RusBox: Towards Efficient and Adaptive Sandboxing for Rust

Wanrong Ouyang Baojian Hua*
School of Software Engineering
University of Science and Technology of China
oywr@mail.ustc.edu.cn bjhua@ustc.edu.cn*

***Abstract*—Rust is a new language for safe system programming, and its strong type system and dynamic bound checking guarantee memory safety. Surprisingly, Rust is still vulnerable to buffer overflows, due to its `unsafe` feature. Recently, there have been a significant amount of studies to protect Rust programs against overflows, however, existing studies have severe limitations: they are either too coarse-grain or of considerable runtime overhead. This paper proposes RusBox, a novel sandboxing software prototype to protect Rust programs against buffer overflow vulnerabilities. The key technical contribution of RusBox is its adaptive combination of static program analysis with sandboxing, to make the protection both effective and efficient. To testify the effectiveness of RusBox, we apply it to three publicly reported CVEs from real-world Rust projects; to evaluate the cost of RusBox, we plan to apply it to 36 widely used open source Rust projects.**

***Index Terms*—Rust, Memory safety, Program analysis, Sandbox technology**

I. Introduction

As an emerging language, Rust [1] provides a strong type system to guarantee memory safety, and has been successful in building system software such as OS kernels [2], browser kernels [3], file systems [4], databases [5], cloud services [6], and blockchains [7]. Surprisingly, Rust is still vulnerable to buffer overflows, due to its `unsafe` feature. As uses a single process memory model, buffer overflows occur in `unsafe` Rust code may affect objects allocated by safe Rust code, which may eventually corrupt data in the entire address space.

Fig. 1 presents a sample code snippet `str::repeat` from the Rust standard library, which has been reported to vulnerable to buffer overflows (CVE-2018-1000810 [8]). Line 2 allocates a vector `buf` of capacity `self.len()*n`. Unfortunately, a smaller buffer `buf` is allocated, if an integer overflow occurs in this multiplication. Thus, line 4 will trigger the overflow in the `unsafe` code.

There have been a lot of research efforts to address this research challenge, which can be classified into two categories: 1) *system-level protection*, to leverage a wide variety of system-level techniques: sandboxing, process isolation, memory permission protection, etc, to explicitly isolate the vulnerable code [11], [12]; 2) *language-based protection*, to construct dedicated memory allocators to achieve memory isolation [13].

* Corresponding author.

```
pub fn repeat(&self, n: usize) -> Vec<T: Copy>{
  let mut buf = Vec::with_capacity(self.len()*n);
  unsafe{
    ptr::copy_nonoverlapping(buf.as_ptr(),
    (buf.as_mut_ptr() as *mut T).add(buf.len()),
     buf.len());}}
```

Fig. 1. A buffer overflow sample code from the Rust standard library

Unfortunately, prior research efforts have severe limitations: first, existing system-level protection are too coarse-grained, in which all foreign function interfaces are sandboxed. As a result, the protection is too costly to protect real-world large Rust projects [12]. Second, the language-based protections are too complex and costly to implement [13]; in addition, to maintain these tools is also laborious and error-prone.

To address these research challenges, this paper proposes a novel software prototype RusBox, which aims to protect Rust programs against buffer overflows efficiently and adaptively.

To guarantee data integrity in Rust, RusBox uses sandboxing [14], [15], to isolate the code that may be vulnerable to buffer overflows. However, a key distinction with prior work is that RusBox leverages a novel static program analysis algorithm to adaptively locate Rust code that may cause overflows, thus should be sandboxed. Designing this algorithm is challenging, as we must account for both the safe and `unsafe` sub-languages of Rust to make the algorithm work across the language boundary. Furthermore, as RusBox's analysis algorithm is sound, the overhead introduced by this protection is low.

To testify the effectiveness of RusBox, we plan to apply it to 3 publicly reported memory security CVEs [8]–[10] from real-world Rust programs. To evaluate the cost of RusBox, we have selected 36 widely used open- source Rust projects from 8 different application domains, consisting of 5,108,432 lines of Rust source code. We plan to apply RusBox to these projects, to measure the runtime overhead of RusBox.

II. Approach

This section presents our approach to design and implement RusBox.

Fig. 2 presents the architecture of RusBox, which consists of several key modules. First, the analyzer module takes as

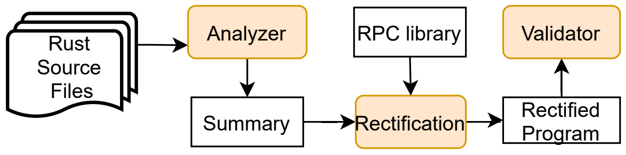

Fig. 2. RUSBOX architecture

input the Rust source files, identifies code that may be vulnerable to buffer overflows. This module leverages static program analysis to identify vulnerable code adaptively and accurately. Due to the propagation characteristics of `unsafe` [16], for the program being analyzed, RUSBOX builds not only the control flow graph, but also the call graph. The analyzer module generates as output a summary information for the program, which essentially records all possible vulnerable code. Like most static analysis, this analysis algorithm is conservative, in that it identifies all potential overflow vulnerabilities.

Second, the rectification module takes as input the summary from the analyzer, and rectifies the vulnerable program by leveraging sandboxing. In our current implementation, RUSBOX leverages the `tarpc` framework [17] to sandbox vulnerable code. There are two reasons to choose `tarpc`: 1) `tarpc` is Rust native, thus it should be easy to integrate it into any Rust projects; 2) `tarpc` is efficient, comparable to Google's gRPC. For code and global variable in the sandbox process, RUSBOX makes of same addresses as the main process, which makes the RPC call easier to create.

Finally, the validator module is used to evaluate the effectiveness, cost and correctness of RUSBOX. First, to testify effectiveness RUSBOX, the validator take as input known vulnerabilities (CVEs) and validate whether these CVEs have been detected. Second, to measure the cost of RUSBOX, the validator runs the program before and after the sandboxing, to compare the overhead introduced. Third, to testify the correctness of RUSBOX, the validator runs the program before and after the rectification, and compares the outputs to guarantee the functional effects (outputs) are identical.

III. EVALUATIONS

The RUSBOX is still under heavy development, and we are conducting experiments with it. First, to evaluate the effectiveness of RUSBOX, we have collected all publicly reported Rust buffer overflow CVEs [8]–[10], and are applying RUSBOX to these CVEs to check whether RUSBOX can defend against these vulnerabilities effectively. To test the cost and correctness of RUSBOX, we have constructed a test suit with 36 open source projects from 8 different application domains, consisting of 5,108,432 lines of Rust source code. And we are applying RUSBOX to this test suit to measure the overhead introduced, and to testify the functional correctness of RUSBOX.

IV. RELATED WORK

Recently, there have been a significant amount of studies to guarantee data integrity of Rust programs. We classified existing studies on Rust memory safety into two categories: system-level protection and language-based protection.

System-level protection. The FC system [11] divides the entire memory into three parts and adds a set of FC call interfaces to allow programmers to control the access permission of a specific region of memory. The Sandcrust framework [12] makes use of sandbox to encapsulate the invocation of C code FFI by converting it into remote procedure calls. However, FC is coarse-grained, and is incomplete, in which rectified programs are still vulnerable to buffer overflows. Sandcrust is not only incomplete but also incurs considerable runtime overhead. On the other hand, RUSBOX is both complete and efficient.

Language-base protection. Liu et al. [13] proposed the XRust framework, in which new memory allocators are added into the Rust runtime, and the Rust compiler is also modified extensively. However, such techniques is laborious and error-prone to maintain and evolve, as the Rust language is still evolving rapidly. On the contrary, RUSBOX is independent of the specific Rust compiler or runtime; thus it's much easier to implement and maintain.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable feedback. This work is supported by a graduate education innovation program of USTC under grant No. 2020YCJC41.

REFERENCES

[1] Steve Klabnik and Carol Nichols. 2018. The Rust Programming Language. https://doc.rust-lang.org/stable/book/2018-edition/
[2] Tock. 2019. Tock Embedded Operating System. https://www.tockos. org/
[3] Servo. The Servo Browser Engine. https://servo.org/
[4] TFS. https://github.com/redox-os/tfs
[5] TiKV. https://github.com/tikv/tikv
[6] TTstack. https://github.com/rustcc/TTstack
[7] Parity. https://github.com/paritytech/parity-ethereum
[8] Pedro Sampaio. 2018. CVE-2018-1000810. https://bugzilla.redhat.com/show_bug.cgi?id=1632932
[9] Rust vase64 project. 2017. CVE-2017-1000430. https://www.cvedetails.com/cve/CVE-2017-1000430/
[10] Pedro Sampaio. 2018. CVE-2018-1000657. https://bugzilla.redhat.com/show_bug.cgi?id=1622249
[11] Almohri H M J, Evans D. Fidelius charm: Isolating unsafe rust code. In Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy. 2018: 248-255.
[12] Lamowski B, Weinhold C, Lackorzynski A, et al. Sandcrust: Automatic sandboxing of unsafe components in rust. In Proceedings of the 9th Workshop on Programming Languages and Operating Systems. 2017: 51-57.
[13] Liu P, Zhao G, Huang J. Securing unsafe rust programs with XRust. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 2020: 234-245.
[14] Brumley D, Song D. Privtrans: Automatically partitioning programs for privilege separation. USENIX Security Symposium. 2004, 57(72).
[15] Bo M, Dejun M, Wei F, et al. Improvements the Seccomp sandbox based on PBE theory. 2013 27th International Conference on Advanced Information Networking and Applications Workshops. IEEE, 2013: 323-328.
[16] Cui M, Chen C, Xu H, et al. SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-Flow Analysis. arXiv preprint arXiv:2103.15420, 2021.
[17] trpc. http://www.github.com/google/tarpc