Security Risks of Transpiling C Programs to Rust

Si Wu Huyao Yang Baojian Hua*
School of Software Engineering, University of Science and Technology of China, China
Suzhou Institute for Advanced Research, University of Science and Technology of China, China
wusi98@mail.ustc.edu.cn yanghuyao@mail.ustc.edu.cn bjhua@ustc.edu.cn

***Abstract*—Rust is a promising systems programming language that provides strong security guarantees without sacrificing efficiency. To fully exploit Rust's security benefits, transpilation is essential to migrate legacy C code to Rust. However, existing studies and practical transpilers all assumed that transpiled Rust programs are more secure and trustworthy than the corresponding C sources. Unfortunately, whether such an assumption truly holds in practice is still unknown. Therefore, a systematic empirical security investigation is urgently needed to evaluate the security risks and implications of C-to-Rust transpilations. In this paper, to fill this gap, we take the first step towards investigating the security risks of transpiled Rust code. To this end, we first create a dataset comprising 25,183 vulnerable C programs, to systematically examine how known vulnerabilities manifest after the transpilation. We then conduct an empirical study of the generated Rust transpiled from the C dataset, and obtain important findings and insights from the results: 1) we find that Rust's built-in checks detect 14,201 vulnerabilities (56.4%) in C2Rust translations and 15,718 (62.4%) in GPT-4 translations, 2) we identify three root causes of semantic correction, behavioral masking, and latent unsafe preservation, leading to detection failures, and 3) we confirm that safety tools like ASan and TSan enhance vulnerability detection in Rust up to 26%. We suggest that: 1) researchers should improve comprehension of Rust security and conduct future research on code transpilation, 2) toolchain builders should refine transpilation approaches to better improve security of transpiled code, and 3) developers should employ security tools to mitigate potential security risks. We believe these findings and suggestions will help researchers, toolchain builders, and developers, by providing better guidelines for code transpilation and Rust security in general.**

***Index Terms*—Rust, Transpilation, Security**

I. Introduction

Rust emerges as a highly promising language especially in security-critical domains such as operating systems [1] [2], embedded systems [3] [4], and network infrastructure [5] [6], where the C language has been dominant. Rust inherits most syntactic features of C, but provides strong security guarantees that C lacks by leveraging novel security mechanisms including the ownership model [7] and borrow checker [8]. Meanwhile, Rust adopts a zero-cost abstraction design philosophy and achieves competitive performance [9] with C. Therefore, Rust is of great potential to address the long-standing security issues that C programs suffer from [10].

To fully exploit Rust's security benefits, there have been significant studies on transpiling legacy C programs to Rust (C-to-Rust) [11] [12] [13] [14]. Generally, C-to-Rust transpilation refers to the automated approaches that migrate legacy C codebases to functionally equivalent Rust code, which are more lightweight and cost-effective than manually rewriting C code from scratch. As a result, existing C-to-Rust transpilation strategies, including rule-based ones (e.g., C2Rust [12]) that transpile code according to a set of syntactic rules and generation-based ones (e.g., GPT-4 [15]) that leverage large language models, have demonstrated promising potential in transpiling large-scale C projects to Rust (e.g., the transpilation of the kernel module of Bareflank [16]).

Unfortunately, while existing studies have made significant advances in C-to-Rust, they all assume that the transpiled Rust code is more secure and trustworthy than the corresponding C sources. However, whether this assumption truly holds in practice remains *unknown*. To the best of our knowledge, there has not been a large-scale empirical investigation into the security risks of the Rust code transpiled from C, including the manifestation of vulnerabilities, their taxonomy, and potential mitigations. We argue the lack of this knowledge is largely due to the common misbelief that C-to-Rust can address the vulnerabilities in C because Rust is a safe language.

However, there is no silver bullet and the current security assumption of transpiled Rust code may not hold due to the following three reasons. First, transpilation tools such as C2Rust are designed with the goal of fidelity to preserve the semantic equivalence between corresponding transpilation sources and targets, instead of rectifying or eliminating vulnerabilities during transpilation. As a result, vulnerabilities may persist in the transpiled Rust code (as evidenced by our results in § IV). Second, transpiled Rust code often leverages `unsafe`, a distinctive insecure Rust feature, to accommodate unsafe operations such as unrestricted memory manipulations in C programs, undermining Rust's security guarantees [17] [18]. Consequently, vulnerabilities in Rust code may escape the tight security belt of Rust. Third, security checking tools for Rust, while powerful, often overlook the nuances of transpiled Rust code because these tools are primarily designed with Rust's safety guarantees in mind and thus cannot process the unique multilingual feature in transpiled Rust. For example, Miri [19], a widely used dynamic security tool for Rust, struggles to detect vulnerabilities in external C libraries in transpiled Rust, because this tool is limited to single-language Rust code. Therefore, a systematic investigation of security risks of transpiled Rust is urgently needed.

In this paper, to fill the present gap, we take the first step towards investigating the security risks of transpiled Rust from C programs. Our investigation combines both quantitative and

* The corresponding author.

qualitative methods, aiming to answer the following research questions: **(1) RQ1: Rust security.** Do Rust's built-in security mechanisms (e.g., strong typing and borrow checker) detect these vulnerabilities? **(2) RQ2: Taxonomy.** What categories of vulnerabilities still persist in the transpiled Rust programs? **(3) RQ3: Root causes.** What are the root causes leading to these persisted vulnerabilities? **(4) RQ4: Tool effectiveness.** Are the current state-of-the-art tools effective in detecting or mitigating these vulnerabilities? **(5) RQ5: Case study.** How do vulnerabilities manifest in real-world program and what are the security implications? Answers to these questions are pivotal to advancing transpilation tooling, guiding secure migration practices, and validating automated translation as a credible path to safety. Moreover, the lack of this knowledge negatively impacts three audiences: researchers are unaware of the research gaps and thus miss opportunities to advance the current state-of-the-art, toolchain builders do not know how to improve their tools based on actual needs and areas that encounter issues, and developers lack knowledge on how to mitigate potential security risks in the transpiled Rust programs.

Conducting a large-scale and systematic investigation of security risks in transpiled Rust is non-trivial and must tackle three technical challenges. (**C1**) There is currently no suitable dataset to systematically evaluate security risks in transpiled Rust. To address this, we construct a dataset from the Juliet Test Suite 1.3 [20], a comprehensive benchmark specifically designed to contain a wide range of real-world C vulnerabilities. While Juliet has been extensively analyzed in C [21] [22] [23], our work is the first to systematically examine how these vulnerabilities behave after translation into Rust. Each program in our dataset is carefully crafted to manifest a distinct security flaw, making it a reliable ground truth for insecure behavior. (**C2**) The absence of well-defined evaluation metrics complicates the assessment of security outcomes after transpilation. To tackle this, we design an automated framework that employs two transpilation tools, C2Rust and GPT-4, to translate vulnerable C programs into Rust. We then analyze the resulting programs and classify their outcomes into three categories: statically rejected, dynamically detected, and silently executed. This framework provides a systematic basis for evaluating Rust's safety guarantees and identifying remaining vulnerabilities. (**C3**) The effectiveness of existing dynamic safety tools in detecting residual vulnerabilities remains unclear. To address this, we integrate widely-used runtime analysis tools such as AddressSanitizer (ASan) and ThreadSanitizer (TSan) to empirically evaluate their detection coverage and limitations.

We obtain important findings and insights from this study. First, we observe that Rust's built-in mechanisms successfully detect a significant number of vulnerabilities. C2Rust translations have 9,319 statically rejected and 4,882 dynamically detected cases (14,201 total), while GPT-4 translations have 11,915 and 3,803 respectively (15,718 total). Second, we categorize vulnerabilities into nine classes and find vulnerabilities related to Logic/Code Structure and Resource Management most often execute silently, showing that such flaws are difficult to catch with Rust's type system or runtime checks. Third, we identify three root causes of such detection failures: semantic correction, behavioral masking, and latent unsafe preservation. Finally, we confirm that safety tools like ASan and TSan enhance Rust's ability to uncover vulnerabilities missed by static checks, yielding up to a 26% increase in detection.

Our findings and suggestions have actionable implications for several audiences. Among others, they 1) help researchers improve their comprehension of Rust security and provide valuable contributions to future research on C-to-Rust transpilations; 2) provide feedback to toolchain builders to refine translation accuracy and enhance safety mechanisms, particularly concerning unchecked unsafe patterns; and 3) assist developers in mitigating security risks by encouraging the use of safety tools and adopting safety coding practices during translation.

Contribution. To summarize, this work represents a *first* step towards investigating the security risks of transpiling C programs to Rust, and makes the following contributions:

- **Dataset.** We construct a dataset of vulnerable C programs and their transpiled Rust counterparts, which serves as a reusable benchmark for future work.
- **Empirical study.** We explore the security risks of C transpiled to Rust programs.
- **Findings and insights.** We present empirical results, findings from the study, as well as implications for these results, future challenges, and research opportunities.
- **Open source.** We make our tool, datasets, and empirical results publicly available in the interest of open science: `https://doi.org/10.5281/zenodo.16751699`.

Outline. The rest of this paper is organized as follows. Section II provides the background of this work. Section III presents the approach we use to perform the analysis. Section IV presents results and root cause analysis. 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 And Motivation

In this section, we first present the necessary background knowledge and motivation for this work, by introducing vulnerabilities in C (§ II-A), the Rust programming language (§ II-B), and C-to-Rust transpilation and motivation (§ II-C).

A. Vulnerabilities in C

C has long been the foundation of systems software for its efficiency and low-level control, but its permissive semantics and manual memory management make it highly prone to security flaws. Most high-impact vulnerabilities in operating systems, browsers, and critical infrastructure originate from C code [24] [25], as classic exploits like stack smashing and return-oriented programming (ROP) [26] enable arbitrary code execution and privilege escalation. Despite decades of secure coding practices and improved tooling, memory-related

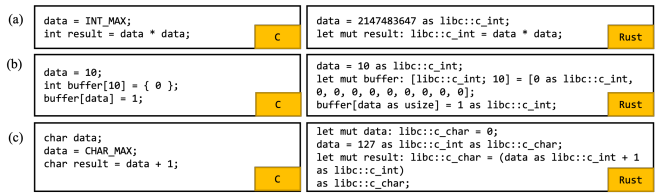


Fig. 1: Vulnerability outcomes of transpiled Rust programs.

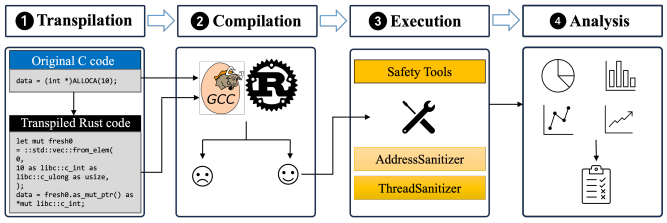


Fig. 2: The workflow of our approach.

vulnerabilities continue to dominate CVE reports, motivating growing interest in safer alternatives such as migrating legacy C code to Rust [27].

B. Rust

Rust enforces strong safety guarantees through its ownership model, borrow checking, and lifetime tracking. Rust permits the use of unsafe features for low-level operations such as hardware interaction and manual memory control [28], yet their misuse underlies all known memory-safety vulnerabilities in Rust [29] [30]. Automated C-to-Rust translation often introduces unsafe blocks to bridge semantic gaps, which, lacking manual review, may create hidden risks. As Rust sees wider adoption in critical systems, assessing and mitigating the security implications of automatically generated unsafe code becomes increasingly vital.

C. C-to-Rust Transpilation and Motivation

The migration of legacy C code to Rust is an emerging trend driven by Rust's safety guarantees and performance benefits. Automated C-to-Rust transpilation has been developed to ease this process and preserve C semantics [12] [31] [32]. However, these tools prioritize behavioral preservation rather than security, often inserting unsafe blocks to emulate operations that violate Rust's ownership and borrowing rules. As a result, some vulnerabilities in the original C code are implicitly fixed by Rust's stricter type system, while others persist or manifest in different forms.

To investigate this, we conduct an initial exploration of transpiling vulnerable C programs into Rust and observe diverse outcomes. As shown in Fig. 1, we observe three distinct behaviors: some programs fail to compile (Fig. 1(a)), others compile successfully but encounter runtime errors (Fig. 1(b)), while a notable portion run successfully despite originating from C code with known vulnerabilities (Fig. 1(c)). These findings point to the need for a thorough empirical study to understand the underlying causes and evaluate the effectiveness of Rust's safety features in such transpiled code.

III. Approach

In this section, we present our approach to conduct the study. We first introduce our workflow (§ III-A) and subsequently define the threat model (§ III-B). Then, we describe each process in detail, including transpilation (§ III-C), compilation (§ III-D), execution (§ III-E) and manual analysis (§ III-F).

A. Workflow

The workflow employed in this study is outlined as Fig. 2. First, we transpile (❶) the original C programs into Rust through two pipelines: C2Rust, a state-of-the-art mechanical transpiler, and GPT-4, a leading LLM-based code generator. Second, we compile (❷) both the original C programs and the transpiled Rust programs using their default build configurations, in order to avoid the influence of additional compiler optimizations. Third, we execute (❸) the compiled binaries with identical inputs in isolated environments and evaluate runtime behavior using dynamic analysis tools to detect vulnerabilities missed by Rust's static checks. Finally, we analyze (❹) the transpiled Rust code through manual inspection of a stratified random sample to identify root causes behind silent correctness.

B. Threat Model

Our study relies on several key assumptions regarding compilation toolchains, the transpilation pipelines and runtime analysis utilities. First, we assume that the compilation toolchains for both C and Rust are reliable. Specifically, we treat `gcc` and `rustc` as correct and stable compilers that generate deterministic binaries under identical inputs. Second, we assume that both C2Rust and GPT-4 produce translations that are largely semantically faithful to the source C programs. While GPT-4 may occasionally introduce subtle deviations due to the nature of LLM-based generation, evaluating such deviations is beyond the scope of this work, and prior research has extensively examined the limitations of LLM-based code generation [33]. Third, we assume runtime analysis tools operate as intended and do not introduce false positives or negatives that would affect the conclusions of our study.

C. Transpilation

First, we transpile C programs into Rust using C2Rust and GPT-4. We select C2Rust as it is the only actively maintained C-to-Rust transpiler currently available. Other tools such as Corrode [13] and Citrus [34] are outdated and incompatible with modern Rust toolchains. To represent the LLM-based approach, we use GPT-4 for three reasons. First, it is one of the most widely adopted large language models in practical code translation workflows. Second, recent empirical studies have shown that GPT-4 achieves the highest success rate among seven evaluated LLMs in generating correct and compilable Rust code from C inputs [33]. Third, our methodology is model-agnostic and can also be applied to other LLMs, making

```
Translate the following C code into semantically equivalent Rust code.
Retain the implementation guarded by #ifndef OMITBAD and #ifdef
INCLUDEMAIN, and ignore the good variant.Print only the Rust code.
$SOURCE_CODE
```

Fig. 3: Prompt template for GPT-4.

GPT-4 a representative and practical choice. Fig. 3 shows the prompt used to guide the LLM in generating semantically equivalent Rust code for vulnerable C functions.

D. Compilation

After transpilation, we compile both the original C code and their transpiled Rust counterparts using standard toolchains. The C programs are compiled with GCC at the default optimization level (-O0), while the Rust programs are built with rustc in its default debug configuration. We adopt these default settings to ensure consistent behavior across languages without introducing additional compiler optimizations that might obscure differences caused by transpilation.

E. Execution

We then execute both the original and transpiled binaries using identical input parameters in isolated runtime environments to ensure fair comparison. For the transpiled Rust programs, we integrate dynamic analysis tools to detect runtime issues missed by Rust's static checks, particularly in unsafe code. Specifically, we use ASan to detect memory safety violations such as buffer overflows and use-after-free errors, and TSan to catch data races. We collect execution results including sanitizer reports, crash traces, and behavioral logs for further inspection.

F. Manual Analysis

Finally, we analyze the subset of transpiled Rust programs that compile and execute successfully (i.e., exit code 0), as these cases may conceal latent vulnerabilities undetected by Rust's safety checks. Because exhaustive review is infeasible, we apply a stratified sampling method that groups programs by vulnerability category and translation approach, ensuring balanced coverage. We then randomly sample each group to achieve a 95% confidence level with a 3% margin of error. For each selected program, we manually inspect the translated code and its runtime behavior to identify translation patterns that cause vulnerabilities to persist, be masked, or manifest differently in Rust.

IV. Experimental Results

In this section, we present the empirical results. We first present the research questions guiding the experiments (§ IV-A), the experimental setup (§ IV-B), the datasets (§ IV-C). We then present the results (§ IV-D to § IV-G).

A. Research Questions

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

RQ1: Rust security. Do Rust's built-in security mechanisms (e.g., strong typing and borrow checker) detect these vulnerabilities?

RQ2: Taxonomy. What categories of vulnerabilities still persist in the transpiled Rust programs?

RQ3: Root cause. What are the root causes leading to these persisted vulnerabilities?

RQ4: Tool effectiveness. Are the current state-of-the-art tools effective in detecting or mitigating these vulnerabilities?

RQ5: Case study. How do vulnerabilities manifest in real-world program and what are the security implications?

B. Experimental Setup

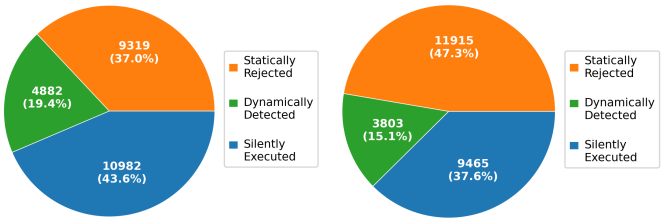
All the experiments and measurements are performed on a server with one 12 physical Intel i7 core (20 hyperthread) CPU and 128 GB of RAM. The machine runs 64-bit Ubuntu 24.04 Linux with kernel version 6.8.0. We use C2Rust version 0.20.0 and GPT-4 as the C-to-Rust translation tools, and run the translated Rust programs with rustc version 1.85.0.

C. Datasets

To conduct our evaluation, we construct a dataset composed of a curated subset of the Juliet Test Suite 1.3 [20]. The Juliet test suite, developed as part of the Software Assurance Reference Dataset (SARD), is a widely adopted benchmark for security-focused static and dynamic analysis. We extract 25,183 small C programs from Juliet that (1) are compatible with compilation and execution on modern Linux systems, and (2) collectively cover a wide range of vulnerability types across 82 CWEs, such as stack-based buffer overflows (CWE-121), integer overflows (CWE-190), and memory leaks (CWE-401). Each test case typically includes two code variants: a bad version that contains a known vulnerability and a good version that implements a safe alternative. In our study, we focus exclusively on the bad variants, as they are designed to trigger the vulnerability under investigation and provide a consistent baseline for analyzing how such security flaws behave after translation.

D. RQ1: Rust security

To answer **RQ1**, we evaluate the extent to which Rust's built-in safety mechanisms, including compile-time checks enforced by the type system and borrow checker as well as runtime checks such as bounds and overflow detection, detect vulnerabilities in transpiled code. We first compile each translated program and record whether compilation fails due to safety violations. This step captures vulnerabilities that are statically prevented. We then execute all successfully compiled binaries under controlled conditions and monitor their runtime behavior. To ensure reproducibility, we repeat each execution 10 times with a 5-second timeout.



(a) C2Rust. (b) GPT-4.

Fig. 4: Distribution of vulnerability outcomes in dataset.

```c
char * data;
char * dataBadBuffer = (char *)
    ALLOCA(50*sizeof(char));
if(staticTrue) {
    data = dataBadBuffer;
    data[0] = '\0';
}
```
C

```rust
let mut data: Vec<u8>;
let mut dataBadBuffer: Vec<u8>
    = vec![0; 50];
if staticTrue {
    data = dataBadBuffer;
    data[0] = b'\0';
}
```
GPT-transpiled Rust

```rust
let mut data: *mut libc::c_char = 0 as *mut libc::c_char;
let mut fresh0 = ::std::vec::from_elem(0,
    (50 as libc::c_int as libc::c_ulong)
        .wrapping_mul(::core::mem::size_of::<libc::c_char>() as
libc::c_ulong) as usize,);
let mut dataBadBuffer: *mut libc::c_char =
    fresh0.as_mut_ptr() as *mut libc::c_char;
if staticTrue != 0 {
    data = dataBadBuffer;
    *data.offset(0 as libc::c_int as isize) = '\0' as i32 as
libc::c_char;
}
```
C2Rust-transpiled Rust

Fig. 5: An example of buffer initialization in transpiled code.

We classify each case into three outcomes that reflect Rust's built-in detection capabilities. (1) **Statically Rejected**, if compilation fails due to violations detected by Rust's type system or borrow checker. This outcome indicates that the compiler successfully blocks unsafe constructs at build time. (2) **Dynamically Detected**, if the program compiles but crashes or panics during execution, reflecting Rust's runtime safety mechanisms such as bounds checking and integer overflow detection. (3) **Silently Executed**, if the program compiles and runs without observable errors, implying that neither static nor runtime checks detect the underlying vulnerability.

Fig. 4 presents the distribution of these outcomes for both C2Rust and GPT-4 transpilation pipelines. For C2Rust, 9,319 cases fail to compile due to safety violations detected by Rust's type system and borrow checker, while an additional 4,882 cases trigger runtime errors, yielding a total of 14,201 vulnerability detections. In comparison, GPT-4 results in 11,915 statically rejected cases and 3,803 dynamically detected cases, amounting to 15,718 detections overall. These findings indicate that Rust's compiler and runtime safety checks effectively prevent a significant portion of unsafe code from executing.

Notably, GPT-4 achieves a higher number of compile-time rejections accompanied by fewer runtime and undetected cases than C2Rust. This improvement is largely attributed to GPT-4's generation-based strategy, which often rewrites unsafe C idioms into safer, more idiomatic Rust abstractions. For example, it frequently replaces raw pointers with data structures such as `Vec` or `Box` and restructures memory management

TABLE I: Vulnerability outcomes of different categories.

	Statically Rejected		Dynamically Detected		Silently Executed	
	C2Rust	GPT-4	C2Rust	GPT-4	C2Rust	GPT-4
MS	1633	3933	2617	1192	4260	3385
NI	3871	4385	1680	2007	3755	2914
IV	1994	966	0	159	0	869
RM	721	869	309	207	894	848
C	108	64	0	25	36	55
LCS	868	1236	47	128	1477	1028
ATC	18	197	141	58	288	192
MC	34	160	88	23	126	65
O	72	105	0	4	146	109

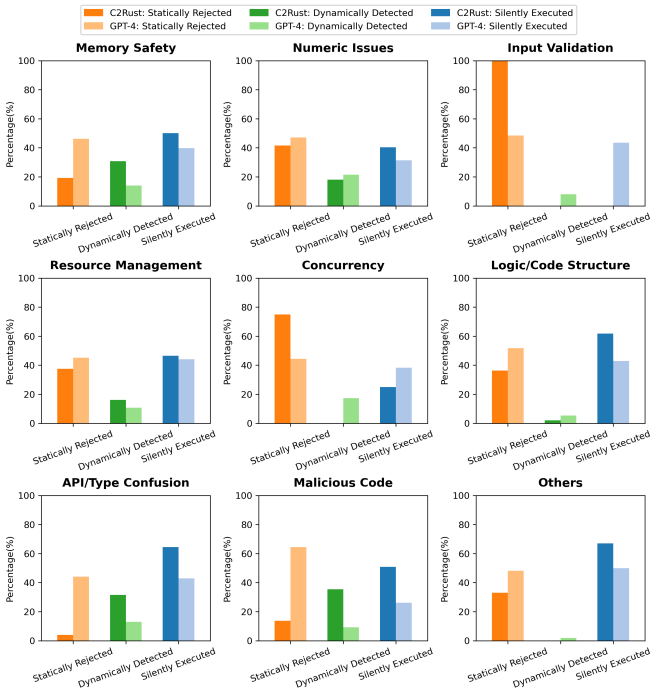


Fig. 6: Vulnerability outcomes of different categories.

patterns, thereby making the code more compatible with Rust's ownership and initialization checks. For example, as illustrated in Fig. 5, GPT-4 translates `char *` buffers into `Vec<u8>`, which inherently triggers ownership and initialization validations during compilation. Conversely, C2Rust preserves raw pointer operations that circumvent these compile-time checks, resulting in code that compiles successfully but may harbor latent vulnerabilities.

E. RQ2: Taxonomy

To address **RQ2**, we categorize the original C vulnerabilities into nine classes, including Memory Safety (MS), Numeric Issues (NI), Input Validation (IV), Resource Management (RM), Concurrency (C), Logic/Code Structure (LCS), API/-Type Confusion (ATC), Malicious Code (MC), and Others (O), and analyze how they manifest in the transpiled Rust programs. Table I summarizes, for both C2Rust and GPT-4 translations, the numbers of cases that are statically rejected, dynamically detected, or silently executed in each vulnerability

class. Fig. 6 illustrates how these outcomes are distributed across the classes, enabling a direct comparison between the two translation approaches.

The results indicate that Rust's built-in safety mechanisms vary significantly in their effectiveness across vulnerability categories. Categories such as IV and C exhibit very high static rejection rates (up to 100% and 75%, respectively, for C2Rust), indicating that Rust's static type system and ownership model robustly prevent many unsafe constructs in these domains. Conversely, MS and NI show a more balanced distribution, with a notable share of cases escaping static checks and leading to runtime failures or silent execution. Particularly, LCS, RM, and O categories stand out with the highest proportions of silently executed vulnerabilities, often exceeding 45% and in some cases above 60%. These results underscore intrinsic limitations of Rust's compile-time and runtime checks in capturing semantic or contextual flaws such as logical errors or resource leaks, which typically require advanced static analysis or formal verification techniques beyond the language's built-in safety guarantees.

While GPT-4 translations generally produce higher rates of static rejection in many vulnerability categories, C2Rust demonstrates stronger static rejection in specific categories such as IV and C. This difference likely stems from C2Rust's rule-based, conservative translation approach, which tends to preserve low-level memory and API usage patterns from the original C code that are prone to triggering Rust's ownership and type system checks. Moreover, both translation approaches leave a substantial number of vulnerabilities undetected during compilation and runtime. These persistent vulnerabilities are especially prevalent in categories related to complex program logic and resource management, indicating that Rust's built-in safety checks—primarily focused on memory safety and type correctness—are insufficient to identify security flaws that require deeper semantic understanding or advanced analysis techniques. This limitation underscores the challenges of relying solely on Rust's safety mechanisms to eliminate all security risks carried over from the original C programs.

F. RQ3: Root cause

To answer **RQ3** and investigate the underlying causes of undetected vulnerabilities, we conduct a qualitative analysis of a stratified sample drawn from both C2Rust and GPT-4 translations.

We perform stratified random sampling within the Silently Executed subset for each translation method, stratifying by vulnerability category to ensures representative coverage across common error types. Using the Krejcie–Morgan formula [35] with a 95% confidence level and 3.0% margin of error, we calculate sample sizes of 972 out of 10,982 for C2Rust and 959 out of 9,465 for GPT-4. Samples are proportionally allocated across categories based on their respective Silently Executed counts. Table II summarizes the distribution of sampled cases per category for both methods.

Following sampling, we classify detection failures into three root causes: semantic correction, behavioral masking, and

TABLE II: Sample sizes for root cause analysis.

Category	MS	NI	IV	RM	C	LCS	ATC	MC	O
C2Rust	377	333	0	79	3	131	25	11	13
GPT-4	343	295	88	86	6	104	19	7	11

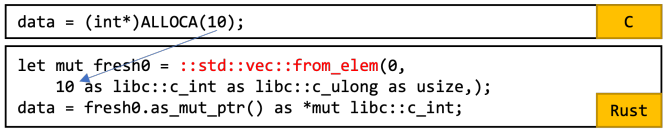


Fig. 7: An example for semantic correction.

latent unsafe preservation. First, semantic correction occurs when the translation from C to Rust modifies the program's semantics to eliminate the original vulnerability. This typically stems from Rust's strong type system, ownership model, and safe memory abstractions, which enforce correct buffer sizing and restrict unchecked pointer access. These language-level guarantees fundamentally alter how memory is allocated and managed, removing opportunities for certain classes of errors to occur. As a result, security flaws arising from improper memory allocation or unchecked accesses are prevented at the semantic level in Rust, effectively eliminating the vulnerability. For example, as shown in Fig. 7, the original C code uses `ALLOCA(10)` to allocate 10 bytes for 10 integers, which is insufficient and leads to potential overflows. The Rust translation replaces this with `vec::from_elem`, ensuring correct allocation based on element type and count, thus preventing the overflow.

Second, behavioral masking arises when vulnerabilities persist in the Rust translation but fail to manifest due to semantic differences between C and Rust, such as variations in memory layout, alignment, initialization, or arithmetic operations. These divergences can prevent buffer overflows, memory corruptions, or integer overflows from triggering erroneous behaviors at runtime, effectively masking the vulnerability. Such discrepancies are often subtle and highly dependent on platform-specific details, making them particularly challenging to detect through conventional testing. For example, as depicted in Fig. 8, the original C operation `INT_MAX + 1` invokes undefined behavior that may cause crashes or exploits, while the Rust translation `i32::MAX.wrapping_add(1)` produces well-defined wrapping behavior (-2147483648) without triggering a panic or program termination. This semantic shift transforms a hazardous operation into a safe value transition, allowing the program to execute normally while concealing the underlying vulnerability.

Third, latent unsafe preservation occurs when unsafe constructs in the translated Rust code reproduce the original C vulnerability, but the resulting flaw remains undetected due to Rust's lack of runtime enforcement within unsafe blocks. Although Rust enforces memory safety in safe code, operations inside unsafe blocks—such as raw pointer arithmetic and unchecked writes—bypass these protections. This issue is exacerbated when the transpilation process produces code

```c
int data;
data = INT_MAX;
int result = data + 1;
```

```rust
let mut data: i32;
data = i32::MAX;
let result = data.wrapping_add(1);
```

Fig. 8: An example for behavioral masking.

```c
char * dataBadBuffer = (char *)ALLOCA(50*sizeof(char));
for (i = 0; i < 100; i++){
    data[i] = source[i];}
```

```rust
let mut fresh0 = ::std::vec::from_elem(0,
    (50 as libc::c_int as libc::c_ulong)
        .wrapping_mul(::core::mem::size_of::<libc::c_char>() as
libc::c_ulong)
        as usize,);
let mut dataBadBuffer: *mut libc::c_char =
    fresh0.as_mut_ptr() as *mut libc::c_char;
i = 0 as libc::c_int as size_t;
while i < 100 as libc::c_int as size_t {
    *data.offset(i as isize) = source[i as usize];
    i = i.wrapping_add(1);
    i;}
```

Fig. 9: An example for latent unsafe preservation.

with extensive unsafe usage that closely mirrors the original C logic, as it effectively nullifies Rust's compile-time guarantees. As illustrated in Fig. 9, the C code allocates 50 bytes and copies a larger buffer into it, causing a buffer overflow. The Rust translation mirrors this logic using unsafe raw pointers. Since no bounds checks or safety assertions are applied at runtime in such contexts, the overflow occurs silently without any error or warning, preserving the latent vulnerability.

G. RQ4: Tool effectiveness

To answer **RQ4**, we evaluate the effectiveness of dynamic safety tools in detecting residual vulnerabilities that escape Rust's built-in security checks after transpilation. Specifically, we evaluate two widely adopted sanitizers (ASan and TSan) across both translation pipelines, as they are capable of capturing a broad range of low-level errors, including memory misuse, concurrency violations, and use of uninitialized data.

Table III and Table IV present the number and proportion of vulnerabilities detected by each tool in C2Rust- and GPT-transpiled code, respectively. Overall, these sanitizers yield moderate improvements in vulnerability detection across both pipelines, with relative increases in detection rates ranging from 0.13% to 26.83%. This confirms that dynamic tools can uncover residual vulnerabilities overlooked by Rust's static checks. ASan shows the most significant improvement in detecting memory safety violations, while TSan contributes to race detection in GPT-translated programs. However, certain vulnerability categories, such as input validation, show limited or no improvements under dynamic analysis, highlighting intrinsic constraints of sanitizers in identifying logical or semantic defects.

Furthermore, for C2Rust-generated code, TSan shows slight decreases in detection for resource management (-0.15%) and logic/code structure (-0.33%). A closer examination reveals that the programs responsible for these decreases belong to CWE-401 (Memory Leak), CWE-563 (Unused Variable),

TABLE III: Vulnerabilities detected by different tools for C2Rust-transpiled code.

	ASan		**TSan**	
	Number	Proportion(%)	Number	Proportion(%)
MS	6,533(+2,283)	76.77(+26.83)	4,385(+135)	51.53(+1.59)
NI	5,582(+31)	59.98(+0.33)	5,585(+34)	60.02(+0.37)
IV	1,994(+0)	100(+0)	1,994(+0)	100(+0)
RM	1,421(+391)	73.86(+20.33)	**1,027(-3)**	**53.38(-0.15)**
C	108(+0)	75(+0)	108(+0)	75(+0)
LCS	915(+0)	38.25(+0)	**907(-8)**	**37.92(-0.33)**
ATC	223(+64)	49.89(+14.32)	202(+43)	45.19(+9.62)
MC	122(+0)	49.19(+0)	122(+0)	49.19(+0)
O	72(+0)	33.03(+0)	72(+0)	33.03(+0)

Numbers in parentheses indicate changes relative to Rust's built-in checks.

TABLE IV: Vulnerabilities detected by different tools for GPT-transpiled code.

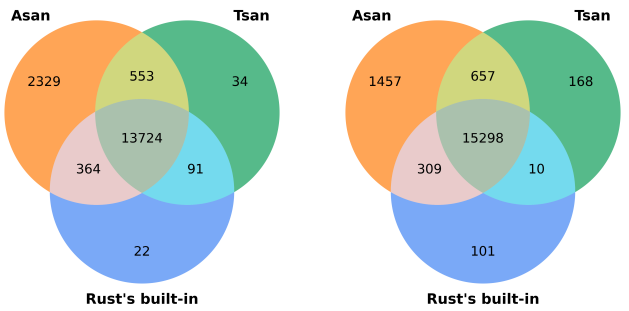
	ASan		**TSan**	
	Number	Proportion(%)	Number	Proportion(%)
MS	6,395(+1,270)	75.15(+14.93)	5,221(+96)	61.35(+0.13)
NI	6,486(+94)	69.70(+1.01)	6,445(+53)	69.26(+0.57)
IV	1,189(+64)	59.63(+3.21)	1,151(+26)	57.72(+1.30)
RM	1,368(+292)	71.10(+15.17)	1,142(+66)	59.36(+3.43)
C	121(+32)	84.03(+22.22)	120(+31)	83.33(+21.52)
LCS	1,462(+98)	61.12(+4.10)	1,426(+62)	59.62(+2.60)
ATC	356(+101)	79.64(+22.59)	279(+24)	62.42(+5.37)
MC	194(+11)	78.23(+4.44)	193(+10)	77.82(+4.03)
O	150(+41)	68.81(+18.81)	156(+47)	71.56(+21.56)

Numbers in parentheses indicate changes relative to Rust's built-in checks.

and CWE-617 (Reachable Assertion). These cases suggest that transpilation can produce code patterns that reduce the likelihood of triggering TSan's runtime checks. For example, memory leaks may be less observable due to modified allocation and deallocation patterns, unused variables may no longer affect execution flow, and assertions may become less reachable as a result of control-flow restructuring. Consequently, TSan may fail to report such errors, reflecting a mismatch between its heuristics and the runtime behavior of transpiled code.

In addition to the quantitative comparison, we visualize the intersection of vulnerabilities detected by tools for the two transpilation approaches in Fig. 10. For C2Rust-transpiled programs, 13,724 vulnerabilities are jointly detected by ASan, TSan, and Rust's built-in checks, with ASan identifying an additional 2,329 mainly related to memory safety and resource management. TSan detects only 34 unique vulnerabilities, with 553 overlapping ASan, reflecting its focus on concurrency issues. GPT-transpiled programs show a similar pattern: 15,298 vulnerabilities are detected by all three tools, ASan adds 1,457, TSan 168, and Rust's built-in checks 101. Pairwise overlaps remain limited. These results highlight the complementary but asymmetric roles of ASan and TSan, with ASan covering a broad range of memory issues and TSan primarily diagnosing concurrency errors.

We also experiment with other dynamic tools, such as Miri, but exclude them from detailed discussion due to space limitations. Miri is largely incompatible with C2Rust-transpiled code because it cannot handle FFI calls to sys-



(a) C2Rust. (b) GPT-4.

Fig. 10: Intersection of vulnerabilities detected by tools.

```c
for (to = user_args, av = NewArgv +
1; (from = *av); av++) {
    while (*from) {
        if (from[0] == '\\' &&
!isspace((unsigned char)from[1])){
            from++;}
        *to++ = *from++;
    }
}
```
C

```
let mut to = user_args.as_mut_ptr();
while { from = *av; !from.is_null() }
{
    while *from != 0 {
        if *from == b'\\' as i8 &&
*from.add(1) != 0 && !(*from.add(1)
as u8).is_ascii_whitespace() {
        from = from.add(1);
        }
        *to = *from;
        to = to.add(1);
        from = from.add(1);
    }
}
```
GPT-transpiled Rust

```
loop {
    from = *av;
    if from.is_null() break;
    while *from != 0 {
        if *from.offset(0 as
libc::c_int as isize) as libc::c_int
== '\\' as i32
&& *(*__ctype_b_loc()).offset(*from.
offset(1 as libc::c_int as isize) as
libc::c_uchar as libc::c_int as
isize,) as libc::c_int & _ISspace as
libc::c_int as libc::c_ushort as
libc::c_int == 0 {
            from = from.offset(1);
            from;
        }
        let fresh0 = from;
        from = from.offset(1);
        let fresh1 = to;
        to = to.offset(1);
        *fresh1 = *fresh0;
    }
}
```
C2Rust-transpiled Rust

Fig. 11: Original and transpiled code for CVE-2021-3156.

tem or C standard library functions. For instance, the translated code for `srand((unsigned)time(NULL))` invokes `libc::srand` and `libc::time`, which Miri cannot execute, leading to premature termination or incomplete analysis.

H. RQ5: Case study

To address **RQ5**, we present two representative case studies illustrating how C program vulnerabilities behave after automated transpilation.

The first case is CVE-2021-3156 [36], a heap-based buffer overflow vulnerability in sudo caused by improper handling of command-line arguments through pointer arithmetic. As depicted in Fig. 11, a trailing backslash makes `from` advance beyond the allocated buffer, leading to out-of-bounds reads and writes (`*to++ = *from++`), thus triggering a heap overflow. The code produced by C2Rust remains close to the original, mapping pointer arithmetic to raw pointer operations in Rust. As a result, the same off-by-one error persists and the overflow can still be triggered. In contrast, the GPT-4 translation adopts more idiomatic Rust constructs, (e.g., using `is_ascii_whitespace` for character checks), while still retains unsafe pointer writes. Nevertheless, both translations preserve the core logic in which pointer from may advance past the buffer boundary, causing the statement `*to++ = *from++`) (or its equivalent) to perform out-of-bounds accesses. This case highlights that vulnerabilities from unsafe pointer arithmetic are unlikely to be eliminated automatically, even in a language with strict safety guarantees.

```c
ssize_t pxa3xx_gcu_write(struct file *file, const char *buff, size_t count,
loff_t *offp) {
    struct pxa3xx_gcu_batch *buffer;
    int words = count / 4;
    ret = copy_from_user(buffer->ptr, buff, words * 4);
    return words * 4;
}
```
C

```
pub unsafe extern "C" fn pxa3xx_gcu_write(mut file: *mut file, mut buff:
*const libc::c_char, mut count: size_t, mut offp: *mut loff_t,) -> ssize_t {
    let mut buffer: *mut pxa3xx_gcu_batch = 0 as *mut pxa3xx_gcu_batch;
    let mut words: libc::c_int = (count / 4 as libc::c_int as size_t) as
libc::c_int;
    ret = copy_from_user((*buffer).ptr, buff as *const libc::c_void,
        (words * 4 as libc::c_int) as size_t,);
    return (words * 4 as libc::c_int) as ssize_t;
}
```
C2Rust-transpiled Rust

```
pub extern "C" fn pxa3xx_gcu_write(file: *mut File, buff: *const u8, count:
size_t, offp: *mut loff_t,) -> ssize_t {
    let buffer: *mut Pxa3xxGcuBatch;
    let words = count / 4;
    unsafe { ret = copy_from_user((*buffer).ptr, buff as *const c_void,
words * 4); }
    (words * 4) as ssize_t
}
```
GPT-transpiled Rust

Fig. 12: Original and transpiled code for CVE-2022-39842.

The second case is CVE-2022-39842 [37], a critical integer overflow vulnerability in the Linux kernel's PXA3XX graphics processing unit driver. The original C code, along with C2Rust and GPT-4 based Rust translations, are shown in Fig. 12 for comparison. The vulnerability originates from assigning the 64-bit unsigned integer variable `count` (of type `size_t`) to the 32-bit signed integer variable `words` after dividing `count` by 4. For large `count`, this truncation causes `words` to overflow, producing an incorrect value. This erroneous value is then used as the size argument to `copy_from_user`, leading to a buffer overflow in `buffer->ptr`. In the C2Rust translation, the expression `(count/4) as libc::c_int` explicitly casts the 64-bit value to a 32-bit signed integer, thus faithfully preserving the overflow-prone behavior of the original code. GPT-4, in contrast, infers `words` as `usize` (a 64-bit unsigned integer) and performs the computation in 64-bit arithmetic, preventing truncation and matching the copy size to the input. This case demonstrates that LLM-based translation can sometimes yield safer code by implicitly leveraging Rust's type system, but such improvements are not systematic or guaranteed without explicit verification.

V. Implications

This paper investigates the safety of Rust programs automatically transpiled from C, highlighting residual vulnerabilities and limitations of current tooling. We outline practical implications for researchers, toolchain designers, and developers, and suggest directions for future work.

For researchers. Despite Rust's strong safety guarantees, vulnerabilities from C can persist after automated transpilation, especially when unsafe blocks are used. Researchers should study how language semantics, memory models, and initialization rules affect vulnerability preservation, evaluate diverse translation techniques, and develop analyses targeting unsafe Rust idioms. Integrating formal verification with transpilation

and examining the effects of compiler optimizations or runtime environments are promising directions.

For toolchain designers. Current transpilers often perform syntax-level conversion, leaving unsafe constructs unchanged and underutilizing Rust's ownership and type systems. Designers should incorporate semantic analysis and post-translation refactoring to convert unsafe idioms into safer abstractions, integrate static analysis and symbolic execution, and provide mechanisms to flag high-risk code regions. Leveraging Rust's borrow checker and type inference can further improve safety and maintainability.

For developers. Automated transpilation tools such as C2Rust or GPT-4 do not guarantee vulnerability-free Rust code. Generated code should be treated as a baseline, with manual refactoring of unsafe blocks, replacement of low-level memory operations, and code review essential for security. Applying dynamic safety tools such as ASan and TSan can uncover issues beyond Rust's static checks. Secure development workflows combining automated transpilation with targeted manual review and testing can significantly enhance code reliability.

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 effects when removal is not possible.

External Validity. Our findings may not generalize to all C codebases or transpilation scenarios in practice. We do not evaluate real-world C projects in our study, as they typically lack labeled ground truth needed to trace how known vulnerabilities manifest after translation. While our evaluation covers a range of vulnerability types and code patterns, it may not reflect the complexity, scale, and coding conventions of large industrial systems.

Internal Validity. Errors in our analysis pipeline, such as misclassification of root causes or incorrect interpretation of sanitizer reports, could affect the validity of our results. While we use stratified sampling and manual inspection to mitigate mislabeling, some edge cases may still be inaccurately analyzed. Additionally, differences in compiler optimization behaviors, runtime environments, or toolchain versions could introduce uncontrolled variables. We attempt to control for this by running all experiments under identical conditions and using consistent compiler flags.

Construct Validity. We use safety tools to detect runtime issues, but some reported findings may be false positives or non-critical. Moreover, these techniques do not cover all vulnerability classes, such as logic errors or subtle API misuse. Furthermore, our analysis also assumes that GPT-4 translations are semantically faithful to the original C programs. However, this assumption may not always hold because LLM-based approaches can introduce subtle deviations or hallucinations that affect program behavior. Such deviations may lead to discrepancies between the intended and observed vulnerability outcomes, thereby limiting the validity of our findings.

VII. Related Work

C Security. The security challenges of C have been extensively studied, resulting in tools for detecting and mitigating vulnerabilities [38]. Static analysis tools such as Clang Static Analyzer [39] and Frama-C [40] detect buffer overflows and use-after-free errors at compile time but often suffer from high false positives and limited scalability. Dynamic tools like AddressSanitizer [41] and Valgrind [42] offer higher accuracy by monitoring execution, though they depend on test coverage and incur significant runtime overhead. Formal verification tools such as CBMC [43] provide strong correctness guarantees but require substantial manual effort, limiting their applicability to large systems. Despite these advances, such techniques do not resolve C's fundamental memory-safety flaws. Consequently, migrating C code to memory-safe languages like Rust has emerged as a complementary solution, raising important questions about the security impact of automated transpilation and motivating systematic evaluation of its effectiveness.

C-to-Rust Translation. Automated C-to-Rust transpilation has progressed notably in recent years, with several tools aiming to convert legacy C code into Rust while preserving functionality and improving code quality [44]. Rule-based tools rely on syntax-driven transformations, with explicit unsafe annotations and lifetime inference to achieve near-complete semantic equivalence at scale [17]. More recently, generation-based approaches employing large language models have emerged, offering greater flexibility in handling complex code patterns beyond traditional rule-based methods [14] [45]. Despite these advances, existing studies primarily evaluate translation correctness and code quality, with little empirical evidence on how vulnerabilities in C programs manifest after translation [46] [47] [48]. Understanding whether and how security issues persist in transpiled Rust code remains an open research challenge, motivating systematic investigation from a security perspective.

VIII. Conclusion

This paper presents the first empirical study on security risks in transpiling C programs to Rust. To this end, we build a dataset of 25,183 vulnerable C programs and translate them using C2Rust and GPT-4. We then compile and execute the transpiled Rust code and find that Rust's built-in checks detect a large portion of vulnerabilities, particularly in categories such as input validation and concurrency. Through root cause analysis, we identify semantic correction, behavioral masking and latent unsafe preservation as key factors behind vulnerabilities escaping detection. Furthermore, dynamic tools improve vulnerability detection by up to 26%. This work represents a first step towards improving the security of C to Rust. We hope our paper inspires a symbiotic ecosystem where researchers, language designers, and developers work together to increase Rust security.

References

[1] S. Lankes, J. Breitbart, and S. Pickartz, "Exploring rust for unikernel development," in *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, 2019, pp. 8–15.

[2] S. D. Simone, "Linux 6.1 officially adds support for rust in the kernel," 2025. [Online]. Available: https://www.infoq.com/news/2022/12/linux-6-1-rust

[3] A. Sharma, S. Sharma, S. R. Tanksalkar, S. Torres-Arias, and A. Machiry, "Rust for embedded systems: Current state and open problems," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2296–2310.

[4] M. Noseda, F. Frei, A. Rüst, and S. Künzli, "Rust for secure iot applications: why c is getting rusty," in *Embedded World Conference, Nuremberg, Germany, 21-23 June 2022*. WEKA, 2022.

[5] P. Kampanakis, "Introducing s2n-quic, a new open-source QUIC protocol implementation in Rust," 2022. [Online]. Available: https://aws.amazon.com/blogs/opensource/introducing-s2n-quic-open-source-protocol-implementation/

[6] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, 2020, pp. 419–434.

[7] "What is ownership?" [Online]. Available: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

[8] "References and borrowing." [Online]. Available: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

[9] N. D. Matsakis and F. S. Klock, "The rust language," in *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*, 2014, pp. 103–104.

[10] O. JE and T. CT, "Why scientists are turning to rust," *Nature*, vol. 588, p. 185, 2020.

[11] A. Fromherz and J. Protzenko, "Compiling c to safe rust, formalized," *arXiv preprint arXiv:2412.15042*, 2024.

[12] "C2rust: Automatic c to rust translation," https://github.com/immunant/c2rust, 2025.

[13] "Corrode: Automatic semantics-preserving translation from c to rust," https://github.com/jameysharp/corrode, 2016.

[14] H. F. Eniser, H. Zhang, C. David, M. Wang, M. Christakis, B. Paulsen, J. Dodds, and D. Kroening, "Towards translating real-world code with llms: A study of translating to rust," *arXiv preprint arXiv:2405.11514*, 2024.

[15] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[16] "Transpiling a kernel module to rust: The good, the bad and the ugly," 2025. [Online]. Available: https://immunant.com/blog/2020/06/kernel_modules/

[17] 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. 1–29, 2021.

[18] J. Hong, "Improving automatic c-to-rust translation with static analysis," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2023, pp. 273–277.

[19] "Miri," 2025. [Online]. Available: https://github.com/rust-lang/miri

[20] P. E. Black and P. E. Black, *Juliet 1.3 test suite: Changes from 1.2*. US Department of Commerce, National Institute of Standards and Technology ..., 2018.

[21] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[22] A. Ouadjaout and A. Miné, "A library modeling language for the static analysis of c programs," in *International Static Analysis Symposium*. Springer, 2020, pp. 223–247.

[23] A. Kaur and R. Nayyar, "A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code," *Procedia Computer Science*, vol. 171, pp. 2023–2029, 2020.

[24] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

[25] R. Bagnara, A. Bagnara, and P. M. Hill, "The misra c coding standard and its role in the development and analysis of safety-and security-critical embedded software," in *Static Analysis: 25th International Symposium, SAS 2018, Freiburg, Germany, August 29–31, 2018, Proceedings 25*. Springer, 2018, pp. 5–23.

[26] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, pp. 1–34, 2012.

[27] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static c code analyzers for vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, 2022, pp. 544–555.

[28] "Unsafe rust," 2025. [Online]. Available: https://doc.rust-lang.org/book/ch20-01-unsafe-rust.html

[29] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-safety challenge considered solved? an in-depth study with all rust cves," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–25, 2021.

[30] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[31] 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 *Proceedings of the ACM/IEEE 44th international conference on software engineering: companion proceedings*, 2022, pp. 354–355.

[32] X. Wu and B. Demsky, "Genc2rust: Towards generating generic rust code from c," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 664–664.

[33] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[34] "Citrus: Convert c to rust," 2025. [Online]. Available: https://gitlab.com/citrus-rs/citrus

[35] R. V. Krejcie and D. W. Morgan, "Determining sample size for research activities," *Educational and psychological measurement*, vol. 30, no. 3, pp. 607–610, 1970.

[36] "Cve-2021-3156," 2025. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156

[37] "Cve-2022-39842," 2025. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-39842

[38] P. Chatzipanagiotou and P. Katsaros, "Vulnerability assessment for c programs using the infer static analyzer," 2022.

[39] "Clang static analyzer," https://clang-analyzer.llvm.org, 2025.

[40] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," in *International conference on software engineering and formal methods*. Springer, 2012, pp. 233–247.

[41] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "{AddressSanitizer}: A fast address sanity checker," in *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012, pp. 309–318.

[42] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[43] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 10*. Springer, 2004, pp. 168–176.

[44] J. Hong and S. Ryu, "Concrat: An automatic c-to-rust lock api translator for concurrent programs," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 716–728.

[45] M. Farrukh, S. Shah, B. Coskun, and M. Polychronakis, "Safetrans: Llm-assisted transpilation from c to rust," *arXiv preprint arXiv:2505.10708*, 2025.

[46] M. Emre, P. Boyland, A. Parekh, R. Schroeder, K. Dewey, and B. Hardekopf, "Aliasing limits on translating c to safe rust," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 551–579, 2023.

[47] J. Hong and S. Ryu, "To tag, or not to tag: Translating c's unions to rust's tagged unions," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 40–52.

[48] V. Nitin, R. Krishna, L. L. d. Valle, and B. Ray, "C2saferrust: Transforming c projects into safer rust with neurosymbolic techniques," *arXiv preprint arXiv:2501.14257*, 2025.