Motivation  C is a popular programming language, but expert opinions and empirical evidence suggest that it promotes insecure code [3,7]. One reason for this is C’s lack of memory safety, which means that bugs involving memory use often constitute exploitable vulnerabilities [5,8].

Recently, programming languages such as Google’s Go [1] and Mozilla’s Rust [2] have been developed in an explicit attempt to provide a fast, low-level, but type- and memory-safe language. Rust is notable in that it does not require garbage collection to ensure temporal memory safety. Garbage collection can produce unacceptable delays and add significant memory overhead [4], which precludes its use in resource-constrained and/or ultra-high performance environments. On the other hand, Rust’s type system strictly controls how heap-allocated memory can be used: An ownership discipline restricts many common forms of aliasing, and proving bounded lifetimes may require additional annotations. Manual memory management is already challenging, compared to using garbage collection, and Rust’s type system makes it moreso. Indeed, there is a temptation—even a need—for programmers to escape the ownership and lifetime discipline by using Rust’s unsafe blocks, thereby threatening safety [6].

We wonder: Can Rust emerge as a safe alternative to C? The answer depends on whether developers who usually prefer C can switch to Rust with a limited impact on development time, effort, and code efficiency while still achieving an increase in security.

Pilot Study  To understand this tradeoff, we designed a two-phase study in which participants perform at-home refresher tasks in both languages, then are randomly assigned to one language for a main, in-person programming task. We manually assessed the resulting code in terms of functional correctness, security, and performance. The main programming task—a three-part parallelized sorting problem—was selected to expose fault lines where complying with Rust’s type system may prove difficult, but where using C (or unsafe blocks in Rust) may result in exploitable memory management errors.

For the pilot study (n=5), we recruited university students with limited classroom experience with both C and Rust. The main portion of the study took 90 minutes, including an exit survey about the experience, and participants were paid $75. The study was approved by the University of Maryland IRB.

Observations  Although data analysis is ongoing, we present some preliminary observations. First, no participant completed all three parts of the main task, and only two participants, both using Rust, managed to complete the first part. We noticed that participants spent a large amount of time customizing the environment, worrying about the remaining time, refreshing background concepts, and rereading and understanding the specification for the tasks, despite it being reviewed before the programming portion started. We note that the above problems could relate to either the design of the study or difficulties in using Rust or C, or both.

Next steps  Results from the pilot will inform updates to the study design. We hope that by changing the second phase to remote, self-administered participation (via Docker), we can collect almost as much data about participants’ work processes while reducing stress and allowing self-pacing. We also plan to revisit the first phase to see if we can refresh additional background concepts more effectively. We are also considering how to restructure the main task to be shorter while maintaining enough complexity to expose challenges in both languages.
Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. CNS-1801545.

References


