How Do Programmers Use Unsafe Rust?

VYTAUTAS ASTRAUSKAS, ETH Zurich, Switzerland
CHRISTOPH MATHEJA, ETH Zurich, Switzerland
FEDERICO POLI, ETH Zurich, Switzerland
PETER MÜLLER, ETH Zurich, Switzerland
ALEXANDER J. SUMMERS, University of British Columbia, Canada

Rust's ownership type system enforces a strict discipline on how memory locations are accessed and shared. This discipline allows the compiler to statically prevent memory errors, data races, inadvertent side effects through aliasing, and other errors that frequently occur in conventional imperative programs. However, the restrictions imposed by Rust's type system make it difficult or impossible to implement certain designs, such as data structures that require aliasing (e.g. doubly-linked lists and shared caches). To work around this limitation, Rust allows code blocks to be declared as *unsafe* and thereby exempted from certain restrictions of the type system, for instance, to manipulate C-style raw pointers. Ensuring the safety of unsafe code is the responsibility of the programmer. However, an important assumption of the Rust language, which we dub the *Rust hypothesis*, is that programmers use Rust by following three main principles: use unsafe code sparingly, make it easy to review, and hide it behind a safe abstraction such that client code can be written in safe Rust.

Understanding how Rust programmers use unsafe code and, in particular, whether the Rust hypothesis holds is essential for Rust developers and testers, language and library designers, as well as tool developers. This paper studies empirically how unsafe code is used in practice by analysing a large corpus of Rust projects to assess the validity of the Rust hypothesis and to classify the purpose of unsafe code. We identify queries that can be answered by automatically inspecting the program's source code, its intermediate representation MIR, as well as type information provided by the Rust compiler; we complement the results by manual code inspection. Our study supports the Rust hypothesis partially: While most unsafe code is simple and well-encapsulated, unsafe features are used extensively, especially for interoperability with other languages.

CCS Concepts: • **Software and its engineering** → *Software libraries and repositories*; **General programming languages**; **Software organization and properties**; • **General and reference** → **Empirical studies**.

Additional Key Words and Phrases: Rust, unsafe code, empirical study, Rust hypothesis

ACM Reference Format:
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 (November 2020), 27 pages. https://doi.org/10.1145/3428204

Authors' addresses: Vytautas Astrauskas, Department of Computer Science, ETH Zurich, Switzerland, vytautas.astrauskas@inf.ethz.ch; Christoph Matheja, Department of Computer Science, ETH Zurich, Switzerland, christoph.matheja@inf.ethz.ch; Federico Poli, Department of Computer Science, ETH Zurich, Switzerland, federico.poli@inf.ethz.ch; Peter Müller, Department of Computer Science, ETH Zurich, Switzerland, peter.mueller@inf.ethz.ch; Alexander J. Summers, Department of Computer Science, University of British Columbia, Canada, alex.summers@ubc.ca.

2475-1421/2020/11-ART136
https://doi.org/10.1145/3428204

(a) Example of Unsafe Blocks

```
let x = 17;
let r = &x; // borrow x
// cast reference r to raw pointer
let p = r as *const i32;
unsafe { assert!(*p == 17); }
```

(b) Example of Unsafe Functions

```
// only safe to call with x == 17
unsafe fn foo(x : i32) { ... }
fn bar() { // safe abstraction
    unsafe{ foo(17); } // safe for 17
}
```

Fig. 1. Examples of unsafe Rust.

1 INTRODUCTION

Rust is a systems programming language whose type system prevents many common errors at compile time. Among other properties, a well-typed Rust program is guaranteed not to exhibit null-pointer dereferences, reading from uninitialised memory, dangling pointers, data races, memory leaks, and inadvertent side effects through aliasing. These strong guarantees are achieved via an ownership type system that governs the *capabilities* to read and modify memory locations.

In Rust, every memory location is owned by precisely one variable; when the owning variable goes out of scope, the memory is freed. The exclusive capability to access this location starts with its owner but can be transferred permanently (along with ownership, via move assignments) or temporarily (via borrowing). For instance, a common idiom is for functions to borrow their arguments from the caller and restore capabilities when the call terminates. Besides exclusive (mutable) borrows, Rust also permits sharing of memory locations via (borrowed) shared references. To prevent data races and inadvertent side effects, the memory locations reachable by a shared reference are enforced to be immutable so long as the shared reference exists. In order to determine when (mutable or shared) borrows go out of scope and the capabilities get restored, Rust associates each borrowed reference with a lifetime and tracks constraints between them.

The rules for ownership and borrowing ensure that, at each point in an execution, a memory location is either accessed exclusively by one function execution or shared immutably. This discipline allows the compiler to eliminate many otherwise-prevalent memory errors. However, it comes at a cost: For example, Rust's ownership system leads to tree-shaped linked data structures. Other mutable data structures, such as doubly-linked lists and graphs, cannot be represented without going outside this strict discipline. Moreover, implementations may suffer from sub-optimal data representations chosen solely to comply with the Rust compiler. For example, reference-counted smart pointers are more flexible but do not follow Rust's ownership system.

To work around these limitations, Rust provides an escape hatch: *Unsafe Rust* supports writing code that need not be subject to all of Rust's default rules. By leveraging unsafe Rust, it is possible to implement, for example, cyclic data structures, hardware abstraction layers, and lock-free algorithms – features that are difficult or even impossible to realise in purely safe Rust. However, this added expressiveness comes at a price. The compiler cannot enforce the above guarantees; this enforcement becomes the responsibility of the developer, entailing significant cognitive effort even for small pieces of unsafe Rust. Examples of the subtleties involved when taking this responsibility are found, for instance, in the discussions of Rust's Unsafe Code Guidelines Working Group [2020]. Importantly, as noted by Jung [2016], the correctness of unsafe code may rely on invariants that could be invalidated by *all* functions modifying the same struct fields; a thorough code review of whether a block of unsafe code is acceptable is not therefore limited to the code within the block itself, but has to include at least all code which could modify these fields.

Rust provides two primary forms of unsafe code, with different purposes. The first form allows programmers to bypass compiler checks. Its core feature is an *unsafe block* defining the scope in which these checks are disabled. In Fig. 1a, for instance, an unsafe block is required to dereference the C-style raw pointer p. By default, it is assumed that such an unsafe block contained within, say, a struct method should use unsafe features in a way which is *encapsulated* from callers of the function; it is the responsibility of the function and not the client code that this code will always execute safely. Alternatively, one can explicitly declare an *unsafe function* (a function annotated with the unsafe keyword). This feature is intended to indicate that the responsibility for the correctness of the unsafe code in the function body lies at least partially with its *callers*[1]. Fig. 1b shows the syntax for such an unsafe function foo, which can be called only from within unsafe blocks. As with any unsafe block, the call within the body of bar combined with the fact that bar is *not* an unsafe function indicates that the implementer of bar intends that this usage of unsafe Rust is safely encapsulated from bar's callers: the developer promises that foo(17) preserves Rust's safety guarantees even though the compiler cannot enforce them.

The second main form of unsafe Rust involves Rust *traits*, which are comparable to Java interfaces. Instead of disabling compiler checks, an *unsafe trait declaration* acts as a documentation feature: it warns developers that all implementations of the trait are expected to satisfy some additional semantic properties such as preconditions, postconditions, or invariants that are not checked, neither at compile nor at run time; furthermore, these properties may be depended upon for the safety of client code *using* these traits. For *trait implementations*, unsafe takes the role of an annotation by which developers acknowledge their responsibility to respect the required semantic properties. We consider usages of unsafe in the above sense as a documentation feature since the compiler does not check the aforementioned properties. However, using unsafe in these cases is *not* optional. Adherence to all documented properties is crucial for upholding Rust's safety guarantees: clients calling an unsafe trait's functions are allowed to rely on these properties for arguing the safety of their own code. Conversely, for a trait not declared as unsafe, all of its clients must ensure safety for *every possible safe implementation* of that trait—regardless of how much it deviates from its originally intended purpose.

Unsafe code must be used with care to retain Rust's strong guarantees. The commonly-advocated practice is that programmers should, as far as possible, use unsafe Rust according to the following three basic principles, which aim to limit the necessary scope of code reviews (cf. Rust Team [2019b], Klabnik and Nichols [2019, Ch. 19], and prominent sources from the Rust community, e.g. Fuchsia Team [2020]; Jung et al. [2020]; Matsakis [2016]; Rust Team [2019a]):

(1) Unsafe code should be used *sparingly*, in order to benefit from the guarantees inherently provided by safe Rust to the greatest extent possible.
(2) Unsafe code blocks should be *straightforward* and *self-contained* to minimise the amount of code that developers have to vouch for, e.g. through manual reviews.
(3) Unsafe code should be *well-encapsulated* behind *safe abstractions*, for example, by providing libraries that do not expose the usage of unsafe Rust (via public unsafe functions) to clients.

Ideally, these principles are implemented by encapsulating unsafe code inside carefully reviewed and tested libraries whose clients can be written in safe Rust and need not be aware of the presence of unsafe code. Parts of the Rust language documentation [Klabnik and Nichols 2019; Rust Team 2019b] claim that programmers can use unsafe code according to these three basic principles – a claim that we refer to as the *Rust hypothesis*.

Understanding how Rust programmers use unsafe code and, in particular, whether the Rust hypothesis holds is essential for users of the Rust language. It allows project managers to judge

[1] In addition, the entire body of an unsafe function is treated implicitly as if it were enclosed in an unsafe block.

to what extent they can rely on Rust's promise to eliminate certain errors, developers to follow (evolving) best practices, testers to determine which properties to check for which parts of the codebase, library designers to identify further idioms of unsafe code that could be safely encapsulated, language designers to devise safe solutions for commonly-used unsafe idioms, and tool builders to support common idioms of unsafe code and their interaction with safe code.

Our work. This paper studies empirically how unsafe code is used in practice. It goes significantly beyond existing studies by analysing a large corpus of Rust projects to assess the validity of the Rust hypothesis and to classify the purpose of unsafe code. To answer these questions, we identify queries that can be answered by *automatically* inspecting the program's source code, its intermediate representation MIR, as well as type information provided by the Rust compiler. For instance, to assess how often unsafe code is used to implement custom concurrency primitives, we collect information about concurrency-related compiler intrinsics such as calls to compare-and-swap. To obtain a deeper understanding of the semantics and intent of unsafe code, we complement this automatically-collected data by manual code inspection.

Our results support the Rust hypothesis partially. Most unsafe code is simple and well-encapsulated behind safe abstractions. However, unsafe code is used quite extensively, especially to interoperate with other programming languages. Interoperability is by far the most prevalent motivation for using unsafe code, followed by implementations of data structures requiring complex sharing (via raw pointers or mutable global data). Other purposes, such as using unsafe concurrency features and *applying unsafe to document semantic properties that are critical for upholding Rust's safety guarantees (the second form of unsafe code mentioned above) are less common.*

Contributions. Our paper makes the following contributions:

- A classification of the motivations for using unsafe code. We identify six main purposes for unsafe code. This classification serves as a basis for our empirical study but is also useful for the systematic documentation of unsafe code and tailoring techniques such as test case generation and program analysis towards specific use cases of unsafe code.
- An empirical study of how unsafe code is used in practice. Our study shows that code that is not concerned with interoperability typically adheres to the Rust hypothesis.
- A discussion of the implications for reasoning about Rust code (in code reviews or during verification).
- Our reusable open-source infrastructure and the analysed data is available online [Qrates Team 2020].

Outline. Sec. 2 classifies the main usages of unsafe code into six different categories. Sec. 3 summarises the methodology of our empirical study and states our core research questions. Sec. 4 presents our general framework for analysing Rust code. The results of our study are presented in Sec. 5 and discussed in detail in Sec. 6. We discuss threats to validity in Sec. 7, summarise related work in Sec. 8, and conclude in Sec. 9.

2 USAGES OF UNSAFE CODE

Rust programs should predominantly use safe code to benefit from Rust's safety guarantees. Nonetheless, there are cases in which unsafe code is either necessary or at least the preferred solution. In this section, we discuss six such use cases. Understanding them – and their prevalence in real-world code – is a central goal of our study.

2.1 Overcoming Aliasing Restrictions

Rust's type system enforces a strict control on aliasing to ensure that programs never exhibit memory errors. It distinguishes between *mutable* references, which must be unique for every memory address, and *shared*, but read-only, references. While the type system is permissive enough for most tasks, there exist important scenarios for which the aliasing restrictions of safe Rust are too restrictive. Our first group of use cases consists of three such scenarios.

2.1.1 Data Structures with Complex Sharing. In Rust, all linked data structures based on mutable references are inherently tree-shaped because the type system prevents any two mutable references from pointing to the same location. Even for shared references, the compiler does not permit the construction of cyclic reference-structures as this could lead to dangling references during deallocation[2]. Implementing topologies other than trees, such as doubly-linked lists, trees with parent-pointers, DAGs, or multigraphs, thus requires developers to bypass the type system, e.g. by using raw pointers, whose dereference is allowed only in unsafe Rust [Cameron et al. 2019, Ch. 11].

Other data structures predominantly rely on shared references but permit specific mutations through otherwise immutable references. For example, both smart pointers based on reference counting and some caching mechanisms rely on the ability to change values through one of the multiple shared references [Rust Team 2020c; The Libra Association 2020].

Rust offers an exception to the immutability requirement of shared references: the content of an `UnsafeCell` can be mutated via a shared reference. However, this *interior mutability* requires unsafe code since `UnsafeCell` essentially exposes a raw pointer to its content. The standard library defines various convenient wrappers of `UnsafeCell`, such as `RefCell` and `Mutex`, that – as proven by Jung et al. [2018] – provide a safe abstraction.

Apart from using raw pointers or relying on existing abstractions provided by the standard library, a third option for implementing complex data structures is to use integers representing indices into a vector instead of references. In other words, one can side-step Rust's default aliasing results by implementing a custom vector-backed heap. This approach allows some form of aliasing – two vector entries can store the same index – while staying within the boundaries of safe Rust. However, it also eliminates many of Rust's strong guarantees for references. The compiler cannot, for example, detect that an index is invalid because it is outside of the underlying vector's bounds; such a program would instead panic at run time.

Note that there exist approaches in the general literature on ownership type systems that can deal with certain forms of mutable aliasing and support the implementation of various standard data structures including cyclic lists (e.g. [Clarke et al. 1998; Clebsch et al. 2015; Gordon 2014; Müller 2002; Potanin et al. 2013]). Incorporating some ideas from these alternative systems into future versions of Rust could potentially provide another way to safely implement complex data structures (although making them compatible with Rust's existing solution for automated memory management seems very challenging).

2.1.2 Incompleteness Issues. Even though Rust's type system is constantly improving, it is necessarily incomplete; some valid programs will be rejected by the compiler. For example, consider the program below which splits a slice (a contiguous sequence of elements in a collection) into two at some specified index `mid`:

```rust
fn split_at_mut<T>(s: &mut [T], mid: usize) -> (&mut [T], &mut [T]) {
    (&mut s[..mid], &mut s[mid..]) // Error: `s` is mutably borrowed twice
}
```

[2] Deallocation is handled automatically by Rust; memory is freed when its owner goes out of scope, including transitively-owned memory.

This program is rejected because the compiler does not distinguish different slice elements; it thus fails to recognise that the capabilities assigned to the returned references do not overlap. To implement this function nonetheless, the standard library reverts to unsafe code: It splits the slice using a raw pointer.

2.2 Emphasise Contracts and Invariants

As noted in Sec. 1, unsafe comes in two different forms: The use cases discussed in the previous subsection are of the first form; they use unsafe code to bypass the restrictions of safe Rust. Here we focus on the second form, where unsafe serves as a documentation feature: Developers may attach unsafe to both functions and traits to indicate that their implementation relies on some contract or invariant that cannot be established by the compiler. The compiler then enforces that unsafe functions are called only from within unsafe blocks (or other unsafe functions) and that implementations of unsafe traits are also marked as unsafe. In other words, by using unsafe, developers acknowledge their responsibility to account for all documented requirements.

A common idiom found in the standard library is to employ unsafe for stressing unchecked preconditions of functions, which need to be established by their callers. For example, consider the function signature below, which is found in the Rust module `std::slice` [Rust Team 2020d].

```
pub unsafe fn from_raw_parts<'a, T>(data: *const T, len: usize) -> &'a [T]
```

This function takes a pointer `data` and an integer `len` as parameters. Its documentation requires that `data` refers to a slice of length at least `len` such that this number of contiguous `T` values in memory can be safely read. The compiler encourages the above idiom by suggesting that programmers “consult the documentation” whenever it detects that an unsafe function is illegally called outside of an unsafe block.

Similarly, the standard library uses unsafe *traits* to highlight other contracts such as postconditions. For instance, the functions defined by the trait `GlobalAlloc` are required (in all implementations of the trait) to never panic. Another interesting example is the unsafe trait `TrustedLen`, whose documentation requires the result of function `size_hint` to return the precise number of elements yet to be returned by the currently-active iterator rather than an approximation.

The correct usage of unsafe for documenting invariants – or even purely logical requirements as in the case of `TrustedLen` – remains controversial: The Rustonomicon notes that there are sensible cases for declaring a trait as unsafe, but also remarks that this has been traditionally avoided [Rust Team 2019b, Ch. 1.1]. One aim of our study is to provide a better understanding of whether and how developers employ unsafe functions and traits for documentation purposes – an aspect that will be addressed alongside other use cases of unsafe Rust in Sec. 3.2.

2.3 Accessing a Lower Abstraction Layer

Rust is designed as an efficient systems programming language. As such, it needs to support interacting with environments operating at a lower level of abstraction, e.g. hardware, operating systems, device drivers, and libraries written in other languages. While Rust goes a long way towards providing safe high-level abstractions, many of these environments are ultimately outside of the compiler’s control; they are necessarily unsafe. To cover these situations, the compiler exposes unsafe functions that give access to – typically faster or more expressive – low-level operations.

2.3.1 Foreign Functions. Rust provides the `extern` keyword to define (unsafe) bindings to functions that are written in foreign languages – typically C – and supplied via shared libraries. For example, the snippet below shows how to bind and subsequently call the C function Z3_mk_solver from the Z3 library. This function uses a Z3_context object, which is not thread-safe. By calling the function from an unsafe block, the developer vouches that they correctly prevented any possible concurrent usage of the Z3_context object.

```
#[link(name = "z3")]
extern "C" {    pub fn Z3_mk_solver(c: Z3_context) -> Z3_solver;    }
fn main() {
    /* ... */ let solver = unsafe { Z3_mk_solver(context) }; /* ... */
}
```

Rust also supports embedding inline assembly via the asm! macro in unsafe blocks; as for any other unsafe block, it is the programmer's responsibility to ensure that the behaviour of inlined-assembly does not violate Rust's guarantees.

2.3.2 Concurrency through Compiler Intrinsics. While Rust offers safe primitives for shared-memory communication between threads in its atomic types module [Rust Team 2020e], the Rust compiler also offers concurrency intrinsic functions that mirror the low-level intrinsics of LLVM [LLVM Team 2020]. All of these intrinsics are marked as unsafe because their incorrect usage may lead to memory errors or data races. For instance, the function atomic_xadd performs an atomic addition; the result is stored at a destination specified by a raw pointer that is passed as a parameter. If that pointer is not aligned, type safety is not guaranteed. Low-level concurrency intrinsics are needed by projects that need to implement their own concurrency primitives. The following snippet from the Redox OS project [Redox developers 2019], for example, relies on the intrinsic atomic_xadd to increment a semaphore counter:

```
pub unsafe extern "C"
fn pte_osSemaphorePost(handle: pte_osSemaphoreHandle, count: c_int)
 -> pte_osResult {
    let semaphore = &mut *handle;
    let _guard = semaphore.lock.lock();
    intrinsics::atomic_xadd(&mut semaphore.count, 1);
    PTE_OS_OK
}
```

2.3.3 Performance. For high-performance applications, developers may occasionally wish to bypass Rust's built-in safety checks in order to gain a speed-up. Prime examples include avoiding the bound checks performed when accessing array elements and providing hints to the optimiser, e.g. that code is unreachable without verifying that this is actually the case[3]. Furthermore, it is possible to modify the custom memory layouts of data structures to speed up type conversions. The following example due to Gjengset [2020] applies such an optimisation to quickly cast an array of bytes (represented with Rust's built-in slice type) to a Rust structure without performing any safety checks:

```
#[repr(C)]
struct SerializedStruct { /* ... */ }

unsafe fn cast_deserialize(i: &[u8]) -> &SerializedStruct {
    &*(i.as_ptr() as *const SerializedStruct)
```

[3] https://doc.rust-lang.org/std/hint/fn.unreachable_unchecked.html

```
}
```

The use cases presented throughout this section are based on anecdotal evidence and examples collected from various libraries. In the remainder of this paper, we study systematically how unsafe code is used in real-world Rust code.

3 METHODOLOGY

Recall that the main goal of our study is to gain insights into how unsafe Rust is used in practice. To this end, we aim to answer the following high-level questions:

- Does the Rust hypothesis hold?
- What are the most prevalent use cases programmers have for using unsafe code?

In this section, we first refine the above questions into five research questions (RQs) that guide our search for a better understanding of unsafe Rust. After that, we break each of these questions down into *more-specific queries*[4] which allow us to infer answers for the original questions. We aim to answer each specific query fully automatically; the details of our approach are presented in Sec. 4. The results gathered by these queries are discussed in Sec. 5.

3.1 The Rust Hypothesis – Do Developers Use Unsafe Rust as Intended?

As introduced in Sec. 1, there are three widely-advocated basic principles for using unsafe Rust:

(1) Unsafe code should be used *sparingly*.
(2) Unsafe code should be *straightforward* and *self-contained*.
(3) Unsafe code should be *well-encapsulated* behind *safe abstractions*.

The Rust hypothesis is that developers typically can and do follow the above principles. To check whether general Rust code in the wild supports this hypothesis, we investigate each principle through dedicated research questions. We first explore how widespread unsafe code is:

RQ 1 (Frequency). *How often does unsafe code appear explicitly in Rust crates?*

The first principle of the Rust hypothesis predicts that unsafe code will rarely appear in our dataset compared to its overall size. To verify this claim, we take a two-pronged approach: First, we identify every usage of unsafe Rust in our dataset and specifically count how many crates (i.e., binaries or libraries) include *any* unsafe code. That is, we count *how many crates contain at least one unsafe block, function, trait definition, or implementation*; all other crates contain only safe Rust. Even small pieces of unsafe code require a significant cognitive effort by developers: they need to be aware of their responsibility to guarantee safety rather than relying on the Rust compiler. Determining what fraction of crates is completely safe allows us to measure how frequently developers aim to avoid this burden by sticking to safe Rust, as we would expect by the first principle.

Second, evaluating frequency solely based on whether a crate contains any unsafe code may lead to a coarse impression. To compensate, we complement this data by measuring the *relative* amount of unsafe code in each crate, i.e. *the ratio of the size of both unsafe blocks and unsafe function bodies to the total size of the crate.* We discuss how we specifically measure the size of code alongside our second research question below.

By the second principle, unsafe code should be *straightforward* – an admittedly subjective notion. To evaluate whether developers prefer to keep their unsafe blocks simple, we measure the *size* of each unsafe block as a proxy for its complexity (where smaller size means lower complexity):

RQ 2 (Size). *What is the size of unsafe blocks that programmers write?*

[4] We highlight all specific queries in *green italic text*.

Attacking this question requires a reasonable means of quantifying the size of an unsafe block. An obvious candidate is to count the lines of Rust code in each unsafe block. However, lines of code are a weak indicator for a block's complexity because some features, such as closures and macros, might realise quite complex behaviour with a few lines of code. Moreover, different indentation and whitespace schemes would inadvertently bias such measurements.

To obtain a measure that is more robust against programming styles of different verbosity, we turn to the Rust compiler's intermediate CFG representation (MIR) in which many (potentially complex) high-level Rust constructs have already been translated to MIR instructions. Our next query thus checks *how many MIR statements does the compiler generate for unsafe blocks.* We also use this query to determine the total amount of unsafe code in a crate, to complement the binary query above.

RQ 3 (Self-containedness). *Is the behaviour of unsafe code dependent only on code in its own crate?*

Unsafe code that is compliant with the second principle of the Rust hypothesis should rarely reach out to other crates in order to keep manual reviews of unsafe code as simple as possible. In particular, unsafe code which relies on the functional behaviour of code from other crates may become vulnerable due to updates to its compilation dependencies.

A naïve query to evaluate this principle would count how many function calls in unsafe blocks have a call target outside the current crate – a low number then indicates a high degree of self-containedness. However, not all call targets are equal: The standard library crates, i.e., `std`, `core`, `alloc`, and `proc_macro`, are used heavily in a wide variety of projects. Since they are thoroughly reviewed, relying on the behaviour of these libraries is – while still technically in violation of self-containedness – arguably less problematic. Moreover, some crates are intended to provide low-level access to libraries written in other languages. For example, so-called "`-sys`" crates (whose name, by convention, ends with `-sys`) mirror the interface of C libraries [Rust Team 2019c]. This can be seen as a separation of concerns since multiple safe abstractions of the same C library may be sensible: the `-sys` gives unfettered access and other crates can implement safe abstractions on top of it. Naturally, such a design leads to dependencies between crates that are justified but in direct conflict to self-containedness.

We consider call targets in the above two categories separately as they amount to *expected and intentional* violations of self-containedness. Hence, we use a refined query that counts *how many function calls in unsafe blocks have a call target which is located in (1) its own crate, (2) a crate belonging to the standard library, (3) a `-sys` crate, or (4) any other crate*. Calls in category (4) point to likely *unintended* violations of the second principle.

Notice that the above query requires detailed knowledge about the targets of function calls. We evaluate it for *standard function calls* whose call targets can always be determined at compile time from the call expression alone. Since unsafe code should be as simple as possible to facilitate manual reviews, we expect unsafe blocks to contain only a few other function calls involving trait methods, closures, or function pointers, which require more manual effort from code reviewers as they have to trace down all possible implementations. To validate our expectation and as another proxy for simplicity, we measure *how many function calls in unsafe blocks and unsafe functions are (a) standard function calls, (b) calls of trait methods, or (c) calls of closures or function pointers*.

RQ 4 (Encapsulation). *Is unsafe code typically shielded from clients through safe abstractions?*

The third principle of the Rust hypothesis requires programmers to shield unsafe code from clients through safe abstractions. In other words, clients should be oblivious to the fact that unsafe code is used internally within a crate. Checking whether developers *succeed* in constructing suitable abstractions amounts to a difficult – if not impossible – task for automatic analyses, for instance, because they would have to check whether executions may exhibit data races. Therefore, we focus on the apparent *design intentions* of developers. That is, are they trying to hide their unsafe functions from other crates as much as possible?

To answer this question, we take a closer look at Rust's concept of visibility. Broadly speaking, there are three main notions: The default is private, meaning only visible within the current module – a user-defined collection of Rust items, such as functions, traits, etc. Alternatively, an item can be visible within either only the current crate or all crates.

We then count *how many unsafe functions are (1) declared private, (2) visible within their crate, and (3) visible to other crates*. Queries (1) and (2) cover all of the unsafe functions that comply with the third principle. Query (3) collects uncompliant cases, i.e., unsafe functions that are exposed to other crates.

3.2 Measuring the Unsafe World – How do Developers Use Unsafe Code?

We now take a closer look at possible reasons for developers to rely on unsafe code. Recall from Sec. 2 our classification of use cases for writing unsafe code ranging from overcoming aliasing restrictions over emphasising contracts and invariants to accessing lower abstraction layers. Our goal is to identify code where these use cases are applied to answer the following:

RQ 5 (Motivation). *What are the most prevalent use cases for unsafe code?*

In the following paragraphs, we present specific queries for identifying individual use cases. All of these queries search for syntactic patterns that are characteristic for the use case in question. As such, they collect evidence for particular use cases rather than precisely capturing them. We intentionally do not attempt to find perfect characterisations because the results of our queries should be gathered *automatically*, possibly with manual follow-up efforts. As is common for most automated program analyses, we thus rely on approximations.

Data structures with Complex Sharing. Since Rust's ownership rules prevent complex sharing, some data structures are notoriously difficult – or even impossible – to implement in safe Rust. Although a few patterns, such as the interior mutability pattern, have evolved in the Rust community to deal with limitations of the ownership system, they all ultimately rely on using raw pointers[5]. Hence, we consider raw pointer dereferences as an indicator of a programmer's intention to bypass the ownership system to allow for complex sharing. To identify this use case, we thus collect *all functions that contain a dereference of a raw pointer*. We explore pointer dereferences at the function level rather than, e.g. studying individual unsafe blocks, to obtain unified results for both safe functions (which need to use unsafe blocks) and unsafe functions (which do not). Moreover, some developers advocate using many minimal unsafe blocks whereas others prefer fewer and larger ones. By phrasing our query at the function level, we keep it agnostic to these different styles.

The above query risks overcounting how frequently developers rely on unsafe code to implement complex data structures because raw pointers are, for example, also used to interoperate with C libraries. To obtain a more conservative estimate, we filter out usages of raw pointers for which we can identify different intentions: we do not count raw pointers appearing in structs that are equipped with attributes, such as `#[repr(C)]`, indicating that they are used for interoperability.

Incompleteness Issues. It is not generally possible to precisely identify all cases in which developers work around incompleteness issues of Rust's type and ownership system[6]. Therefore, we focus on unsafe functions for which the Rust documentation lists overcoming limitations of the compiler as

[5] Another common pattern is to use a vector-backed custom heap and implement the data structure using integer indices instead of pointers. However, this approach stays within safe Rust and is, consequently, outside the scope of our study.
[6] If it were, the compiler could use the same analysis to prevent the incompleteness in the first place!

a use case: we *collect all calls of unsafe functions involving explicit type casts*. For instance, both the "incredibly unsafe" function `transmute` and its close relative `transmute_copy` reinterpret the bits of a value as another type and are suggested by the Rust documentation to work around limitations of lifetimes, e.g. extending a lifetime or shortening an invariant lifetime [Rust Team 2020b].

Emphasise Contracts and Invariants. Recall that the `unsafe` keyword in Rust may also serve as a documentation feature when attached to functions or traits. To understand whether developers have used `unsafe` to document contracts and invariants, we run two queries: First, we *search for unsafe functions whose body contains only safe Rust code*. Since there is no technical reason why these functions need to be declared as `unsafe`, we expect that any such functions are declared unsafe either accidentally, or in order to document some implicit contract or invariant that is critical for upholding safety (for example, of unsafe code in the same module). Second, we *count the number of both safe and unsafe traits declared*. For traits, `unsafe` is always a documentation feature. Based on the low number of unsafe traits in the Rust standard library – there are only 11 in total – we expect to find only a few unsafe trait declarations.

Concurrency through Compiler Intrinsics. The Rust compiler provides access to low-level concurrency intrinsics through dedicated unsafe functions. To measure how frequently developers rely on them, we collect a list of unsafe functions wrapping concurrency intrinsics in the `std::intrinsics` module. We then run a query *collecting all unsafe blocks that call one of the collected functions.*

Foreign Functions. Interoperability with other languages, e.g. accessing C/C++ libraries, is mentioned by Qin et al. [2020] as a frequent use case for writing unsafe code. To identify code that is likely to interoperate with foreign code, we exploit the following observations in our queries:

- The attribute `#[repr(C)]` ensures that the memory layout of a type is interoperable with the C programming language; unsafe code that relies on such types is thus likely to exchange data with foreign code. Hence, we count *how many types are equipped with `#[repr(C)]`*.
- The name of crates that wrap C system libraries ends – by convention – with the suffix `-sys`. We thus classify *all `-sys` crates* as belonging to this use case.
- Rust allows functions to be declared `extern` with a custom Application Binary Interface (ABI) such that foreign code with the specified ABI can call them. Consequently, we determine *how many unsafe functions are declared with a foreign ABI*.
- Finally, we search for *usages of inline assembly* via the `asm!` macro. A closer analysis of the motivation for using inline assembly, e.g. low-level optimisations or interacting with hardware devices, is subject to manual inspection.

Performance. We consider two standard optimisations that use unsafe code to boost performance: First, we *search for unchecked functions* (those with "unchecked" in their name). By convention, these functions sacrifice run-time checks for better performance; they are consequently unsafe. The standard library, for example, adheres to this naming convention and frequently provides both a safe (checked) and an unsafe (unchecked) variant of the same functionality.

Second, we determine whether *unsafe blocks contain the special union type `MaybeUninit`*. The Rust documentation describes this type as a highly unsafe variant of optional types that avoids any safety checks at run time (cf. [Rust Team 2018]). In fact, it may reintroduce dangling references as developers may use it to define uninitialised references in unsafe blocks.

4 A FRAMEWORK FOR QUERYING RUST CRATES

To automatically evaluate the queries presented in the previous section, we developed a framework – called Qrates– for *Q*uerying *R*ust Cr*ates* for a large dataset of publicly available Rust code. It is inspired by both an idea proposed by Matsakis [2017] (one of the project leaders of the Rust compiler team) for adding Datalog output to the Rust compiler and commercial tools such as Code QL [2020] for analysing other programming languages. In this section, we briefly outline the main components of Qrates.

Intuitively, our framework works as follows: We first create a database of Rust crates enriched with metadata, e.g. the crates' origin and whether it compiles to a binary or a library. After that, we run a set of queries on the database to extract statistical data that enables answering the questions from the previous section. Answering a research question then amounts to combining the data gathered by one or more queries.

To construct the database, we implemented a plugin for the Rust compiler that extracts information such as the program CFG as well as type information for a given Rust crate during compilation and stores it in a local database. Extracting data through a compiler plugin has immediate benefits: It integrates well into Rust's existing build infrastructure, including its widely-used package manager `cargo`. Moreover, we gain access to the analysed code in various intermediate formats, such as the CFG representation (MIR) in which all types and static function calls are fully resolved. Another consequence is that we consider only crates that compile successfully.

For the compilation, Qrates is based on the Rustwide library, which was mainly developed by Albini [2020] from the Rust infrastructure team. Its original purpose is to run ecosystem-wide tests of the Rust compiler. Rustwide provides a Docker image with the necessary dependencies needed to compile most publicly available crates.

After creating local databases for all crates of interest, Qrates merges them into one comprehensive database with additional cross-link information. Once the final database has been constructed, the user can query it to gather statistics. Prime examples of supported queries include – but are not limited to – all *highlighted queries* from Sec. 3. In particular, Qrates can count how often a specific Rust feature, say a call to a function of interest, appears in (unsafe) blocks, functions, entire crates, or across all crates in the database.

It is noteworthy that, since crates may specify fixed versions for their dependencies, our database may contain different versions of the same crate. To prevent double-counting, our queries report results only for a single version of each crate (for this study, we chose the latest version). We still keep all crate versions in the database, because some queries are concerned with dependencies. For example, when analysing the call targets of a function, some of them may be defined in older versions of a dependency.

Furthermore, our database keeps precise information about the origin of the analysed code in order to avoid counting the same code, e.g. statically linked libraries, twice.

We note that all queries concerning functions are based on Rust's intermediate CFG representation (MIR). On the one hand, this is a mild advantage because *unreachable* code has already been eliminated. On the other hand, this means the compiler has already performed various transformations, such as macro expansion, desugaring pattern-match constructs, and generating code for `#[derive(...)]` attributes. Consequently, some care is needed to check whether a piece of code can be attributed to the programmer, e.g. by considering the unsafe block check mode provided by the compiler; otherwise, we risk overcounting features that are predominantly introduced by the Rust compiler.

5 EMPIRICAL RESULTS

In this section, we present the results automatically gathered by our queries, complemented by some manual inspections, and provide answers to the research questions from Sec. 3. We first discuss our data sets, then the experimental setup, and finally the results for each research question. We provide a detailed discussion of our findings in Sec. 6.

Unsafe Feature	#crates	%
None	24,360	76.4
Some	7,507	23.6
Blocks	6,414	20.1
Function Declarations	4,287	13.5
Trait Implementations	1,591	5.0
Trait Declarations	280	0.9

Table 1. Rust crates with and without any unsafe code grouped by feature. A crate may contain multiple unsafe features.

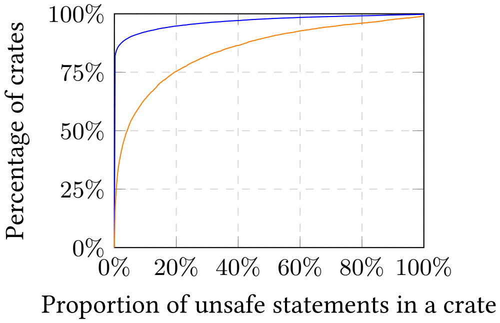

Fig. 2. The cumulative proportion of statements in unsafe blocks and functions in all crates (blue) and in crates that have at least one such statement (orange).

5.1 Datasets and Experimental Setup

We evaluated our queries on a dataset that comprises the most-recent version (as of 2020-01-14) of all 34445 *packages* published on central Rust repository crates.io. The implementation of a package can be composed of multiple *crates*, one of which is usually primary and determines the name of the package. We excluded 5,459 packages (15.8%) whose most recent version did not successfully compile. For packages with conditional compilation features, we used the default flags specified in the manifest. In cases where a package failed to compile with the default flags, but succeeded with different ones (when compiled as a dependency of another package) we selected a random build for analysis. As a result, our dataset consists of 31867 crates. Most of these crates are compiled to Rust libraries (76.0%), or binaries (20.0%). The remaining crates are procedural macros (4.0%).

Our experiments were conducted on a computer equipped with an Intel Xeon E5-4627 processor (3.30GHz, 16 cores), 252 GB of RAM, running Ubuntu 16.04.6 as an operating system and version `nightly-2020-02-03` of the Rust compiler. Since our experiments do not depend on timings or performance, it should be straightforward to reproduce our results on different hardware. We collected all of our results in Jupyter notebooks for follow-up analyses using Python [Jupyter Team 2020]; these are also available online [Qrates Team 2020].

5.2 The Rust Hypothesis – Do Developers Use Unsafe Rust as Intended?

We first answer the research questions related to the Rust hypothesis, i.e., the claim that developers typically use unsafe Rust (1) sparingly and in a way such that its behaviour is (2) both straightforward and self-contained, and (3) well-encapsulated behind safe abstractions.

5.2.1 RQ 1 (Frequency): How often does unsafe code appear explicitly in Rust crates? Table 1 shows in both absolute and relative numbers how many crates contain unsafe code, and which unsafe features they use (our first query), while Fig. 2 shows the relative amount of statements in unsafe blocks and functions in (1) all crates and, for readability, (2) crates that contain at least one unsafe statement (our second query)[7]. The majority of crates (76.4%) contain no unsafe features at all. Even in most crates that do contain unsafe blocks or functions, only a small fraction of the code is unsafe: for 92.3% of all crates, the unsafe statement ratio is at most 10%, i.e., up to 10% of the codebase consists of unsafe blocks and unsafe functions. However, with 21.3% of crates containing

[7] Notice that Fig. 2 does not account for unsafe traits. Consequently, the percentage of crates without any unsafe blocks or functions (78.7%) is slightly larger than the percentage of entirely safe crates (76.4%).

some unsafe statements and – out of those crates – 24.6% having an unsafe statement ratio of at least 20%, we cannot claim that developers use unsafe Rust sparingly, i.e., *they do not always follow the first principle* of the Rust hypothesis.

Nevertheless, if we compare our results with the ones from Evans et al. [2020], we can see that they report *higher* percentages of unsafe crates across *all* features in their experiments. Their experiments are based on a 16 months *older* snapshot (from September 2018) of the central Rust repository crates.io. In the meantime, more than 10,000 crates have been added to the repository and, in particular, the percentage of unsafe crates dropped from 29% to 23.6%. This finding differs from Evans et al. [2020], who observed that the amount of unsafe code in the most downloaded crates slightly *increased* over 10 months. One possible explanation for these observations is that the most downloaded crates provide the necessary extensions to the language or standard library (for example, an efficient random number generator) that cannot be implemented in safe code and, therefore, the amount of unsafe in the most popular crates does not change while a significant portion of the newly added crates are application code that does not need to use unsafe. Another possible explanation of these observations is that they may reflect concerted efforts within the Rust community to reduce the overall usage of unsafe code, such as the “Rust Safety Dance” project by the security working group of the Rust Team [2019a].

From Table 1, we can also see that the most used unsafe features are unsafe blocks and unsafe function declarations. Both unsafe trait declarations and unsafe trait implementations are rare – the former are found in less than 1% of all crates; given that implementations generally do have interesting contracts and invariants, this low number suggests that programmers do not find it useful to highlight those via the `unsafe` keyword.

5.2.2 RQ 2 (Size): What is the size of unsafe blocks that programmers write? Recall from Sec. 3.1 that we measure the number of MIR statements the compiler generates for an unsafe block, #MIR for short, as a proxy for its code complexity. Fig. 3 shows the cumulative distribution of MIR statements generated for each unsafe block, cropped at 100 #MIR to improve readability; the depicted graph covers 97.4% of all unsafe blocks. The size of most blocks is quite small: 75% of all unsafe blocks comprise at most 21 #MIR, which almost coincides with the mean of 22.0 #MIR. For comparison, the compiler already generates 12 MIR statements – more than the overall median of 10 – for the small unsafe block shown in Fig. 4. Upon closer manual inspection, there is a significant share, namely 14.4%, of tiny unsafe blocks that either wrap an expression (without function calls) or call a single unsafe function whose arguments were computed before the unsafe block. Conversely, there is a small number (78 or 0.02%) of huge outliers whose size ranges from 2,000 to 21,306 #MIR. Most of these unsafe blocks are automatically generated, e.g. through user-written macros or external scripts.

In summary, the size of unsafe blocks is typically small. Assuming that the number of MIR statements adequately approximates the complexity of unsafe blocks, we conclude that *most developers keep their unsafe blocks simple*, which supports the second principle of the Rust hypothesis.

5.2.3 RQ 3 (Self-containedness): Is the behaviour of unsafe code dependent only on code in its own crate? In our dataset, we have in total 772,228 calls in unsafe blocks. As shown in Fig. 5, more than three-quarters of them are calls to standard functions while calls to trait methods and calls to closures and function pointers are only 18.0% and 3.6%, respectively. If we compare with the distribution of the entire dataset (shown in Fig. 6) that includes also the compiler-generated code[8], we can see that calls in unsafe blocks have a significantly larger proportion of standard function calls (78.3% in unsafe blocks vs. 56.3% in the entire dataset) and, correspondingly, a significantly smaller

[8] The compiler-generated unsafe code contains only standard function calls.

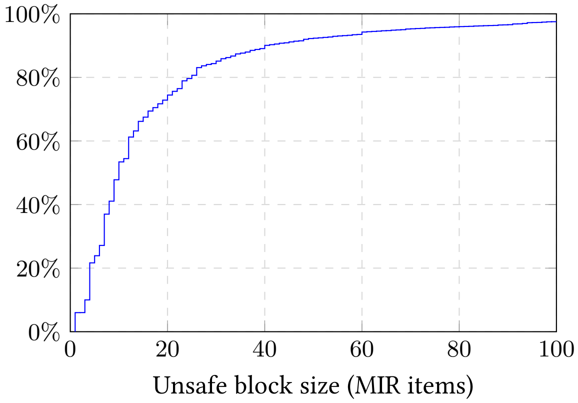

Fig. 3. Cumulative distribution of the size of unsafe blocks, cropped at 100 MIR instructions (#MIR).

```
fn nop(x: u32) {}

fn main() {
    let x = 42;
    unsafe {
        nop(x);
        nop(x);
    }
}
```

Fig. 4. Unsafe block of size 12 #MIR, more than the overall median of 10.

proportion of calls to trait methods (18.0% in unsafe blocks vs. 42.9% in the entire dataset). Even though 18.0% is still a substantial proportion, the relatively low number confirms our expectation that developers avoid those calls in unsafe blocks to keep the code simpler and more self-contained. In particular, a manual inspection of 100 randomly-selected calls to trait methods in unsafe blocks revealed that in 82 cases, the call target can be determined statically, just by looking at the function containing the unsafe block. Therefore, these calls do not add substantially to the complexity of the unsafe code.

To understand why the proportion of calls to closures and function pointers is larger in unsafe blocks than in all code (3.6% vs. 0.7%), we manually looked into several examples and observed three main patterns. The first one is parameterising the behaviour of unsafe code with a closure that is passed in as an argument to the safe wrapper. A typical example of this pattern is the `sort_by` function on the primitive type slice, which takes a comparison function as an argument. The second pattern is using function pointers to call functions from dynamically-loaded libraries (which can be done only from within an unsafe block), and the third pattern is using function pointers to implement callbacks to system libraries.

Besides the different forms of calls, we analysed where the call targets of standard calls are implemented, to assess the extent to which unsafe code is self-contained. Fig. 7 illustrates the distribution of targets of calls to standard functions, grouped into four categories. The majority (52.1%) of all function calls are into the standard library; as argued in Sec. 3.1, we consider such function calls only a minor violation of self-containedness. Most of the remaining calls (25.9%) stay within the same crate. Only 7.4% of all calls targets are located in other crates. We manually inspected a few of these crates and found that most of them, similarly to `-sys` crates, encapsulate system libraries. So in summary, for codebases written purely within Rust, very few calls actually violate the self-containedness principle of the Rust hypothesis.

We also analysed how the distribution of call targets changes when we consider only calls to *unsafe* functions (which, as we will see in Sec. 5.2.5, is the most common motivation for using an unsafe block). As shown in Fig 8, the share of calls to `-sys` crates is significantly higher, whereas the share of calls that stay within the same crate remains almost the same. This suggests that developers hesitate to call *unsafe* functions that reach out to other crates unless they explicitly wish to interact with system libraries.

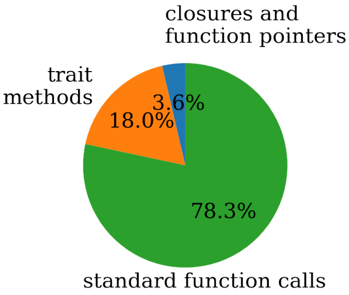

Fig. 5. Distribution of types of calls in unsafe blocks.

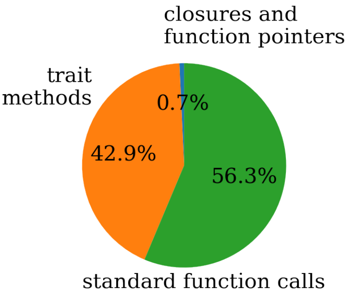

Fig. 6. Distribution of types of calls in the entire dataset (including the compiler-generated code).

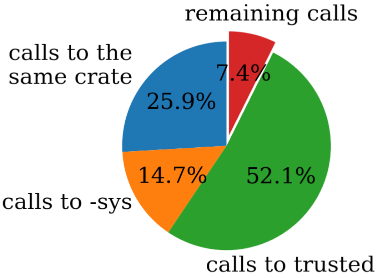

Fig. 7. The call targets of standard calls.

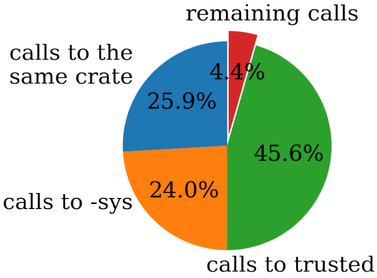

Fig. 8. The call targets of standard calls when only calls to *unsafe* functions are considered.

5.2.4 RQ 4 (Encapsulation): Is unsafe code typically shielded from clients through safe abstractions? In Table 2, we classify unsafe functions based on their visibility, which may be private (only callable from this submodule), visible within a restricted module, or public. We use visibility as an indication for the programmer's intention to encapsulate unsafe implementations from client code. Our metric is based on the information in a function's declaration, and does not differentiate between using the public modifier to enable calls from other submodules within the *same* crate, or from different crates entirely. For the latter, the functions would also need to be declared visible in the root module of the crate, which is a separate decision. Note that as soon as a function is declared public, its call-sites are in general unknown and may change over time. We removed from this analysis all unsafe trait methods (only 687, 0.1% of all unsafe functions), as their visibility is implicit.

We observed that only 12.0% of unsafe functions are not visible to arbitrary code (say, in other crates) because they are either private or restricted to a module. The vast majority (88.0%) of unsafe functions are declared to be public. At first glance, this suggests that programmers rarely shield their unsafe code from clients. To investigate this, we also studied the *ratio* of public unsafe functions compared to all unsafe functions *in each crate* containing at least one unsafe function. That is, a ratio close to 1 indicates that a crate poorly encapsulates unsafe functions (as all of these functions are public). Conversely, a ratio close to 0 indicates strong encapsulation as almost all unsafe functions are not publicly visible. The results are depicted in Fig. 9. Based on this metric, we get a clearer picture: most crates (78.5%) have either all or none of their unsafe functions declared public. In particular, 34.7% of all crates seem to be well encapsulated: they declare unsafe functions but none of them are visible from the outside.

Visibility	# functions	%
Private	65,230	11.7
Restricted	1,535	0.3
Public	489,928	88.0

Table 2. Visibility of unsafe function definitions (excluding trait methods).

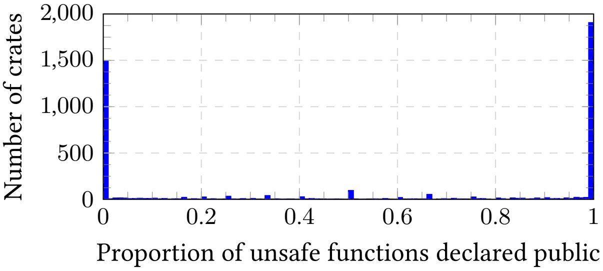

Fig. 9. The number of crates for each ratio of public unsafe functions compared to all unsafe functions.

Moreover, 43.8% of crates declare all of their unsafe functions public; more precisely, these crates contain 274,434 (49.2%) unsafe functions[9]. Continuing our investigation, we queried how many public unsafe functions within these crates provide raw bindings to system libraries, as it is common practice to make these bindings public. At first, we checked the ABIs of the functions. We found that 163,650 (59.6%) have foreign item ABI, which means that they are bindings of foreign items (most likely C functions). We also found that 571 functions (0.2%) have C ABI, which means that they can be called from C code and, therefore, it makes sense to have them public. The vast majority of the remaining functions (110,212 or 40.2%) have Rust ABI and, therefore, it is hard to automatically tell whether they are bindings or not. Therefore, we checked the meta information of the crates that contain these functions and found that 9,642 (3.5%) are assigned to categories that indicate them as crates that wrap system libraries and 49,363 (18.0%) are assigned to categories related to embedded programming. Finally, we manually reviewed 30 crates from the remaining list that have most unsafe functions (in total 41,063 functions or 15.0%) and found that they either provide APIs to microcontrollers or OpenGL bindings. After our analysis we are left with only 10,148 functions (3.7%) that are public and which may not be from the crates that provide bindings.

To summarise, even though the large number of crates that provide bindings make it hard to draw definitive conclusions, it seems that Rust programmers at least attempt to not expose unsafe functions to their clients because we found that 34.7% of all crates using unsafe functions do not declare a single public one; conversely crates that declare a lot of public unsafe functions can often be attributed to cases where encapsulation is not intended.

5.2.5 RQ 5 (Motivation): What are the most prevalent use cases for unsafe code? To answer this question, we first identified a set of independent reasons for which the compiler requires unsafe blocks and functions to be declared unsafe. We extracted these reasons from the source code of the Rust compiler [Rust Team 2020a]; they are therefore complete. Then, we collected which reasons apply to the implementation of each function (either the body of an unsafe function or the unsafe blocks inside a safe function). The results are summarised in Table 3. As the data shows, calls to unsafe functions are by far the main reason why unsafe code is unsafe, followed by dereferencing raw pointers[10].

A block or function may be unsafe for multiple reasons. We found that for 83.5% of all functions that have at least one reason of unsafety, calling unsafe functions is the *only* reason of unsafety. In 93.6% of the functions, unsafety is due only to the first 2 entries of the table, and that in 99.4%

[9] All percentages below are with respect to this number of unsafe functions.

[10] While all reasons mentioned in the table *should* require to use unsafe code, the reason "borrow of a packed field" does not due to a compiler bug; see https://github.com/rust-lang/rust/issues/27060 for details.

of the functions, all reasons for unsafety are among the top 3 entries of the table. This data will enable, for instance, developers of static analysers and verification tools for Rust to prioritise which features of unsafe Rust code should be supported in their tools.

The classification in Table 3 indicates why the compiler requires a block or function to be declared unsafe, but does not explain why programmers chose to use these unsafe features. To understand their motivation, we studied the prevalence of the specific use cases proposed in Sec. 3.2, as we discuss next.

Data Structures with Complex Sharing. To assess how often unsafe code is used to implement data structures with mutable aliases, we measured how many functions dereference a raw pointer and how many structs have raw pointer fields. Our database contains 7,385,690 function definitions, out of which only 46,263 (0.6%) dereference a raw pointer in their implementation. In particular, this is done in 9,273 out of a total of 557,380 unsafe functions (1.7%), in 35,761 out of 6,221,053 safe functions (0.6%), and in 1,229 of 607,257 closure declarations (0.2%). Overall, 7.0% of all crates have unsafe code that dereferences at least one raw pointer. Regarding the raw pointer fields, we found that 6.6% of all crates have types with raw pointer fields. After filtering out raw pointers in structs whose attributes indicate that they are likely intended for interoperability, this number reduces to 4.6% of all crates.

Given that the restriction to tree-shaped data structures seems to be a major limitation of safe Rust, the number of raw-pointer dereferences is sizeable, yet rather low. It seems that raw pointers are rarely used to implement more complex data structures. A possible explanation is that sharing occurs especially in standard data structures, such as cyclic lists, doubly-linked lists, smart pointers, and trees with parent-pointers. However, such data structures are provided by the standard library and, thus, do not often occur in the form of custom implementations in application code. Another possible explanation is that Rust programmers choose designs that can be implemented without mutable sharing in safe Rust rather than resorting to unsafe manipulation of raw pointers. Finally, developers may circumvent the ownership system while staying within safe Rust by relying on custom vector-backed heaps and using less constrained integer indices instead of references.

Even though not necessarily related to data structures, the use of mutable static variables (the third most-prevalent reason for unsafety in Table 3) is also a form of sharing because global data can be accessed and mutated by multiple functions – behaviour that one could alternatively achieve via aliased raw pointers.

Incompleteness Issues. In safe Rust, the type system is able to prevent, for instance, usage of references whose target *might* have been deallocated in some preceding conditional branch. These checks are a form of static analysis subject to incompleteness, as they conservatively reject some otherwise valid programs. Since incompleteness cases cannot be precisely identified automatically, we instead measured the calls to unsafe functions involving explicit type casts (`transmute` and `copy_transmute`) as a proxy to assess how frequently programmers need to work around incompleteness issues of the type checker. We found that 28,469 out of 319,600 unsafe blocks (8.9%) call a `transmute` function, and that 4.5% of all crates contain at least one call to a `transmute` function. Interestingly, only 1.7% of all crates have more than 3 unsafe blocks with a call to those functions. This confirms our expectation that calls to `transmute` functions, including workarounds for incompletenesses of the compiler, are rare, and that when crates have to make those calls, they use them sparingly. However, there still exist some outliers that make thousands of calls to `transmute`. After manual inspection, we found these crates to contain code generated by scripts or recursive macros, which explains the anomaly.

Table 3. Reasons why blocks and functions need to be declared unsafe, aggregated on the function level. For a specific function, there can be more than one reason why it needs to be declared as unsafe.

Reason	**#functions**	%
call to unsafe function	403,307	89.76
dereference of raw pointer	46,263	10.30
use of mutable static variable	25,888	5.76
access to union field	1,426	0.32
use of extern static variable	548	0.12
use of inline assembly	493	0.11
borrow of packed field	326	0.07
initialising type with `rustc_layout_scalar_valid_range` attr	41	0.0
assignment to non-Copy union field	3	0.0
pointer operation (in a `const` function)	2	0.0
cast of pointer to int (in a `const` function)	1	0.0
borrow of layout-constrained field with interior mutability	0	0.0
mutation of layout-constrained field	0	0.0

Emphasise Contracts and Invariants. To check how prevalent `unsafe` is used as a documentation feature, our queries gathered data for unsafe traits and unsafe functions with safe implementations.

We found 1,093 unsafe trait declarations, which amounts to only 2.5% of all trait declarations. We conclude that developers rarely use unsafe traits, possibly because (1) the compiler never forces them to and (2) there are no decisive guidelines for using unsafe traits. Instead, the Rust documentation seems to discourage developers from frequently declaring traits as unsafe [Rust Team 2019b, Ch. 1.1]. Notably, we observed that a few developers embraced unsafe traits enthusiastically: Five crates are responsible for 40.4% of all unsafe trait declarations.

Regarding unsafe functions, our experiments yield that 36.1% of all unsafe functions are written in completely safe Rust. We found this number surprisingly high. After all, the compiler does not force developers to declare such functions as `unsafe` – in contrast to other unsafe features. Rather, a programmer has to intentionally type an additional keyword. Hence, at first glance, it seemed that developers frequently were using unsafe functions for the same reason as unsafe traits: to document properties, e.g. invariants that are potentially critical for upholding Rust's safety guarantees.

To find explanations for the surprisingly high number of unsafe functions with safe implementations, we performed manual inspections: We manually inspected the ten crates with the highest overall count of unsafe functions with completely safe bodies. All of these crates are automatically generated to provide peripheral access to various microcontrollers. The involved code generation seems to be conservative and frequently use unsafe functions even if it does not have to. Moreover, we randomly selected a few additional unsafe functions with safe bodies for manual inspection. Among these functions, a few were equipped with explicitly documented invariants. Other functions seem to be marked as unsafe primarily for legacy reasons. So these extra inspections suggest that most of these functions are declared unsafe almost accidentally, rather than to intentionally highlight contracts and invariants. Therefore, the discrepancy between unsafe traits and unsafe functions seems much smaller than the initial numbers suggest.

Overall, there is no clear evidence that unsafe functions are frequently used for documenting contracts and invariants, except when those contracts overlap with (and perhaps protect against) situations in which unsafe Rust features are used in the functions' implementations.

Concurrency through Compiler Intrinsics. To measure to what extent compiler intrinsics are used to implement fine-grained concurrency, we collected all unsafe blocks that call one of the 89 compiler concurrency intrinsics defined in the `core::intrinsics` module, or their re-export from `std::intrinsics`. These functions are used by only 4 crates in our dataset: `core` (8 calls), `compiler_builtins` (7 calls), `rs_lockfree` (6 calls), and `hsa` (1 call). We thus conclude that compiler intrinsics are not widely used, probably because they are still marked as *experimental* and require a nightly version of the compiler.

Interestingly, while analysing the results, we found that the concurrency intrinsics exposed by the `compiler_builtins` crate are incorrectly *not* marked as `unsafe` even though they internally dereference a raw pointer passed as parameter. It is, thus, possible for safe code to dereference a null pointer from safe Rust code by calling these intrinsics. We reported this unsoundness in the API, which was confirmed by the library developers [Compiler-builtins developers 2020].

Foreign Functions. To detect interoperability with other languages we first measured how many types are equipped with `#[repr(C)]`, to have a memory layout compatible with C structures. Out of 1,486,978 definitions of structures and enumerations, we found that only 3.9% are annotated with `#[repr(C)]`. This annotation is used in 6.2% of all crates.

As a second query, we collected all crates whose name ends with `-sys`, to find those that adhere to the `-sys` naming convention for providing public bindings to a C system library. We found 650 crates (2.0% of all crates) whose names end with `-sys`, but we also noticed that other crates use different naming conventions: for 24 crates the name ends with `-ffi`, for 13 with `-bindings`, and for 10 with `-bindgen`. These suffixes all clearly mark a crate that provides public bindings to C libraries, as `-sys` crates should do. By further manual inspection of popular crates, we also found various crates such as `libc`, `gl`, and `winapi` that provide bindings to system libraries without using any naming convention. This plethora of cases suggests that the `-sys` convention is known, but not consistently applied by library developers.

As a third query, we measured how many unsafe functions are declared with a foreign ABI, to detect bindings to system library functions. We found that 248,522 (44.6%) out of 557,380 unsafe function definitions are actually static bindings to foreign items. This large percentage – which does not include functions that provide bindings to *dynamically* loaded libraries – shows that interoperability with foreign functions is actually a very common pattern of unsafe code. Overall, 1,599 crates (5.0% of all crates) contain at least one function with a foreign ABI. This reinforces the hypothesis that the `-sys` naming convention by itself is not enough to completely detect the crates that wrap system libraries.

Finally, as a fourth query, we measured usage of inline assembly. Out of more than 7 million function definitions we only found 493 cases of functions that use assembly. In particular, we found that 10 low-level and hardware-related crates actually contain 69.8% of all the functions that make use of inline assembly. This strongly suggests that inline assembly is in general rarely used.

Performance. Regarding our anticipated usages of unsafe code to improve performance, we found that 5.9% of unsafe calls in unsafe blocks involve unchecked functions spread across 4.3% of all crates. Avoiding run-time checks does not appear to be frequently used by all developers. However, it plays a significant role for some performance-oriented crates. For example, the Rust bindings of the X Window System call 4,852 unchecked functions in a single crate.

Developers rarely use the union `MaybeUninit`, which allows declaring uninitialised variables: We detected it in only 1,816 unsafe blocks, which appear in 0.55% of all crates.

In summary, performance optimisations using unsafe Rust seem to be a niche problem: They are mostly concentrated among a few crates. Within these crates, however, they are heavily used.

6 DISCUSSION

We now discuss the overall results of our study of unsafe Rust. Moreover, we address possible implications of our findings for the Rust community.

6.1 Does the Rust Hypothesis Hold?

One of the main research questions which motivated our study is the Rust hypothesis, i.e., the claim that Rust developers both can and typically do write unsafe code according to the three basic principles introduced in Sec. 1. That is, unsafe code should be (1) *used sparingly*, (2) *straightforward, self-contained*, and (3) *well-encapsulated*. While this claim is widely-advocated in the Rust community, the results of our study support it *only partially*, and must be qualified by the fact that we discovered that unsafe code concerned with interoperability is far more prevalent than might be expected.

We found strong evidence that programmers usually adhere to the second principle, i.e., they keep their unsafe blocks simple (RQ 2) and self-contained (RQ 3). This is encouraging, as the rationale underlying this principle is to reduce the amount of code whose safety relies on manual efforts by programmers instead of automated guarantees by the compiler.

However, the first principle is not widely adhered to if one examines our data set as a whole; crates containing at least some unsafe code are not at all uncommon: we discovered that almost a quarter of all crates contain at least some unsafe code (RQ 1). The total ratio between safe and unsafe code also indicates that unsafe code is used quite extensively. When compared to previous studies, we observed that the usage of unsafe Rust is most-likely *decreasing* over time: possibly a reflection of efforts in the Rust community to reduce the overall reliance on unsafe code.

Regarding the third principle, while our initial measurements suggest that unsafe functions are extremely prevalent, our follow-up steps paint a clearer picture: Rust developers frequently seem to attempt to hide all of their unsafe functions from clients. Moreover, a great many of the exposed unsafe functions originate from crates that provide bindings for interoperating with hardware or libraries written in other languages (typically C and C++). After several attempts to classify these cases, we were left with sufficiently few public unsafe functions to conclude that for unsafe functions *implemented in Rust alone*, programmers avoid making at least the vast majority publicly visible (RQ 4). A precise measurement is made challenging by the fact that crates performing interoperability not only contribute many unsafe functions to our data set but do not, in general, adhere to naming conventions designed to make them easy to identify (i.e., `-sys` suffix for crates providing C library bindings).

Overall, our results appear to support the Rust hypothesis for at least the majority of Rust code which is not for interoperability. However, there appear to be non-trivial exceptions to all three principles; something that project managers may want to keep in mind when adding dependencies to their own codebases, and software testers should consider paying close scrutiny to. Our Qrates analysis framework could be repurposed in this setting as a means of gathering important metadata about the usages of unsafe code in a crate under consideration. Simple checks could also be added to the official Rust linter Clippy [Clippy developers 2019] to warn of potentially-risky visibility of unsafe functions.

6.2 How is Unsafe Rust Used?

The second key question of our study was concerned with finding the most prevalent use cases of unsafe Rust. To this end, we explored all reasons that the Rust compiler used to enforce the use of unsafe blocks, and considered six specific use cases based on our own manual classification.

In general, calls to unsafe functions suffice to explain why 83.5% of all unsafe blocks and unsafe functions need to be declared unsafe; unsafe functions are the main feature that tool developers who wish to support unsafe Rust should prioritise. This percentage climbs to 93.6% if we additionally consider raw pointer dereferences and to 99.4% by also including access to mutable static variables.

Taking a closer look at the individual use cases, we observed that the number of raw pointer dereferences is sizeable, yet low compared with calls of unsafe functions. It seems that Rust developers rarely use raw pointers for implementing custom C-style data structures; they seem to prefer either the standard library abstractions, or different implementation patterns, such as using vector-based heaps and integer indices. This has interesting consequences for tool developers, as many automated program analysis techniques, e.g. shape analysis [Wilhelm et al. 2002], work well on tree-shaped data structures, but suffer dramatically in the presence of complex sharing. If complex sharing is predominantly achieved through a few standard abstractions and patterns, tools may have a chance to recognise and exploit idiomatic usages of popular abstractions.

Rust's type system and run-time checks rarely seem to concern developers enough to take the risk of circumventing the rules; they rarely (in fewer than 9% of all unsafe blocks) opted to disable compiler checks or transmute types to either gain performance or manually override the standard type system.

While concurrency-related compiler intrinsics appear only in a tiny fraction of crates, our manual follow-up on our experiments confirms that they can be quite dangerous: during our analysis, we discovered a bug that allowed us to dereference a null pointer from safe Rust code by calling one of them (albeit in an unstable feature under development).

We observed that many functions are declared unsafe despite not employing *any* of the language features which the compiler requires to be used only in unsafe code. This suggests either confusion on the part of the programmer as to when such blocks are required, legacy reasons (e.g. the same function must be unsafe on a different platform), or that programmers employ such blocks to document either their own ad hoc correctness properties, or properties that are required for upholding safety even though the Rust compiler neither checks them nor associates them with unsafe Rust. An example of the latter case could be to indicate a precondition for a function which is necessary for the preservation of some *invariant* that the safety of *other* functions depends upon. While the latter case matches with the community's expectations[11], we encountered only few instances in which unsafe functions explicitly document contracts or invariants. Similarly, hardly any programmer declares unsafe traits, which act purely as documentation features (warning implementers to take care of a particular property). One reason for the low adoption of this feature could be that contracts and invariants are not enforced by the compiler. This could be changed by allowing developers to attach more formal specifications to traits and functions. For example, a recently-developed Rust verifier [Astrauskas et al. 2019] provides a plugin for the Rust compiler that allows the annotation of Rust code with specifications via dedicated attributes.

Finally, we observed that a large amount of unsafe code serves to interoperate with libraries written in other programming languages. Many crates purely act as an interface for these libraries. To facilitate future code analyses, it would be beneficial if these could be made easier to detect and classify, e.g. through an explicit attribute for low-level crates. We observed that naming conventions (in particular, the `-sys` convention for crates providing C bindings) appear to be insufficient for this purpose, presumably because programmers have their own preferences for crate naming.

7 THREATS TO VALIDITY

In this section, we present threats to the validity of our empirical study and its evaluation.

Dataset. The main threat to validity is the selected dataset, and any biases this may introduce compared with current practice using the Rust language in general. By taking all currently-compiling

[11] As expressed in communications with developers via the Rust Secure Code Working Group.

crates from crates.io, we have a substantial slice of the open-source Rust code available, but other complementary sources (such as github.com) could also be solicited.

On the other hand, it is possible that we are sampling *too much* code from our chosen source; our dataset likely includes crates which are no longer maintained, for example. Arguably, these crates are not flawed as sources for our study, but the older they are, the less significant their features are for evaluating current practice.

Since we currently pre-filter crates by whether they compile, there is a chance that some crates have been missed because we were not able to identify the right compiler version or settings. We mitigated this last specific threat by taking such compiler arguments from the crate's own manifest; unfortunately, determining the intended compiler version is not possible in general.

Generated Code. Some occurrences of unsafe code are generated internally by the compiler. For example, the desugaring of pattern-matching constructs can yield unsafe code, even for source code which never uses unsafe Rust. To avoid, for example, potentially classifying such crates as containing usages of unsafe (RQ 1), we eliminate all compiler-generated unsafe blocks in advance (the compiler never generates unsafe functions) for all relevant queries in our experiments. Since we filter out compiler-generated unsafe blocks but keep all compiler-generated safe code, our measurements may underestimate the ratio of unsafe to safe code. Similarly, the distribution of types of function calls reported for our entire dataset (Fig. 6) may change when filtering out all compiler-generated code.

The measurements for RQ 2 (Size) show a wide distribution, with a few huge outliers. As explained in Sec. 5, rather than simply reporting our answers (and averages) directly, we were able to identify manually that these outliers are due to generated code. However, our dataset might contain other forms of non-standard Rust code (e.g. generated by scripts), which we did not identify. Nevertheless, the cumulative distribution we chart forms an adequate basis for our conclusion that the vast majority of unsafe blocks are small.

Comprehensiveness of Queries. Our results are mostly based on queries that collect data automatically. Some of these queries check proxies for the properties of interest, for example, the size of unsafe blocks as a measure of complexity and the visibility of unsafe functions as an indication for the presence of safe abstractions. There is a risk that our choice of proxies (and queries) does not faithfully capture the properties of interest. To mitigate this risk, we complemented our automatic queries by manual code inspections.

Overlapping Motivations. Our presented analysis of the motivation for programmers using unsafe Rust (RQ 5) does not include a detailed analysis of the *overlap* between multiple motivations (for example, blocks containing both raw pointer operations and assembly code). Our framework and data set *do* provide this information, and exploring these correlations could be interesting future work.

Errors in the Implementation. Most of our results are based on our Qrates framework and subsequent data processing in Jupyter notebooks. Errors in the implementation could invalidate our findings. To mitigate this risk, we subjected all implementations to careful code reviews and tested them extensively.

Errors in the Rust Compiler. Some of our measurements, e.g. the reasons for unsafety in Table 3, rely on internal data computed by the Rust compiler—they are, consequently, sensitive to compiler bugs. While we checked Rust's bug tracker and asked for feedback from the Rust community to account for known compiler issues, our results may be influenced by so-far unknown bugs.

8 RELATED WORK

In this section, we focus mainly on other studies of Rust code; other references to relevant related work are mentioned throughout the paper where appropriate.

Closest to our work is a recent study by Evans et al. [2020] that performs a quantitative evaluation of unsafe Rust. Since their dataset relies on a 16 months older snapshot of the same repository (crates.io), we can observe a few developments over time: in particular, the percentage of crates containing *any* unsafe code (a measurement made in both studies) has dropped from 29% to 23.6% (noting though that these comparisons relate figures which were computed by different experimental methods).

Apart from such basic statistics, the two studies are complementary as they take fundamentally opposite perspectives on the usage of unsafe code: we study (intended) compliance with the Rust hypothesis, i.e., commonly advocated best practices for writing unsafe code; in particular, our work takes into account whether programmers aim to shield their unsafe code behind safe abstractions. By contrast, Evans et al. [2020] measure how the unsafe keyword permeates call chains and, thus, how many functions *transitively* depend on unsafe code; they consider all of these functions as tainted, or, in their terminology, "possibly unsafe". This notion ignores whether a function (1) depends on unsafe code through a safe abstraction that *takes responsibility* (by declaring exposed functions as safe) for Rust's guarantees, or (2) is *explicitly exposed* to unsafe code (by declaring all functions in the call chain as unsafe).

These different perspectives can lead to quite different results for the same data set. For example, assume the corpus under analysis consists of two idealised codebases, say A and B, where A strictly adheres to the Rust hypothesis, and B frequently violates it. Moreover, suppose that many (declared safe) functions are at least transitively clients of the unsafe code in A and B, respectively. Our methodology *distinguishes* these codebases, concluding that some developers (those of A) aim to apply commonly advocated guidelines, and some do not (those of B). In contrast, the approach of Evans et al. [2020] cannot distinguish them: both codebases are possibly unsafe. Furthermore, our study provides an in-depth analysis of the reasons *why* programmers employ unsafe code, which was not an apparent primary focus of Evans et al. [2020] (where this question was explored via a survey rather than by analysing a codebase).

Our work evaluates the extent to which programmers *intend* for their unsafe code to be encapsulated from clients. However, we do not attempt to judge whether programmer's implementations *correctly* provide such an abstraction. This question has many subtle dimensions (as has been explained by e.g. Jung [2016]) and addressing it, in general, requires extremely sophisticated formal reasoning, as explored in the context of the RustBelt project [Jung et al. 2018].

Qin et al. [2020] perform manual code inspections of 850 selected instances of unsafe Rust, i.e., unsafe blocks or functions, to elicit the motivation for using unsafe code and to explore the memory safety and concurrency errors caused by these instances of unsafe code. They select examples for their study by analysing bug reports and filtering commit messages for keywords indicating memory errors. Their work includes an analysis of incorrect usages of unsafe code but does not assess how and why unsafe code is used in general. Our much larger dataset yields a different perspective in several respects: for example, they observe that "calling unsafe functions counts for 29% of the total unsafe usages", whereas the percentage across our full data set is much larger, namely 84.6%. We can also confirm over our larger data set their observation that a significant number of unsafe functions need not (from the compiler's perspective) be labelled as unsafe.

Ozdemir [2016] performs an early study of how unsafe Rust is used. The high-level approach is somewhat similar to our own; source code analysis is performed by a compiler plugin that collects statistical data about unsafe blocks and functions. The presented results are mostly covered by our first two research questions. In particular, Ozdemir notes that 30% of all crates contain some unsafe code; this confirms the trend we noted in Sec. 5 alongside RQ 1 that the overall number of unsafe crates appears to be declining over time. Given the significant changes to the Rust language and its user base over the last four years, we do not compare all results in detail.

In addition to the above studies, which specifically consider unsafe Rust, there have also been qualitative studies on how well-suited Rust could be for certain applications: Mindermann et al. [2018] study the usability of Rust's cryptographic libraries, whereas both Balasubramanian et al. [2017] and Levy et al. [2017] evaluate Rust in general as a systems programming language.

9 CONCLUSION

We have presented a large-scale study addressing the current practices of Rust programmers with respect to unsafe Rust, as well as an investigation of the *motivations* for why unsafe code is employed in practice. We identified three commonly-held expectations regarding unsafe Rust practice (the Rust hypothesis), and our study showed partial support for these. In particular, while our study shows that unsafe code is very commonly used in small and self-contained quantities, it is much less scarce overall than one might expect, and in total a great many unsafe functions are exposed to arbitrary client code across crate boundaries. On the other hand, a very sizeable portion of these functions ultimately result from the need to provide interoperability with custom hardware and native code written in C; when one eliminates the vast majority of these cases, it becomes clear that most unsafe functions written in Rust are not actually publicly-accessible, indicating a common effort by programmers to encapsulate the unsafe aspects of their implementations. On the other hand, many of the unsafe functions we investigated manually did not document the intended requirements imposed on their callers; this suggests a weakness in development practice which programmers should be aware of, and software testers should look to highlight.

Our paper presents the Qrates framework with which we have carried out our study, along with a large repository of harvested data on recent unsafe code usage. Our framework and experimental methodology can be straightforwardly reused for a wide variety of Rust-related empirical studies (not limited to unsafe code), and could, for example, be used to complement the annual Rust Survey [Rust Language Team 2019] with data on current coding practice in the community. This survey asks specifically for recommendations for improvement; the widespread reuse of C libraries we have observed during our study (e.g. those providing GUI libraries such as Qt with no existing Rust alternative, or facilities to dynamically-load code, which is not natively possible in Rust) provides empirically-justified directions for future language and library development.

Our investigation of the *reasons* why unsafe code is employed shows that the vast majority of unsafe code is used to call unsafe functions, while only few other causes arise commonly. These results have important implications for the potential development of Rust analysis tools, particularly where the aim is to help programmers reason about whether their unsafe code is correct; an ability to specify or otherwise support reasoning about external function calls is a critical concern, while our study shows that support for, say, inline assembly need not be prioritised.

An advantage of our overall methodology (and our developed framework Qrates in particular) is that running follow-up studies of a complementary nature is relatively straightforward. Indeed, since we have made Qrates available to the community [Qrates Team 2020], it will be possible for others to author-related studies by building upon our technical approach. As future work, we are considering developing on Qrates to provide an analysis tool capable of simply answering queries and generating metadata about a crate in question; this could have important applications for code review and software teams considering potential dependencies. The highly automated nature of our framework also opens up the possibility of longer-term studies on the *evolution* of programmer practice, making it possible to judge how common-practice and accepted standards for unsafe Rust usage are changing over time.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive feedback which, in particular, led to a refined version of the queries considered in RQ 5. We are grateful to Ana Nora Evans for showing us how to avoid pitfalls related to code generated by macros when extracting information from the Rust compiler. Moreover, we thank Ralf Jung and the Rust Secure Code Working Group for their feedback and for clarifying the intended purpose of unsafe traits in the Rust community.

This work was partially funded by the Swiss National Science Foundation (SNSF) under Grant No. 200021_169503.

REFERENCES

Pietro Albini. 2020. The Rustwide library. https://crates.io/crates/rustwide Accessed May 11, 2020.

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 147:1–147:30. https://doi.org/10.1145/3360573

Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamaric, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. *Operating Systems Review* 51, 1 (2017), 94–99. https://doi.org/10.1145/3139645.3139660

Nick Cameron, Sebastien Celles, and Contributions by the Rust Community. 2019. Rust For Systems Programmers. https://github.com/nrc/r4cppp Accessed May 11, 2020.

David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998*, Bjørn N. Freeman-Benson and Craig Chambers (Eds.). ACM, 48–64. https://doi.org/10.1145/286936.286947

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE! 2015)*. ACM, 1–12.

Clippy developers. 2019. Clippy. https://github.com/rust-lang/rust-clippy Accessed April 4, 2019.

Code QL. 2020. Website of Code QL. https://semmle.com/codeql Accessed May 11, 2020.

Compiler-builtins developers. 2020. Safety of intrinsics. https://github.com/rust-lang/compiler-builtins/issues/355 Accessed September 7, 2020.

Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers? *CoRR* abs/2007.00752 (2020). arXiv:2007.00752 https://arxiv.org/abs/2007.00752

Fuchsia Team. 2020. Fuchsia Documentation - Unsafe Code in Rust. https://fuchsia.googlesource.com/fuchsia/+/master/docs/development/languages/rust/unsafe.md Accessed May 11, 2020.

Jon Gjengset. 2020. Demystifying unsafe code (Talk at Rust NYC). https://youtu.be/QAz-maaH0KM Accessed on March 19, 2020.

Colin Stebbins Gordon. 2014. *Verifying Concurrent Programs by Controlling Alias Interference.* Ph.D. Dissertation. University of Washington.

Ralf Jung. 2016. The Scope of Unsafe. https://www.ralfj.de/blog/2016/01/09/the-scope-of-unsafe.html Accessed April 4, 2019.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2020. Safe Systems Programming in Rust: The Promise and the Challenge. *Commun. ACM (to appear)* (2020). https://people.mpi-sws.org/~dreyer/papers/safe-sysprog-rust/paper.pdf

Jupyter Team. 2020. The Jupyter project. https://jupyter.org/ Accessed May 11, 2020.

Steve Klabnik and Carol Nichols. 2019. The Rust Programming Language. https://doc.rust-lang.org/book/ Accessed May 11, 2020.

Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The Case for Writing a Kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems* (Mumbai, India) *(APSys '17)*. ACM, New York, NY, USA, Article 1, 7 pages. https://doi.org/10.1145/3124680.3124717

LLVM Team. 2020. LLVM Atomic Instructions and Concurrency Guide. https://llvm.org/docs/Atomics.html Accessed May 11, 2020.

Nicholas D. Matsakis. 2016. Unsafe abstractions. http://smallcultfollowing.com/babysteps/blog/2016/05/23/unsafe-abstractions Accessed on May 5th, 2020; Matsakis is co-leader of the Rust compiler team.

Nicholas D. Matsakis. 2017. Project idea: datalog output from rustc. https://smallcultfollowing.com/babysteps/blog/2017/02/17/project-idea-datalog-output-from-rustc/ Accessed May 11, 2020.

Kai Mindermann, Philipp Keck, and Stefan Wagner. 2018. How Usable Are Rust Cryptography APIs?. In *2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, July 16-20, 2018*. IEEE, 143–154. https://doi.org/10.1109/QRS.2018.00028

Peter Müller. 2002. *Modular Specification and Verification of Object-Oriented Programs*. Lecture Notes in Computer Science, Vol. 2262. Springer. https://doi.org/10.1007/3-540-45651-1

Alex Ozdemir. 2016. Unsafe in Rust: Syntactic Patterns. https://cs.stanford.edu/~aozdemir/blog/unsafe-rust-syntax Accessed on May 5th, 2020.

Alex Potanin, Monique Damitio, and James Noble. 2013. Are your incoming aliases really necessary? counting the cost of object ownership. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 742–751. https://doi.org/10.1109/ICSE.2013.6606620

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. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 763–779. https://doi.org/10.1145/3385412.3386036

Redox developers. 2019. Snippet from Redox OS Repository. https://github.com/redox-os/relibc/blob/2cbc78f238b3eda426171def100f44707cfe8ae3/src/platform/pte.rs#L337-L345 Accessed May 11, 2020.

Rust Language Team. 2019. Rust Survey 2019 Results. https://blog.rust-lang.org/2020/04/17/Rust-survey-2019.html Accessed May 15, 2020.

Rust Team. 2018. Rust: The Reference. https://doc.rust-lang.org/reference/ Accessed May 11, 2020.

Rust Team. 2019a. Mission Statement of the Secure Code Working Group. https://github.com/rust-secure-code/wg Accessed May 11, 2020.

Rust Team. 2019b. The Rustonomicon. https://doc.rust-lang.org/nomicon/ Accessed May 2, 2020.

Rust Team. 2019c. The Cargo Book. https://doc.rust-lang.org/cargo/reference/build-scripts.html#-sys-packages Accessed May 11, 2020.

Rust Team. 2020a. File: check_unsafety.rs. https://github.com/rust-lang/rust/blob/27ae2f0d60d9201133e1f9ec7a04c05c8e55e665/src/librustc_mir/transform/check_unsafety.rs Accessed May 11, 2020.

Rust Team. 2020b. Rust Documentation of Transmute. https://doc.rust-lang.org/std/mem/fn.transmute.html Accessed September 10, 2020.

Rust Team. 2020c. Rust: The Reference — Module std::cell. https://doc.rust-lang.org/std/cell/index.html Accessed May 11, 2020.

Rust Team. 2020d. Rust: The Reference — Module std::slice. https://doc.rust-lang.org/std/slice/index.html Accessed May 11, 2020.

Rust Team. 2020e. Rust: The Reference – Module std::sync::atomic. https://doc.rust-lang.org/std/sync/atomic/ Accessed October 6, 2020.

Qrates Team. 2020. Qrates artefact. https://doi.org/10.5281/zenodo.4026639 Source code and dataset: https://github.com/rust-corpus/qrates.

The Libra Association. 2020. Core Repository of the Libra Project. https://github.com/libra/libra/blob/8d9bba00629e602051e40bab2b80e7ed89f40c0b/storage/storage-client/src/state_view.rs#L96-L97 Accessed May 11, 2020.

Unsafe Code Guidelines Working Group. 2020. Project website. https://github.com/rust-lang/unsafe-code-guidelines Accessed August 17, 2020.

Reinhard Wilhelm, Thomas W. Reps, and Shmuel Sagiv. 2002. Shape Analysis and Applications. In *The Compiler Design Handbook: Optimizations and Machine Code Generation*, Y. N. Srikant and Priti Shankar (Eds.). CRC Press, 175–218. https://doi.org/10.1201/9781420040579.ch5