BorrowSanitizer

IAN MCCORMACK, OLIVER BRAUNSDORF, JOHANNES KINDER, JONATHAN ALDRICH, and JOSHUA SUNSHINE

The Rust programming language provides static safety guarantees via the rules of its aliasing model. However, Rust is increasingly used in interoperation with languages that have far weaker aliasing restrictions. These multi-language applications are particularly susceptible to violations of Rust's aliasing rules, which can reintroduce the types of safety issues that Rust was designed to prevent. Despite this risk, existing dynamic analysis tools lack either the capability or performance necessary to effectively detect Rust-specific undefined behavior in this context. We propose BorrowSanitizer, an LLVM-based dynamic instrumentation tool for finding violations of Rust's latest *Tree Borrows* aliasing model in applications that interoperate with C and C++. Our design will leverage shadow metadata propagation alongside static optimizations to ensure that it will be performant enough to be useful in production and compatible with fuzzing techniques. Our 30-minute presentation will cover the motivation, design, and future goals of the project, including a live demonstration of any features that are functional at the time of the workshop.

MOTIVATION

Memory safety issues are a common source of severe security vulnerabilities. To counteract this threat, developers have started to write new software components in languages that provide inherent security guarantees [11]. Rust has been a uniquely popular choice for this transition because it provides these guarantees through static restrictions, avoiding the run-time overhead associated with garbage collection [9, 18]. Porting existing, large-scale systems to Rust is time-consuming. Instead, developers are increasingly adopting Rust in interoperation with components written in memory-unsafe languages, like C and C++ [1]. Rust can still provide security benefits in this context. However, developers will need to maintain Rust's invariants across foreign function boundaries to avoid reintroducing the types of safety issues that Rust was designed to prevent.

Foreign function calls are one of several "unsafe" operations that bypass Rust's static restrictions. Incorrect usage of these operations can violate Rust's aliasing model. Aliasing violations undermine Rust's core safety guarantees, causing optimizations to be applied incorrectly and making it possible to trigger severe security vulnerabilities through externally "safe" APIs [7, 22]. The Rust community mitigates these issues by keeping unsafe code minimal and encapsulated, making it easier to document and audit [8]. These tasks become impractical for Rust components that interoperate with C and C++ applications, which introduce far more unsafe operations. Libraries that share memory across foreign boundaries will be at a higher risk of having aliasing violations.

Miri, a Rust interpreter, is the only tool that can detect violations of Rust's evolving aliasing models [10, 19, 21]. However, Miri has two key limitations that prevent it from finding these Rust-specific bugs in multi-language applications. First, Miri cannot detect undefined behavior triggered by foreign functions. The Krabcake project [12] proposed solving this problem by creating a plugin for Valgrind [17], but it is not fully functional yet, and it has not had active development since June of 2023 [13]. McCormack et al. [14] extended Miri to support interpreting LLVM bitcode and found 46 instances of undefined or undesired behavior in 37 libraries—including one maintained by the Rust project. However, neither of these approaches fully addresses Miri's second limitation: it is slow—up to 1000 times slower than native execution [3]. Krabcake would significantly reduce this overhead, but Valgrind's baseline is still between four and five times the speed of native execution [20]. This limits the potential effectiveness of any static optimizations. An approach based on compile-time instrumentation would theoretically allow us to reach native speed if we can soundly remove run-time checks for components that are verifiably safe. This would be preferable for compatibility with fuzzing tools.

For an incremental transition to Rust to be effective, developers need a high-performance tool for finding aliasing bugs in multi-language applications. We will provide these capabilities with **BorrowSanitizer**: an LLVM-based dynamic instrumentation tool for detecting violations of Rust's Tree Borrows [21] aliasing model. It is currently an early-stages of development, and it is not yet functional. Our primary goal is to develop a production-ready tool that comprehensively supports interoperation between Rust, C, and C++. Our ongoing work is open source and publicly available.[1]

DESIGN PRINCIPLES

BorrowSanitizer (BSAN) is an LLVM-based sanitizer. It is seamlessly integrated into the Rust and LLVM toolchains, and it uses the same interfaces as existing sanitizers. Users can enable BorrowSanitizer directly by providing configuration flags, but we have also created a Cargo plugin (e.g. `cargo bsan`) that prepares an instrumented version of Rust's standard library. This plugin will intercept invocations of Clang and the Rust compiler to enable BorrowSanitizer and automatically handle cross-language link-time optimization. When BSAN is enabled, our modified Rust compiler will insert special "retag" instructions as LLVM intrinsic functions (e.g. `@llvm.retag`). Under Tree Borrows, retags are inserted to update the permission of a reference when it is created or passed into a function. Our retag intrinsics will capture all of the Rust-specific type information that we need to detect aliasing violations at the LLVM level. We will not modify Clang, aside from adding BSAN to the list of supported sanitizers.

Aside from retags, all other run-time checks will be inserted by an LLVM pass. This pass will replace retag intrinsics with calls into our runtime library, and it will add additional run-time instrumentation for tracking the *provenance* of each pointer. Like Miri, our provenance metadata will consist of an allocation identifier and a *borrow tag*, which links each pointer to its permission under Tree Borrows. We will store provenance metadata in a disjoint memory space consisting of a shadow stack and a two-level directory table [16]. This preserves the memory layout of the source program to allow for ABI compatibility with uninstrumented libraries. Inspired by the lock-and-key approach of CETS [15], we will detect aliasing violations by comparing a pointer's provenance with the values stored in a "lock" location associated with each memory allocation. The lock will contain a copy of the allocation identifier, the base and bounds of the allocation, and a pointer to a collection of the data structures used by Tree Borrows to store the state of the tree. Our run-time checks will pass through LLVM's sanitizer runtime interface into an external library implemented in Rust. This will allow us to reuse Miri's existing implementation of Tree Borrows.

Our core motivation is to provide the Rust community with a usable tool for detecting aliasing violations across language boundaries. We also intend for BorrowSanitizer to support future research on static analysis techniques and fuzzing heuristics. Like Miri, our initial prototype will conservatively check all memory accesses for undefined behavior. However, recent results from Braunsdorf et al. [4] indicate that eliding sanitizer checks from safe components can significantly increase performance. Prior work has demonstrated that the gradual typing methodology [6] can be applied to dataflow analysis [5]. It may also be effective in this context for leveraging the partial, static information from the borrow checker to soundly remove run-time checks from applications with unsafe components. We also predict that test inputs which cause programs to access the same allocations on both sides of a foreign function boundary will be more likely to trigger aliasing violations. We can observe these behaviors at run time to create heuristics that could direct a coverage-guided fuzzer toward these execution paths [2]. We are proposing this talk to gather requirements from potential users in the Rust community and to obtain feedback on our design in support of these future research efforts.

[1] borrowsanitizer.com

REFERENCES

[1] Jen Engel Alex Rebert, Chandler Carruth and Andy Qin. 2024. *Safer with Google: Advancing Memory Safety.* Google. https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[3] Keaton Brandt. 2022. Data-driven performance optimization with Rust and Miri. https://medium.com/source-and-buggy/data-driven-performance-optimization-with-rust-and-miri-70cb6dde0d35

[4] Oliver Braunsdorf, Konrad Hohentanner, and Johannes Kinder. 2024. Poster: Ensuring Memory Safety for the Transition from C/C++ to Rust. In *Network and Distributed System Security Symposium* (San Diego, CA) *(NDSS '24)*. Internet Society, Geneva, Switzerland, 3 pages. https://www.ndss-symposium.org/wp-content/uploads/ndss24-posters-37.pdf

[5] Sam Estep, Jenna Wise, Jonathan Aldrich, Éric Tanter, Johannes Bader, and Joshua Sunshine. 2021. Gradual Program Analysis for Null Pointers. arXiv:2105.06081 [cs.PL] https://arxiv.org/abs/2105.06081

[6] Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting gradual typing. *SIGPLAN Not.* 51, 1 (Jan. 2016), 429–442. https://doi.org/10.1145/2914770.2837670

[7] Ossi Heralla and Jerome Froelich. 2021. *CVE-2021-45720.* MITRE. https://www.cve.org/CVERecord?id=CVE-2021-45720

[8] Sandra Höltervennhoff, Philip Klostermeyer, Noah Wöhler, Yasemin Acar, and Sascha Fahl. 2023. "I wouldn't want my unsafe code to run my pacemaker": An Interview Study on the Use, Comprehension, and Perceived Risks of Unsafe Rust. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 2509–2525. https://www.usenix.org/conference/usenixsecurity23/presentation/holtervennhoff

[9] Jesse Howarth. 2020. *Why Discord is Switching from Go to Rust.* Discord. https://discord.com/blog/why-discord-is-switching-from-go-to-rust

[10] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371109

[11] Christoph Kern. 2024. *Secure by Design at Google.* Technical Report. Google Security Engineering.

[12] Felix S. Klock and Bryan Garza. 2023. Krabcake: A Rust UB Detector. https://pnkfx.org/presentations/krabcake-rust-verification-2023-april.pdf

[13] Felix S. Klock and Bryan Garza. 2023. Umbrella repository for Krabcake experiments. https://github.com/pnkfelix/krabcake-vg

[14] Ian McCormack, Joshua Sunshine, and Jonathan Aldrich. 2024. A Study of Undefined Behavior Across Foreign Function Boundaries in Rust Libraries. arXiv:2404.11671 [cs.SE] Will appear in the 2025 International Conference on Software Engineering (ICSE).

[15] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *Proceedings of the 2010 International Symposium on Memory Management* (Toronto, Ontario, Canada) *(ISMM '10)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/1806651.1806657

[16] Nicholas Nethercote and Julian Seward. 2007. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (San Diego, California, USA) *(VEE '07)*. Association for Computing Machinery, New York, NY, USA, 65–74. https://doi.org/10.1145/1254810.1254820

[17] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. https://doi.org/10.1145/1273442.1250746

[18] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2021. Ranking programming languages by energy efficiency. *Science of Computer Programming* 205 (2021), 102609. https://doi.org/10.1016/j.scico.2021.102609

[19] Rust Team. 2025. Miri. https://github.com/rust-lang/miri

[20] Valgrind Developers. 2024. Valgrind User Manual. https://valgrind.org/docs/manual/manual.html

[21] Neven Villani, Derek Dreyer, and Ralf Jung. 2023. Tree Borrows. https://github.com/Vanille-N/tree-borrows/blob/master/full/main.pdf

[22] Guido Vranken and Alex Crichton. 2023. *CVE-2023-30624.* MITRE. https://www.cve.org/CVERecord?id=CVE-2023-30624