# Fortifying the Seams Between C/C++ and Rust: Characterizing Bugs in Interop Tools

XUEMENG CAI, Singapore Management University, Singapore
JIAKUN LIU*, Harbin Institute of Technology, Faculty of Computing, China
CUNYANG LIU, Zhejiang University, China
LINGFENG BAO, Zhejiang University, China
YIJUN YU, The Open University, UK
LINGXIAO JIANG, Singapore Management University, Singapore

Rust has become increasingly popular in recent years due to its safety and high performance. Despite these advantages, Rust projects rarely start from scratch in practice, and many Rust-based systems instead use hybrid programming, where Rust interoperates with existing C/C++ code. To reduce the manual effort involved in this interoperation (interop) process, several interop tools have been proposed to facilitate hybrid programming between Rust and C/C++. However, the challenges and limitations of these tools remain largely unexplored, leaving developers unclear about the future directions and users unclear about the appropriate usage scenarios. To fill the gap, we mined 320 bugs in the popular interop tools, i.e., Bindgen, Cbindgen, and CXX. We observe that the main obstacle is not tool crashes or performance issues, but the inability to generate correct and functional code. The leading types of the failures are generating code that is unfaithful to the original code's intent and generating incomplete code. Memory layout mismatch is the most common cause with Bindgen. Input with complex *struct* is the biggest challenge across tools, followed by the usage with specific tool configuration options for customized binding generation. Even input with primitive data types can be problematic. We also observe a common fix pattern that updates the control flow and condition. Based on our findings, we propose a series of suggestions to developers, users, and future researchers. For example, we suggest the developers validate configuration combinations, provide clearer documentation, and implement default-safe mechanisms to prevent failures related to unexpected tool configuration options.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**.

Additional Key Words and Phrases: Rust, interop, C/C++, empirical study, bug characteristics

## 1 Introduction

Rust has experienced a significant rise in popularity over the past few years due to its combination of strong memory safety and high performance [11–13, 24, 31, 32]. One line of work aims to migrate C/C++ legacy code by translating existing C code to Rust [3, 14, 95]. However, it often produces non-idiomatic and unsafe code with low readability, which is challenging to understand and maintain. Some studies have leveraged large language

*Corresponding authors

Authors' Contact Information: Xuemeng Cai, xuemengcai@smu.edu.sg, Singapore Management University, Singapore; Jiakun Liu, Harbin Institute of Technology, Faculty of Computing, Harbin, China, jiakunliu@hit.edu.cn; Cunyang Liu, cunyangliu@zju.edu.cn, Zhejiang University, Hangzhou, China; Lingfeng Bao, lingfengbao@zju.edu.cn, Zhejiang University, Hangzhou, China; Yijun Yu, y.yu@open.ac.uk, The Open University, UK; Lingxiao Jiang, lxjiang@smu.edu.sg, Singapore Management University, Singapore.


ACM 1557-7392/2026/2-ART
https://doi.org/10.1145/3795532

models (LLMs) for translation [4, 15, 22, 30], showing promise in handling diverse language features in C and Rust. However, the translated code still exhibits errors, especially in large-scale projects.

In practice, rather than directly translating existing C/C++ code to Rust, many Rust-based systems adopt hybrid programming [89], where Rust interoperates with existing C/C++ components. For example, Amazon Firecracker [2] and Servo [28] use hybrid programming between C/C++ and Rust to leverage well-established C/C++ libraries, functions, and codebases. To enable interoperation (interop) between Rust and C/C++, Foreign Function Interface (FFI) binding code is often required. Given the fundamental differences between the languages, writing FFI binding code manually often demands huge, tedious, and error-prone effort. For example, developers may need to ensure Application Binary Interface (ABI) compatibility and convert C++ templates, inheritance, structs, enums, etc., into Rust equivalents.

To facilitate interop and binding code generation between Rust and C/C++, a series of tools have been made available, e.g., Bindgen [53], Cbindgen [67], CXX [79]. Considering the rise of Rust, the potential of using interop tools appears promising. However, the challenges and limitations of these tools remain unclear. Understanding existing interop tools can shed light on their future directions for developers and offer practical and effective usage guidance to users.

To close this gap, we comprehensively analyze all the bugs of the interop tools. In total, we collected 320 bug reports and 148 associated pull requests or direct commits from GitHub of the three tools. Qualitative analyses are conducted with reference to these reports and additional information collected by reproducing available test cases. Given that we are the first work that focuses on the challenges and limitations of Rust and C/C++ interop tools, we would like to explore how these interop tools manifest their challenges and limitations. Therefore, we first ask:

**RQ1: What are the symptoms of the failures?** We observe five types of symptoms, i.e., crash, hang, generating an uncompileable code, generating a functionality error code, and performance degradation. The main obstacle is not tool crashes or performance issues, but the inability to generate correct and functional code.

Then, to understand the limitations of the tools that lead to the failures, we ask:

**RQ2: What causes these symptoms?** We identify 11 causes of the symptoms. Generating code that faithfully preserves the original code's intent and generating complete code are the most challenging across tools. Memory layout mismatch is the most common cause in Bindgen.

Further, to understand what kind of inputs and configurations are challenging for tools, we ask:

**RQ3: What triggers the failures?** We identify 25 types of triggers with more than one occurrence. Complex *struct* is the biggest challenge across tools, followed by tool configuration options for customizing the generated bindings. Surprisingly, we observe that even primitive data types can be problematic.

Finally, considering the distinct characteristics of the Rust language, we would like to understand whether there is a new fixing strategy for Rust-related failures. Therefore, we ask:

**RQ4: What are the common fix patterns?** We identify 10 fix patterns with more than one occurrence, largely overlapping with what were reported [21, 23, 27]. Fixes to conditions and updates to control flows are common across tools.

Based on our findings, we provide a series of suggestions to developers, users, and future researchers. For example, we observe that tool configuration options can trigger failures. We suggest that developers validate the combinations of different configurations, provide clearer documentation, and implement default-safe behaviors to prevent failures due to unexpected configurations. For users, we observe that over 80% of the reported issues involve generated code that fails to compile or exhibits incorrect functionality. This indicates that users still need to validate the generated code carefully with various supporting tools and test cases.

The remainder of this paper is organized as follows. We describe the background knowledge related to the interop between Rust and C/C++ in Section 2. The details of our methodology are presented in Section 3. We answer the research questions in Section 4. The discussions related to the implications of the study and the threats

```rust
// Declare an external C function (main.rs)
extern "C" { fn add(x: i32, y: i32) -> i32; }

// Call the C function in an unsafe block (main.rs)
unsafe { let result = add(2, 3); }
```

(a) Rust source files (main.rs)

```c
// Declare the function in a C header (add.h)
int add(int x, int y);

// Define the function in a C source file (add.c)
int add(int x, int y) { return x + y; }
```

(b) C source files (add.h & add.c)

Fig. 1. Calling a C function from Rust

```rust
// Expose a Rust function to be called from C (lib.rs)
#[no mangle]
pub extern "C" fn multiply(x: i32, y: i32) -> i32 { x * y }
```

(a) Rust source files (lib.rs)

```c
// Declare the Rust function for use in C (multiply.h)
int multiply(int x, int y);

// Call the Rust function from C (main.c)
#include "multiply.h"
int result = multiply(4, 5);
```

(b) C source files (main.c & multiply.h)

Fig. 2. Calling a Rust function from C



to validity are in Section 5. Closely related papers are briefly discussed in Section 6. Finally, we conclude our paper in Section 7.

## 2 Background

### 2.1 Interop between Rust and C

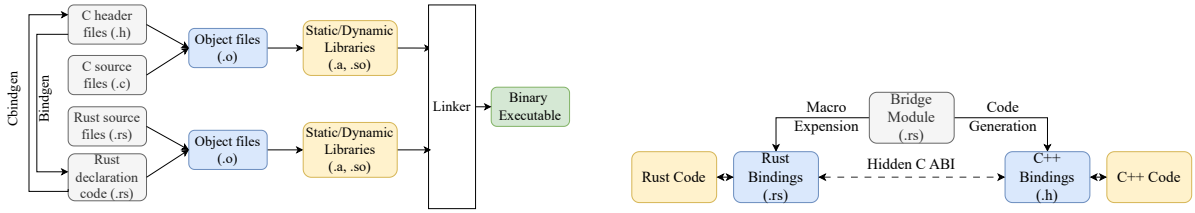
To integrate Rust with existing C codebases or system libraries, Rust provides a Foreign Function Interface (FFI) [86] that allows developers to call C functions from Rust code.

To call C functions from Rust, developers should declare the foreign function using the `extern "C"` keyword and provide appropriate type mappings in Rust. Figure 1 shows an example of calling a C function from Rust. Such function calls are marked as `unsafe`, indicating that such function calls lack compiler guarantees for memory safety and that their safety instead relies on the correctness of the underlying C and Rust implementations. On the C side, the header and source files are compiled into object files, which are then linked with the Rust crate either statically or dynamically to produce the final executable.

To call Rust functions from C, the Rust function needs to be declared with `#[no_mangle]` and `extern "C"` keywords to ensure a stable, unmangled symbol name and a C-compatible calling convention. Figure 2a shows an example of calling a Rust function from C. The exposed functions should be declared in a corresponding C header file (see 2b). Then, the Rust crate is compiled as a static or dynamic library, which can be linked into the C project during compilation.

### 2.2 Interop between Rust and C++

Due to C++ features such as function overloading, class hierarchies, and templates that are not directly compatible with Rust's FFI mechanism, interoperating between Rust and C++ is more challenging. As a result, direct interaction with C++ functions and types using `extern "C"` is only feasible when the C++ code exposes a C-compatible interface. Therefore, more work is needed for developers to achieve hybrid programming between Rust and C++.




(a) The overall pipeline of interop between Rust and C. (b) The workflow for Rust and C++ interop using CXX

Fig. 3. Workflows for interoperability between Rust and C/C++.



## 2.3 Interop Tools

To help developers interop more C/C++ features, Bindgen, CXX, and Cbindgen are proposed and widely used in academia and industry to provide an idiomatic mechanism for calling C++ code from Rust and vice versa. Figure 3a shows the workflow of Bindgen and Cbindgen, and Figure 3b shows the workflow of CXX.

**Bindgen** [53] is a **translation-based** tool that automatically generates Rust bindings from C or C++ header files by leveraging *libclang* to parse the source declarations. It translates function signatures and data structures into their Rust equivalents, which can then be used to call native C/C++ functions from Rust code via FFI. As of Dec. 2025, the Bindgen crate has been downloaded on the order of 200 million times from crates.io (approximately 209M downloads) [1], indicating that it is one of the most widely used Rust & C/C++ interop tools in practice.

**Cbindgen** [67] is another **translation-based** tool that complements Bindgen by automatically generating C or C++ headers for Rust libraries that expose a public C API, leveraging the Rust syntax tree. As of Dec. 2025, the Cbindgen crate has roughly 66M downloads on crates.io [1], fewer than Bindgen but still on the scale of tens of millions, indicating widespread adoption for integrating Rust libraries into existing C/C++ codebases.

**CXX** claims to provide a safe mechanism for calling C++ code from Rust and vice versa in the form of a **bridge-based** tool. As the most recently introduced of the three tools we study, it has nevertheless already seen substantial adoption: as of Dec. 2025, the CXX crate has roughly 49M downloads on crates.io [1]. Using CXX, users need to declare which Rust types and functions should be exposed to C++, and which C++ APIs should be made accessible from Rust in a `#[cxx::bridge]` bridge module. This bridge module serves as the central interface specification for safe, bi-directional interop. Then, CXX uses a pair of code generators, namely a Rust macro via `#[cxx::bridge]` and a C++ header generator, to generate the glue code. This includes data type mappings and `extern "C"` declarations on both sides, enabling function calls between Rust and C++ through a hidden C ABI. To support these mappings, besides primitive data types, CXX provides a set of built-in types that serve as bridges when a native equivalent is missing in one language. For example, CXX introduces `UniquePtr<T>` and `SharedPtr<T>` on the Rust side to interoperate with C++'s smart pointer `std::unique_ptr<T>` and `std::shared_ptr<T>`, respectively. Similarly, Rust's `Box<T>` can be exposed to C++ as `rust::Box<T>` via CXX.

## 3 Methodology

Figure 4 demonstrates the overview of our methodology. To investigate the challenges of Rust and C/C++ interop tools, we begin by collecting issues labelled with bugs from GitHub. Then, we perform qualitative analyses using open and axial coding to construct a taxonomy of the failures, focusing on the symptom, cause, trigger, and fix pattern perspectives.

### 3.1 Data Collection

To understand the challenges and limitations of interop tools between C/C++ and Rust, we focus on the issues labeled with the 'bug" tag. These issues are the functionalities promised by the tool but are challenging for developers to implement, which, therefore, are the main subject of our study. Other types of issues are excluded because they are unlikely in the scope of the original design of the tool (e.g., issues labeled with "feature request" or "enhancement"), or related to the unclear documentation of the tools (e.g., issues labeled with "docs").

To do so, we collected all the issues labelled as bugs in the Bindgen, CXX, and Cbindgen when we started our research (February 2025). As a result, we collected 237, 24, and 59 bugs from Bindgen, CXX, and Cbindgen, respectively (320 in total). Table 1 shows the statistics of the collected issues. We present the number of open and closed issues at that time. Specifically, Bindgen, which is recognized as the most well-developed and popular tool, has the greatest number of bug-labeled issues. In contrast, CXX, a relatively newer tool, has fewer bug reports in comparison.

| Tool | #Closed issues | #Open issues | #Total issues |
|---|---|---|---|
| Bindgen | 143 | 94 | 237 |
| Cbindgen | 35 | 24 | 59 |
| CXX | 22 | 2 | 24 |
| **Total** | 120 | 200 | 320 |

Table 1. Statistics of the studied issues for the three tools.

To summarize the fix patterns for these issues, we collected the merged pull requests/direct commits associated with them. To correctly understand the fix patterns, we focused on valid and complete fixes; thus, we excluded the issues that were resolved with more than one pull request (which would imply the complexity of the fixes and contain intermediate incorrect fixes). As a result, we collected the pull requests of 101 issues from Bindgen, 22 from CXX, and 25 from Cbindgen.

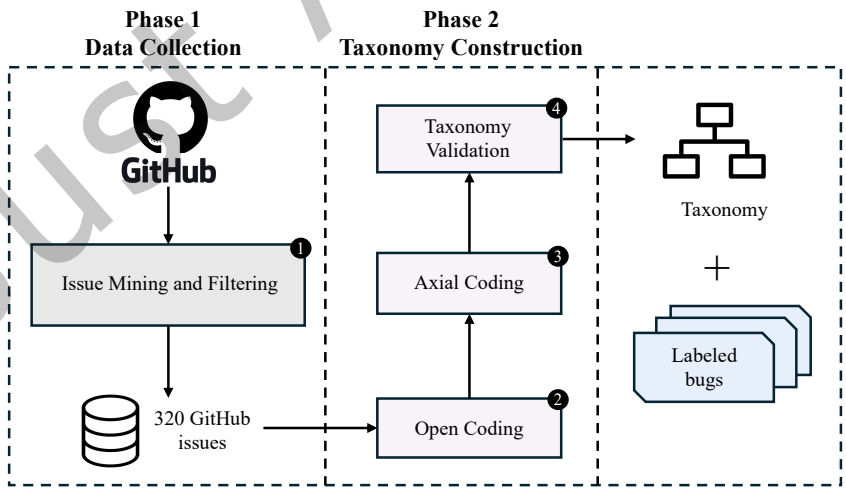



Fig. 4. Overview of our methodology

## 3.2 Taxonomy Construction

To answer our research questions from the introduction, we conduct three iterations of open coding, then an iteration of axial coding to characterize the issues in the studied tools, following prior studies [5, 33, 92, 98, 100]. The open coding phase involves the initial analysis of the issues, where we identify and label the key concepts and categories present in the data. This is followed by axial coding, where we refine and group the identified codes into higher-level categories, establishing a taxonomy that captures the relationships between different concepts. Finally, we validate the taxonomy through a review session with experienced researchers in the field.

*3.2.1 Open Coding.* In the first iteration of the open coding phase, the first two authors of the paper, each with over three years of research experience in software engineering, independently and manually analyzed 50 issues that were randomly sampled from the collected 320 issues. For each bug, each author carefully reviewed the bug description and executed the available test cases using the latest release version of the tool. The goal is not to reproduce the bug, but to determine (i) whether any open issues had been silently fixed and (ii) whether the reported problems in closed issues had regressed in the current codebase [18]. For open issues, executing the available test cases can help authors gain a deeper understanding of the reported behavior and even gather more information from the execution (e.g., detailed compiler error messages explaining why a compilation error occurred, which may not have been included in the original report). For closed issues, executing the available test cases allows authors to verify whether the problem has been resolved and examine the discrepancy between the correct and faulty behavior. Each author drafted a set of initial codes based on their understanding and interpretation of the issues in terms of (1) the symptoms that manifest the issue; (2) the cause(s) of the symptom; (3) the language components of the input or tool features that trigger the issue; and (4) the fix pattern(s) adopted by developers.

For example, **to understand the symptoms of the issues**, for closed issues, we rely on the symptoms explicitly described by the issue reporters. For open issues, we initially adopt the reporter's description as a draft label. When test cases are available, we further re-execute the test case to observe the symptom. As a result, we update the symptoms of two issues. For example, in Bindgen issue #943 [46], the error message in the issue report indicates a symptom of *incorrect output*, as the layout of the struct in the generated binding code does not match the size specified in the input C header. However, due to subsequent version updates, the generated binding code we reproduced triggered a *compilation error*, specifically due to conflicting `packed` and `align` for one struct. Consequently, we revised the symptom label as *compilation error*. In our labeling process, each reported issue is annotated with exactly one symptom category. However, if the available test case passes in our reproduction, we use the symptom described in the issue report, assuming the issue may have been fix silently, and thus has not been confirmed by the authors. Specifically, 20 issues are fix silently. For example, Bindgen issue #2240 [51] reports a compilation error caused by the conflicting memory layout attributes in a generated struct. When we reproduced it with the latest version, the issue no longer occurred, and a follow-up comment noted that "the patch seems to be working" by a user, suggesting it may have been silently fixed.

Similarly, **to understand the causes of the failures**, we annotate the causes of open issues with test cases that allow authors to explore and the closed issues. Specifically, we reference the original issue reports, as well as the structured error codes in the error messages when executing test cases. Rust defines structured error codes [87]. However, we observe that only 26.5% of the Bindgen issues include structured error codes in their error messages. This motivates the further analysis of the causes of the failures.

Finally, a discussion is held to compare the initial codes and reach a consensus on the labels and descriptions of the codes. The initial fine-grained codebook was then refined and expanded based on this discussion.

Then, the first two authors independently applied the initial fine-grained codebook to the remaining 270 issues, which were randomly sampled from the collected 320 issues. Another round of discussion was held to resolve any discrepancies in the coding process and refine the fine-grained codebook further. The Jaccard Index [93]

```
// Issue 1592 in bindgen:
// Extraneous padding in struct with bitfield
// and flexible array member
struct {              //  Struct
       char a : 1;    //  Bitfield
       void *b[];     //  Flexible array member
};
```

Fig. 5. An example of a failure triggered by more than one language component. Specifically, this failure is triggered by a C header input defining a struct with both a bitfield and a flexible array member, leading to unintended padding in the generated Rust bindings.

is used to measure the inter-annotator agreement between the two authors, which is calculated as the size of the intersection divided by the size of the union of the two sets of labels. However, the Jaccard Index for this round of open coding is 0.44, indicating a medium degree of consistency between the two authors. This is because we observe that one issue may involve multiple symptoms, causes, or triggers, and the two authors may have different interpretations of the same issue. Figure 5 shows an example of a failure that is triggered by more than one language component in Bindgen issue #1589 [42]. This failure is triggered by a C header input defining a struct with both a bitfield and a flexible array member, leading to unintended padding in the generated Rust bindings. Therefore, we consider that the issue is simultaneously triggered by three distinct language components: *struct*, *bitfield*, and *flexible array member*.

To address the issue of a single bug being associated with multiple labels, we decided to enable multi-label annotation for the issues, which means that each bug can be assigned multiple labels from the fine-grained codebook. Therefore, we perform the third iteration of open coding, where the first two authors independently applied the refined fine-grained codebook to the all 320 issues. The first two authors then discuss their coding labels and finalize the fine-grained codebook. The Jaccard Index for this round of open coding is 0.87, indicating a high degree of consistency between the two authors.

*3.2.2 Axial Coding.* After establishing the fine-grained codebook, the first two authors collaboratively grouped related fine-grained codes into higher-level conceptual categories through iterative content analysis, discussion, and refinement. For each category, two authors discussed representative examples from the dataset to clarify its scope, resolve ambiguities, and ensure consistent interpretation. As a result of this process, we constructed a draft codebook consisting of three dimensions.

*3.2.3 Validation.* After constructing the draft codebook, the first two authors invited another two authors to conduct a face-to-face review session. Specifically, one has over nine years and the other over fifteen years of experience in software engineering research. During the review sessions, the first two authors presented the draft codebook, including category names, descriptions, and at least one concrete example from the dataset for each category to clarify its scope and practical relevance. Through open discussion and critical feedback from all four authors, the version of our codebook was finalized.

## 4 Findings

### 4.1 Analysis of Issue Symptoms

We identify five categories of symptoms that indicate the failures of the tools, which are common to the symptoms of the bugs in other systems [20, 91, 98, 99]:

- **Crash** refers to the tool aborting before producing complete output on either the Rust or C++ side.

*Example.* In CXX Issue 833 [75], placing the inner attribute `#![deny(missing\_docs)]` inside a `cxx::bridge` module triggers a crash during macro expansion because its implementation mishandled inner attributes.

- **Hang** refers to the tool failing to terminate within a reasonable amount of time when generating bindings.
  *Example.* In Bindgen Issue 1590 [82], a minimal C++ header that uses `alignof` within a template alias triggers infinite recursion in *libclang* during constant evaluation, making Bindgen appear to hang.
- **Generating uncompilable code** refers to the fact that the generated bindings are uncompilable when using Bindgen or Cbindgen as a standalone program, or will cause the project to fail to build when the tool is used as a library within a Rust project.
  *Example.* In Bindgen Issue 1514 [81], Bindgen fails to generate a type parameter to an inner class field, causing a Rust compilation error due to missing generic arguments.
- **Generating functionality error code** refers to the generated bindings that compile successfully but exhibit behavior that deviates from the intended semantics.
  *Example.* In Cbindgen Issue 470[62], Cbindgen generates `typedef uintptr_t (*Function)();`, which omits the explicit `void` and diverging from the intended semantics.
- **Performance degraded** refers to cases where the tool is able to start up or produce bindings, but takes substantially longer than expected as reported by users or developers.
  *Example.* In Bindgen Issue 1532 [80], generating bindings for certain C/C++ headers takes an unexpectedly long time.

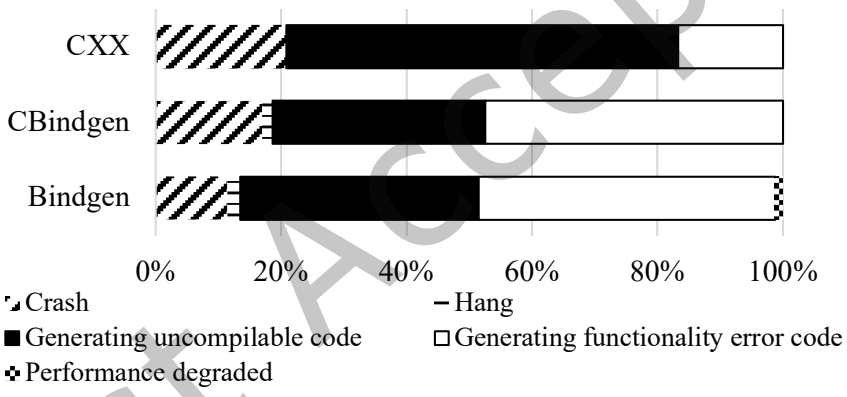



Fig. 6. Distribution of different symptoms across tools.

Figure 6 shows the distribution of different symptoms across tools. **In our bug corpus, crashes account for only a minority of the reported failures.** We observe that around 20% or even fewer failures crash for each tool, while the rest show up as non-crash symptoms such as incorrect bindings, misgenerated code, or compilation errors without a tool crash. We also observe that the overall proportion of crashes in the open issues is 10%, which is lower than that in the closed issues. This shows that developers prioritize the resolution of crashes.

**In our bug corpus, the largest fraction of the reported issues relate to correctness, i.e., producing code that is compilable and functions as intended.** Specifically, among the issues we studied, generating uncompilable code is the dominant issue in CXX, while generating functionally error code is the most common issue in Bindgen and Cbindgen. This indicates that Bindgen and Cbindgen might generate syntactically correct but semantically incorrect bindings. Once the generated code can be compiled, CXX seems more reliable in generating correct behavior than the other two.

Failures in C/C++ and Rust interop tools have the same symptoms in other systems, i.e., crash, hang, generating an uncompilable code, generating a functionality error code, and performance degradation. The main obstacle is not tool crashes or performance issues, but the inability to generate correct and functional code.

## 4.2 Analysis of Cause of the Symptoms

We identify 11 top-level causes and 47 sub-causes that have more than one occurrence, covering 94% of the issues.

- **Unfaithful Code (UFC)** refers to the case where the generated compilable binding is not faithful to the intended design of the original code. For example, the generated function signatures, data types, constant values, identifiers, comments, mutability qualifiers, and namespace mappings are not the same as the original code, e.g., translating the signature of `void a(int arg1)` to `fn a()`.
- **Incomplete Code (INC)** refers to the cases where the generated bindings are structurally or semantically incomplete, e.g., missing definitions of types, functions, constants, and aliases; missing generic arguments for parameterized constructs such as structs or templates; missing lifetime bounds required to correctly express type relationships; and omissions of necessary wrappers or attributes.
- **Tool Behavior Error (TBE)** refers to abnormal or unintended behaviors related to the tool itself, rather than errors related to the input/output. These include misbehaving configuration options that behave unexpectedly or have no effect; unexpected compiler or runtime warnings; and internal errors related to attribute or lifetime handling. Additionally, this category includes internal logic issues in the tool itself, such as infinite loops or unbounded recursion during execution or startup, which usually lead to slow startup or execution times.
- **Memory Error/Mismatch (MEM)** includes issues where the generated bindings introduce errors related to memory.These include size, alignment, and offset mismatches between Rust and C/C++ structures; generation of conflicting `__packed` and `__aligned` attributes, which violate Rust's layout constraints; and overflow errors caused by exceeding representable memory bounds.
- **Compatibility Issue (CPI)** captures problems that arise when the generated bindings or the tool itself fail to work correctly in different environments. Specifically, *Version compatibility issues* refer to incompatibilities caused by specific toolchain versions (e.g., `libclang` in Bindgen). *Platform compatibility issues* arise when the same generated code behaves differently or fails on different operating systems (e.g., Linux, macOS, or Windows). *Target compatibility issues* occur when the tool fails to support or correctly generate bindings for specific compilation targets (e.g., `--target=armv7-unknown-linux-gnueabihf`).
- **Identifier Resolution Error (IDRE)** occurs when the generated bindings fail to resolve or manage identifiers. Specifically, *invalid identifier* refers to the cases where the generated identifier violates Rust naming rules, such as reserved keywords. *Duplicated definition* refers to the cases where the same identifier is defined multiple times. *Alias cycle* occurs when a type alias refers to itself directly or indirectly in the generated code. *Name shadowing error* occurs when a generated local variable unintentionally reuses the name of an existing parameter or field.
- **Tool Runtime Error (TRE)** refers to causes that lead to terminate abruptly during build or execution of generated code or tool itself, such as panics and segmentation faults. A *panic* occurs when the tool encounters an unrecoverable internal error and terminates abruptly. A *segmentation fault* indicates a low-level memory access violation that causes the tool to crash during execution.
- **Derived Traits Error (DTE)** refers to cases where the tool fails to handle trait derivation during binding generation. Traits in Rust define shared behavior for types and enable structured interoperability. Many common traits, such as `Copy`, `Clone`, and `Debug`, can be automatically derived by the compiler, reducing boilerplate and improving code maintainability, known as derived traits. For instance, using `#[derive(Copy, Clone)]` on a simple struct like `struct Point { x: i32, y: i32 }` allows it to be copied implicitly without manual implementation. Bindgen supports the generation of derived traits on demand when producing Rust bindings

from C/C++ structures, while CBindgen emits corresponding trait-based implementations in C/C++ based on the `#[derive(...)]` annotations present in the original Rust input. However, we observe that the tool fails to generate required `#[derive]` annotations or their corresponding C/C++ implementations (i.e., missing derived traits), or derives traits that are invalid for a given type or its fields (i.e., wrong derived traits).

- **Import Resolution Error (IMRE)** includes issues where the tool fails to resolve external dependencies such as headers, modules, or crates. Specifically, *incorrect import path* occurs when the generated bindings specify a header or module path that cannot be resolved by the compiler. *Missing import statement* refers to cases where the generated bindings omit a required import declaration in C/C++ or Rust, resulting in unresolved references during parsing. *Crate/Module parsing failure* refers to cases where the tool fails to parse a Rust external crate or internal module.
- **FFI-Safety Violation (FSV)** captures issues in the generated bindings that violate Rust's expected safety guarantees at the FFI boundary. This includes *unsafe FFI*, which refers to cases where the generated bindings violate FFI requirements, such as the use of data types that are not FFI-safe, or unsafe initialization and exception handling implementations for built-in types generated by CXX.
- **Invalid Redundant Code (IRC)** refers to cases where the tool generates code containing redundant elements that make the bindings invalid, leading to compilation errors. This category includes *unused type parameters* that introduce uninstantiable generics, *duplicated* `const` *qualifiers* such as `const const int`. We also observe *invalid shim functions*, that are incorrectly emitted for abstract or opaque C++ types, which cannot be constructed or passed by value on the Rust side.

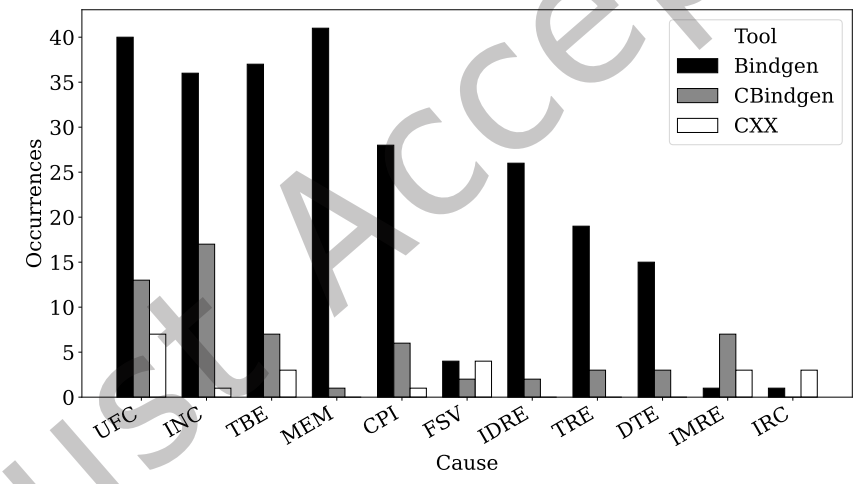



Fig. 7. Distribution of causes across tools

Figure 7 and Figure 8 shows the distribution and normalized distribution of the causes across tools. **Faithfulness and completeness are major bottlenecks.** We observe that 18.8% of issues across tools fail because the tool cannot generate code that is faithful to the design of the original code (i.e., UFC), and 16.9% of the issues across tools fail because the tool cannot generate complete code (i.e., INC). In addition, CXX and Bindgen can also generate invalid redundant code (IRC). This shows that the **tools fall short in understanding and preserving the intended semantics of the original code**.

**Memory layout mismatches account for the highest proportion of Bindgen issues, at 17.3% of all reported Bindgen bugs.** This indicates that Bindgen often fails to correctly infer or preserve the intended

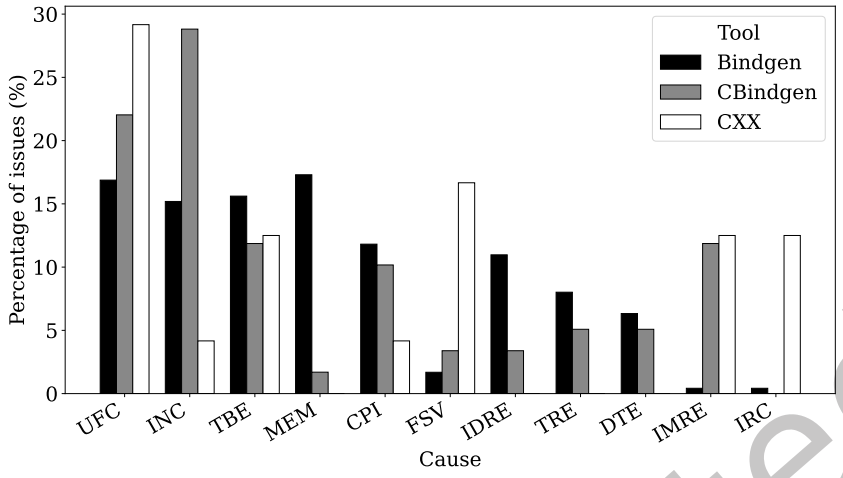



Fig. 8. Normalized distribution of causes across tools.

memory layout (i.e., size, alignment, and field offsets) between C/C++ declarations and their corresponding Rust FFI representations. One possible reason is that tools incorrectly assume how data is laid out in memory. This highlights the lack of deep modeling of data layout, which is often platform and compiler-specific (e.g., padding, ABI, alignment constraints). Additionally, tools may not account for differences in default alignment and packing rules between languages, leading to mismatches when structures are converted. This indicates that the Bindgen's safety modeling and layout annotation inference are flawed.

**Name and reference resolution is a critical weakness.** We observe that Identifier Resolution Errors (IDRE) and Import Resolution Errors (IMRE) stem from difficulties in accurate scoping, naming, and dependency tracking. Common problems include alias cycles, missing imports, name shadowing, or invalid identifiers. These errors suggest that **tools lack sufficient identifier management systems and dependency inference.**

Finally, we note that in our corpus, about 12.5% of issues are associated with more than one cause category. For example, Bindgen Issue #1284 [7] reports that the same header produces different layouts when bindings are generated natively on the Raspberry Pi versus cross-generated on an *x86_64* host. This discrepancy arises because the tool does not consistently use the target's pointer size when computing the layout. This issue is classified under both CPI and MEM: CPI because it stems from target-specific pointer-size and cross-compilation configuration differences, and MEM because the generated *struct* has incorrect size and padding. Together, they are underlying causes of the observed symptom of generating functionally erroneous code. Such cases illustrate that our multi-label classification scheme faithfully reflects the fact that real-world bugs can have multiple triggers and causes.

> We identify 11 causes of the failures of the tools. Generating code that is faithful to the original code's design and generating complete code are the most challenging tasks across tools. Memory layout mismatches are the most common issue with Bindgen.

## 4.3 Analysis of Triggers

Figure 9 shows the distribution of the top ten triggers across Bindgen, Cbindgen, and CXX, respectively. We identify a total of 25 types of triggers with more than one occurrence (covering at least 92% of the collected bugs

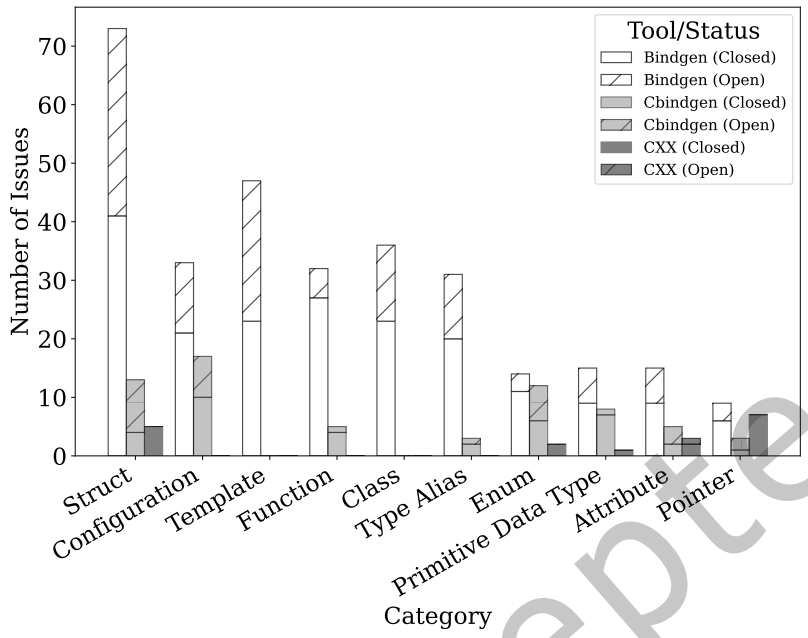



Fig. 9. Distribution of triggers across three tools

for each tool, ensuring that our analysis remains representative while minimizing the influence of corner cases).[1] Moreover, most issues in our corpus have multiple triggers, since a single bug can be simultaneously triggered by several interacting language constructs or configuration patterns. There are five types of triggers that occur in all three tools (can be observed in Figure 9). Due to the different designs of the tools, different tools have different distinct triggers. For example, Bindgen has seven distinct triggers, which is the largest, while Cbindgen only has two distinct types of triggers, which is the fewest among tools. To understand the common challenges in the interop of Rust and C/C++, here, we focus on the top ten triggers in terms of the occurrences across tools (a more detailed discussion on each type of trigger can be found in our replication package). Figure 10 shows the relationship between different symptoms, triggers, and causes.

**Issues involving complex `struct` types account for the largest portion of reported problems across all tools**, accounting for 30.8%, 22.0%, and 20.8% issues in Bindgen, Cbindgen, and CXX, respectively. In C or C++, developers can declare basic types in a struct, and even bitfields, flexible array members, pointers, nested structs, unions, and alignment control attributes (e.g., `__attribute__((packed))`. This indicates that a structure can be customized and complex, which is challenging for tools.

To use the struct, translation-based interop tools (i.e., Bindgen and Cbindgen) generate binding code in the target language by fully translating input headers in the original language. However, we observe that Cbindgen (8 out of 13) commonly fails due to the oversight of certain types in the struct or attributes associated with it, such as generic arguments (e.g., issue #746 [65]) and Rust annotations `#[repr(transparent)]` (e.g., issue #690 [64]) (i.e.,

[1]These triggers are: Class, Target Platform, Primitive Datatype, Constant, Macro, Vector, Function, Type Alias, Dependent Crate, Array, Lifetime, Template, Tool Configuration Option, Enum, Union, Bitfield, Builtin type, Comment, Namespace, Pointer, Attribute, Reference, Module Path, Struct, Flexible Array Member. A more detailed discussion on each type of trigger can be found in our replication package.

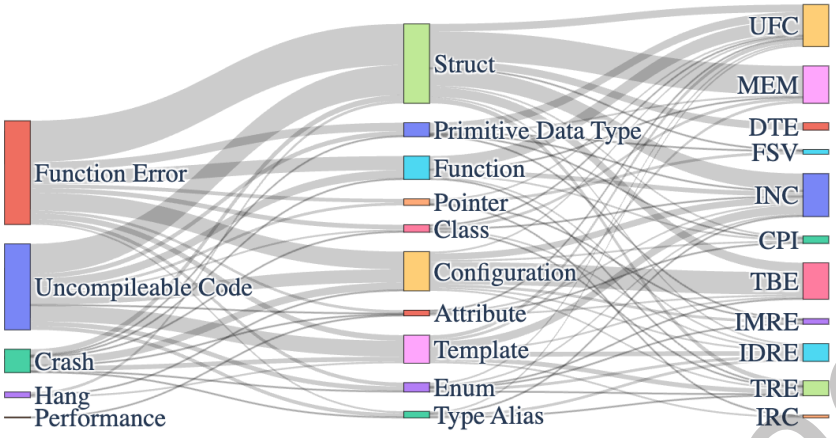



Fig. 10. Sankey diagram between symptoms, triggers, and causes.

INC). In addition, Bindgen (26 out of 75) commonly fails with structs that have bitfields, flexible array members, nested structs, unions, and alignment control attributes (e.g., `__attribute__((packed))`) because the memory layout issues (including size, alignment, and offset) between structures in different programming languages are not correctly resolved (e.g., issue #1947 [49]) (i.e., MEM).

Different from translation-based tools, bridge-based interop tools (i.e., CXX) devise built-in types to facilitate the mapping of complex data types between Rust and C++, such as implementing `rust::Vec<T>` in C++ to map with `Vec<T>` in Rust, and `rust::String` to `String`. CXX (3 out of 5) commonly fails where structs containing built-in types are not properly mapped or declared across the language boundary. For example, in issue #277 [74], a user defines a struct containing a `Vec<T>` field, but the generated C++ implementation of `rust::Vec<T>` lacks required declarations for internal runtime support types, resulting in compilation errors (i.e., INC).

**In our bug corpus, issues related to tool configuration options are the second most frequent trigger category in both Bindgen and Cbindgen**. Developers can customize the generated bindings in Bindgen and Cbindgen with different API methods, command-line flags, comment annotations, or configuration files. However, we observe that these tools can fail on certain options. In Bindgen, the most error-prone options are *blocklist/allowlist* (12 out of 33) and *opaque-type* (4 out of 33). The *blocklist* and *allowlist* options control which types, functions, and global variables Bindgen should include or exclude in the generated bindings. Incorrect or fragile implementation of these options by the tool can lead to the unintended inclusion of blocklisted code or the omission of necessary definitions, resulting in incomplete or incorrect bindings (e.g., issue #1774 [50]). The *opaque-type* option determines whether types that are not translatable by Bindgen should be represented as opaque blobs of bytes with a size and alignment. While opaque types are expected to suppress the generation of inner type details, their interaction with the *allowlist* option can lead to missing type definition errors, as the tool may still emit references to internal members that were represented as opaque blobs (e.g.,#801 [38]).

For Cbindgen, the most common option that triggers failures is *parse_deps* (5 out of 17), which controls whether the tool should parse dependent crates and include their types in the generated output. However, this option can lead to parsing failures, as the internal parsing algorithm may fail to detect or resolve all transitive dependencies (e.g., #381 [54]).

**Issues related to *template*, *function*, *class*, and *type alias* are frequently occurred in Bindgen, and issues related to *enum* and *attribute* are common across tools**. This shows the specific limitations of tools on these language entities. For example, *function*-related issues can be triggered by various aspects, including

function arguments, return types, and identifier resolution or variadic functions in handling corner cases such as functions returning other functions (e.g., Bindgen issue #2713 [52]), and function bindings may be omitted under certain target-specific configurations (e.g., Bindgen issue #1941 [45]). One possible reason is that supporting these may exceed the capability of *libclang* used in Bindgen.

**Even primitive data types can be problematic**. To use *primitive data type*, all tools need to map primitive data types between C/C++ and Rust. However, we observe that in Bindgen, primitive data types that trigger such issues are more diverse, e.g., signed and unsigned integers of various widths (e.g., `uint64_t`, `size_t`) and floating-point types (e.g., `double`), especially for types like `size_t` whose exact mapping to Rust may vary across platforms and compilers. One interesting fact is that 6 out of the 8 *primitive data type*-related issues in Cbindgen are explicitly related to the basic `char` type, typically due to mismatches between Rust's `char` and C/C++ character or wide-character types on specific platforms (e.g., `wchar_t` on Windows [8]). For CXX, we only observed one reported issue related to `uint64_t` in our bug corpus.

> Complex struct is the biggest challenge across tools, followed by tool configuration options for customization. Even primitive data types can be problematic.

### 4.4 RQ4. Fixing Pattern

Table 2 shows the identified fix patterns with more than one occurrence, which cover 76.2%, 76.0%, and 50.0% solved issues for Bindgen, Cbindgen, and CXX, respectively.

**The fix patterns we identified largely overlap with those reported in prior work** [21, 23, 27], such as inserting conditional branches and reordering code blocks. This suggests that, despite the significant differences between Rust and other languages in diverse aspects (e.g., type systems, memory models, and syntax), the repair strategies required for fixing issues in interop tools are similar. One possible reason is that many of these issues stem from general programming constructs and control-flow logic errors rather than language-specific semantics. For instance, binding tools often need to generate conditionals, wrapper functions, or control guards to manage cross-language data access, error handling, or safety checks. Regardless of the target language, repairs need to adjust logic conditions, wrap method calls, or reorder code blocks to restore intended behavior. These fix patterns reflect a need for semantic correctness at the program logic level, which transcends specific language syntax. This observation implies that repair techniques for language binding tools may be broadly transferable and reusable across ecosystems, laying a foundation for building language-agnostic automated repair frameworks or training general-purpose models that can handle multilingual software repair tasks.

**Changing logic and implementing related code is the most common.** This is because developers need to handle various corner cases. For example, **the mutate match arm (inserting new arms, deleting redundant ones, or updating the statements within specific patterns) is the most common fix pattern across tools.** It is widely used in scenarios where the original match logic fails to account for all possible cases. For example, developers may add new arms to support language keywords overlooked in earlier implementations (e.g., `"dyn"`), enable handling of compiler-version-gated language features (e.g., `repr(packed(N))` introduced in Rust 1.33), or support for additional enum variants in `TypeKind` or `CursorKind`, enabling more accurate type recognition and mapping. `TypeKind` is an internal representation used to classify C/C++ types (e.g., structs, enums, pointers) during parsing. For example, in Bindgen PR #1592 [41], the developers added a dedicated match arm to handle zero-length arrays (`TypeKind::Array`) with a custom layout computation, ensuring that such constructs receive correct alignment information instead of being left unhandled by the default case. Without this explicit handling, the layout inference would yield `None`, resulting in missing or incorrect layout information for the type. By mutating match arms to explicitly handle these cases, developers ensure semantic completeness, improve tool robustness, and maintain compatibility with evolving language specifications. This fix pattern appears frequently

Table 2. Fix patterns across tools. The rightmost column shows the distributions of the proportion of the fixed pattern among all fixed issues of the tool. ■ for BINDGEN, ■ for CBINDGEN, and □ for CXX. Identifiers in the "Example" column correspond to representative pull requests or commits cited in this paper.

| Fix Patterns | Code Change Pattern | Description | Example | Distribution |
|---|---|---|---|---|
| **Insert Method Argument** | `- fn mtd1(arg1)`<br>`- x = mtd1(arg1)`<br>`+ fn mtd1(arg1, arg2)`<br>`+ x = mtd1(arg1, arg2)` | The method definition or invocation is modified by adding one or more parameters, typically to support new data requirements or extended functionality. | BINDGEN PR 1531 [44]<br>CBINDGEN PR 473 [60]<br>CXX PR 94 [71] |  |
| **Mutate Method Chain** | `- x.mtd1().mtd2()`<br>`+ x.mtd1().mtd3().mtd2() // insert`<br>`+ x.mtd1() // delete`<br>`+ x.mtd3().mtd2() // replace`<br>`+ x.mtd2().mtd1() // reorder` | A method chain is modified by inserting, removing, replacing, or reordering method calls to correct logic or adapt to API changes. | BINDGEN Commit bce1330 [39]<br>CBINDGEN PR 454 [63]<br>CXX PR 806 [76] |  |
| **Insert Wrapping Method** | `- x = stmt1`<br>`+ x = mtd1(stmt1)` | An existing statement is wrapped inside a new method call. | BINDGEN PR 1515 [40]<br>CXX PR 828 [77] |  |
| **Mutate Condition** | `- if cond1`<br>`+ if cond2 && cond1`<br>`+ if cond2 \|\| cond1` | A condition is mutated by inserting, deleting, or modifying logical subconditions (e.g., via && or \|\|) | BINDGEN PR 1861 [48]<br>CBINDGEN PR 1006 [66]<br>CXX PR 443 [78] |  |
| **Insert Match Block** | `- if (cond1){ ... } // optional`<br>`+ match value1 {`<br>`+  Pattern1 => stmt1,`<br>`+  ..., }` | A new `match` block is introduced to implement pattern-based control flow. In some cases, it replaces an existing `if-else` chain to enable more structured branching logic. | BINDGEN PR 926 [37]<br>CBINDGEN PR 321 [55]<br>CXX PR 311 [72] |  |
| **Mutate Match Arm** | `match value1 {`<br>`  Pattern1 => stmt1,`<br>`- Pattern2 => stmt2,`<br>`  }` | A match expression is modified by inserting new arms, deleting existing ones, or updating the statements associated with specific patterns. | BINDGEN PR 1678 [43]<br>CBINDGEN PR 396 [59]<br>CXX PR 263 [73] |  |
| **Insert `if-else` block** | 1. Insert conditional block without return<br>`+ if (cond1) { stmt1 }`<br>`+ else { stmt2 } // optional`<br>2. Insert guard clause with early return<br>`+ if (cond1) {`<br>`+  return value1`<br>`+ }` | An `if-else` block is inserted to introduce conditional execution. This includes (1) conditional blocks that execute statements when a condition is met, and (2) guard clauses that enforce early return when the condition holds. | BINDGEN PR 1775 [47]<br>CBINDGEN PR 540 [61]<br>CXX PR 585 [70] |  |
| **Insert Wrapping `if-else`** | `+ if (cond1) {`<br>`+  stmt1`<br>`+ }`<br>`+ else { stmt2 } // optional` | An existing statement is wrapped within an `if` or `else` block to constrain its execution to specific runtime scenarios. | BINDGEN PR 1081 [36]<br>CBINDGEN Commit 60d60aa [56]<br>CXX PR 311 [72] |  |
| **Insert Conditional Branch** | `- if (cond1){ stmt1 }`<br>`+ if (cond2) { stmt2 }`<br>`+ else if(cond1){ stmt1 }` | A new conditional branch is added (e.g., `if`, `else if`, or `else`) to handle an additional case. | BINDGEN PR 1094 [34]<br>CBINDGEN PR 371 [58] |  |
| **Reorder Code Block** | `- stmt1`<br>`// other statements`<br>`+ stmt1` | An existing code block is moved to a different position to change its execution order | BINDGEN PR 808 [35]<br>CBINDGEN PR 355 [57] |  |

in our dataset, likely **due to the extensive use of `match` expressions in Rust**, especially in interop tools that heavily rely on variant-based dispatch for parsing and code generation. However, less frequent or semantically subtle cases are sometimes overlooked during initial implementation. These omissions manifest when the tools are applied to diverse, real-world C/C++ codebases where such edge cases occur.

> The fix patterns we identified largely overlap with those reported in prior work, and control-flow and condition fixes are common across tools.

# 5 Discussion

## 5.1 Implications for developers of interop tools

In Section 4.2, we observe that many reported issues involve generating bindings that either do not compile or do not behave as intended, i.e., the generated code is not faithful to the original intent of the C/C++ interface. One possible factor is the difficulty of ensuring precise semantic alignment and accurate cross-language mappings between Rust and C/C++. We suggest that **future developers consider improving datatype system alignment and on developing more comprehensive models for function and interface representation and mapping**, so that mismatches in data types, data layouts, and function signatures can be detected and handled more systematically.

In Section 4.2, our study shows that Unfaithful Code (UFC) and Memory Mismatches (MEM) are the two most frequent cause categories for Bindgen. One possible reason is related to Bindgen 's reliance on *libclang*'s abstract syntax trees (AST), which can be incomplete or expose ambiguous or unexposed nodes. For example, in Bindgen, a developer reported that *enum* values in a templated C++ class are not properly retrieved by *libclang*, resulting in incorrect *enum* values in the generated code [85]. Moreover, in Bindgen Issue 2713 [52], developers also mentioned that Bindgen generates wrong function arguments due to some quirks with getting the right function type from *libclang*. Similarly, Bindgen Issues 1894 [83] and 2189 [84] document memory-layout mismatches due to ambiguity in the *libclang* AST, which causes Bindgen to infer incorrect sizes or alignments. These issues illustrate that relying on a single upstream analysis library can be fragile in corner cases.

To address this limitation, we recommend enhancing Bindgen by combining *libclang*'s output with supplementary semantic sources. For example, Bindgen could continue using *libclang* for bulk parsing but invoke a lightweight *Clang* plugin for suspicious or unexposed nodes in the AST. Moreover, large language models (LLMs), which can capture rich contextual patterns in source code, could be leveraged to propose candidate interpretations for missing or incomplete AST elements based on similar patterns in training data. This suggestion should serve as heuristic hints and would need to be checked or confirmed by deterministic analyses before being adopted.

In addition, when ambiguity or uncertainty is detected, the tool could present multiple binding options (e.g. class method pointer versus function pointer in Issue 1894 [83]), or at least provide explicit warnings to the user, rather than silently producing potentially incorrect code, which is especially risky in practice when thousands of lines of bindings are produced and hard to verify manually.

In Section 4.2, we also identify several recurring subcategories of causes, such as attribute conflicts (`#[repr(packed)]` vs `#[repr(align)]`), invalid identifiers, and duplicate definitions, that can be addressed systematically through a post-processing checker or repair system. As these issues involve syntactic or structural patterns that are easy to detect but hard to intercept during the initial AST traversal, the tool can detect and automatically fix these issues using predefined rules. Moreover, **integrating test generation and verification techniques** can help ensure the correctness and faithfulness of generated bindings.

In Section 4.2, we observe that Derived Traits Error (DTE) and FFI-Safety Violation (FSV) can cause failures of tools. One possible reason is that Rust's stricter safety and trait systems introduce additional complexity. For example, tools may generate invalid `#[derive(...)]` annotations, or non-FFI-safe types such as `u128`. We suggest

that **tool implementations explicitly account for Rust's core principles, such as Rust's trait system, FFI guarantees, and ownership rules, beyond syntax-level generation.**

In Section 4.2, we observe that tools can also fail because of Identifier Resolution Errors (IDRE) and Import Resolution Errors (IMRE). One possible reason is that tools lack sufficiently robust identifier management and dependency inference. These findings suggest that **tool implementations could benefit from leveraging static analysis and compiler integration to infer and name references more comprehensively during binding generation**, for example, by tracking symbol scopes and module paths explicitly and validating that all referenced identifiers are resolvable in the generated bindings.

In Section 4.2, we observe that tools can fail because of compatibility issues. In Section 4.3, we observe that tools can fail because the same primitive data types can have different sizes on different platforms (e.g., usize). This indicates that developers need to do platform-specific debugging to preserve the usefulness of interoperation tools in cross-platform projects. The tools fail to "write once, bind anywhere". We suggest that **tools dynamically resolve platform-specific data type mapping to generate portable bindings.**

In Section 4.3, we observe that tool options for customization can trigger issues. We suggest that **tools include robust validation of configuration combinations, clearer documentation, and default-safe behaviors** to prevent these configuration options from unexpected behavior, especially when multiple options are used together.

## 5.2 Implications for future researchers

Our taxonomy are well-defined and supported by extensive and verifiable examples and test cases, which can support the automatic classification of newly reported issues and serve as a foundation for fine-tuning, few-shot learning, or prompt engineering in LLM-based systems. In addition, our dataset can serve as a benchmark for evaluating future interoperability tools and as training data for fine-tuning LLMs on at least two key tasks: (1) generating correct bindings from C/C++ headers, and (2) automatically repairing incorrect outputs produced by interoperability tools.

In Section 4.3, we identify common triggers that are prone to binding errors, along with concrete examples. For instance, we characterize complex *structs* that are prone to layout mismatches, such as those with bitfields and nested anonymous unions. These data can be used to predict the risk of incorrect bindings and to proactively alert users before binding generation.

In Section 4.4, we identify 10 fix patterns, which are similar to the existing fix patterns reported in prior studies, e.g., Insert Method Argument or Mutate Method Chain. Considering the promises of existing automatic program repair approaches, we see **an opportunity to train models or design repair techniques on these patterns for automate repair.** For example, the summarized patterns could serve as a patch vocabulary for LLM fine-tuning or rule-based patching engines.

Finally, our findings also suggest architectural directions for future interop tools. In Section 5.1, we mentioned several Bindgen bugs we analyzed stem from limitations or quirks in upstream components such as *libclang*. We therefore recommend designing interop tools with explicit awareness of the limitations and failure modes of their external dependencies. For example, future tools can validate or cross-check the outputs of analysis libraries and design recovery strategies or diagnostics for cases where the underlying dependency produces incomplete, ambiguous, or inconsistent information.

## 5.3 Implications for users of interop tools

In Section 4.1, we observe that in our bug corpus for each tool, only around 20% of the reported issues involve tool crashes, whereas a large fraction is about producing code that does not compile or function as intended. This suggests that **although users can expect the tools to run without crashing most of the time, they still**

**need to validate the generated code carefully using compilation tooling, tests, or code review, as code that compiles is not guaranteed to be functionally correct.** In particular, users should be especially cautious when the input contains triggers that are error-prone (e.g., complex `structs`, mixed configuration options, or platform-specific types whose size or layout differs across 32-bit and 64-bit targets, as shown in our study), and treat bindings involving these patterns as higher-risk code that warrants extra checks.

### 5.4 Threats To Validity

*5.4.1 Internal Validity.* Our study may be affected by manual labeling errors and subjective biases. To mitigate this threat, two authors with expertise in code analysis analyzed the issues independently and conducted multiple rounds of discussion to resolve discrepancies. We employed established qualitative methods (open and axial coding) to build our taxonomy, following best practices in empirical software engineering. Furthermore, we validated the taxonomy through a review session with two more senior researchers with over nine and fifteen years of experience in software engineering research, respectively, thereby enhancing the reliability of our taxonomy.

*5.4.2 External Validity.* Our study focuses on three widely-used interoperability tools: Bindgen, CXX, and Cbindgen, but they may not fully represent all interop tools or future developments in Rust and C/C++ tooling. To mitigate this threat, we deliberately selected these three tools with different design goals and implementation strategies. This diversity increases the generalizability of our findings.

Our scope focuses on hybrid programming settings where Rust interoperates with C/C++ via FFIs and interop tools. As a result, our taxonomy may not be generalized to other software systems or programming languages. Nevertheless, several types of causes in our taxonomy, for example, platform-specific data type mapping bugs (e.g., 32-bit vs. 64-bit targets), and tool configuration-related bugs, are not specific to Rust-C/C++ and may inform the design and analysis of hybrid-programming tools in other language ecosystems.

Moreover, our dataset only includes GitHub issues explicitly labeled as "bug" by tool developers, which may miss some relevant interoperability problems that were unlabeled. We intentionally focus on these issues because they represent confirmed, developer-verified faults that have been triaged by maintainers, reducing noise from usage misunderstandings or configuration mistakes. To assess mislabeling risk, we inspected a sample of issues that are not labeled for each project and did not find new symptom or cause categories beyond those in our taxonomy so far. While confirmed "bug" issues capture the most urgent and impactful problems, we acknowledge that undocumented failures may exist and our taxonomy may not cover all categories and we plan to incorporate additional labels and data sources in future work.

*5.4.3 Construct Validity.* The studied issues are diverse and complex in nature, which makes consistent categorization challenging, and thus the constructed taxonomy and implications and suggestions for various kinds of interop stakeholders may only be valid for some limited aspects. We may need to consider the interop ecosystem across C/C++, Rust, and even other languages to validate and enhance our study results.

## 6 Related Work

### 6.1 Studies about leveraging existing C/C++ codebases in Rust

Considering the advantages of the Rust language and the large volume of existing C/C++ code base, many researchers focused on translating existing C/C++ code to Rust [4, 10, 14, 15, 17, 22, 30, 95, 97]. For example, C2SafeRust [22] leverages large language models (LLMs) to clean up output produced by C2Rust, improving code readability and reducing unsafe patterns. Moreover, some compiler-based approaches further aim to lift raw C pointers into safer Rust references and owned pointers. For example, Emre et al. propose a compiler-based approach that leverages Rust's borrow checker to turn part of the raw-pointer translated code into safer and

ownership-aware Rust [9], while Zhang et al. propose a technique that infer ownership models of C pointers and automatically translate the pointers into safe Rust equivalents [101]. These techniques mitigate specific unsafety in translated code to some extend. However, despite their effort, the translated code remains buggy and non-idiomatic in general, making it less practical for a large-scale application in the real world.

Crubit [68] is an open-source bidirectional bindings generator for C++ and Rust that aims to integrate the two ecosystems. Its document [69] provides only a high-level analysis of each tool's advantages and limitations. Our study goes further by systematically analyzing bugs across three tools and constructing a well-defined taxonomy supported by extensive, verifiable examples and test cases.

The work by Sharma et al. [29] is the one that is the most similar to ours, i.e., they also focused on the interop between Rust and C/C++. However, different from our work that focuses on characterising the failures of interop tools, Sharma et al. [29] aimed to understand the interop in embedded systems. Though they considered interop tools (i.e., Bindgen and Cbindgen) in their study, they only explored the effectiveness of these tools through several examples. In contrast, we collect and characterize failures of interop tools from real-world reported bugs in widely used projects, providing more practical and realistic implications.

### 6.2 Empirical studies on bug characteristics

Prior studies analyzed the characteristics of bugs in real-world projects across programming languages and domains. [5, 6, 19, 20, 23, 26, 88, 90–92, 94, 98, 99, 102]. For example, Wan et al. [90] characterized the bug reports from eight open-source blockchain systems and identified ten common bug categories. Ren et al. [26] used topic modeling to analyze the bug reports of Ubuntu and identified three general bug categories. Yang and Cai [98] conducted the first comprehensive study of real-world cross-language bugs between Python-C and Java-C, observing that those bugs commonly arise from logic and boundary errors at assignments and function calls, manifesting as incorrect outputs, crashes, or memory leaks.

In the field related to Rust, some studies focus on characterising specific types of Rust bugs, e.g., memory safety, concurrency [16, 25, 96]. For example, Qin et al. [25] focus on memory and thread safety issues in real-world Rust applications. They observe that over 50% of safety issues stem from misuse of unsafe code, while concurrency bugs can involve incorrect use of channels or atomic operations. Yu et al. [100] characterize 790 real-world bugs of Rust programs and identify 15 root causes and six symptoms, revealing that Rust programs encounter diverse bug types beyond memory safety issues.

Unlike prior studies, which focused only on a single programming language or hybrid programming between Python-C and Java-C, our paper focuses on the bugs when interop between Rust and C/C++. Our findings show that a significant portion of bugs are caused by the mismatches between Rust and C/C++, which can only be observed in hybrid programming scenarios. This shows the difference between our work and prior studies, showing the necessity of our work.

## 7 Conclusion

Our study presents the first comprehensive analysis of real-world failures in Rust and C/C++ interop tools. We construct a taxonomy by identifying key symptoms, root causes, triggers, and fix patterns from GitHub issues across three widely-used interop tools. Our findings reveal that the primary challenge lies in generating binding code that is faithful to the original input headers and functionally complete. Failures frequently stem from complex data structure definitions and tool configuration options for customization. Based on our observations, we offer actionable insights for both tool developers and end users. The taxonomy developed from our study can guide the design of more robust and trustworthy interop tools, inform testing strategies for robustness evaluation, and inspire techniques for automated bug detection and repair.

## Data Availability

Our experimental results and dataset are publicly available at: https://github.com/Cxm211/IT1

## Acknowledgments

This research is supported by the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (Award ID: MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

## References

[1] 2025. crates.io: The Rust Community's Crate Registry. https://crates.io/. Official Rust package registry used by Cargo.

[2] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: lightweight virtualization for serverless applications. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'20)*. USENIX Association, USA, 419–434.

[3] C2Rust Developers. 2020. C2Rust: Migrate C code to Rust. https://github.com/immunant/c2rust.

[4] Xuemeng Cai, Jiakun Liu, Xiping Huang, Yijun Yu, Haitao Wu, Chunmiao Li, Bo Wang, Imam Nur Bani Yusuf, and Lingxiao Jiang. 2025. RustMap: Towards Project-Scale C-to-Rust Migration via Program Analysis and LLM. arXiv:2503.17741 [cs.SE] https://arxiv.org/abs/2503.17741

[5] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *J. Syst. Softw.* 152, C (June 2019), 165–181. doi:10.1016/j.jss.2019.03.002

[6] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating systems errors. *SIGOPS Oper. Syst. Rev.* 35, 5 (Oct. 2001), 73–88. doi:10.1145/502059.502042

[7] Bindgen Contributors. 2018. Wrong padding size generated from different arch. https://github.com/rust-lang/rust-bindgen/issues/1284. Accessed: 2025-12-08.

[8] Cbindgen Contributors. 2019. *Rust* `char` *is not* `wchar_t` *e.g. on Windows.* https://github.com/mozilla/cbindgen/issues/373 GitHub issue #373 in the `mozilla/cbindgen` repository.

[9] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating C to Safer Rust. *Proceedings of the ACM on Programming Languages* 5, OOPSLA, Article 121 (Oct. 2021), 121:1–121:29 pages. doi:10.1145/3485498

[10] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2025. Towards Translating Real-World Code with LLMs: A Study of Translating to Rust. arXiv:2405.11514 [cs.SE] https://arxiv.org/abs/2405.11514

[11] Kasra Ferdowsi. 2023. The Usability of Advanced Type Systems: Rust as a Case Study. arXiv:2301.02308 [cs.PL] https://arxiv.org/abs/2301.02308

[12] Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. 2021. Benefits and drawbacks of adopting a secure programming language: rust as a case study. 20 pages.

[13] Jake Goulding. 2020. What is Rust and why is it so popular? https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/

[14] Jaemin Hong and Sukyoung Ryu. 2024. Don't Write, but Return: Replacing Output Parameters with Algebraic Data Types in C-to-Rust Translation. *Proc. ACM Program. Lang.* 8, PLDI, Article 176 (June 2024), 25 pages. doi:10.1145/3656406

[15] Jaemin Hong and Sukyoung Ryu. 2025. Type-migrating C-to-Rust translation using a large language model. *Empirical Software Engineering* 30, 3 (2025). doi:10.1007/s10664-024-10573-2

[16] Oey Kevin Andrian Santoso, Catherine Kwee, William Chua, Ghinaa Zain Nabiilah, and Rojali. 2023. Rust's Memory Safety Model: An Evaluation of Its Effectiveness in Preventing Common Vulnerabilities. *Procedia Comput. Sci.* 227, C (Jan. 2023), 119–127. doi:10.1016/j.procs.2023.10.509

[17] Anirudh Khatry, Robert Zhang, Jia Pan, Ziteng Wang, Qiaochu Chen, Greg Durrett, and Isil Dillig. 2025. CRUST-Bench: A Comprehensive Benchmark for C-to-safe-Rust Transpilation. arXiv:2504.15254 [cs.SE] https://arxiv.org/abs/2504.15254

[18] Kaixuan Li, Jian Zhang, Sen Chen, Han Liu, Yang Liu, and Yixiang Chen. 2024. PatchFinder: A Two-Phase Approach to Security Patch Tracing for Disclosed Vulnerabilities in Open-Source Software. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 590–602. doi:10.1145/3650212.3680305

[19] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now? an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability* (San Jose, California) *(ASID '06)*. Association for Computing Machinery, New York, NY, USA, 25–33.

doi:10.1145/1181309.1181314
[20] Di Liu, Yang Feng, Yanyan Yan, and Baowen Xu. 2023. Towards understanding bugs in Python interpreters. *Empirical Softw. Engg.* 28, 1 (Jan. 2023), 39 pages. doi:10.1007/s10664-022-10239-x
[21] Yunbo Ni, Yang Feng, Zixi Liu, Runtao Chen, and Baowen Xu. 2024. PanicFI: An Infrastructure for Fixing Panic Bugs in Real-World Rust Programs. arXiv:2408.03262 [cs.SE] https://arxiv.org/abs/2408.03262
[22] Vikram Nitin, Rahul Krishna, Luiz Lemos do Valle, and Baishakhi Ray. 2025. C2SaferRust: Transforming C Projects into Safer Rust with NeuroSymbolic Techniques. arXiv:2501.14257 [cs.SE] https://arxiv.org/abs/2501.14257
[23] Kai Pan, Sunghun Kim, and E. James Whitehead Jr. 2009. Toward an understanding of bug fix patterns. *Empirical Software Engineering* 14, 3 (2009), 286–315. doi:10.1007/s10664-008-9077-5
[24] Jeffrey M. Perkel. 2020. Why scientists are turning to Rust. *Nature* 588, 7836 (2020), 185–186. doi:10.1038/d41586-020-03382-2
[25] 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 Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. doi:10.1145/3385412.3386036
[26] Xiaoxue Ren, Qiao Huang, Xin Xia, Zhenchang Xing, Lingfeng Bao, and David Lo. 2018. Characterizing Common and Domain-Specific Package Bugs: A Case Study on Ubuntu. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 01. 426–431. doi:10.1109/COMPSAC.2018.00065
[27] Mohammad Robati Shirzad and Patrick Lam. 2024. A study of common bug fix patterns in Rust. *Empirical Software Engineering* 29, 44 (2024). doi:10.1007/s10664-023-10437-1
[28] Servo Developers. 2023. Servo: A Parallel Browser Engine Project. https://github.com/servo/servo.
[29] Ayushi Sharma, Shashank Sharma, Sai Ritvik Tanksalkar, Santiago Torres-Arias, and Aravind Machiry. 2024. Rust for Embedded Systems: Current State and Open Problems. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 2296–2310. doi:10.1145/3658644.3690275
[30] Momoko Shiraishi and Takahiro Shinagawa. 2024. Context-aware Code Segmentation for C-to-Rust Translation using Large Language Models. arXiv:2409.10506 [cs.SE] https://arxiv.org/abs/2409.10506
[31] Stack Overflow. 2020. Stack Overflow Developer Survey 2020. https://survey.stackoverflow.co/2020#technology-most-loved-dreaded-and-wanted-languages-loved
[32] Stack Overflow. 2020. Why the developers who use Rust love it so much. https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/
[33] Anselm L. Strauss and Juliet Corbin. 2004. Open Coding. In *Social Research Methods: A Reader*. Routledge, 303–306.
[34] The Bindgen Project Contributors. 2017. Fix mistakenly derive hash for struct that contains IncompleteArrayField. https://github.com/rust-lang/rust-bindgen/pull/1094.
[35] The Bindgen Project Contributors. 2017. Fix tracing of opaque types. https://github.com/rust-lang/rust-bindgen/pull/808.
[36] The Bindgen Project Contributors. 2017. Handle unsigned integer constants greater than u32::MAX in codegen. https://github.com/rust-lang/rust-bindgen/pull/1081.
[37] The Bindgen Project Contributors. 2017. ir: When something has a definition, return unresolved type references until we parse it. https://github.com/rust-lang/rust-bindgen/pull/926.
[38] The Bindgen Project Contributors. 2017. Opaque types with static members produce invalid bindings. https://github.com/rust-lang/rust-bindgen/issues/801.
[39] The Bindgen Project Contributors. 2017. Resolve through type refs and aliases when marking type params used. https://github.com/rust-lang/rust-bindgen/commit/bce13307645172be62e8895e82d9362f0c4e4fd6.
[40] The Bindgen Project Contributors. 2019. codegen: Append implicit template parameters to base members. https://github.com/rust-lang/rust-bindgen/pull/1515.
[41] The Bindgen Project Contributors. 2019. ir: Properly find the layout of incomplete arrays. https://github.com/rust-lang/rust-bindgen/pull/1592.
[42] The Bindgen Project Contributors. 2019. Issue #1589: Extraneous padding in struct with bitfield and flexible array member. https://github.com/rust-lang/rust-bindgen/issues/1589.
[43] The Bindgen Project Contributors. 2019. Warn rather than panic on unknown namespace prefix. https://github.com/rust-lang/rust-bindgen/pull/1678.
[44] The Bindgen Project Contributors. 2019. Work around a libclang bug / limitation. https://github.com/rust-lang/rust-bindgen/pull/1531.
[45] The Bindgen Project Contributors. 2020. bindgen does not emit functions for wasm32-unknown-emscripten. https://github.com/rust-lang/rust-bindgen/issues/1941.
[46] The Bindgen Project Contributors. 2020. bindgen issue #943: Padding bytes are not ignored in PartialEq/Eq/Ord/PartialOrd/Hash. https://github.com/rust-lang/rust-bindgen/issues/943. Accessed: 2025-05-29.
[47] The Bindgen Project Contributors. 2020. Do not emit Rust method wrapper for blacklisted functions. https://github.com/rust-lang/rust-bindgen/pull/1775.

[48] The Bindgen Project Contributors. 2020. Fix constness of multidimensional arrays. https://github.com/rust-lang/rust-bindgen/pull/1861.
[49] The Bindgen Project Contributors. 2020. Issue #1947: Wrong layout generated for some structures with complex bit fields. https://github.com/rust-lang/rust-bindgen/issues/1947.
[50] The Bindgen Project Contributors. 2020. Rust method wrapper is still emitted for blacklisted functions. https://github.com/rust-lang/rust-bindgen/issues/1774.
[51] The Bindgen Project Contributors. 2022. Issue #2240: __attribute__((aligned(2), packed)) generates both repr(packed) and repr(aligned). https://github.com/rust-lang/rust-bindgen/issues/2240.
[52] The Bindgen Project Contributors. 2024. Inappropriate generated type for function-returning-function. https://github.com/rust-lang/rust-bindgen/issues/2713.
[53] The Bindgen Project Developers. 2025. rust-bindgen: Automatically generates Rust FFI bindings to C and C++ libraries. https://github.com/rust-lang/rust-bindgen
[54] The Cbindgen Project Contributors. 2019. cbindgen panics parsing sodiumoxide crate. https://github.com/mozilla/cbindgen/issues/381.
[55] The Cbindgen Project Contributors. 2019. Escape chars. https://github.com/mozilla/cbindgen/pull/321.
[56] The Cbindgen Project Contributors. 2019. Fix #254: Correct module resolution in Rust 2018 edition. https://github.com/mozilla/cbindgen/commit/60d60aaf0d8bd290b7fc3d5ceb6284f2157e7cbf.
[57] The Cbindgen Project Contributors. 2019. Fix the crate parsing order. https://github.com/mozilla/cbindgen/pull/355.
[58] The Cbindgen Project Contributors. 2019. Make running cbindgen with own crate expansion from build.rs. https://github.com/mozilla/cbindgen/pull/371.
[59] The Cbindgen Project Contributors. 2019. Map char to char32_t. https://github.com/mozilla/cbindgen/pull/396.
[60] The Cbindgen Project Contributors. 2020. bindgen: enable ”void prototype” while writing struct field. https://github.com/mozilla/cbindgen/pull/473.
[61] The Cbindgen Project Contributors. 2020. Do not accept array as function arguments. https://github.com/mozilla/cbindgen/pull/540.
[62] The cbindgen Project Contributors. 2020. Fix generated typedefs for function types without arguments. https://github.com/mozilla/cbindgen/issues/470
[63] The Cbindgen Project Contributors. 2020. Handle new line in doc attribute. https://github.com/mozilla/cbindgen/pull/454.
[64] The Cbindgen Project Contributors. 2021. Issue #690: #[repr(transparent)] struct with a NonZero field. https://github.com/mozilla/cbindgen/issues/690.
[65] The Cbindgen Project Contributors. 2022. Issue #746: _CBINDGEN_IS_RUNNING should be exposed and documented in some form. https://github.com/mozilla/cbindgen/issues/746.
[66] The Cbindgen Project Contributors. 2024. Fix local override of enum prefix-with-name. https://github.com/mozilla/cbindgen/pull/1006.
[67] The Cbindgen Project Developers. 2025. cbindgen: Generate C/C++ bindings from Rust code. https://github.com/mozilla/cbindgen
[68] The Crubit Project Contributors. 2022. Crubit: C++/Rust interoperability project. https://github.com/google/crubit/tree/main
[69] The Crubit Project Contributors. 2022. High-level design of C++/Rust interop. https://github.com/google/crubit/blob/main/docs/design/design.md
[70] The CXX Project Contributors. 2020. Consider mut receiver for allowing mut C++ return type. https://github.com/dtolnay/cxx/pull/585.
[71] The CXX Project Contributors. 2020. Do not emit UniquePtr::new for opaque C types. https://github.com/dtolnay/cxx/pull/94.
[72] The CXX Project Contributors. 2020. Fix return Result<Box<T» from Rust to C++ [v2]. https://github.com/dtolnay/cxx/pull/311.
[73] The CXX Project Contributors. 2020. Handle &mut reference in more places. https://github.com/dtolnay/cxx/pull/263.
[74] The CXX Project Contributors. 2020. Missing some definitions in generated code involving Vec. https://github.com/dtolnay/cxx/issues/277.
[75] The CXX Project Contributors. 2021. Bridge attribute breaks inner attributes. https://github.com/dtolnay/cxx/issues/833.
[76] The CXX Project Contributors. 2021. Fill in angle brackets with appropriate span if elided from impl key. https://github.com/dtolnay/cxx/pull/806.
[77] The CXX Project Contributors. 2021. Fix undeclared lifetime when fn ptr arg contains outer lifetime. https://github.com/dtolnay/cxx/pull/828.
[78] The CXX Project ContributorsThe CXX Project Contributors. 2020. Fix placement of commas in C++ member functions that call Rust methods. https://github.com/dtolnay/cxx/pull/443.
[79] The CXX Project Developers. 2025. cxx: Safe interop between Rust and C++. https://github.com/dtolnay/cxx
[80] The rust-bindgen Project Contributors. 2019. bindgen startup is incredibly slow. https://github.com/rust-lang/rust-bindgen/issues/1532
[81] The rust-bindgen Project Contributors. 2019. Generated code for inner class of template-class doesn’t use type-parameter. https://github.com/rust-lang/rust-bindgen/issues/1514
[82] The rust-bindgen Project Contributors. 2019. Infinite recursion on small input with alignof. https://github.com/rust-lang/rust-bindgen/issues/1590
[83] The rust-bindgen Project Contributors. 2020. C++ class method pointer treated as C function pointer. https://github.com/rust-lang/rust-bindgen/issues/1894

[84] The rust-bindgen Project Contributors. 2022. #pragma pack(1) struct containing another packed struct is not packed itself. https://github.com/rust-lang/rust-bindgen/issues/2189
[85] The rust-bindgen Project Contributors. 2024. Silently wrong code generated for enums in templated classes. https://github.com/rust-lang/rust-bindgen/issues/2871
[86] The Rust Project Developers. 2019. The Rust Programming Language: Calling an Unsafe Function or Method. https://doc.rust-lang.org/book/ch20-01-unsafe-rust.html#calling-an-unsafe-function-or-method Accessed: 2025-05-21.
[87] The Rust Project Developers. 2025. Rust Compiler Error Index. https://doc.rust-lang.org/error_codes/error-index.html.
[88] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An Empirical Study of Bugs in Machine Learning Systems . In *2012 IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, Los Alamitos, CA, USA, 271–280. doi:10.1109/ISSRE.2012.22
[89] TrustInSoft. 2025. *Rust's Rise in Embedded Systems: Why Hybrid Code Needs Advanced Analysis.* https://www.trust-in-soft.com/resources/blogs/rusts-rise-hybrid-code-needs-advanced-analysis
[90] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug Characteristics in Blockchain Systems: A Large-Scale Empirical Study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 413–424. doi:10.1109/MSR.2017.59
[91] Ziyuan Wang, Yun Fang, and Nannan Wang. 2025. An empirical study on bugs in TypeScript programming language. *Journal of Systems and Software* 226 (2025), 112445. doi:10.1016/j.jss.2025.112445
[92] Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. 2025. Towards Understanding the Characteristics of Code Generation Errors Made by Large Language Models. arXiv:2406.08731 [cs.SE] https://arxiv.org/abs/2406.08731
[93] Wikipedia contributors. 2025. Jaccard index. https://en.wikipedia.org/wiki/Jaccard_index
[94] Menghan Wu, Yang Zhang, Jiakun Liu, Shangwen Wang, Zhang Zhang, Xin Xia§, and Xinjun Mao. 2022. On the Way to Microservices: Exploring Problems and Solutions from Online Q&A Community. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 432–443. doi:10.1109/SANER53432.2022.00058
[95] Xiafa Wu and Brian Demsky. 2025. GenC2Rust: Towards Generating Generic Rust Code from C . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 664–664. doi:10.1109/ICSE55347.2025.00127
[96] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. 2021. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 3 (Sept. 2021), 25 pages. doi:10.1145/3466642
[97] Aidan Z. H. Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. 2024. VERT: Verified Equivalent Rust Transpilation with Large Language Models as Few-Shot Learners. arXiv:2404.18852 [cs.PL] https://arxiv.org/abs/2404.18852
[98] Haoran Yang and Haipeng Cai. 2025. Dissecting Real-World Cross-Language Bugs. In *ACM International Conference on the Foundations of Software Engineering (FSE)*. 1–23. doi:10.1145/3715777 (artifact evaluated; badges: Available).
[99] Yilin Yang, Tianxing He, Zhilong Xia, and Yang Feng. 2022. A comprehensive empirical study on bug characteristics of deep learning frameworks. *Information and Software Technology* 151 (2022), 107004. doi:10.1016/j.infsof.2022.107004
[100] Chengquan Zhang, Yang Feng, Yaokun Zhang, Yuxuan Dai, and Baowen Xu. 2024. Beyond Memory Safety: an Empirical Study on Bugs and Fixes of Rust Programs. In *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*. 272–283. doi:10.1109/QRS62785.2024.00035
[101] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. 2023. Ownership Guided C to Rust Translation. In *Computer Aided Verification (CAV 2023), Part III (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 459–482. doi:10.1007/978-3-031-37709-9_22
[102] Yi Zhang, He Jiang, Shikai Guo, Xiaochen Li, Hui Liu, and Chongyang Shi. 2025. Toward Understanding FPGA Synthesis Tool Bugs. *ACM Trans. Softw. Eng. Methodol.* (Feb. 2025). doi:10.1145/3718737 Just Accepted.