RUSTCHECK: Safety Enhancement of Unsafe Rust via Dynamic Program Analysis

Lei Xia, Yufei Wu, and Baojian Hua*
School of Software Engineering, University of Science and Technology of China, Hefei 230026, China
Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou 215123, China
xialeics@mail.ustc.edu.cn, wuyf21@mail.ustc.edu.cn, bjhua@ustc.edu.cn*
* Corresponding author.

Abstract—Rust is a modern system-level programming language providing strong security guarantees, which has been widely applied in building software infrastructures. However, *unsafe* Rust, a language feature introduced for programming flexibility and efficiency, can be prone to memory vulnerabilities due to the lack of compile-time and run-time checks. Worse yet, it is challenging to diagnose such memory vulnerabilities in Rust programs, due to the subtle interactions between the safe and unsafe code. This paper presents RUSTCHECK, the first memory safety enhancement framework for dynamic program analysis of Rust programs. The key idea of RUSTCHECK is to dynamically detect memory safety issues caused by the improper use of *unsafe* Rust through static instrumentations. We have implemented a software prototype for RUSTCHECK and conducted experiments to evaluate the effectiveness and performance of it by applying RUSTCHECK to 56 CVES from real-world Rust projects. And experimental results showed that RUSTCHECK can successfully detect all of 65 memory vulnerabilities in CVEs, with low runtime overhead (3.30% on average) to the Rust projects being checked.

Keywords–Unsafe Rust, Security, Dynamic analysis

1. INTRODUCTION

Rust [1] is an emerging system-level programming designed to address the memory safety issues of traditional languages, such as C/C++. It provides strong security guarantee via its novel language features and strict safety checks. In view of its technical advantages, Rust is gaining widespread adoptions in building software infrastructures.

However, certain scenarios arise in which low-level operations or interactions with external code demand greater flexibility than the safe Rust programming model can offer. Consequently, *unsafe* feature [2] is introduced as a deliberate part of Rust for programming flexibility and efficiency, while it comes with the potential drawback of increased susceptibility to memory vulnerabilities due to the absence of compile-time and run-time checks. Furthermore, diagnosing these memory vulnerabilities in Rust programs can be especially difficult, owing to the intricate interplay between safe and unsafe code segments. The pervasive use of *unsafe* feature in Rust has posed severe threats to Rust safety [3] [4] [5].

Recognizing this problem, a considerable amount of security studies has been conducted on Rust (*e.g.*, vulnerability detection [6] [7] [8] [9]). Unfortunately, while prior studies have made considerable progress on the safety enhancement of the *unsafe Rust*, they still have two limitations. On one hand, the utilization of static program analysis by these established tools can result in false positives and false negatives, potentially causing confusion to developers. On the other hand, the error messages provided by these tools lack the necessary depth to assist developers in accurately diagnosing the root causes of potential vulnerabilities.

To address these limitations, this paper presents RUSTCHECK, the *first* fully automatic memory safety enhancement framework for dynamic program analysis of Rust programs through static instrumentations. This framework possesses the capability to identify and report intricate causes of bugs, utilizing runtime information that is challenging to acquire through static analysis. RUSTCHECK takes the following three key steps to detect and diagnose bugs in Rust programs: 1) performing static program analysis to annotate all memory-related operations and identity proper positions for the following instrumentation; 2) inserting necessary runtime checks against the potential memory vulnerabilities; and 3) re-executing the instrumented Rust programs to identify the vulnerabilities and analysis root causes of the bugs or crashes.

We have built a software prototype of RUSTCHECK and conducted initial experiments with it on 56 memory-related CVEs to evaluate the validity of it. Experimental results showed that RUSTCHECK successfully detected all of 65 memory vulnerabilities in the 56 known CVEs, and offered comprehensive analysis of the root causes for each test case, which is valuable to help Rust developers to fix the problems. Besides that, we measured the execution time of the instrumented programs, and the experimental results showed that RUSTCHECK incurs low runtime overhead, with 3.30% on an average.

2. APPROACH

In this section, we elucidate our approach by expounding on the design and implementation of RUSTCHECK.

2.1. Design

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

Front-end. The frontend module takes as input the target Rust source files being detected and diagnosed, and generates the MIR (Middle-level Intermediate Representation) representations for the programs.

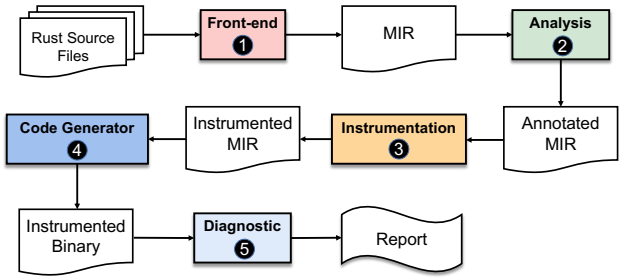

Figure 1: RUSTCHECK Architecture

Analysis. This module takes as input the generated MIR from the frontend module, and builds a control-flow graph for each function in the program. Then, it will identify all memory-related operations, such as allocation, read, write, or reclaim, and identify the sites that require security instrumentations.

Instrumentation. This module generates instrumented MIR from annotated MIR, by inserting necessary safety checks, according to the corresponding type of potential memory bugs.

Code generator. The code generator generates instrumented binaries from the instrumented MIR. In this pass, the instrumented binaries are also linked against a customized security library RUSTCHECK provides, which designed to collect runtime informations and help RUSTCHECK to perform root causes analysis.

Diagnostic The diagnostic module takes as input the annotated binaries and re-executes it, to confirm the vulnerabilities and generate detailed reports recording program outputs, logs, running time, etc, for subsequent manual analysis.

2.2. Implementation

We have implemented a software prototype which works as a customized Rust compiler driver, following the architecture of RUSTCHECK. This prototype utilizes the official `rustc` compiler for parsing Rust source code and producing instrumented binaries. Additionally, it employs specific passes that operate on MIR supplied by the Rust compiler to enact the analysis algorithm and implement instrumentation. To complement this, we offer a customized security library, intended to be linked with the instrumented binaries for conducting root cause analysis.

3. EVALUATION

RUSTCHECK is undergoing significant developmental advancements, and we have undertaken preliminary experiments utilizing the current implementation. First, we created a dataset consisting of 56 memory-related CVEs collected from the official CVE database and curating ground truth for it. Then, to evaluate the effectiveness and performance of RUSTCHECK, we applied RUSTCHECK to this dataset. The experimental results demonstrated that RUSTCHECK effectively detected all 65 memory vulnerabilities in all 56 known CVEs and provided comprehensive root cause analyses for each case, with an acceptable runtime overhead of 3.37% on average.

4. RELATED WORK

A significant amount of work on Rust security has been made in general. Qin et al. [3] conducted an empirical study of memory and concurrency security vulnerabilities in Rust applications. Xu et al. [4] investigated 186 Rust memory security-related CVEs and proposed a taxonomy. Astrauskas et al. [5] studied the use of *unsafe* in 31867 Rust crates and summarized the usage scenarios of *unsafe*. SafeDrop [6], Rudra [7], Mirchecker [8], and Rupair [9] all perform vulnerability detection based on static program analysis. However, our work first employs dynamic program analysis for vulnerability detection and root cause diagnosis in Rust programs.

5. SUMMARY

In this work, we present the first software prototype RUSTCHECK, to enhance the safety of unsafe Rust, by performing static program instrumentation on MIR. We performed priliminary experiments with RUSTCHECK, demonstrating its effectiveness in diagnosing all existing memory-related Rust CVEs, with low runtime cost.

REFERENCES

[1] "Rust programming language," https://www.rust-lang.org/.

[2] "Unsafe," https://doc.rust-lang.org/std/keyword.unsafe.html.

[3] 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*, vol. 31, no. 1, pp. 3:1–3:25, Sep. 2021.

[4] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2020, pp. 763–779.

[5] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging rust types for modular specification and verification," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, Oct. 2019.

[6] M. Cui, C. Chen, H. Xu, and Y. Zhou, "Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis," Apr. 2021.

[7] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: Finding memory safety bugs in rust at the ecosystem scale," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, Oct. 2021, pp. 84–99.

[8] Z. Li, J. Wang, M. Sun, and J. C. Lui, "Mirchecker: Detecting bugs in rust programs via static analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, Nov. 2021, pp. 2183–2196.

[9] B. Hua, W. Ouyang, C. Jiang, Q. Fan, and Z. Pan, "Rupair: Towards automatic buffer overflow detection and rectification for rust," in *Annual Computer Security Applications Conference*, Dec. 2021, pp. 812–823.