astrauskas et al. 2020; Qin et al. 2020] our investigation is specifically about automatically translated Rust programs, which may have a different distribution of unsafe code than Rust programs written by developers.

2.1 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.

We have collected 17 open source C programs of various sizes and application domains, as shown in Table 1. 11 of the programs came from the `c2rust` manual [Immunant inc. 2020a] (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 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[2], and annotations such as `#[no_mangle]` and `#[repr(C)]` to make the Rust code compatible with the C ecosystem.

Table 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.

[1] The accompanying artifact including our evaluation is available at [Emre and Schroeder 2021].
[2] 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.

Table 1. Corpus 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 programs are abbreviated as TI and RFK, respectively.

Program	Application 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 analysis	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

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 2.2 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.

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 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.

2.2 Provenance of Unsafety

The Rust Reference [The Rust developers [n.d.]] 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,[3] `malloc`, or `free`; or making an indirect call via a function pointer;[4]
- **Cast**: unsafe casting using `std::mem::transmute`;
- **InlineAsm**: using inline assembly.

We collect our data on a function-level because (1) `c2rust` marks functions `unsafe` rather than inserting `unsafe` blocks,[5] and (2) existing work on quantifying unsafe behavior of Rust programs in general [Astrauskas et al. 2020] 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

[3] `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 3.2, so we do not count these functions as external functions in our study.

[4] 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.

[5] 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.

Table 2. How many times different categories of unsafety appear in each corpus program. The meaning of each column is explained in Section 2.2.

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

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 Rust programs to Rust High-level IR (HIR)[6], 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[7]; 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 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 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.

[6]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.

[7]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. 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 2.2. The percentages with respect to the total number of unsafe functions are notated with a superscript. Some behavior does not occur uniquely in any program, in that case, we do not include the percentage for that column.

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	$3^{100\%}$	0	0	0	0	0	0
grabc	0	$3^{50\%}$	$1^{17\%}$	$4^{67\%}$	0	$0^{0\%}$	$0^{0\%}$	$4^{67\%}$	$1^{17\%}$	$5^{83\%}$	0	$0^{0\%}$	$0^{0\%}$	$0^{0\%}$	$0^{0\%}$
libcsv	0	$0^{0\%}$	$1^{4\%}$	$1^{4\%}$	0	$0^{0\%}$	$0^{0\%}$	$9^{39\%}$	$13^{57\%}$	$22^{96\%}$	0	$4^{17\%}$	$0^{0\%}$	$0^{0\%}$	$0^{0\%}$
urlparser	0	$0^{0\%}$	$0^{0\%}$	$14^{67\%}$	0	$0^{0\%}$	$0^{0\%}$	$20^{95\%}$	$0^{0\%}$	$17^{81\%}$	0	$1^{5\%}$	$0^{0\%}$	$19^{90\%}$	$0^{0\%}$
RFK	0	$0^{0\%}$	$0^{0\%}$	$15^{88\%}$	0	$0^{0\%}$	$1^{6\%}$	$15^{88\%}$	$0^{0\%}$	$7^{41\%}$	0	$0^{0\%}$	$0^{0\%}$	$2^{12\%}$	$1^{6\%}$
genann	0	$0^{0\%}$	$0^{0\%}$	$14^{52\%}$	0	$0^{0\%}$	$1^{4\%}$	$24^{89\%}$	$0^{0\%}$	$21^{78\%}$	0	$13^{48\%}$	$1^{4\%}$	$18^{67\%}$	$2^{7\%}$
xzoom	0	$1^{10\%}$	$1^{10\%}$	$10^{100\%}$	0	$0^{0\%}$	$0^{0\%}$	$9^{90\%}$	$0^{0\%}$	$8^{80\%}$	0	$0^{0\%}$	$0^{0\%}$	$4^{40\%}$	$0^{0\%}$
lil	0	$0^{0\%}$	$2^{1\%}$	$73^{46\%}$	0	$0^{0\%}$	$1^{1\%}$	$134^{84\%}$	$14^{9\%}$	$148^{93\%}$	0	$52^{33\%}$	$1^{1\%}$	$100^{63\%}$	$2^{1\%}$
snudown	0	$0^{0\%}$	$1^{1\%}$	$37^{40\%}$	0	$0^{0\%}$	$0^{0\%}$	$63^{68\%}$	$19^{21\%}$	$90^{98\%}$	0	$0^{0\%}$	$0^{0\%}$	$34^{37\%}$	$1^{1\%}$
json-c	0	$62^{35\%}$	$10^{6\%}$	$49^{28\%}$	0	$0^{0\%}$	$4^{2\%}$	$114^{64\%}$	$24^{13\%}$	$144^{81\%}$	0	$49^{28\%}$	$1^{1\%}$	$51^{29\%}$	$11^{6\%}$
bzip2	0	$0^{0\%}$	$3^{2\%}$	$79^{63\%}$	0	$0^{0\%}$	$7^{6\%}$	$85^{67\%}$	$23^{18\%}$	$82^{65\%}$	0	$3^{2\%}$	$2^{2\%}$	$26^{21\%}$	$6^{5\%}$
libzahl	0	$0^{0\%}$	$0^{0\%}$	$115^{50\%}$	0	$111^{48\%}$	$0^{0\%}$	$114^{50\%}$	$90^{39\%}$	$230^{100\%}$	0	$0^{0\%}$	$0^{0\%}$	$110^{48\%}$	$0^{0\%}$
TI	0	$0^{0\%}$	$0^{0\%}$	$13^{6\%}$	0	$0^{0\%}$	$1^{0\%}$	$104^{45\%}$	$74^{32\%}$	$175^{76\%}$	0	$73^{32\%}$	$1^{0\%}$	$16^{7\%}$	$49^{21\%}$
tinycc	0	$286^{46\%}$	$5^{1\%}$	$492^{79\%}$	0	$0^{0\%}$	$8^{1\%}$	$498^{80\%}$	$54^{9\%}$	$577^{92\%}$	0	$244^{39\%}$	$1^{0\%}$	$358^{57\%}$	$30^{5\%}$
optipng	0	$57^{10\%}$	$4^{1\%}$	$297^{52\%}$	0	$0^{0\%}$	$14^{2\%}$	$371^{65\%}$	$126^{22\%}$	$487^{85\%}$	0	$57^{10\%}$	$7^{1\%}$	$141^{25\%}$	$29^{5\%}$
tmux	0	$569^{42\%}$	$9^{1\%}$	$710^{52\%}$	0	$0^{0\%}$	$9^{1\%}$	$1030^{75\%}$	$244^{18\%}$	$1328^{97\%}$	0	$489^{36\%}$	$1^{0\%}$	$653^{48\%}$	$5^{0\%}$
libxml2	0	$198^{7\%}$	$28^{1\%}$	$2220^{74\%}$	0	$0^{0\%}$	$39^{1\%}$	$2359^{78\%}$	$369^{12\%}$	$2740^{91\%}$	0	$1156^{38\%}$	$0^{0\%}$	$1268^{42\%}$	$183^{6\%}$
Total	0	$1176^{18\%}$	$65^{1\%}$	$4143^{62\%}$	0	$111^{2\%}$	$85^{1\%}$	$4953^{74\%}$	$1054^{16\%}$	$6081^{91\%}$	0	$2141^{32\%}$	$15^{0\%}$	$2800^{42\%}$	$319^{5\%}$

Table 3, in contrast to Table 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 $\exists!$ 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 2, and 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 2), at the function level it compares much more closely to Global and Extern (Table 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.

2.3 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 our programs (`libzahl`) for architecture-specific optimizations. We examine the remaining categories in more detail below.

2.3.1 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:

- The raw pointer appears as part of the public signature of an API implemented by the program. This is a common occurrence in our corpus programs 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 2.2:

(1) Number of declared pointers involved in that category (Table 4);
(2) Number of dereferences of pointers in that category that appear in the code (Table 5);
(3) Number of unsafe functions that use pointers from that category (Table 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, 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 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 [Steensgaard 1996] and Rust's type system [The Rust developers [n.d.]]. 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

Table 4. Raw pointer declarations, grouped by category. ∃! and $\exists_{\geq 2}$ are explained in Section 2.2. Lifetime category contains only unique (∃!) causes by definition. The percentages are relative to the Total column.

Program	VoidPtr		PtrArith		ExternPtr		Lifetime	Total
	∃!	$\exists_{\geq 2}$	∃!	$\exists_{\geq 2}$	∃!	$\exists_{\geq 2}$	∃!	
qsort	$0^{0\%}$	$0^{0\%}$	$2^{50\%}$	$0^{0\%}$	$0^{0\%}$	$0^{0\%}$	$2^{50\%}$	4
grabc	$0^{0\%}$	$0^{0\%}$	$1^{8\%}$	$0^{0\%}$	$5^{38\%}$	$0^{0\%}$	$7^{54\%}$	13
libcsv	$7^{19\%}$	$10^{27\%}$	$0^{0\%}$	$7^{19\%}$	$2^{5\%}$	$3^{8\%}$	$18^{49\%}$	37
urlparser	$0^{0\%}$	$70^{89\%}$	$0^{0\%}$	$70^{89\%}$	$4^{5\%}$	$70^{89\%}$	$5^{6\%}$	79
RFK	$0^{0\%}$	$0^{0\%}$	$1^{50\%}$	$0^{0\%}$	$1^{50\%}$	$0^{0\%}$	$0^{0\%}$	2
genann	$0^{0\%}$	$61^{84\%}$	$0^{0\%}$	$62^{85\%}$	$6^{8\%}$	$62^{85\%}$	$5^{7\%}$	73
xzoom	$0^{0\%}$	$24^{83\%}$	$1^{3\%}$	$25^{86\%}$	$3^{10\%}$	$25^{86\%}$	$0^{0\%}$	29
lil	$1^{0\%}$	$314^{72\%}$	$60^{14\%}$	$316^{72\%}$	$10^{2\%}$	$317^{72\%}$	$50^{11\%}$	438
snudown	$0^{0\%}$	$159^{65\%}$	$2^{1\%}$	$161^{66\%}$	$47^{19\%}$	$156^{64\%}$	$31^{13\%}$	244
json-c	$13^{4\%}$	$227^{76\%}$	$1^{0\%}$	$227^{76\%}$	$9^{3\%}$	$211^{71\%}$	$41^{14\%}$	297
bzip2	$43^{19\%}$	$89^{39\%}$	$47^{21\%}$	$70^{31\%}$	$9^{4\%}$	$89^{39\%}$	$37^{16\%}$	227
libzahl	$9^{2\%}$	$324^{71\%}$	$114^{25\%}$	$322^{70\%}$	$3^{1\%}$	$319^{70\%}$	$7^{2\%}$	457
TI	$15^{2\%}$	$41^{5\%}$	$724^{84\%}$	$41^{5\%}$	$4^{0\%}$	$41^{5\%}$	$82^{9\%}$	866
tinycc	$18^{1\%}$	$1100^{81\%}$	$16^{1\%}$	$1094^{81\%}$	$24^{2\%}$	$1084^{80\%}$	$191^{14\%}$	1352
optipng	$12^{1\%}$	$1016^{72\%}$	$121^{9\%}$	$987^{70\%}$	$51^{4\%}$	$1013^{72\%}$	$207^{15\%}$	1407
tmux	$265^{6\%}$	$3550^{76\%}$	$17^{0\%}$	$3311^{71\%}$	$177^{4\%}$	$3554^{77\%}$	$622^{13\%}$	4645
libxml2	$451^{5\%}$	$8332^{84\%}$	$171^{2\%}$	$7729^{78\%}$	$152^{2\%}$	$8336^{84\%}$	$839^{8\%}$	9950
Total	$834^{4\%}$	$15317^{76\%}$	$1276^{6\%}$	$14422^{72\%}$	$507^{3\%}$	$15280^{76\%}$	$2142^{11\%}$	20116

Table 5. Raw pointer dereferences, grouped by category. ∃! and $\exists_{\geq 2}$ are explained in Section 2.2. Lifetime category contains only unique (∃!) causes by definition. The percentages are relative to the Total column.

Program	VoidPtr		PtrArith		ExternPtr		Lifetime	Total
	∃!	$\exists_{\geq 2}$	∃!	$\exists_{\geq 2}$	∃!	$\exists_{\geq 2}$	∃!	
qsort	$0^{0\%}$	$0^{0\%}$	$6^{60\%}$	$0^{0\%}$	$0^{0\%}$	$0^{0\%}$	$4^{40\%}$	10
grabc	$0^{0\%}$	$0^{0\%}$	$2^{10\%}$	$0^{0\%}$	$4^{19\%}$	$0^{0\%}$	$15^{71\%}$	21
libcsv	$0^{0\%}$	$26^{15\%}$	$0^{0\%}$	$26^{15\%}$	$0^{0\%}$	$17^{10\%}$	$148^{85\%}$	174
urlparser	$0^{0\%}$	$2^{3\%}$	$0^{0\%}$	$2^{3\%}$	$0^{0\%}$	$2^{3\%}$	$58^{97\%}$	60
RFK	$0^{0\%}$	$0^{0\%}$	$24^{100\%}$	$0^{0\%}$	$0^{0\%}$	$0^{0\%}$	$0^{0\%}$	24
genann	$0^{0\%}$	$312^{92\%}$	$22^{6\%}$	$313^{92\%}$	$4^{1\%}$	$313^{92\%}$	$0^{0\%}$	339
xzoom	$0^{0\%}$	$37^{22\%}$	$23^{13\%}$	$114^{66\%}$	$12^{7\%}$	$105^{61\%}$	$23^{13\%}$	172
lil	$0^{0\%}$	$895^{54\%}$	$127^{8\%}$	$897^{54\%}$	$8^{0\%}$	$897^{54\%}$	$636^{38\%}$	1668
snudown	$0^{0\%}$	$489^{58\%}$	$35^{4\%}$	$493^{59\%}$	$185^{22\%}$	$474^{56\%}$	$129^{15\%}$	842
json-c	$9^{0\%}$	$1639^{89\%}$	$39^{2\%}$	$1646^{89\%}$	$56^{3\%}$	$1433^{78\%}$	$93^{5\%}$	1843
bzip2	$1704^{45\%}$	$1192^{32\%}$	$173^{5\%}$	$627^{17\%}$	$11^{0\%}$	$1195^{32\%}$	$679^{18\%}$	3764
libzahl	$1^{0\%}$	$1220^{50\%}$	$1183^{48\%}$	$1220^{50\%}$	$22^{1\%}$	$1191^{48\%}$	$31^{1\%}$	2457
TI	$426^{23\%}$	$184^{10\%}$	$1237^{67\%}$	$184^{10\%}$	$0^{0\%}$	$184^{10\%}$	$0^{0\%}$	1847
tinycc	$28^{0\%}$	$4525^{80\%}$	$122^{2\%}$	$4522^{80\%}$	$9^{0\%}$	$4491^{80\%}$	$946^{17\%}$	5632
optipng	$5^{0\%}$	$5212^{86\%}$	$203^{3\%}$	$5043^{83\%}$	$36^{1\%}$	$5208^{86\%}$	$606^{10\%}$	6062
tmux	$1002^{5\%}$	$17687^{82\%}$	$131^{1\%}$	$16449^{76\%}$	$345^{2\%}$	$17694^{82\%}$	$2486^{11\%}$	21658
libxml2	$986^{2\%}$	$45764^{87\%}$	$372^{1\%}$	$41475^{79\%}$	$235^{0\%}$	$45771^{87\%}$	$5175^{10\%}$	52546
Total	$4161^{4\%}$	$79184^{80\%}$	$3693^{4\%}$	$73011^{74\%}$	$927^{1\%}$	$78975^{80\%}$	$11025^{11\%}$	99109

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).

Table 6. Functions using raw pointers in a given category. $\exists!$ and $\exists_{\geq 2}$ are explained in Section 2.2. Lifetime category contains only unique ($\exists!$) causes by definition. The percentages are relative to the Total column.

Program	VoidPtr		PtrArith		ExternPtr		Lifetime	Total
	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	$\exists_{\geq 2}$	$\exists!$	
qsort	$0^{0\%}$	$0^{0\%}$	$2^{67\%}$	$0^{0\%}$	$0^{0\%}$	$0^{0\%}$	$1^{33\%}$	3
grabc	$0^{0\%}$	$0^{0\%}$	$0^{0\%}$	$0^{0\%}$	$1^{50\%}$	$0^{0\%}$	$1^{50\%}$	2
libcsv	$1^{6\%}$	$5^{28\%}$	$0^{0\%}$	$3^{17\%}$	$0^{0\%}$	$4^{22\%}$	$12^{67\%}$	18
urlparser	$0^{0\%}$	$5^{71\%}$	$0^{0\%}$	$5^{71\%}$	$0^{0\%}$	$5^{71\%}$	$2^{29\%}$	7
robotfindskitten	$0^{0\%}$	$0^{0\%}$	$0^{0\%}$	$0^{0\%}$	$0^{0\%}$	$0^{0\%}$	$2^{100\%}$	2
genann	$0^{0\%}$	$6^{67\%}$	$0^{0\%}$	$6^{67\%}$	$0^{0\%}$	$6^{67\%}$	$3^{33\%}$	9
xzoom	$0^{0\%}$	$9^{100\%}$	$0^{0\%}$	$5^{56\%}$	$0^{0\%}$	$8^{89\%}$	$0^{0\%}$	9
lil	$1^{1\%}$	$62^{79\%}$	$9^{12\%}$	$61^{78\%}$	$0^{0\%}$	$62^{79\%}$	$6^{8\%}$	78
snudown	$0^{0\%}$	$47^{85\%}$	$0^{0\%}$	$47^{85\%}$	$2^{4\%}$	$47^{85\%}$	$6^{11\%}$	55
json-c	$6^{8\%}$	$50^{65\%}$	$0^{0\%}$	$50^{65\%}$	$1^{1\%}$	$47^{61\%}$	$20^{26\%}$	77
bzip2	$7^{17\%}$	$28^{68\%}$	$2^{5\%}$	$19^{46\%}$	$0^{0\%}$	$28^{68\%}$	$4^{10\%}$	41
libzahl	$0^{0\%}$	$79^{88\%}$	$9^{10\%}$	$79^{88\%}$	$0^{0\%}$	$79^{88\%}$	$2^{2\%}$	90
tulipindicators	$2^{1\%}$	$3^{2\%}$	$96^{66\%}$	$3^{2\%}$	$0^{0\%}$	$3^{2\%}$	$45^{31\%}$	146
tinycc	$3^{3\%}$	$76^{64\%}$	$5^{4\%}$	$75^{63\%}$	$0^{0\%}$	$65^{55\%}$	$35^{29\%}$	119
optipng	$4^{2\%}$	$175^{67\%}$	$12^{5\%}$	$171^{66\%}$	$5^{2\%}$	$177^{68\%}$	$62^{24\%}$	260
tmux	$26^{5\%}$	$483^{88\%}$	$2^{0\%}$	$467^{85\%}$	$7^{1\%}$	$482^{88\%}$	$31^{6\%}$	549
libxml2	$20^{3\%}$	$532^{68\%}$	$6^{1\%}$	$492^{63\%}$	$7^{1\%}$	$535^{69\%}$	$210^{27\%}$	779
Total	$70^{3\%}$	$1560^{70\%}$	$141^{6\%}$	$1483^{66\%}$	$23^{1\%}$	$1548^{69\%}$	$441^{20\%}$	2241

Tables 4 to 6 contain the results of our analysis. Tables 4 and 5 can be used 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`.

2.3.2 External Function Calls. Removing unsafety due to external function calls can only be done by replacing those external calls. We investigated our corpus prorgrams in an effort to prioritize which external functions would be most beneficial to replace. The extended version of this paper [Emre et al. 2021b] goes over the prevalence of external functions and which external functions have the highest impact on safety. To summarize our findings, the programs collectively declare 409 external functions; 73% of the external functions are unique to one program; and 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. Finally, the external functions used in most programs and the external functions having the highest impact (in terms of functions that call that external function, directly or indirectly) implement string manipulation, I/O, or error handling.

2.3.3 C-style Unions. We manually inspected all C-style unions declared in our corpus programs. 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 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.

2.3.4 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[8] 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.

2.4 Observations and Discussion

From Table 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 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 1 shows the results of this calculation. We include both the results for all functions in our corpus programs, 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 the necessary information required to produce verifiably safe code. Ultimately, unsafety stems from the fact

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

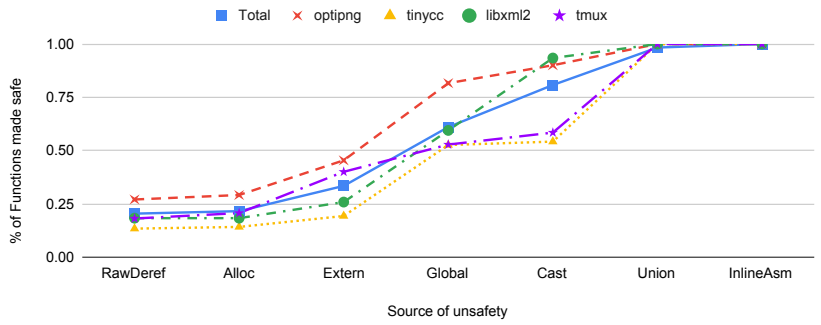

Fig. 1. Cumulative percentage of functions made safe by fixing the given unsafety category. The "Total" line shows this number for all functions across all programs.

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 linear 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.

Nevertheless, translating C to safer Rust is a worthwhile goal and, we believe, a reasonable endeavour to undertake. We show in the next section 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.

3 TURNING RAW POINTERS INTO SAFE REFERENCES

In this section 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 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 technique consists of three stages after the `c2rust` translation of the original C program:

(1) Connect the definitions and uses of types and functions across modules, and remove unnecessary unsafety and mutability markers. (Section 3.2)

(2) Determine initial lifetimes to convert unsafe raw pointers into safe references. (Section 3.3)

(3) Iteratively rewrite the program to resolve lifetime inference and borrow checking errors. (Section 3.4)

The rest of this section is structured as follows: We (1) give a description of our algorithm (Section 3.1), explain the algorithm with a running example (Sections 3.2 to 3.4), and finally we discuss the algorithmic complexity of our technique (Section 3.5). More details of our technique is given in [Emre et al. 2021a].

3.1 Description of Our Algorithm

Our algorithm uses the lattice of *configurations* described in Figure 2. 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 representation 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

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

Fig. 2. 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. $\mathscr{P}$ is the powerset operation.

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 2, 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 [Weiss et al. [n.d.]]). We use the type equivalence constraints to propagate which locations in the program should have a raw pointer using a Steensgaard-style alias analysis [Steensgaard 1996], and use the subtyping constraints to propagate which locations should be owned using an Andersen-style analysis [Andersen 1994].

We implement the initial optimistic rewrite (Section 3.3), and the iterative rewrite (Section 3.4) as a single algorithm, known as ResolveLifetimes (Figure 3). ComputeTaintAnalysis computes the taint analysis from Section 2.3.1 and a subset-based variant of it. The rest of the functions in the algorithm are described in Sections 3.2 to 3.4. $\bot$ 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 3.3 and 3.4, and we give the precise rewrite rules in [Emre et al. 2021a]. 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.

3.1.1 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 3). The error numbers refer to the ones in Rust Compiler Error Index [noa [n.d.]].

```
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)
```

Fig. 3. Our algorithm for ResolveLifetimes, the parts of our method after merging struct definitions and resolving extern functions.

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.

3.2 Connecting Functions and Data Structures Across Modules

This and the following two sections explain our technique in relation to an example C program shown in Figure 4a, which implements a binary search tree. Figure 4b shows the result of running

```c
// bst.h: BST node definition
typedef struct node_t {
  node_t* left;
  node_t* right;
  int value;
};

node_t* find(int value, node_t* node);
void insert(int value, node_t* node);

// bst.c: BST implementation
#include "bst.h"

node_t* find(int value, node_t* node) {
  if (value < node->value && node->left) {
    return find(value, node->left);
  } else if (value > node->value && node->
      right) {
    return find(value, node->right);
  } else if (value == node->value) {
    return node;
  }
  return NULL;
}

void insert(int value, *node_t node) {
  // Implementation omitted for brevity.
}

// main.c: program entry point
#include "bst.h"

int main() {
  node_t* tree = malloc(sizeof(node_t));
  tree->value = malloc(sizeof(int));
  *(tree->value) = 3;
  insert(1, tree);
  insert(2, tree);
  *(find(3, tree)->value) = 4;
  return 0;
}
```

(a) A C program implementing a binary search tree. We omit the implementation of insert for brevity.

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

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

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

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

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

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

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

(b) The Rust program produced from Figure 4a. 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).

Fig. 4. An example a of C program translated to Rust by `c2rust`.

`c2rust` on the C program. We will step through each stage of our technique in the remainder of this section, demonstrating on the given example.

The original C program may consist of multiple compilation units (e.g., Figure 4a has two: `bst.c` and `main.c`). `c2rust` translates each compilation unit separately into its own Rust module (e.g., Figure 4b 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 4b, 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.

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

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

pub unsafe fn find(value: c_int, node: *mut node_t) -> *mut node_t {
  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_t;
}
pub unsafe fn insert(value: c_int, node: *mut node_t) { /*...*/ }

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

pub unsafe fn main_0() -> int {
    let mut tree = malloc(::std::mem::size_of::<node_t>()) as * mut node_t;
    (*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;
}
```

Fig. 5. The Rust program from Figure 4b after deduplicating struct definitions and converting extern functions to imports. The unnecessary mutability annotations have been removed from the function arguments.

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 4b the type `bst::node_t` and the type `main::node_t` are 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 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 5 shows our example after this process.

3.3 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.[9] 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:

```
pub struct node_t<'a1, 'a2> {
    pub left: Option<&'a1 mut node_t<'a1, 'a2>>,
    pub right: Option<&'a1 mut node_t<'a1, 'a2>>,
    pub value: Option<&'a2 mut c_int>,
}
```

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

```
fn find<'a1, 'a2, 'a3, 'a4, 'a5, 'a6>(value: c_int, mut node: Option<&'a1 mut node_t<'a2, 'a3>>) ->
    Option<&'a4 mut node_t<'a5, 'a6>>;
fn insert<'a1, 'a2, 'a3>(value: c_int, mut node: Option<&'a1 mut node_t<'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 avoid consuming the original object in the contexts that it is not assigned to or deliberately consumed:

[9] 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.

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

Fig. 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.

- When using an immutable reference, we clone it so the original object is not destroyed.[10]
- 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 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)`.
- 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 7 shows our example program after all of these transformations.

3.4 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 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.

[10]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`.

```
1  // bst.rs
2  use std::os::raw::c_int;
3
4  pub struct node_t<'a1, 'a2> {
5      pub left: Option<&'a1 mut node_t<'a1, 'a2>>,
6      pub right: Option<&'a1 mut node_t<'a1, 'a2>>,
7      pub value: Option<&'a2 mut c_int>,
8  }
9  impl<'a1, 'a2> std::default::Default for node_t<'a1, 'a2> {  /* ... */ }
10 pub fn insert<'a1, 'a2, 'a3>(mut value: c_int,
11                             mut node: Option<&'a1 mut node_t<'a2, 'a3>>) { /* ... */ }
12
13 pub fn find<'a1, 'a2, 'a3, 'a4, 'a5, 'a6>(mut value: c_int, mut node: Option<&'a1 mut node_t<'a2, 'a3>>)
14  -> Option<&'a4 mut node_t<'a5, 'a6>> {
15     if value < **(**node.as_ref().unwrap()).value.as_ref().unwrap() && !(**node.as_ref().unwrap()).left.
       is_none() {
16         return find(value, borrow_mut(&mut (*node.unwrap()).left))
17     } else {
18         if value > **(**node.as_ref().unwrap()).value.as_ref().unwrap() && !(**node.as_ref().unwrap()).
       right.is_none() {
19             return find(value, borrow_mut(&mut (*node.unwrap()).right))
20         } else { if value == **(**node.as_mut().unwrap()).value.as_mut().unwrap() { return node } }
21     }
22     return None;
23 }
24
25 // main.rs
26 use std::os::raw::c_int;
27 use bst::{node_t, insert, find};
28
29 pub fn main_0() -> int {
30     let mut tree = Some(Box::new(node_t::default()).as_mut());
31     **(**tree.as_mut().unwrap()).value.as_mut().unwrap() = 3;
32     insert(1, borrow_mut(&mut tree));
33     insert(2, borrow_mut(&mut tree));
34     **(**find(3, borrow_mut(&mut tree)).as_mut().unwrap()).value.as_mut().unwrap() = 4;
35     return 0;
36 }
```

Fig. 7. The Rust program from Figure 5 after converting raw pointers into references.

- 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 2.3.1 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 2.3.1 and marking the references promoted to owned references as sinks.

We demonstrate these steps on the example program in Figure 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_t>>`).[11] 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 8 shows the final fixed `find` function, Section 4 of [Emre et al. 2021a] contains the full fixed program.

```
1  pub fn find<'a1, 'a2, 'a3, 'a4, 'a5, 'a6>(mut value: c_int, mut node: Option<&'a1 mut node_t<'a2, 'a3>>)
2   -> Option<&'a4 mut node_t<'a5, 'a6>>
3  where 'a1: 'a4, 'a5: 'a2, 'a6: 'a3, 'a3: 'a6, 'a2: 'a5
4  {
5      if value < **(**node.as_ref().unwrap()).value.as_ref().unwrap() && !(**node.as_ref().unwrap()).left.
         is_none() {
6          return find(value, borrow_mut(&mut (*node.unwrap()).left))
7      } else {
8          if value > **(**node.as_ref().unwrap()).value.as_ref().unwrap() && !(**node.as_ref().unwrap()).
         right.is_none() {
9              return find(value, borrow_mut(&mut (*node.unwrap()).right))
10         } else { if value == **(**node.as_mut().unwrap()).value.as_mut().unwrap() { return node } }
11     }
12     return None;
13 }
```

Fig. 8. The `find` function after applying all steps of our technique. The rest of the program is same as Figure 7. We reproduced the full program at this stage on [Emre et al. 2021a] .

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

[11] We could potentially make it a `Box<node_t>` 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.

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

3.5 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 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 3 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[12] 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 the lattice is lexicographically ordered, we discard all lifetime constraints when we promote a pointer; in total ResolveLifetimes may have $O((c + l)r \max_i |f_i|)$ 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.

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 3.1, 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 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.

[12] We do not process lifetimes in nested functions

Table 7. Number of unsafe functions due uniquely to using raw pointers. ResolveImports and ResolveLifetimes are the two phases of our method explained in Section 4.1; 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	Original	ResolveImports	ResolveLifetimes	Remaining	Total 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%

4.1 Evaluation Setup

We evaluate our technique in two parts, which we label in our tables as described below:

- **ResolveImports:** This is the first step of our technique, described in Section 3.2, 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 Sections 3.3 and 3.4, which converts `Lifetime` raw pointers (as described in Section 2.3.1) into safe references. As we did in Section 3 we will use the term "raw pointers" throughout to mean specifically `Lifetime` raw pointers.

We collect the following statistics, similar to Section 2.3.1, 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.

4.2 Results

Table 7 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 2.3.1. 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

Table 8. 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	Fixed (%)	Orig.	Remaining	Fixed	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%

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.
- 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 8 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. We reproduce a simplified version of the code snippet with this unsafe behavior in [Emre et al. 2021a].
- 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.

4.2.1 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 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 3.4; 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.

5 RELATED WORK

This section covers relevant background regarding C to Rust translation and other related work. Rust's ownership system and memory management method are reviewed for unfamiliar readers in Section 1 of our supplementary material [Emre et al. 2021a].

5.1 Translating C to Rust

There have been several early tools for translating C code to unsafe Rust code, such as Citrus [Citrus Developers [n.d.]] and Corrode [Sharp 2020]. Both of these tools are now superseded by `c2rust` [Immunant inc. 2020b], an industry-backed C99 to Rust translator. It is made of three parts: (1) the *translator* translates the C program to an unsafe Rust program that mirrors the C code; (2) the *refactoring tool* helps the programmer refactor and rewrite the unsafe Rust program into a mostly-safe Rust program by providing program-wide refactoring operations and scripting support; and (3) the *cross-check tool* allows for comparing the execution traces of two programs on a test input.

Since `c2rust` does not have any formal guarantees, it relies on the cross-check tool to validate that the initial Rust program behaves the same as the original C program and that the incrementally refactored Rust programs preserve that behavior. Our technique's implementation leverages the translator tool to provide an initial unsafe Rust program and the cross-check tool to validate that the output of our technique behaves the same as the original C program.

5.2 Characterizing Unsafe Code in Rust

Astrauskas et al. [2020] investigate usage of `unsafe` in practice across the open-source Rust ecosystem. They use manual inspection and automated queries to analyze program structure, types, and other information produced by the compiler. They find that most unsafe code is simple and well-encapsulated, however interoperability with other languages causes unsafe features to be used extensively. They report that 44.6% of the unsafe function definitions they found in the Rust ecosystem are bindings for foreign functions used for linking against C libraries. Their results support that porting these C libraries to safer Rust versions would significantly reduce the overall amount of unsafe dependencies in the Rust ecosystem.

Qin et al. [2020] empirically investigate the usage of Rust's safety mechanisms and `unsafe` in open source Rust projects. They also build two static bug detectors based on their study results, and revealed previously unknown bugs. Their results show that 66% of unsafe operations are due to unsafe memory operations such as type casting and raw pointer manipulation. They also report that the most common (42%) purpose of `unsafe` usage is to reuse existing code, including C code that performs pointer manipulation and calling into external libraries like `glibc`. These results indicate that converting C code to safe Rust is an important problem to increase trust in Rust code, and that converting raw pointer operations to safe Rust references accounts for a significant portion of this conversion.

Evans et al. [2020] keep track of potentially unsafe functions by examining the call graph because the use of unsafe may be gated behind an internal `unsafe` block in another function. Because they do not propagate the ultimate causes of unsafety, they deem 89.2% of the potentially unsafe functions as unsafe because of calling other unsafe functions. They note that most instances of unsafety is through library dependencies rather than using the `unsafe` keyword in the crate itself, and the most frequently called unsafe functions belong to the Rust standard library (65%), and calls to external C functions (22.5%). The next most common causes of unsafety in their classification are raw pointer dereferences and using globals, which is similar to our results in Section 2.2.

5.3 Using Compilers as Oracles

There are several program transformation tools that use compilers as oracles to check the annotations computed by the tool, and to provide counterexamples. Flanagan et al. [2001]; Flanagan and Leino [2001] describe a system that generates candidate annotations for ESC/Java [Flanagan et al. 2002], and refines the annotations until ESC/Java verifies all of them. CANAPA [Cielecki et al. 2006] propagates non-null constraints in Java programs by running ESC/Java2 on the input program, and analyzing the errors produced by it to add non-nullness annotations. Similarly, Cascade [Vakilian et al. 2015] processes type checker errors for a program to infer type qualifiers. To propagate these qualifiers, Cascade considers data flow edges (pseudo-assignments) as subtyping constraints and propagates any qualifier that is on the left-hand side of a pseudo-assignment to the right-hand side.

5.4 Formalizing Rust Ownership and Type Systems

There are several Rust formalizations in the literature [Benitez 2016; Jung et al. 2017; Reed 2015; Weiss et al. [n.d.]]. Here, we cover the formalizations that involve the Rust ownership system or borrow checker, as our technique interacts with both components.

Patina [Reed 2015] is a formal semantics for a pre-1.0 version of Rust. It focuses on using a syntactic version of the borrow checking algorithm based on lexically-scoped lifetimes. Since then, Rust has added support for non-lexical lifetimes and other new features, making safety now less restrictive than in pre-1.0 Rust.

The RustBelt project [Jung et al. 2017] describes a mechanised formal semantics for a Rust 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. However, since our technique requires reasoning only about Rust programs translated from C which do not use all Rust features (e.g., traits), we do not need a complete Rust specification. Therefore, we decided to base our choice of field-based taint analysis, and the specific rules for promoting references and adding lifetime constraints on Oxide [Weiss et al. [n.d.]], a simpler formalization that operates closer to Rust's source level and only involves Rust features observed in our input programs. Oxide handles explicit mutability and lifetime annotations with the aim of capturing *the essence of Rust*. Oxide is close to Rust's high-level IR (HIR), and does not model Rust's module or trait systems.

There have also been extensions of existing semantic modeling and verification tools to support Rust. Baranowski et al. [2018] extend the SMACK verifier to work on Rust programs, and KRust [Wang et al. 2018] is an implementation of Rust's semantics on the K-framework.

6 CONCLUSION

In this paper we have investigated the problem of automatically translating C programs into *safer* Rust programs—that is, Rust programs that improve on the safety guarantees of the original C programs. First, we conducted an in-depth study into the underlying causes of unsafety in translated programs and the relative impact of fixing each cause. We find that 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. We have ordered the categories by their impact to help determine their relative priorities.

Second, 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.

This paper presents the first empirical study of unsafety in *translated* Rust programs (as opposed to programs originally written in Rust) and also the first technique for automatically removing causes of unsafety in translated Rust programs. It lays the groundwork for future research into removing even more unsafety from these programs. That future research will address the other categories of unsafety outlined in this paper and ultimately extend the project to handle multi-threaded programs and C++.

REFERENCES

[n.d.]. Rust Compiler Error Index. https://doc.rust-lang.org/error-index.html
[n.d.]a. Rust support hits linux-next. https://lwn.net/Articles/849849/
[n.d.]b. Supporting Linux kernel development in Rust. https://lwn.net/Articles/829858/
2021a. NVD - CVE-2021-21148. https://nvd.nist.gov/vuln/detail/CVE-2021-21148

2021b. NVD - CVE-2021-3156. https://nvd.nist.gov/vuln/detail/CVE-2021-3156

Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Technical Report.

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

Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (Nov. 2020), 27 pages. https://doi.org/10.1145/3428204

Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. 2018. Verifying Rust programs with SMACK. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 528–535.

Sergio Benitez. 2016. Short Paper: Rusty Types for Solid Safety. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. Association for Computing Machinery, New York, NY, USA, 69–75. https://doi.org/10.1145/2993600.2993604

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

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

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

Maciej Cielecki, Jędrzej Fulara, Krzysztof Jakubczyk, and Łukasz Jancewicz. 2006. Propagation of JML non-null annotations in Java programs. In *Proceedings of the 4th international symposium on Principles and practice of programming in Java (PPPJ '06)*. Association for Computing Machinery, New York, NY, USA, 135–140. https://doi.org/10.1145/1168054.1168073

Citrus Developers. [n.d.]. Citrus / Citrus. https://gitlab.com/citrus-rs/citrus

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

Mehmet Emre and Ryan Schroeder. 2021. *Artifact for "Translating C to Safer Rust"*. https://doi.org/10.5281/zenodo.5442253

Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021a. *Supplementary Material on "Translating C to Safer Rust"*. https://cs.ucsb.edu/~benh/research/papers/oopsla21-supplementary.pdf

Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021b. *Translating C to Safer Rust – Extended Version*. https://cs.ucsb.edu/~benh/research/papers/oopsla21-extended.pdf

Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 246–257.

Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. 2001. Annotation inference for modular checkers. *Inform. Process. Lett.* 77, 2 (2001), 97–108. https://doi.org/10.1016/S0020-0190(00)00196-4

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, José Nuno Oliveira, and Pamela Zave (Eds.). Vol. 2021. Springer Berlin Heidelberg, Berlin, Heidelberg, 500–517. https://doi.org/10.1007/3-540-45251-6_29 Series Title: Lecture Notes in Computer Science.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation (PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 234–245. https://doi.org/10.1145/512529.512558

Tim Hutt. 2021. Would Rust secure cURL? https://timmmm.github.io/curl-vulnerabilities-rust/

Immunant inc. 2020a. c2rust Manual Examples. https://c2rust.com/manual/examples/index.html

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

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

S. Klabnik and C. Nichols. 2018. *The Rust Programming Language*. No Starch Press. https://doc.rust-lang.org/book/

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

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

21–26. https://doi.org/10.1145/2818302.2818306

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

Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. (2020), 763–779. https://doi.org/10.1145/3385412.3386036

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

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

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

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

The Rust developers. [n.d.]. The Rust Reference. https://doc.rust-lang.org/stable/reference/

Mohsen Vakilian, Amarin Phaosawasdi, Michael D. Ernst, and Ralph E. Johnson. 2015. Cascade: A Universal Programmer-Assisted Type Qualifier Inference Tool. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, Florence, Italy, 234–245. https://doi.org/10.1109/ICSE.2015.44

F. Wang, F. Song, M. Zhang, X. Zhu, and J. Zhang. 2018. KRust: A Formal Executable Semantics of Rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 44–51. https://doi.org/10.1109/TASE.2018.00014

Aaron Weiss, Daniel Patterson, Nicholas D Matsakis, and Amal Ahmed. [n.d.]. Oxide: The Essence of Rust. ([n. d.]), 27.