How Do Programmers Use Unsafe Rust?

ANONYMOUS AUTHOR(S)

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 sparingly, make it easy to review, and hide 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 structure, its intermediate representation MIR, as well as type information provided by the Rust compiler, and 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.

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

1 INTRODUCTION

Rust is a systems programming language whose type system prevents many common errors at compile time. A well-typed Rust program is guaranteed not to exhibit, amongst others, 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 how memory is accessed and shared.

In Rust, every memory location is owned by precisely one variable; when the owning variable goes out of scope, the memory is freed. Ownership can be transferred permanently (via move assignments) or temporarily, via borrowing. For instance, a common idiom is for functions to borrow their arguments from the caller and restore ownership when the call terminates. Besides exclusive (mutable) borrows, Rust also permits sharing of memory locations via shared references. To prevent data races and inadvertent side effects, the memory locations reachable by a shared reference are enforced to be immutable as long as the shared reference exists. In order to determine when (mutable or shared) borrows go out of scope and exclusive ownership is 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...
Unsafe code can be hidden from clients behind safe abstractions by wrapping it inside safe functions. The function bar2 in Fig. 1b, for example, provides a safe abstraction of function foo—the developer promises that foo(17) preserves Rust’s safety guarantees even though the compiler cannot enforce them.

The second form involves traits, which are comparable to Java interfaces. Instead of disabling compiler checks, an unsafe trait declaration warns developers that the trait relies on semantic properties such as preconditions, postconditions, or invariants that are not checked, neither at compile nor at run time. For trait implementations, unsafe takes the role of an annotation by which developers acknowledge their responsibility to respect the required semantic properties.

Unsafe code must be used with care to retain the strong guarantees Rust aims to provide. The commonly-advocated practice is that programmers should, as far as possible, use unsafe Rust according to the following three basic principles (cf. Rust Team [2019b, Ch. 1], Klabnik and Nichols [2019, Ch. 19], and prominent sources from the Rust community, e.g., Fuchsia Developers [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 to clients.

Fig. 1. Examples of Unsafe Rust.
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 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 can 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 structure, its intermediate representation MIR, as well as type (and borrowing) 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 also by far the most prevalent motivation for using unsafe code, followed by complex sharing (via raw pointers or mutable static fields). Other purposes, such as using unsafe concurrency features and applying unsafe as a documentation feature (the second form of unsafe code mentioned above) are far less common.

Contributions. Our paper makes the following contributions:

- A classification of the motivation for using unsafe code. We identify seven 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 for 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 adheres to the Rust hypothesis.
- A discussion of the implications for reasoning about Rust code (in code reviews or during verification).
- The Rust corpus infrastructure, which is useful also for other empirical studies of Rust programs (beyond unsafe code). We will submit the infrastructure as well as the data we collected as an artefact.

Outline. Sec. 2 classifies the main usages of unsafe code into seven 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 seven 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 safe Rust is too restrictive. Our first-class 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 to construct cyclic references as this could lead to dangling references during deallocation. 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 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 value through one of the multiple shared references [Rust Team 2020b; 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 as 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.

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 vector into two at some specified index mid:

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

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

2.2 Emphasize Contracts and Invariants

As noted in Sec. 1, unsafe comes in two different forms: The use cases discussed in the previous subsection 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. For example, consider the function below taken from Rust Team [2020c].

```rust
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 a parameter. Its documentation requires that `data` refers to a slice of length at least `len` such that it can be safely read. The compiler encourages the above idiom by suggesting programmers to “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` should never panic. Another interesting example is the unsafe trait `TrustedLen` whose documentation requires the result of function `size_hint` to return the precise length of an 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.

### 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 takes a context and returns a solver type. By calling it from an unsafe block, the developer vouches for its safety within the calling context.

```rust
#![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.

#### 2.3.2 Concurrency through Compiler Intrinsics

The Rust compiler offers concurrency intrinsic functions that mirror the lower-level intrinsics of LLVM [LLVM Team 2020]. All of them are marked as unsafe because they 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 \texttt{atomic\_xadd} to increment a semaphore counter:

\begin{verbatim}
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
}
\end{verbatim}

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 optimizer,
e.g., that an addition never overflows. 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 optimization to quickly cast an array of bytes to a Rust structure without
performing any safety checks:

\begin{verbatim}
#\[repr(C)\]
struct SerializedStruct { /* ... */ }
unsafe fn cast_deserialize(i: &[u8]) -> &SerializedStruct {
    *(i.as_ptr() as *constSerializedStruct)
}
\end{verbatim}

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\footnote{We highlight all specific queries in \textit{green italic text}.} 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 \textit{sparingly}. 

, Vol. 1, No. 1, Article . Publication date: May 2020.
Unsafe code should be straightforward and self-contained.

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 identify every usage of unsafe Rust in our dataset, and specifically count how many crates (that is, 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. Determining what fraction of crates is completely safe allows us to measure how frequently developers avoid writing unsafe code altogether—as we would expect by the first principle. We measure the amount of unsafe code separately, see 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?

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 realize quite complex behaviour with a few lines of code.

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 second 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.

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. To separate concerns, safe abstractions need to be built on top of these crates, which naturally leads to a dependency that is in direct conflict to self-containedness.

We consider call targets in the above two categories separately as they amount to expected 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 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 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 unsafe function 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 emphasizing 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 characterizations because the results of our queries should be gathered automatically, possibly with manual follow-up efforts. As it 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.

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 of these different styles.

Incompleteness Issues. It is not generally possible to precisely identify cases in which developers work around incompleteness issues of Rust’s type and ownership system. If it was, the compiler could use the same analysis to prevent the incompleteness in the first place. Therefore, we focus on cases in which developers explicitly work around static checks of the compiler. More precisely, 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 the lifetimes inference.

Emphasize 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 them to rely on some contract or invariant (unless unsafe is used accidentally or for legacy reasons). 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 create a whitelist of unsafe functions wrapping concurrency intrinsics. We then run a query collecting all unsafe blocks that call a whitelisted function.

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 optimizations or interacting with hardware devices, is subject to manual inspection.

Performance. We consider two standard optimizations that use unsafe code to boost performance: First, we search for unchecked functions (those with “unchecked” in their name). By convention, these functions sacrifice runtime 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 runtime (cf. [Rust Team 2018]). In fact, it may reintroduce dangling references as developers may use it to define uninitialized 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 Querying Rust Crates 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 CodeQL [2020], for analyzing 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, i.e., algorithms that extract statistical data from the database to answer specific questions. 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 CFG as well as type, and borrowing information, about 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 analyzed code in various intermediate formats, such as the CFG representation (MIR). 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 discussed 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 our database may contain several versions of the same crate, e.g. if another crate depends on an outdated version. However, all of our queries report results only for the most recent version of each crate. We still keep outdated crates in the database, because some queries are concerned with dependencies. For example, when analyzing the call targets of a function, we also need to consider the outdated dependencies.

Furthermore, 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; otherwise, we risk overcounting features that are predominantly introduced by the Rust compiler. To detect potential discrepancies, we run our queries both on the original code and a copy of the original code in which we eliminate macros and attributes before compilation.
How Do Programmers Use Unsafe Rust?

<table>
<thead>
<tr>
<th>Unsafe Feature</th>
<th>#crates</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>22,857</td>
<td>75.9</td>
</tr>
<tr>
<td>Any</td>
<td>7,262</td>
<td>24.1</td>
</tr>
<tr>
<td>Blocks</td>
<td>6,187</td>
<td>20.5</td>
</tr>
<tr>
<td>Function Declarations</td>
<td>4,171</td>
<td>13.8</td>
</tr>
<tr>
<td>Trait Implementations</td>
<td>1,549</td>
<td>5.1</td>
</tr>
<tr>
<td>Trait Declarations</td>
<td>274</td>
<td>0.9</td>
</tr>
</tbody>
</table>

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

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 will provide a detailed discussion of our findings in Sec. 6.

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 primary and usually determines the name to the package. We excluded 5677 packages (16%) whose most recent version did not successfully compile. For crates with conditional compilation features, we used the default flags specified in the manifest. As a result, our dataset consists of 30119 crates. Most of these crates are compiled to Rust libraries, namely 77.4%, or binaries, namely 18.5%. The other crates are procedural macros (4.2%).

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] (we plan to make these available as a part of our accompanying artifact).

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, while Fig. 2 shows the proportion of statements in unsafe blocks and functions in crates that have at least one such statement.

Fig. 2. The proportion of statements in unsafe blocks and functions in crates that have at least one such statement.

The majority of crates (75.9%) contain no unsafe features at all. Even in most crates that do contain statements in unsafe blocks and functions, only a small fraction of the code is unsafe.
However, with 24.1% of crates containing some unsafe code, 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 24.1%. This observation 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, capped at 100 #MIR to improve readability; the depicted function covers 98.5% 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 22.1 #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 inspection, there is a significant share, namely 14.9%, of tiny unsafe blocks that either wrap an expression (without function calls) or call a single unsafe function without changing any of its arguments. Conversely, there is a small number (0.25%) of huge outliers whose size ranges from 2000 to 21306 #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 763,215 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% and 3.7%, respectively. If we compare with the distribution of the entire dataset (shown in Fig. 6), we can see that calls in unsafe blocks have a significantly larger proportion of standard function calls (78.3% in unsafe blocks vs. 55.8% in the entire dataset) and, correspondingly, a significantly smaller proportion of calls to trait methods (18% in unsafe blocks vs. 43.5% in the entire dataset). Even though 18% is still a substantial proportion, the relatively low number confirms our expectation that developers avoid those calls in unsafe blocks to keep the code simple. 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,
Fig. 3. Cumulative distribution of the size of unsafe blocks, capped at 100 MIR instructions (#MIR).

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

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 (3.7% vs. 0.7%), we manually looked into several examples and noticed 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, and the third pattern is using function pointers to implement callbacks to foreign libraries.

Besides the different forms of calls, we analysed where the call targets are implemented, to assess whether unsafe code is self-contained. Figure 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 (25.7%) stay within the same crate. Only 7.5% of all calls targets are located in other crates; we manually inspected a few of these crates and found that most of them, similar to -sys crates, encapsulate foreign libraries. So in summary, very few calls actually violate the self-containedness principle of the Rust hypothesis.

We also explored the distribution of targets of calls to unsafe functions. 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 foreign libraries.

5.2.4 RQ 4 (Encapsulation): Is unsafe code typically shielded from clients through safe abstractions? We classified in Table 2 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.) As soon as a function is declared public, its call-sites are in general unknown and may change over time.
We observed that only 12% 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%) of unsafe functions are declared to be public. We decided to check how many of these functions are from crates that provide raw bindings to foreign libraries and library intrinsics because it is a common practice to make these bindings public. At first, we checked the ABIs of the functions. We found that 235,481 (43.1% of all functions) have foreign item ABI, which means that they are bindings of foreign items (most likely C functions). We also found that 3,233 (0.6%) functions 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 (242,116 or 44.3% of all functions) 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 134,535 (24.6%) are assigned to categories that indicate them as crates that wrap system libraries. Finally, we manually reviewed 30 crates from the remaining list that have most public unsafe functions (in total 90,152 functions, or 16.5% of all functions) and found that all of them as well are crates that wrap system libraries. After our analysis we are left with only 17,429 (or 3.2%) functions that are public and which may not be from the crates that provide bindings.

To summarise, even though the large number crates that provide bindings make it hard to make definitive conclusions, it seems that the Rust programmers at least intend to not expose unsafe functions to their clients because we found that 11.9% of functions are non-public while at most 3.2% of public functions are not bindings.
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 why 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]. 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.

A block or function may be unsafe for multiple reasons. We found that 85% of all functions that have at least one reason of unsafety, calling unsafe functions is the only reason of unsafety. In 94% of the functions, unsafety is due only to the first 2 entries of the table, and that in 99% of the functions, all reasons for unsafety are among the top 3 entries of the table. This data will enable e.g. developers of static analysers to prioritise which features of unsafe Rust code should be supported in their tools.

The classification in Table 3 indicates why a 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. We identified 7,117,244 function definitions in our database, out of which only 44,974 (0.6%) dereference a raw pointer in their implementation. In particular, this is done in 8,758 out of a total of 546,705 unsafe functions (1.6%), in 34,950 out of 5,990,197 safe functions (0.6%), and in 1,266 of 580,342 closure declarations (0.2%). Overall, 7.2% of all crates have unsafe code that dereferences at least one raw pointer.

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 sizable, yet rather low. 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 (which is not part of our dataset) and, thus, do not often occur 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.

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.

Incomplete Incompleteness Issues. Since incompleteness cases cannot be precisely identified automatically, we 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.
Table 3. Reasons why blocks and functions need to be declared unsafe, aggregated on the function level.

<table>
<thead>
<tr>
<th>Reason</th>
<th>#functions</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>call to unsafe function</td>
<td>431,926</td>
<td>85.4</td>
</tr>
<tr>
<td>dereference of raw pointer</td>
<td>44,974</td>
<td>8.9</td>
</tr>
<tr>
<td>use of mutable static variable</td>
<td>25,971</td>
<td>5.1</td>
</tr>
<tr>
<td>access to union field</td>
<td>1,421</td>
<td>0.3</td>
</tr>
<tr>
<td>use of extern static variable</td>
<td>547</td>
<td>0.09</td>
</tr>
<tr>
<td>use of inline assembly</td>
<td>480</td>
<td>0.09</td>
</tr>
<tr>
<td>borrow of packed field</td>
<td>318</td>
<td>0.06</td>
</tr>
<tr>
<td>initializing type with rustc_layout_scalar_valid_range attr</td>
<td>41</td>
<td>0.01</td>
</tr>
<tr>
<td>assignment to non-Copy union field</td>
<td>3</td>
<td>0.00</td>
</tr>
<tr>
<td>pointer operation (in a \texttt{const} function)</td>
<td>2</td>
<td>0.00</td>
</tr>
<tr>
<td>cast of pointer to int (in a \texttt{const} function)</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>borrow of layout-constrained field with interior mutability</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>mutation of layout-constrained field</td>
<td>0</td>
<td>0.00</td>
</tr>
</tbody>
</table>

of the type checker. We found that 27,661 out of 649,473 unsafe blocks (4.3%) call a \texttt{transmute} function, and that 4.6% of all crates contain at least one call to a \texttt{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 \texttt{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 \texttt{transmute}. After manual inspection, we found these crates to contain code generated by scripts, which explains the anomaly.

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

We found 1,084 unsafe trait declarations, which amounts to only 2.5% of all trait declarations. This number is likely too high due to possibly overcounting of re-exported 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 almost 41% of all unsafe trait declarations.

Regarding unsafe functions, our experiments yield that 36% 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 \texttt{unsafe}—in contrast to other unsafe features. Rather, a programmer has to intentionally type an additional keyword. Hence, it seems that developers frequently employ unsafe functions for documentation purposes.

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 the 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 numbers suggest.

Overall, there is no clear evidence that unsafe functions are frequently used for documenting contracts and invariants. One possible reason is that contracts are not enforced by the compiler.

ConcURRENCY through Compiler Intrinsics. To measure to what extent unsafe code is 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: 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. Even if the crate is under active development and can be used only with a nightly version of the compiler, we could not ignore the existence of these dangerous functions and will report them to the developers.

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 1,439,653 definitions of structures and enumerations, we found that only 3.9% are annotated with #[repr(C)]. This annotation is used in 6.5% 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 and provide public bindings to a C system library. We found 635 crates (2.1% of all crates) whose name end with -sys, but we also noticed that other crates use different naming conventions: for 23 crates the name ends with -ffi, for 13 with -bindings, and for 8 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 like libc, g1, and winapi that provide bindings to system libraries without actually 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 242,650 (44.3%) out of 546,705 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, 1,353 crates (4.5% 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 480 cases of functions that use assembly. In particular, we found that 10 low-level and hardware-related crates actually contain 75% of all the functions that make use of inline assembly. This strongly suggests that inline assembly is in general rarely used.
Regarding our anticipated usages of unsafe code to improve performance, we found that 7.4% of all calls to unsafe functions involve unchecked functions spread across 5.8% of all crates. Avoiding runtime 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 \texttt{MaybeUninit}, which allows declaring uninitialised variables: We detected it in only 1,807 unsafe blocks, which appear in 0.56% of all crates.

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 contains 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 further stages of manual inspection paint a clearer picture: 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. \texttt{-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 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 contributors 2019] to warn of potentially-risky visibility of unsafe functions.

\(\text{Vol. 1, No. 1, Article . Publication date: May 2020.}\)
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 85% 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 prioritize. This percentage climbs to 95% if we additionally consider raw pointer dereferences and to 99% 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 sizable, yet low compared to calls of unsafe functions. It seems that Rust code frequently depends on tree-shaped data structures, which can be implemented in safe Rust, or else employs e.g. standard library abstractions for constructing more-complex reference structures. 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.

Moreover, Rust’s type system and runtime checks rarely seem to concern developers enough to take the risk of circumventing the rules; they rarely (in fewer than 5% 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 concerns beyond those that the Rust compiler chooses to associate with unsafe Rust.

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 adaption 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 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
crates from crates.io we have a substantial slice of the open-source 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.

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 complement 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.

### 8 RELATED WORK

We focus mainly on other studies of Rust code; other references to related work are mentioned throughout the paper.

Evans et al. [2020] perform a quantitative evaluation concerning unsafe code and, in particular, measure how unsafe code permeates call chains and, thus, how much code is transitively dependent on functionality implemented with unsafe code. Their work does not take the Rust hypothesis into account as it does not distinguish between (1) call chains where safe abstractions on the lower layers assume responsibility (by declaring exposed functions as safe) for the Rust guarantees and (2) call chains where unsafety is explicitly exposed to clients (by declaring exposed functions
as unsafe). Interestingly, since their dataset relies on a 16 months older snapshot of the same crates.io repository, we can observe that the percentage of crates containing any unsafe code (a measurement made in both studies) has apparently dropped from 29% to 24.1% (noting though that these comparisons relate figures which were computed by different experimental methods).

Our study additionally evaluates the self-containedness and intended encapsulation of unsafe code written in practice, as well as providing an in-depth analysis of the reasons why programmers employ unsafe code.

While our study includes evaluating the extent to which programmers intend for their unsafe code to be encapsulated from clients, 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 analyzing 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%. 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 noticed 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 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 motivation 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 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 plan for QRATES to be made available to the community as an artefact, 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.

REFERENCES


Steve Klabnik and Carol Nichols. 2019. The Rust Programming Language.
How Do Programmers Use Unsafe Rust?


