RUSTY: Effective C to Rust Conversion via Unstructured Control Specialization

Xiangjun Han, Baojian Hua*, Yang Wang*, and Ziyao Zhang
School of Software Engineering, University of Science and Technology of China, China
pnext@mail.ustc.edu.cn, bjhua@ustc.edu.cn, angyan@ustc.edu.cn, zhangziyao21@mail.ustc.edu.cn
*corresponding authors

2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C) | 979-8-3503-1991-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/QRS-C57518.2022.00122

***Abstract*—Rust is an emerging programming language designed for both performance and security, and thus many research efforts have been conducted recently to migrate legacy code bases in C/C++ to Rust to exploit Rust's safety benefits. Unfortunately, prior studies on C to Rust conversion still have three limitations: 1) complex structure; 2) code explosion; and 3) poor performance. These limitations greatly affect the effectiveness and usefulness of such conversions. This paper presents RUSTY, the *first* system for effective C to Rust code conversion via unstructured control specialization. The key technical insight of RUSTY is to implement C-style syntactic sugars on top of Rust, thus eliminating the discrepancies between the two languages. We have implemented a software prototype for RUSTY and conducted experiments to evaluate the effectiveness and testify the usefulness of it by applying RUSTY to micro-benchmarks, as well as 3 real-world C projects: 1) Vim; 2) cURL; and 3) the silver searcher. And experimental results demonstrated that RUSTY is effective in eliminating unstructured controls, reducing the code size by 16% on average with acceptable overhead (less than 61 microseconds per line of C code).**

***Keywords*—Rust security; code conversion; control specialization**

I. INTRODUCTION

Rust [1] is an emerging programming language with design goals of memory safety, type safety and thread safety, and thus has been successfully used to build a diverse range of low-level software infrastructures. Recently, migrating legacy C/C++ code bases to Rust [2] has shown promising potential, due to its two advantages: 1) safety; and 2) economy. First, such migrations can significantly improve safety of legacy software in light of Rust's safety advantages without sacrificing efficiency [3]. Second, it is often more economic to migrate legacy code to Rust than to rewrite every line of code from scratch, as most algorithm design and library implementation can be reused during the migration. As a result, there have been many academic studies as well as industry efforts on this research topic.

Technically, two approaches can be utilized for the legacy code migration from C to Rust: 1) manual rewriting; and 2) automated conversion. First, manual rewriting can be employed to re-implement the existing C code bases using Rust by Rust developers. Although manual rewriting is promising for its convenience and flexibility, its engineering efforts are considerable, especially for large C projects with huge code bases. Second, automated conversion translates legacy code to Rust by leveraging the so-called transpilers [4]. Due to its advantages of practical usefulness and capability to process large-scale programs automatically, automated conversion is studied extensively [5] [6] [7], with many practical tools proposed [8] [9] [10] [11].

Unfortunately, while prior studies on automated code migration from C to Rust have made considerable progress, they still have three limitations. First, the Rust code generated by the existing automated conversion techniques has more complex structures than the original C code, making the subsequent code maintenance difficult. Second, existing studies and tools suffer from code explosion problem, by generating Rust code of undesired sizes. Third, the generated Rust code has poor performance, compared with the original C code.

To address these limitations, this paper presents the *first* system for effective C to Rust conversion via unstructured control specializations. First, we explored root causes for leading to such limitations and identified the main cause is the discrepancy of control structures between C and Rust. We argue that this root cause is not unique to C to Rust conversion but general in many similar conversions (e.g., from C to WebAssembly). Second, we present RUSTY, the first framework for one-pass C to Rust code migration via unstructured control specialization, in which the key technical insight is to implement C-style syntactic sugars on top of Rust. To eliminate the effect of unstructured control structures, RUSTY incorporates a tree rewriter to introduce specialized unstructured code patterns.

We have built a software prototype of RUSTY and conducted initial experiments with it on 10 micro-benchmarks with arbitrary `goto` statements as well as 3 real-world C projects from diverse application domains. Experimental results showed that RUSTY successfully reduced the code size by 16% on average with acceptable overhead (less than 61 microseconds per line of C code).

II. APPROACH

This section discusses our approach by presenting the design and implementation of RUSTY.

Design. As presented by Figure 1, the architecture of RUSTY consists of six key modules, which will be discussed next in detail, respectively.

Parsing. The C parser takes as input the C source programs and parses them into abstract syntax trees (ASTs). The rationale for choosing AST as one key intermediate representation is that AST has explicit marking of source-level `goto` statements.

CFG Generator. The CFG generator takes as input the C AST and builds special control-flow graphs (CFGs). In this phase, the generator replaces the `goto` and the label statements in C with dedicated non-transfer statements. As a

2693-9371/22/$31.00 ©2022 IEEE
DOI 10.1109/QRS-C57518.2022.00122

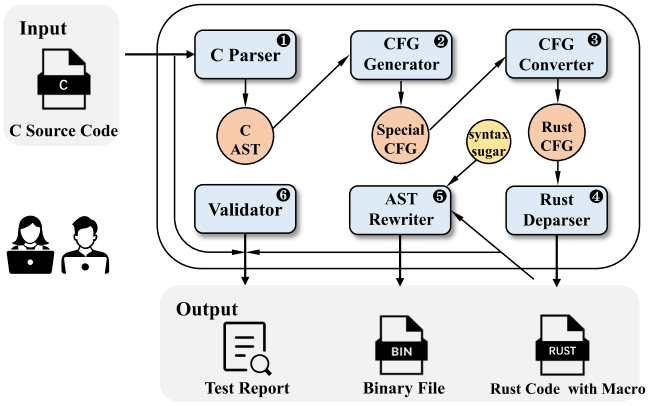

Figure 1. Architecture of RUSTY

result, the generated control flow graphs do not contain explicit goto and label statements.

CFG Converter and Rust Deparser. The CFG converter takes as input the special CFG and converts it into the corresponding Rust CFG, by compiling C statements into equivalent Rust statements. One key design point is that the unstructured controls in C are represented as special syntactic sugars in Rust. Finally, the Rust deparser takes as input the Rust CFG and decompiles it to normal Rust AST, in which syntactic sugars are preserved.

AST Rewriter and Validator. The AST rewriter takes as input the Rust AST, expands the syntactic sugars by overriding the AST, and generates the Rust AST conforming to the standard Rust syntax. To evaluate the effectiveness, complexity, correctness and cost of RUSTY, the validator takes as input the original C programs as well as the generated Rust programs, and compares them in terms of functional correctness, code sizes, and performance.

Implementation. We have implemented a software prototype, following the architecture of RUSTY. This prototype leverages Clang to parse C sources, and C2Rust to generate, convert, and deparse CFG. We modified an off-the-shelf Rust library `forward_goto` to rewrite the C CFG. Finally, we implemented the validator using bash and Python scripts.

III. EVALUATION

The RUSTY is still under heavy development, and we have conducted some initial experiments with it. First, to evaluate the effectiveness of RUSTY, we applied it to 10 micro-benchmarks, and experimental results demonstrated that RUSTY is effective in reducing the code sizes of the generated Rust target code by 16% on average. Second, to evaluate the complexity of RUSTY, we applied it to 3 real-world C projects: 1) Vim; 2) cURL; and 3) the silver searcher, and experimental results demonstrated that RUSTY is effective to reduce the cognitive complexity of Rust target code by 65.3% on average. Third, to evaluate the correctness and the cost of RUSTY, we applied it to all the benchmarks, and experimental results demonstrated that RUSTY does not affect the functional correctness of Rust target code and the overhead RUSTY introduced is less than 61 microseconds per line of C. Finally, to evaluate the usefulness of RUSTY, we conducted a developer study, and the survey results demonstrated that RUSTY is helpful to end-users in converting C to Rust in a fully automated manner.

IV. RELATED WORK

Recently, studies of transpilers converting C to Rust have become a hot research topic.

Transpilers. There have been several transpilers converting C to Rust. Bindgen [8] generates Rust FFI bindings for C libraries automatically. Corrode [11] is semantics-preserving transpiler intending for partial automation. Similar to Bindgen, Cirtus [10] generates function bodies but without preserving C semantics. As the successor of Corrode, C2Rust [9] supports large-scale automatic conversion while preserving semantics.

C to Rust automated conversion. Existing studies on C to Rust automated conversion focus on safety. Emre et al. [5] 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. [6] proposed an approach to lift raw pointers to arrays with type constraints. Ling et al. [7] 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.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work is partially supported by the National Natural Science Foundation of China (No.62072427, No.12227901), the Project of Stable Support for Youth Team in Basic Research Field, CAS (No.YSBR-005), Academic Leaders Cultivation Program, USTC.

REFERENCES

[1] Rust, "Rust programming language," https://www.rust-lang.org/.
[2] Google, "Mitigating memory safety issues in open source software," https://security.googleblog.com/2021/02/mitigating-memory-safety-issues-in-open.html.
[3] P. Chifflier, and G. Couprie, "Writing Parsers Like it is 2017," *2017 IEEE Security and Privacy Workshops (SPW)*, 2017, pp. 80-92, May 2017.
[4] R. Kulkarni, A. Chavan, and A. Hardikar, "Transpiler and it's advantages," *International Journal of Computer Science and Information Technologies*, vol. 6, pp. 1629-1631, April 2015.
[5] 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.
[6] T. Y. Hong, and Bryan, "From c towards idiomatic & safer rust through constraints-guided refactoring," Doctoral dissertation, National University of Singapore, 2022.
[7] 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," *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 354-355, 2022.
[8] Rust, "Bindgen," https://github.com/rust-lang/rust-bindgen.
[9] Galois, and Immunant, "C2rust," https://c2rust.com/.
[10] "Citrus," https://gitlab.com/citrus-rs/citrus.
[11] "Corrode," https://github.com/jameysharp/corrode.