An Empirical Study of C to Rust Transpilers

Lei Xia Baojian Hua* Zhiyuan Peng
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
{xialeics, pengzhiyuan}@mail.ustc.edu.cn bjhua@ustc.edu.cn*

***Abstract*—C to Rust transpilers, which automatically convert C programs into Rust programs, have become increasingly popular in migrating legacy C codebases to Rust to take advantage of Rust's safety features. However, existing academic studies and industrial engineering practices mainly focus on optimizing the Rust code generated by C to Rust transpilers, assuming their trustworthiness and reliability, while neglecting the question of whether this assumption holds in practice.**

In this paper, we conduct, to the best of our knowledge, the *first* and most *comprehensive* large-scale empirical study of C to Rust transpilers, to gain an understanding of the reliability, limitations, and remaining research challenges of state-of-the-art C to Rust transpilation tools. We first designed and implemented a software prototype TOUCHSTONE, then used it to study the state-of-the-art C to Rust transpilers C2Rust. We obtained important findings and insights from empirical results, such as: 1) we proposed 4 root causes leading to transpilation failures; 2) we revealed 2 reasons hurting performance; 3) we identified 1 root cause affecting correctness of transpilation; 4) we proposed a measurement metric for the quality of tarnspiled Rust code; and 5) we investigated the resilience of C to Rust transpilers against code obfuscation. We suggest that: 1) transpiler builders should enhance their transpilers in terms of effectiveness, performance, and quality of transpiled code; and 2) Developers working on migration from legacy C projects to Rust should make better use of C to Rust transpilers based on the suggestions in this study. We believe these findings and suggestions will help tanspiler builders, Rust developers, and security researchers, by providing better guidelines for C to Rust transpiler studies.

***Index Terms*—Empirical study, Rust, Transpiler**

I. INTRODUCTION

Rust is an emerging programming language with two design goals of safety and efficiency. First, Rust achieves its safety goal via its unique ownership and borrowing system [1] [2], which is the cornerstone of Rust's memory safety. Combined with automatic lifetime-based memory management [3] and strict security checking rules, Rust guarantees memory safety. Second, Rust achieve its efficiency design goal by embracing a zero-cost abstraction philosophy. Specifically, Rust incorporates the programming concept of explicit lifetime[3], which incurs no runtime overhead without using garbage collections. As a result, due to its safety and efficiency advantages, Rust is gaining widespread adoptions in building software infrastructures, such as operating system kernels [4], language runtimes [5], databases[6], and blockchains [7].

In view of Rust's technical advantages, a considerable number of academic studies [8] [9] [10], as well as industry efforts [11] [12] [13] [14], have been conducted to migrate legacy C/C++ code bases to Rust. These migrations offer three notable advantages: 1) safety; 2) efficiency; and 3) economy. First, while C/C++ has been overwhelmingly used in system programming due to their flexibility and efficiency, their lack of security guarantees has made them vulnerable to cyber attacks [15]. Therefore, migrating legacy C/C++ code bases to Rust can significantly improve safety [16]. Second, migrating C/C++ to Rust does not compromise efficiency because of Rust's zero-cost abstraction design philosophy and its competitive performance with C [17]. Third, migrating C/C++ code to Rust allows maximum reuse of existing infrastructures, such as algorithm designs, data structures, and modular architectures, which makes migration more economical than rewriting Rust code from scratch.

While manual C/C++ migrations are flexible, automatic migrations using *transpilers* are often more practical, particularly for large C/C++ projects. Unlike compilers that translate high-level languages (*e.g.*, C or Java) to low-level languages (*e.g.*, x86 or ARM), transpilers translate programs between languages at roughly the same abstraction level (*e.g.*, from C to Java [18]), without modifying the functionality of the programs. Due to of its full automation and functionaltiy preservation advantages, transpiler-based automatic code migration has become an active area of research with a considerable number of academic studies [8] [9] [10] and engineering efforts [19] [20] [21] [22]. These studies have focused not only on investigating the theoretical foundations but also on developing practical tools. For instance, C2Rust[21], a state-of-the-art transpiler for automatically migrating C to Rust without any manual intervention, has been successfully used to convert large C projects.

Although prior studies have made significant progress in migrating C to Rust via transpilers, they have assumed that transpilers are trustworthy and reliable. However, it remains unknown whether such an assumption holds in practice. To the best of our knowledge, there has not been any empirical studies on C to Rust transpilers in terms of their success rates, faithfulness, performance, and the quality of their generated Rust code. We speculate that this situation may be due to the misconception that transpilers are easy to implement because of the roughly same abstraction level between the source language (*e.g.*, C) and the target language (*e.g.*, Rust).

However, transpilers from C to Rust still face three key technical challenges: 1) language discrepancies; 2) safety guarantees; and 3) idiomatic styles. To begin with, transpilers need to

address the syntactic discrepancies between the source and target languages, especially source language features absent from the target [23]. Furthermore, Rust, as a language guaranteeing safety, possesses certain idiosyncrasies (*e.g.*, explicit lifetime) that make transpilation challenging. Lastly, transpilers should produce the target code with idiomatic styles, which aid in maintaining and evolving the code.

Some research questions regarding C to Rust transpilers are still unanswered, which are as follows: What are the success rates and functional correctness of transpilation performed by these tools? What is the performance of these tanspilers? How does the quality of Rust code transpiled from these tools against to the input C code in terms of safety, overhead, and complexity? Without answers to these research questions, transpiler developers might base their work on wrong assumptions and thus miss opportunities to improve these tools. Developers working on migrating legacy C projects to Rust will not benefit from state-of-the-art.

Our work. To fill this gap, this paper presents the *first* and most *comprehensive* empirical study of C to Rust transpilers, in terms of success rates, failure factors, performance, functional correctness, and quality of the target code. First, we designed and implemented novel software tool prototype TOUCHSTONE to conduct this study. Second, we selected and created three datasets to perform the empirical study: 1) a random C dataset containing 84,125 executable C programs generated by Csmith[24] and YARPGen[25]; 2) a vulnerable C dataset containing 117 vulnerable C programs; and 3) a dataset containing 117 vulnerable C programs. Third, we have selected C2Rust, one advanced C to Rust transpiler, which has been widely used in academic researches [8] [10] and industrial practice[22]. Finally, we perform an empirical study in terms of success rates, failure factors, performance, functional correctness, and quality of transpiled Rust code.

We obtained important findings and insights from these empirical results, such as: 1) we investigated the success rates of these C2Rust and proposed 4 failure factors and 4 root causes for failures; 2) we studied the performance of C2Rust and revealed the root cause of inefficiency; 3) we proposed the key factor affecting the faithfulness of C2Rust; and 4) we studied the quality of Rust code translated from C2Rust and presented a quantitative metric in terms of safety, overhead and complexity to measure it.

Our findings, tools, and result have actionable implications for several audiences. Among others, they 1) help transpiler developers further improve transpilers by increasing the success rates, faithfulness, and improve quality of translated Rust code; 2) help transpilers users to make more effective use of transpilation tools to benefit from state-of-the-art; and 3) help Rust language designers to imporve the design of language such as adding support for bitfield[26] to promote the adoption of rust in areas such as embedded systems programming and network programming.

Contributions. To the best of our knowledge, this is the first and most comprehensive empirical study of C to Rust transpilers. To summarize, this work makes the following contributions:

- **Empirical study and tools.** We present the first empirical study of Rust transpilers, with a novel software prototype TOUCHSTONE we created.
- **Findings, insights, and suggestions.** We present interesting findings and insights, as well as suggestions, based on the empirical results.
- **Open source.** We make our tool, data and empirical results available in the interest of open science, at `https://doi.org/10.5281/zenodo.7871547`.

Outline. The rest of this paper is organized as follows. Section II introduces the background and motivations for this work. Section III presents the approach we used to perform this study. Section IV presents the empirical results we obtained, and answers to the research questions based on these results. Section V and VI discuss the implications of this work, and threats to validity, respectively. Section VII discusses the related work and Section VIII concludes.

II. BACKGROUND

To be self-contained, this section presents necessary background knowledge and motivations for this work.

A. Rust

Capsule history. Rust is an emerging programming language designed for building reliable and efficient system software. It originated as a personal project by Graydon Hoare in 2006 and was later officially sponsored by Mozilla in 2009 [27]. Rust 1.0 was released in 2015, marking a stable and production ready version of the language. and its latest stable version is 1.68.2 (as of this study). With over 15 years of active development, Rust is becoming more mature and productive.

Advantages. Rust emphasizes safety, efficiency, and productivity. First, Rust provides safety guarantees via a unique ownership and borrowing system, alongside a sound type system based on linear logic [28] [29] and alias types [30] [31]. These advanced language features not only rule out memory vulnerabilities such as dangling pointers, memory leaking, and double frees, but also enforce thread safety by preventing data races and deadlocks. Second, Rust achieve high efficiency through the ownership-based explicit memory management [32] and a lifetime model, without any garbage collectors. Both the ownership and lifetime are checked and enforced at compile-time, thus incurring zero runtime overhead. Third, Rust provides productivity via advanced programming features for object-oriented programming such as traits and generics, as well as for functional programming such as pattern matching and closures.

Wide adoptions. Rust has been widely adopted across diverse domains since its release. For example, Rust has been used successfully to build software infrastructures, such as operating system kernels [33] [34] [35] [36][37] , Web browsers[38], file systems [39] [40], cloud services [41], network protocol stacks [42], language runtime [43], databases [44], and blockchains [45]. Rust has also been a favored option for game development, as exemplified by games such

as Oxide [46] and Second Life [47]. Moreover, Rust is gaining more adoptions in the industry, such as Microsoft[12][48], Google[49][50], and even Linux [51] are beginning to use Rust for the development of system software. The increasing adoption of Rust indicates its ability to solve complex problems and deliver efficient and reliable software, making it a promising language for the future.

B. Transpiler for Language Conversion

Concept. A *transpiler* (translating compiler) converts programs in one programming language into semantic equivalent programs in another language(e.g. C2Rust, a C to Rust transpiler[21] converts C99-compliant programs to Rust programs preserving functionality, in which both C and Rust are system programming languages), or different standards of the same language(e.g. JS transpiler babel[52], which transpiles the arrow function in the ES6 standard[53] into the anoymous function in the ES5 standard [54]).

Transpilers have been extensively studied with many tool proposed (e.g., C to Rust [19] [20] [21] [55], Python to JavaScript [56] [57], JavaScript to Python [58]). In the future, with increasing adoptions of domain-specific languages, transpilers will be used more widely for program conversion between different languages.

Workflow. Source-to-source language *transpilation* (translating compilation) requires three primary steps: 1) source code processing; 2) translation; and 3) target code generation. First, the input source is processed into an internal program representation, which are often abstract syntax trees (ASTs) as they convey almost all source information that subsequent translation needed. Second, these internal program representations are translated to another language's internal representations with different syntax but equivalent semantics. Third, target programs are generated from the converted internal representations. Notably, although transpilers look simple from a conceptual point of view, they are actually large and complex software. For example, the C2Rust transpiler consists of more than 80,000 lines of Rust code, even excluding the supporting libraries.

C. Challenges for Transpilation

Source-to-Source language transpilation is a complex and intricate task, there are three key technical challenges in implementing transpilers from C to Rust: 1) language discrepancies; 2) safety guarantees; and 3) idiomatic styles.

First, addressing the syntactic discrepancies between the source and target languages is a universal challenge in transpilation. In the case of transpiling C to Rust, language features can be classified into three categories as the Venn diagram in Fig.1 presents: 1) common features (region B); 2) source language features absent from the target (region A); and 3) target language features absent from the source (region C). Features in regions B and C are relatively easy to handle, as they can be translated straightforwardly. However, features in region A require encoding with other target language features to mimic functionality, which could be complex and produce unsatisfactory results. For example, Rust does not support unstructured control flow features in C, such as the `goto` statement [59], making it challenging to convert C code to Rust using only structured control flow features [23].

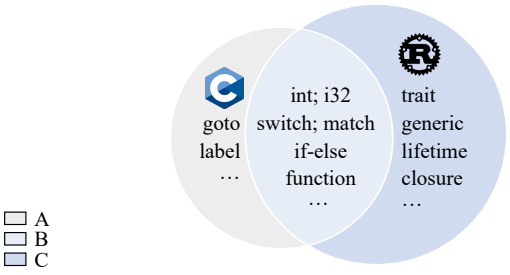


Fig. 1: A Venn diagram illustrating the syntactic discrepancies between C and Rust.

Second, Rust, as a language guaranteeing safety, have special peculiarities that make transpilation challenging. For example, explicit lifetime [3] is a necessary and fundamental safety feature of Rust, transpilers have to add appropriate lifetimes of references to data to allow the compilers (*e.g.*, rustc [60]) to ensure memory safety by keeping track of them, otherwise the rust programs will not pass the compiler's static detection. However, adding lifetimes automatically is challenging, due to ambiguous borrowing rules for ensuring memory safety.

Third, transpilers should produce target code with idiomatic styles, benefiting subsequent code maintenance and evolution. Consequently, a transpiler must possess a profound comprehension of the idioms and characteristics of both the source and target languages and the mappings between them, including the flexibility to modify transpilation strategies based on the context of the source code.

Given the challenges involved, designing and implementing a transpiler is complex and laborious. However, prior studies have made signicant progress in migrating C to Rust via transpilers, assuming that transpilers are trustworthy and reliable. To the best of our knowledge, there has not been any empirical studies on C to Rust transpilers in terms of their success rates, correctness, performance and the quality of their generated Rust code, thus it remains unknown whether such an assumption holds in practice.

III. Approach

This section describes our approach to conduct an empirical study on C to Rust transpilers, which faces challenges in automation and scalability when dealing with a large dataset of C programs. Specifically, two key reasons make it difficult to study large datasets with hundreds of thousands of C programs in a fully automatic manner: the need for automation and the need for scalability. While manual code inspection can complement automated analysis, a fully automated study is necessary to handle such a large dataset. Furthermore, the study must be scalable to analyze different transpilers,

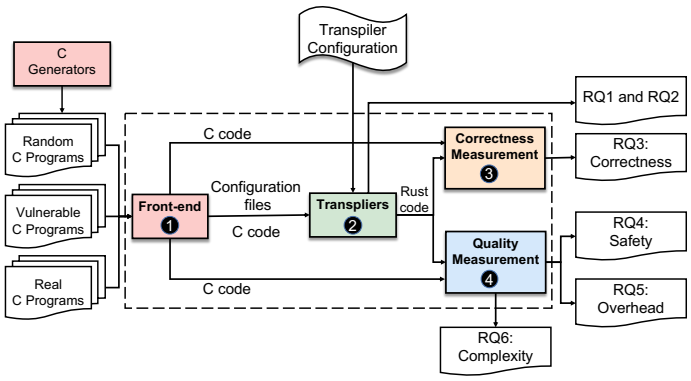


Fig. 2: TOUCHSTONE Architecture

including potential ones in the future, even transpilers from other languages to Rust.

To address these challenges, we have developed a software prototype called TOUCHSTONE, which enables automated and scalable investigation of research questions. In this section, we present the architecture of TOUCHSTONE (Section III-A) and its different modules, including the front-end module (Section III-B), the C-to-Rust transpilation module (Section III-C), the effectiveness measurement module (Section III-D), and the quality measurement module (Section III-E). By utilizing TOUCHSTONE, we can conduct empirical studies on C to Rust transpilers in an automated and scalable manner, and evaluate their success rates, performance, correctness and quality of generated Rust code.

A. The Architecture

TOUCHSTONE is designed with one important principle of modularity and extensibility, so that it is straightforward to make modifications suitable for different needs, such as adding new C programs datasets, experimenting with new rust transpilers' optimizations, evaluating new rust transpilers, or studying new evaluation metrics.

Based on this design principle, we present, in Fig. 2, the architecture of TOUCHSTONE, consisting of four key modules. First, the front-end module (❶) takes as input C programs, and process them by filtering out anything not related to compilation and generate specific configuration files required by transpilers.

Second, the transpilation module (❷) takes as input the C programs and the generated configuration files from the preceding module, transpile the C soure code, and generates the Rust programs according to a user-specified transpiler configuration. Transpilation results from this module are used to investigate the first research questions (**RQ1** and **RQ2**), that is, the success rates and performance of transpilers.

Third, the correctness measurement module (❸) takes as input both transpiled Rust programs and corresponding C programs, measures the functional correctness of transpilation by performing a differential testing which is a widely used technique for compiler testing. This result is further used to answer the third research question (**RQ3**).

TABLE I: C-Generators leveraged by TOUCHSTONE

Name	Language	Supported Languages	Open Source
CSmith[24]	C++	C	✓
YARPGen[25]	C++	C/C++	✓

Finally, the quality measurement module (❹) takes as inputs both an C program and the corresponding Rust program, and evaluate the quality of the transpiled Rust program in 3 aspects: 1)safety, 2) overhead and 3) complexity. The results from this measurement are used to answer three research questions (**RQ4**, **RQ5** and **RQ6**), respectively.

In the following sections, we discuss the design and implementation of each module, respectively.

B. The Front-end

The front-end process C code input with respect to their ways of building and generate specific configuration files according to the requirements of transpilers. First, depending on the building methods of input C programs, the front-end module filters out anything not related to compilation such as documents, and keep only code-related parts. Second, generateing specific configuration files required by transpilers, such as `compile_commands.json` C2Rust relying on.

Although integrating the front-end with other modules is possible, the current design of TOUCHSTONE, from a software engineering perspective, has two key advantages: 1) it makes TOUCHSTONE feasible to process different types of C source code and different compilation methods; and 2) it is more effective by processing the peculiarities of the C source code at an early stage, simplifying subsequent phases significantly.

C. Transpilation

The transpilation module transpiles C programs into corresponding Rust programs, based on user-supplied configuration. The transpiler configuration specifies the configuration to control transpilers, such as transpiler options and output format.

The module employs state-of-the-art transpilers that 1) should be fully automatic, which enables large datasets can be processed without human intervention, 2) can generate executable rust projects according to the input C projects. 3) can preserve functionality during the transpilation, and 4) are frequently updated and maintained. To this end, C2Rust has been selected and used in TOUCHSTONE, which supports automatic transpilation on C99-compliant code while preserving semantics. To the best of our knowledge, it is comprehensive and represent the state-of-the-art of C to Rust transpiling tools.

Some C to Rust transpilers are omitted in our study. Among them, Bindgen [19] generates Rust FFI bindings for C libraries automatically, but not the correspoding Rust libraries. Citrus [55] generates function bodies but without preserving semantics of input C code. Corrode [20], the first semantics-preserving transpiler intending for partial automation, is, unfortunately, no longer maintained and thus unusable.

However, the architecture of TOUCHSTONE (Fig. 2) is neutral to the specific C to Rust transpilers selected and used. Furthermore, the modular design of TOUCHSTONE facilitates the integration of additional decompilers with ease..

D. Functional correctness Measurement

Technically, C to Rust transpilers might produce Rust programs without preserving functionality compared to the C programs input. The functional correctness measurement module in TOUCHSTONE is designed to measure the the correctness of transpilation by leveraging a differential testing approach. The experimental results are further used to answer **RQ2** (Section IV-A).

In order to automatically and efficiently test the functional correctness of transpilation, TOUCHSTONE has for the first time applied the C-generators, which are widely used to stress-test C compilers, to test the C to Rust transpilers. We have chosen Csmith and YARGen, the two most widely used C generators which can generate generate random C programs statically and dynamically conforming to the C99 standard, to test the degree of C2Rust's support for C99 syntax. Each generated C program consists of several files and after being compiled and run produces a decimal number, which is revealed to the global variable values in this C program. The numbers produced by C programs input and transpiled Rust programs will be compared to check for correctness. If they are same, then the transpiler has successfully generated corresct Rust code for the C program. Otherwise, it is likely to encounter a transpiler bug.

E. Quality Measurement

As Rust is a language combining efficiency and safety, the transpiled Rust code is excepted to have improved safety and performance comparable to that of the input C code. Additionally, the transpiled Rust code should also be readable and maintainable, making it easy for developers to understand and maintain.

To this end, TOUCHSTONE incorporates a quality measurement module to evaluate the quality of the transpiled Rust code in 3 aspects: 1)safety, 2) overhead and 3) complexity. First, to check whether safety improved we applied TOUCHSTONE to the vulnerable C dataset created from CWE [61], a set of widely used "Weaknesses in Software Written in C". The Rust programs transpiled from this dataset are classified into 2 categories: a) corresponding vulnerabilities are detected by rustc during the compilation of the transpiled Rust programs; and b) vulnerabilities, as in the corresponding C code still exists and escapes the static detection of rustc. We believe that the first categorie can be considered as proof of safety improvement. The results on this aspect are used to answer **RQ4** (Section IV-A).

Second, to measure the overhead brought to the transpiled Rust programs, we applied TOUCHSTONE to the random C dataset to measure the run time of C programs and the corresponding Rust programs. The results on this aspect are used to answer **RQ5** (Section IV-A).

TABLE II: C to Rust transpiler

Name	Language	Source Language	Target Language	Open Source
C2Rust	Rust	C99	Rust	✓

Third, to answer **RQ6** (Section IV-A), we specifically applied TOUCHSTONE to the actual C dataset. To calculate the code structure complexity, we used the cognitive complexity (CC) metric, which explicitly measures code understandability. CC is widely used in academic studies [62] [63] and integrated into widely-used static code analysis tools [64] [65].

IV. EMPIRICAL RESULTS

This section presents our empirical results by answering the research questions.

A. Research Questions

By presenting the empirical results, we mainly investigate the following research questions:

RQ1: Success rate and root cause analysis. What is the success rates of transpilers? What are the root causes leading to transpilation failures?

RQ2: Performance. What is the average execution time of transpilers? Are they performant enough for practical usage?

RQ3: Functional correctness. Are transpilers faithful in generating Rust code with functional equivalence compared to the corresponding C code?

RQ4: Safety. Does Rust cdoe generated by transpilers behave better in safety, compared with C programs?

RQ5: Overhead. Does the transpilation introduce extra overhead to the generated Rust code?

RQ6: Complexity. What is the complexity of transpiled Rust code?

B. Evaluation Setup

All evaluations and measurements are performed on a server with one 20 physical Intel i7 core CPU and 64 GB RAM running Ubuntu 20.04.

C. Transpilers and Datasets

We first describe decompilers and datasets created and used in this study, which has been released in our open source.

Transpilers. We selected C2Rust from transpilers presented in TABLE II. To the best of our knowledge, it can represent state of-the-art C to Rust transpilers.

Datasets. To conduct the empirical study, we selected and created three datasets to perform the empirical study: 1) a random Cdataset containing 84,125 executable C programs generated by CSmith and YARPGen (presented in TABLE I), two proven C-generators widely used in testing C compilers; 2) a real C dataset containing 3 real-world C applications in several domains, two principles guide our selection of real-world C projects. First, the project we selected should be widely used and publicly available, thus the evaluation in this work can be reproduced on such public projects by others.

TABLE III: The actual C dataset of 3 real-world projects.

Project	Domain	LoC	#Files	Github Stars (k)
Vim[**?**]	System Tools	358,707	186	28.4
cURL[66]	Web	145,802	535	26.5
Silver searcher[67]	Dev Tools	3,932	12	24.1

Second, to cover as many C usage scenarios as possible, we aim to include as many domains in our study as possible. According to the aforementioned project selection criteria, we collected C projects from 3 different domains: system tools, Web, and dev tools. The selected domains and projects are presented in Table III. For each of the projects included, we give the name of the selected projects in the corresponding domain, the sizes of these projects (measured by lines of C source code), the numbers of C source files, and the GitHub stars reflecting their popularity. and 3) a vulnerable C dataset containing 45 vulnerable C programs from CWE[61], a set of widely used "Weaknesses in Software Written in C"[68].

D. RQ1: Success Rate and Root Cause Analysis

To answer **RQ1** by investigating transpilation success rate of C2Rust, we first applied TOUCHSTONE to the random C dataset. A successful transpilation should meet the following two requirements: 1) C2Rust must successfully output the Rust code transpiled from the C code input; and 2) the output Rust code must be executable.

TABLE IV presents the empirical results. The first column lists the two phases of transpilation. The next 2 columns present transpilation failures as well as success rates. The last 4 columns present the failure factors, which have been classified into four categories: stack overflows, compatibility errors, bitfield errors, and type errors.

The empirical results give interesting findings and insights. On the one hand, only 5 times did c2rust fail to output the corresponding rust programs based on the input c programs, demonstrating a high success rate (beyond 99.99%) of output, on the other hand, compared to the high success rates of output, relatively low rate (72.64%) of the output Rust code can execute successfully.

We then conduct an analysis of the 5 output failures. All of them are due to stack overflows caused by large recursion depths. C2Rust uses recursion for function calls that are normally treated as no-jumping code blocks in context-free grammars (CFG). However, if multiple consecutive function calls are encountered, it is possible to trigger a stack overflow error due to excessive recursion.

We then explore the root causes leading to execution failures, and identified 3 key reasons: 1) compatibility error; 2) bitfield error; and 3) type error. First, C2Rust still use unaligned references which were previously accepted by the Rust compiler but has been phased out [69]; it becomes a hard error in rustc 1.63.0 [70] C2Rust v0.17.0 relied on.

Second, C2Rust does not provide complete support for transpilation of bitfields. On the one hand, C2Rust does not take into account the use of bitfields in unions which results the mistranslation of bitfields in unions as showing in Figure 3. On the other hand, although structs with bitfields are treated specifically during transpilation, the automatic alignment of structs is not taken into account. For example, in `struct S1` (presented in fig. 4(b)), the bit range of `f1` is contradictory to the bytes allocated for `f0_f1`.

Third, C2Rust is not comprehensive enough for type transpilation. For explicit type cast, c2rust provide a comprehensive support which takes full account of some special type such as `_Bool`, as showing in the Fig.5(b), in the transpiled Rust code, comparing with zero to get the bool value. However, implicit type cast is not supported sufficiently, the transpiled Rust code attempts to cast a numeric type to a bool is not allowed in Rust [71].

```c
union U0 {
  // f0: a bit field
  unsigned int f0:2;
  int f1;
} g;
```

(a) C code snippet.

```rust
pub union U0 {
    // f0: not a bit field
    pub f0: c_uint,
    pub f1: c_int,
}
```

(b) Rust code snippet.

Fig. 3: Sample code illustrating the transpilation error in unions with bitfields.

```c
struct S1 {
  signed f0 : 2; unsigned f1 : 32;
};
```

(a) C code snippet.

```rust
pub struct S1 {
  // ... ...
  #[bitfield(name="f0", ty="c_int", bits="0..=1")]
  // bit range after alignment
  #[bitfield(name="f1", ty="c_uint", bits="32..=63")]
  // bytes before alignment
  pub f0_f1: [u8; 5],
}
```

(b) Rust code snippet.

Fig. 4: Sample code illustrating the transpilation error in structs with bitfields.

```c
_Bool i = (_Bool)0;
// cast as '_Bool'
i += 1;
```

(a) C code snippet.

```rust
let mut i: bool = 0 != 0;
// cannot cast as `bool`
i=(i as c_int + 1)as bool;
```

(b) Rust code snippet.

Fig. 5: Sample code illustrating the transpilation error in type cast.

TABLE IV: Success rates, failures, failure factors of the 2 indicators of transpilation, on the random C dataset.

P of Transpilation	Result		Failure Factors			
	Failures	Success Rates	Stack Overflows	Compatibility Errors	Bitfield Errors	Type Errors
Translation	5	99.99%	5	0	0	0
Execution	23,739	72.65%	0	9,758	8,769	5,212
Total	23,744	72.64%	5	9,758	8,769	5,212

TABLE V: Performance.

Performance	Time per Test (ms)		Time per Line (ms)	
	Avg	SD	Avg	SD
C2Rust	536	235.18	0.292	7.8

Summary: C2Rust guarantees a high success rate (over 99.99%) for output, but the output Rust code has a relatively low executable rate (72.65%). We identified that output failures were caused by stack overflows, and have revealed 3 root causes for execution failures: compatibility error, bitfield error and type error.

E. RQ2: Performance

To answer **RQ2** by investigating the performance of C2Rust, we applied TOUCHSTONE to the random C dataset. Each random C program is transpiled in 10 rounds to calculate an average transpilation time. TABLE V presents the average transpilation time per file and its standard deviation, average transpilation time per line of C code and its standrd deviation, respectively.

The empirical results give interesting findings and insights. First, the transpilation time grow nearly linearly with file sizes. On average, C2Rust can process 3,424 lines of C code per Second, the standard deviation of the transpilation time per line is relatively high(SD = 7.8).

We then explored the root causes of the high standard deviation of transpilation time per line, based on a manual inspection of the C2Rust's sources. This inspection revealed 2 key reasons. First, transpiling unstructured control flow in C which is absent in Rust is time consuming. To address the aforementioned challenge of unstructured control discrepancy, C2Rust have proposed the Relooper algorithm [23], and second, depth-first algorithms are extensively used in C2Rust, such as depth-first iterator of the four C AST types and depth first visitor pattern for Rust ASTs, which affect performance of C2Rust.

Summary: The performance of C2Rust is affected by the size of the input C program file and the type of C program code, its own use of the relooper algorithm and depth-first recursive algorithm has low performance.

F. RQ3: Functional correctness

To answer **RQ3** by investigating the functional correctness of transpilation performed by C2Rust, we applied TOUCHSTONE to the random C dataset to verify the functional equivalence between transpiled Rust code and original C code.

TABLE VI: Success rates, failures, failure factors for the functional correctness, on the random C dataset.

Indicators of Transpilation	Result		Failure Factor
	Failures	Success Rates	Side effect
Functional Correctness	187	99.67%	187

```c
#include <stdio.h>
int g_1 = 0;
int func() {
   // change the value of global_v
   g_1 = 1;
   return g_1;
}
int main(){
   // the result of func() is not used, but necessary
   -func();
   // ... ...
}
```

(a) C code snippet.

```rust
pub static mut g_1: c_int = 0 as c_int;
pub unsafe extern "C" fn func() -> c_int {
   g_1 = 1 as c_int;
   return g_1;
}
unsafe fn main_0() -> c_int {
   // ignore the translation of "-func();"
   // ... ...
   return 0;
}
```

(b) Rust code snippet.

Fig. 6: Sample code illustrating the functional incorrectness about side effects.

As we present, in TABLE VI, 187 transpiled Rust programs executed with different results from the corresponding C programs and C2Rust guarantees a high success rate (99.67%) in preserving functionality during transpilation from C to Rust.

We then investigate the root causes for failures, based on manual code inspection. This investigation reveals the key reason is the incomplete handling of side effect[72]. As showing in the Fig. 6(a), although the result of `func()` is not used, but it changes the value of `g_1` which is called the

TABLE VII: Experimental results of safety enhancements on the vulnerable C dataset.

Vulnerability Type	Num	Rustc	
		Error Report	Rates(%)
OOB	8	1	12.5
Buffer Overflow	7	0	0
Uaf/DF	6	0	0
Integer Overflow	7	0	0
Divison by Zero	8	1	12.5
Memory leak	9	0	0
Total	45	2	4.44

side effect of expression. However, during the transpilation performed by C2Rust, this kind of expressions will be ignored.

Summary: C2Rust has a high success rate (99.67%) in preserving functionality during the transpilation.

G. RQ4: Safety

To answer **RQ4** by investigating the safety enhancement during the transpilation, we applied TOUCHSTONE to the vulnerable C dataset to analyse whether the safety of the transpiled Rust code is improved compared to the input C code.

TABLE VII presents the empirical results. The first column lists 6 types of common vulnerabilities. The next column lists the numbers of test cases each type of vulnerability contains. In the following column, we show the number of transpiled Rust programs, in which vulnerabilities were detected by `rustc` during the compilation of the transpiled Rust programs. The last column lists the success rate of safety enhancement from C to Rust.

The empirical results give interesting findings and insights. First, success rate of safety improvement from C to Rust is very low, the total success rate is only 4.44%. Second, in terms of memory safety, `rustc` doesn't show the capability of static detection that it should have, only 2 errors were reported at the compile stage.

We then investigate the root causes for the low success rates of safety enhancements, based on manual code inspection. First, almost all of the transpiled Rust code is wrapped in `unsafe` [8], which greatly affects Rustc's capability of static detection [68]. Second, for some functions in C libraries which are prone to safety vulnerabilities, such as `malloc`, `free`, etc., C2Rust will use the language feature: `extern` [73] to import them and treat them as FFIs [74], which `rustc` can not detect them[68].

Summary: Transpilation of C2Rust does not significantly improve the safety of the programs. Tranpiled rust programs do not take full advantage of the safe language features of Rust.

TABLE VIII: Overhead introduced by transpilation.

Quality of Rust Code	Time Per File (ms)		Time Per Line (ms)	
	C	Rust	C	Rust
Overhead	19.7	79.2	0.1089	0.1303

H. RQ5: Overhead

To answer **RQ5** by investigating the overhead of transpiled Rust programs compared to the original C programs, we applied TOUCHSTONE to the random C dataset.

TABLE VIII presents empirical results of execution time of transpiled Rust programs and the C programs. Each program is executed for 10 rounds. First, the average execution time per file of Rust is 4 times longer than that of C, however, the average execution time per line of Rust just 1.2 times longer than that of C.

We further explored the potential root causes and identified the key reason is code duplication caused by the Relooper algorithm. For example, in Fig. 7a, to convert the left CFG, Relooper will duplicate the node E as a new node E', thus forming the CFG on the right of Fig. 7a. The duplication and addition of nodes affect the performance of the program by incurring more cache misses, especially for large code blocks E with many instructions.

Summary: The overall performance of generated Rust programs is lower than that of the original C program.

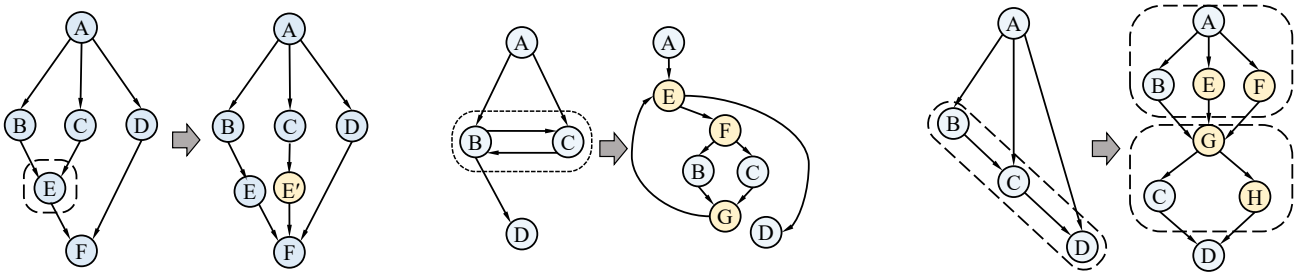
I. RQ6: Complexity

To answer **RQ6** by investigating the complexity of the transpiled Rust code, which reflects the readability and maintainability of the code, We applied TOUCHSTONE to the real-world C dataset consisting of 3 large-scale C projects: Vim, cURL, Silver searcher.

Table IX presents the experimental results. The first column lists three real-world C projects we used, whereas the next three columns in (**Code Explosion**) give the lines of C code, lines of Rust code, and increment which is calculated by $INC = RustLoc/CLoc - 1$, The last three columns in (**Cognitive Complexity (CC)**) present the CC metric of the orginal C code, the Rust code generated by C2Rust, and the increment, respectively. The decrement is calculated by $INC = CCode/Rustcode - 1$.

The empirical results give interesting findings and insights. On the one hand, the transpiled three Rust projects all have significant code explosion compared to the original C code (average increment is 90.24%); on the other hand, the average increment in CC metric is 31.46%.

We further explored the root causes and identified 2 key reasons. First, the Relooper algorithm often generates Rust code exhibiting more complex structures than the original C code due to irreducibility of the CFG. For example, as presented by Fig. 7b, the subgraph (on the left) consisting of nodes B and C is irreducible. To make the subgraph reducible, Relooper algorithm adds three new nodes E, F,



(a) Poor performance (b) Complex structures (c) Code explosion

Fig. 7: Three sample CFG conversions to illustrate the three limitations: complex structures, code explosion, and poor performance.

TABLE IX: Code explosion and cognitive complexity(CC) in the real-world dataset.

Program	Code Explosion			Cognitive Complexity (CC)		
	C Loc	Rust Loc	INC[1] (%)	C code	Rust code	INC[1] (%)
Vim	358,707	673,399	87.72	7989	9954	24.59
cURL	145,802	287,498	97.18	6445	8987	39.44
Silver searcher	3,932	6,343	61.32	278	401	44.24
Average	169,480	322,413	90.24	4904	6447	31.46

[1] is abbreviations of INCrement.

and G (on the right), and corresponding edges between them. The addition of such new nodes and edges complicated the structure of the programs considerably; and second, the irregularity of the CFG caused by Relooper algorithm leads to such code explosions. For example, in Fig. 7c, the CFG on the left cannot be converted to a *Multiple* construct directly as nodes B, C and D have different successors. To convert this CFG, Relooper adds four new nodes E, F, G, and H, as shown on the right of Fig. 7c. Although the addition of the four new nodes makes the two subgraphs in the dash boxes on the right in Fig. 7c easily convertible, it does increase the code size considerably (by 2X).

Summary: The generated Rust projects all have a significant code explosion (average increment is 90.24%) and increment of cognitive complexity (average increment is 31.46%).

V. Implications

This work presents the first and most comprehensive empirical studies of C to Rust transpilation. This section discusses some implications of this work along with some important directions for future studies.

For transpiler developers. The results of this work provide transpiler developers with important insights into their transpilers, in terms of success rates, performance, faithfulness, and so on. On the one hand, transpiler developers can leverage the insights proposed by this work to better improve the quality and reliability of current tools. For example, developers should pay special attention to the transpilation of source language features absent from the target such as bitfield available in C, but not in Rust. On the other hand, developers might utilize the software prototype TouchStone we proposed as well as the datasets we created in this work to testify their implementations.

For transpiler users. The results and suggestions in this work can benefit users of C to Rust transpilers who are working on migration from legacy C projects to Rust. On the one hand, our empirical results demonstrated that these transpilers have high success rates of translation, so developers should make use of these tools to aid in initial transpilation from C to Rust preserving functionality. Furthermore, users of C to Rust transpilers should pay special attention to the unstructured controls such as `goto` in C programs and other language discrepancies between C and Rust. On the other hand, our study results also demonstrated that the quality of transpiled Rust code is not good enough in terms of safety, overhead and complexity, users will still need to use other automatic optimization tools [8], or optimise the Rust code manually.

For Rust designers. The results of this work can help Rust language designers to complete the design of Rust, introducing and improve some language features to promote the development of rust. For example, bitfield[26], a crucial data structure enabling the developers to pack multiple bits of data into a single byte, has not been natively supported by Rust. Introducing the support to bitfield could make Rust language programming better in efficiency, speed, and memory optimization, thus promoting the adoption of rust in areas such as embedded systems programming and network programming.

VI. Threats to Validity

As in any empirical study, there are threats to the validity of our work. We attempt to remove these threats where possible

and mitigate the effect when removal is not possible.

Transpilers. In this work, we have used C2Rust to conduct this study. Although this C to Rust transpiler is the most widely used and thus represent state-of-the-art, there may be other transpilers available (Section III). Furthermore, new C to Rust transpilers might be developed in the future. Fortunately, the modular design of TOUCHSTONE makes it straightforward to testify to new C to Rust transpilers. Besides that, TOUCHSTONE is also applicable to test other language transpilers. For example, we can apply TransFuzz [75] in TOUCHSTONE to test JS transpilers such as babel and swc with ease.

Datasets. In this work, we utilized a random C dataset, a real-world C dataset, and a vulnerable C dataset. As the random C dataset is generated by proven C generators: Csmith and YARPGen which are widely used for testing C compilers; the real-world C dataset are selected from widely used C applications with high code quality; and the vulnerable C dataset is collected from CWE which is public and widely used. On the other hand, there are some other datasets available. Fortunately, the architecture of TOUCHSTONE is neutral to any specific dataset used, so a new dataset can always be added without difficulties.

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

VII. Related Work

A significant amount of work on transpilers has been made in general, and on automated C to Rust transpilation particularly. However, work in this study stands for a novel contribution to these fields.

Transpilers. There have been several studies on converting C to Rust. Bindgen [19] generates Rust FFI bindings for C libraries automatically. Corrode [20] is semantics-preserving transpiler intending for partial automation. and Citrus[55] generates function bodies but without preserving C semantics. As the successor of Corrode, C2Rust[21] supports large-scale automatic conversion while preserving semantics. This is the first large-scale empirical study into C to Rust transpilers, and for the first time applies the C-generators, which are widely used to stress-test C compilers[76][77], to test the C to Rust transpilers.

Automatic migration from C to Rust. There have been several studies on the optimization in the automatic migration from C to Rust. Emre et al. [8] first analyzed the sources of unsafety in Rust code generated by C2Rust and proposed a technique to convert raw pointers into references in translated programs, which hooks into the rustc compiler to extract type and borrow checker results. Hong et al. [9] proposed an approach to lift raw pointers to arrays with type constraints. Ling et al. [10] presented CRustS, which eliminates non-mandatory unsafe keywords in function signatures and refines unsafe block scopes inside safe functions using code structure pattern matching and transformation.

However, these studies as well as optimization tools and transpilation frameworks do not discuss the reliablities of the C to Rust transpilers they leveraged but just assumed these transpilers are correctly implemented and thus trustworthy.

In contrast, this work, for the first time, we conducted the empirical study of the C to Rust transpilers, which is orthogonal to prior studies and thus complemented them.

VIII. Conclusion

In this work, we presented the first and most comprehensive study of C to Rust transpilers. By designing and implementing a software prototype TOUCHSTONE, we proposed root causes leading to transpiler failures and have submitted these bugs with suggestions for fixing them to transpiler developers, all of which have been accepted. We also revealed reasons for hurting performance, and we identified the bug affecting the faithfulness of transpilation and have fixed it. Besides that, we also evaluated the quality of the transpiled Rust code such as safety, readability, and overhead. We provided suggestions to transpiler developers, users of transpilers, and Rust language designers to promote a healthier ecosystem for Rust.

References

[1] "Ownership," https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html.

[2] "Borrowing," https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html.

[3] "Lifetimes - the rust programming language," https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/lifetimes.html.

[4] "Tock embedded operating system."

[5] "Deno — a modern runtime for javascript and typescript," https://deno.land/.

[6] "Meilisearch database."

[7] "Diem blockchain."

[8] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating c to safer rust," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 121:1–121:29, Oct. 2021.

[9] T. Y. Hong, "From c towards idiomatic & safer rust through constraints-guided refactoring," p. 85.

[10] M. Ling, Y. Yu, H. Wu, Y. Wang, J. R. Cordy, and A. E. Hassan, "In rust we trust – a transpiler from unsafe c to safer rust," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022, pp. 354–355.

[11] "Quantum - mozillawiki."

[12] MSRC. Team, "A proactive approach to more secure code – microsoft security response center."

[13] "Mitigating memory safety issues in open source software."

[14] C. Wiltz, "A brief history of rust at facebook," Apr. 2021.

[15] T. Saito, R. Watanabe, S. Kondo, S. Sugawara, and M. Yokoyama, "A survey of prevention/mitigation against

memory corruption attacks," in *2016 19th International Conference on Network-Based Information Systems (NBiS)*, Sep. 2016, pp. 500–505.
[16] P. Chifflier and G. Couprie, "Writing parsers like it is 2017," in *2017 IEEE Security and Privacy Workshops (SPW)*, 2017, pp. 80–92.
[17] W. Bugden and A. Alahmar, "Rust: The programming language for safety and performance," Jun. 2022.
[18] "mtsystems - c source code to java source code translation."
[19] "Bindgen," The Rust Programming Language, Aug. 2022.
[20] J. Sharp, "Corrode: Automatic semantics-preserving translation from c to rust," Aug. 2022.
[21] "Immunant/c2rust: Migrate c code to rust," https://github.com/immunant/c2rust.
[22] "Curl-rust: Rewrite memory leak related modules for curl using rust," https://gitee.com/openeuler/curl-rust.
[23] "Emscripten — proceedings of the acm international conference companion on object oriented programming systems languages and applications companion."
[24] "Csmith-project/csmith: Csmith, a random generator of c programs," https://github.com/csmith-project/csmith.
[25] "Intel/yarpgen: Yet another random program generator," https://github.com/intel/yarpgen.
[26] "Bit field," *Wikipedia*, Nov. 2022.
[27] "Internet for people, not profit."
[28] J.-Y. Girard, "Linear logic," *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–101, Jan. 1987.
[29] P. Wadler, "Linear types can change the world!" in *Programming Concepts and Methods*, vol. 3. Citeseer, 1990, p. 5.
[30] J. Boyland, "Alias burying: Unique variables without destructive reads," *Software: Practice and Experience*, vol. 31, no. 6, pp. 533–553, 2001.
[31] D. Clarke and T. Wrigstad, "External uniqueness is unique enough," in *European Conference on Object-Oriented Programming*. Springer, 2003, pp. 176–200.
[32] D. J. Pearce, "A lightweight formalism for reference lifetimes and borrowing in rust," *ACM Transactions on Programming Languages and Systems*, vol. 43, no. 1, pp. 3:1–3:73, Apr. 2021.
[33] "Tock embedded operating system," https://www.tockos.org/.
[34] S. Lankes, J. Breitbart, and S. Pickartz, "Exploring rust for unikernel development," in *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, ser. PLOS'19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 8–15.
[35] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis, "The case for writing a kernel in rust," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, ser. APSys '17. New York, NY, USA: Association for Computing Machinery, Sep. 2017, pp. 1–7.
[36] A. Light, "Reenix: Implementing a unix-like operating system in rust."
[37] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with intel memory protection keys," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '20. New York, NY, USA: Association for Computing Machinery, Mar. 2020, pp. 143–156.
[38] "Servo, the parallel browser engine," https://servo.org/.
[39] "Redox-os/tfs: Mirror of https://gitlab.redox-os.org/redox-os/tfs," https://github.com/redox-os/tfs.
[40] "High velocity kernel file systems with bento — usenix," https://www.usenix.org/conference/fast21/presentation/miller.
[41] "Rustcc/ttstack: A private-cloud solution for smes !" https://github.com/rustcc/TTstack.
[42] "Smoltcp-rs/smoltcp: A smol tcp/ip stack," https://github.com/smoltcp-rs/smoltcp.
[43] "Tokio - an asynchronous rust runtime," https://tokio.rs/.
[44] "Tikv/tikv: Distributed transactional key-value database, originally created to complement tidb," https://github.com/tikv/tikv.
[45] "Blockchain infrastructure for the decentralised web — parity technologies," https://www.parity.io/.
[46] "Oxide - home," https://oxidemod.org/.
[47] "Official site — second life - virtual worlds, virtual reality, vr, avatars, and free 3d chat," https://secondlife.com/.
[48] "Microsoft to explore using rust — zdnet," https://www.zdnet.com/article/microsoft-to-explore-using-rust/.
[49] "Google online security blog: Rust in the android platform," https://security.googleblog.com/2021/04/rust-in-android-platform.html.
[50] "Google funds project to secure apache web server with new rust component."
[51] "Rust for linux," https://github.com/Rust-for-Linux.
[52] "Babel · babel," https://babeljs.io/.
[53] "Ecmascript 2015 language specification – ecma-262 6th edition," https://262.ecma-international.org/6.0/.
[54] "Ecmascript language specification - ecma-262 edition 5.1," https://262.ecma-international.org/5.1/.
[55] "Citrus / citrus · gitlab," https://gitlab.com/citrus-rs/citrus.
[56] A. Berti, "Javascripthon: Javascript for refined palates: A python 3 to es6 javascript translator."
[57] T. E. Crosley, "Why jiphy?" Jun. 2022.
[58] P. Dabkowski, "Piotrdabkowski/js2py," Aug. 2022.
[59] M. Nagappan, R. Robbes, Y. Kamei, É. Tanter, S. McIntosh, A. Mockus, and A. E. Hassan, "An empirical study of goto in c code," PeerJ PrePrints, Preprint, Feb. 2015.
[60] "What is rustc? - the rustc book," https://doc.rust-lang.org/rustc/what-is-rustc.html.
[61] "Cwe - cwe-658: Weaknesses in software written in c (4.10)," https://cwe.mitre.org/data/definitions/658.html.
[62] G. A. Campbell, "Cognitive complexity: An overview and evaluation," in *Proceedings of the 2018 International Conference on Technical Debt*, ser. TechDebt '18. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 57–58.
[63] J. Bogner and M. Merkel, "To type or not to type? a sys-

tematic comparison of the software quality of javascript and typescript applications on github," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 658–669.

[64] "Automatic code review, testing, inspection & auditing — sonarcloud."

[65] "Cognitivecomplexity - intellij ides plugin — marketplace."

[66] "Curl/curl: A command line tool and library for transferring data with url syntax, supporting dict, file, ftp, ftps, gopher, gophers, http, https, imap, imaps, ldap, ldaps, mqtt, pop3, pop3s, rtmp, rtmps, rtsp, scp, sftp, smb, smbs, smtp, smtps, telnet, tftp, ws and wss. libcurl offers a myriad of powerful features," https://github.com/curl/curl.

[67] "Ggreer/the_silver_searcher: A code-searching tool similar to ack, but faster." https://github.com/ggreer/the_silver_searcher.

[68] S. Hu, B. Hua, L. Xia, and Y. Wang, "Crust: Towards a unified cross-language program analysis framework for rust," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, Dec. 2022, pp. 970–981.

[69] "Tracking issue for future-incompatibility warning 'unaligned_references' · issue #82523 · rust-lang/rust," https://github.com/rust-lang/rust/issues/82523.

[70] "Rustc_ast - rust," https://doc.rust-lang.org/1.63.0/nightly-rustc/rustc_ast/index.html.

[71] "E0054 - error codes index," https://doc.rust-lang.org/error_codes/E0054.html.

[72] "Side effect (computer science)," *Wikipedia*, Mar. 2023.

[73] "Extern - rust," https://doc.rust-lang.org/std/keyword.extern.html.

[74] "Ffi - the rustonomicon," https://doc.rust-lang.org/nomicon/ffi.html.

[75] "Saner 2023 - macao, china," https://saner2023.must.edu.mo/index.

[76] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "History-guided configuration diversification for compiler test-program generation," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. San Diego, California: IEEE Press, Feb. 2020, pp. 305–316.

[77] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for c and c++ compilers with yarpgen," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 196:1–196:25, Nov. 2020.