Demo: VRLifeTime -- An IDE Tool to Avoid Concurrency and Memory Bugs in Rust

Ziyi Zhang
USTC, Pennsylvania State University

Boqin Qin
BUPT, Pennsylvania State University

Yilun Chen
HoneycombData Inc

Linhai Song*
Pennsylvania State University

Yiying Zhang
University of California, San Diego

ABSTRACT

As a young programming language designed for systems software development, Rust aims to provide safety guarantees like high-level languages *and* performance efficiency like low-level languages. Lifetime is a core concept in Rust, and it is key to both safety checks and automated resource management conducted by the Rust compiler. However, Rust's lifetime rules are very complex. In reality, it is not uncommon that Rust programmers fail to infer the correct lifetime, causing severe concurrency and memory bugs. In this paper, we present VRLifeTime, an IDE tool that can visualize lifetime for Rust programs and help programmers avoid lifetime-related mistakes. Moreover, VRLifeTime can help detect some lifetime-related bugs (*i.e.*, double locks) with detailed debugging information. A demo video is available at https://youtu.be/L5F_XCOrJTQ.

ACM Reference Format:
Ziyi Zhang, Boqin Qin, Yilun Chen, Linhai Song, and Yiying Zhang. 2020. Demo: VRLifeTime -- An IDE Tool to Avoid Concurrency and Memory Bugs in Rust. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3372297.3420024

1 INTRODUCTION

Rust is a young programming language designed to develop low-level software. Its main design goal is to provide the same runtime performance as C/C++, while ruling out safety issues in C/C++ through strict compile-time checks. In recent years, Rust has gained increasing popularity. According to surveys on Stack Overflow, Rust was the most beloved language in the last four years. It has already been used to implement many safety-critical software systems, ranging from OSes [10, 13] to browsers [12] and blockchain applications [4, 7].

The core of Rust's safety checks is two concepts: *ownership* and *lifetime*. The basic rule only permits each value to have one owner variable and a value is *dropped* (or freed) when its owner's lifetime ends. Rust extends the basic rule with a set of rules to allow the ownership to be *move*d and *borrow*ed, while still guaranteeing memory safety and thread safety. Rust's safety rules essentially ensure that all accesses to a value are within the value's lifetime and prohibit the combination of aliasing and mutability. All these rules are checked and enforced at compilation, so that Rust can achieve the performance as good as C/C++ during runtime.

```
1  fn do_request(client: &LeaderClient) {
2    //client.inner: Arc<RwLock<Inner>>
3  -  match connect(client.inner.read().unwrap().m) {
4  +  let result = connect(client.inner.read().unwrap().m);
5  +  match result {
6       Ok(_) => {...}
7       Err(_) => {
8         if let Err(e) = client.reconnect() {...}
9       }
10    }
11 }
12
13 impl LeaderClient {
14   fn reconnect(&self) {
15     let mut inner = self.inner.write().unwrap();
16     ...
17   }
18 }
```

Figure 1: A double-lock bug in TiKV.

Besides safety, Rust's lifetime mechanism is also widely used to achieve automated resource management, since all resources allocated to a variable are automatically freed when the variable ends its lifetime. For example, there is no explicit `Unlock()` in Rust. A `Mutex.Lock()` function call returns a reference of the shared variable protected by the mutex. All accesses to the shared variable are conducted using the reference, so that the Rust compiler can verify all accesses happen while the lock is held. The lock is automatically released by the Rust compiler (through implicitly calling `Unlock()`), when the reference variable ends its lifetime.

However, Rust proposes many new language features, making its lifetime rules very complex. In the real world, it is difficult for Rust programmers to correctly infer Rust variables' lifetime scopes. Thus, some allocated resources are held longer or shorter than programmers' expectations, leading to severe concurrency and memory bugs, *e.g.,* double locks, use-after-free bugs.

A double-lock deadlock in TiKV is shown in Figure 1. Variable `client`'s `inner` field is an `Inner` object protected by a `RwLock`. Line 3 acquires the read lock and uses field `m` to call function `connect()`. If `connect()` returns `Err` (line 7), function `client.reconnect()` is called at line 8, which in turn acquires the write lock at line 15. However, the lifetime of the reference returned by `client.inner.read()` doesn't end until the end of the `match` block at line 10, so that the read lock is held until line 10. Therefore, a double-lock bug happens when function `connect()` return `Err`. The fix is to save the return of `connect()` to a local

*Linhai Song was supported by NSF grant CNS-1955965.

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11.
https://doi.org/10.1145/3372297.3420024

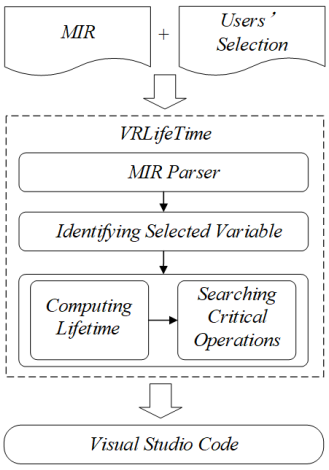

Figure 2: VRLifeTime's Architecture

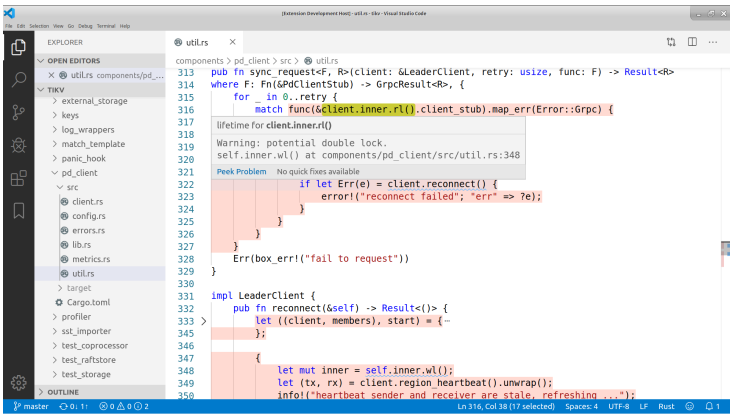

Figure 3: VRLifeTime's IDE Display

variable `result` and use it as the condition of the `match` block at line 5. After fixing, the read lock is released at line 4.

The bug in Figure 1 demonstrates the difficulty in reasoning the lifetime of a Rust variable and writing correct Rust programs. Programmers have to know where the implicit `Unlock()` is called by the Rust compiler is related to the lifetime of the reference returned by `Mutex.lock()` (or `RwLock.read()`), how the Rust compiler computes the lifetime for a variable if the variable is used in the condition of a `match` block, and whether using a variable to invoke a function can impact its lifetime. With the complex syntax in Rust, bugs caused by misunderstanding Rust's lifetime rules widely exist in the real world [9].

In this paper, we present VRLifeTime, an IDE tool that can visualize the lifetime scope for a user-selected variable. We implement VRLifeTime as a plugin for Visual Studio Code [5]. Besides the lifetime visualization, we also integrate the double-lock detector built in our previous work [9] into VRLifeTime. For each double lock identified by the detector, VRLifeTime highlights the corresponding locking operations and also provides a detailed explanation. The difference between our previous paper [9] and this paper is that our previous paper conducted a bug study and implemented two bug detectors for Rust, while this paper focuses on highlighting Rust variables' lifetime scopes to avoid lifetime-related bugs.

In summary, we make the following contributions.

- First, we conduct Rust-specific program analysis to accurately compute the lifetime scope for a given variable.
- Second, we provide a user-friendly GUI with comprehensive information for programmers to understand identified issues.

2 RELATED WORK

Rust uses LLVM as its compiler backend. Thus, many static and dynamic bug detectors designed for C/C++ [2, 14] can also be applied to Rust. However, we anticipate these tools are not effective at identifying lifetime-related bugs in Rust, since those bugs are usually caused by Rust's unique language features. The Rust team builds and releases two detectors. They either cover limited buggy code patterns [11] or depend on user-provided inputs [6], and neither of them aims to detect concurrency bugs.

There are existing techniques to visualize lifetime in Rust [1, 3]. However, their goals are to help developers understand errors reported by the Rust compiler and help developers change code to pass Rust's compiler checks, while VRLifeTime aims to warn potential concurrency and memory bugs that cannot be detected by the Rust compiler (*e.g.*, Figure 1).

3 OVERVIEW

An overview of VRLifeTime's architecture is shown in Figure 2. When a programmer saves the Rust source code file she is editing, VRLifeTime runs the Rust compiler to generate the MIR of the Rust file. When the programmer selects a piece of source code in the IDE, VRLifeTime first identifies which variable's lifetime the programmer wants to analyze, and then computes the lifetime scope for the variable through analyzing the generated MIR. VRLifeTime also invokes the double-lock detector to analyze the computed lifetime scope for potential bugs. In the end, the computed lifetime scope and the identified bugs are visualized in the IDE.

Figure 3 shows how VRLifeTime visualizes the bug in Figure 1. After the read locking operation (line 316) is selected, VRLifeTime colors the selection in yellow and colors the computed lifetime scope of the temporary reference returned by the read locking in pink. Function `client.reconnect()` is called inside the lifetime scope, so that all its instructions are considered as inside the lifetime scope and are also colored in pink. Since `client.reconnect()` acquires the write lock and causes the double lock, VRLifeTime puts a tilde under the invocation of `client.reconnect()` to warn the programmer. If the programmer moves his mouse onto the invocation, a window is popped up to explain the warning (*i.e.*, a potential double lock) and provide more information (*i.e.*, the write lock acquisition at line 348).

4 IMPLEMENTATION

We implement VRLifeTime as a plugin for Visual Studio Code (VSCode). VRLifeTime takes the generated MIR and a user-selected piece of code as input, and computes the visualization information in four steps: parsing MIR, pinpointing the selected variable,

computing the lifetime scope, and searching critical operations. We only present the details for the first three steps in this paper, since the last step is discussed in our previous paper [9].

Parsing MIR. We parse the generated MIR and build the control flow graph (CFG) for each function in it. Given a program element (*e.g.*, variable, instruction), MIR provides its rendering location, including the row number and the range of column numbers. We record the rendering information to interact with VSCode. Given two variables `a` and `b`, if `a` is moved to `b` (indicated by keyword `move`), we consider `a` and `b` are equivalent. We compute equivalent variable sets for each function. We also compute the points-to set for each pointer variable and reference variable.

Pinpointing the selected variable. We expect a user to select a few characters in the edited Rust file. A particular event in VSCode can provide the row number and the range of the columns for the selected characters. VRLifeTime identifies the selected variable as the one with the smallest column range among all variables whose rendering locations can cover the selected characters. For a temporary variable, VRLifeTime considers its rendering location the same as the operation creating the variable.

Computing the lifetime scope. MIR leverages two special instructions `StorageLive` and `StorageDead` to mark the lifetime start and the lifetime end for each variable respectively. We define the lifetime scope of a variable as a set of instructions. Each instruction is along a path on CFG starting from the variable's `StorageLive` and ending at the variable's `StorageDead`. If a variable has equivalent variables (*i.e.*, variables with the moving relationship), we compute the union of lifetime scopes for all its equivalent variables and use the union as the lifetime scope.

To handle a function call, we consider the following two cases. First, if a function is called in the lifetime scope of a variable and the variable is not moved into the function, we consider all the instructions executed by the function directly or indirectly as in the lifetime scope of the variable. Second, if the variable is moved into the function, then the variable's lifetime ends right after the call site in the caller function, and we continue to analyze the callee function to extend the variable's lifetime scope.

For a pointer variable, we visualize the lifetime of the object it points to. The reason is that a pointer variable is in a primitive type and its own lifetime scope has nothing to do with memory bugs or concurrency bugs, while inspecting whether a pointer is used inside the lifetime of the object it points to can avoid potential use-after-free bugs. Given a reference variable, we visualize its own lifetime, since the Rust compiler guarantees that all references live within the lifetime of its owner variable and using a reference cannot cause any use-after-free bug.

5 EXPERIMENTAL EVALUATION

Methodology. Our experiment is designed to understand the proportion of real-world double locks VRLifeTime can detect (*coverage*) and how accurate VRLifeTime's detection results are (*accuracy*). In our previous work, we collected 70 real-world concurrency bugs from popular Rust applications and libraries [9]. Among them, 30 are double locks. We apply VRLifeTime to the 30 bugs and inspect how many of them can be detected to understand the coverage. For accuracy, we apply VRLifeTime to the latest versions of the Rust applications and libraries used in our previous work [9]. We examine all reported results to count true bugs and false positives.

Coverage. For 21 (out of 30) collected double locks, the Rust compiler can compile the buggy application version or we successfully inject the bug into a version that can be compiled. We apply VRLifeTime to the 21 bugs directly. VRLifeTime detects 18 bugs and fails to report the other three. For the remaining nine bugs, the compiler cannot compile their buggy versions and we cannot do the bug injection either, because their buggy functions have already been removed. We cannot apply VRLifeTime to the nine bugs directly. Thus, we conduct a manual study on them and find that VRLifeTime can detect four of them. Overall, VRLifeTime can detect a relatively large portion of real-world double-lock deadlocks.

There are two possible reasons why a bug is not detected by VRLifeTime. First, our alias analysis is not precise, and we fail to identify the same lock is acquired twice for four bugs. Second, our call-graph analysis is not good enough (*e.g.*, unable to handle function pointers). There are four bugs due to this reason.

Accuracy. After applying VRLifeTime to the latest application versions, VRLifeTime detects six previously unknown bugs without reporting any false positive. We report all the detected bugs to developers. Developers have fixed all of them [8].

6 CONCLUSION AND FUTURE WORK

Facing the increasing adoption of Rust in developing safety-critical software systems, we build VRLifeTime, an IDE tool that can visualize the lifetime scope for a user-selected variable and help programmers avoid lifetime-related bugs. VRLifeTime also has the capability to detect double-lock deadlocks, while providing detailed information to facilitate the bug validation and fixing. In the future, we plan to extend VRLifeTime to detect other types of bugs and conduct a user study to understand which debugging information is more valuable.

REFERENCES

[1] David Blaser. 2019. Simple Explanation of Complex Lifetime Errors in Rust. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/David_Blaser_BA_Report.pdf.

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *OSDI '08*.

[3] Dietler Dominik. 2018. Visualization of Lifetime Constraints in Rust. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Dominik_Dietler_BA_report.pdf.

[4] Facebook. 2019. Libra's mission is to enable a simple global currency and financial infrastructure that empowers billions of people. https://developers.libra.org/

[5] Microsoft. 2019. Visual Studio Code - Code Editing. Redefined. https://code.visualstudio.com/.

[6] Miri. 2019. *An interpreter for Rust's mid-level intermediate representation.* https://github.com/rust-lang/miri

[7] Parity-ethereum. 2019. The Parity Ethereum Client. https://www.parity.io/ethereum/

[8] Boqin Qin. 2020. ethcore/client: fix deadlock caused by double-read lock. https://github.com/openethereum/openethereum/pull/11766

[9] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. In *PLDI '20*.

[10] Redox. 2019. The Redox Operating System. https://www.redox-os.org/

[11] Rust-clippy. 2019. A bunch of lints to catch common mistakes and improve your Rust code. https://github.com/rust-lang/rust-clippy

[12] Servo. 2019. The Servo Browser Engine. https://servo.org/

[13] Tock. 2019. Tock Embedded Operating System. https://www.tockos.org/

[14] Wei Zhang, Chong Sun, and Shan Lu. 2010. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS '10*.