Optimizing Tree Borrows with Lazy Allocation in Miri

1 Introduction

Memory safety bugs are simultaneously elusive and pervasive throughout software engineering. It is estimated that around 70% of severe vulnerabilities in unsafe codebases are due to memory safety bugs [6]. Much of systems software engineering of the past has been written in memory unsafe languages, such as C and C++, where programmers take on the burden of memory management.

One solution to memory safety is to write purely safe Rust. The Rust Programming Language ensures memory safety by default through its strict ownership rule system at compile time. However, these rules are too restrictive in many circumstances; programmers cannot implement cyclic data structures, lock-free algorithms, hardware abstraction layers, or foreign function interfaces (FFI) without violating memory ownership [1]. In response, Rust allows functions and blocks of code to be wrapped in the `unsafe` keyword to evade ownership checks. For these unsafe abstractions, programmers must find another way of preventing memory safety bugs.

Miri is a dynamic undefined behavior (UB) detection tool for unsafe Rust [3]. It sanitizes a single execution at a time searching for a myriad of memory bugs including data races, leaks, misalignment, and aliasing. For aliasing bugs particularly, Miri uses a reference-tracking model to detect aliasing violations, the most recent of which is Tree Borrows [7]. Aliasing violations in Tree Borrows, such as out-of-bounds reads, not only leave programs vulnerable to attack but can be fatal to system functionality—as was the case in 2024 for the machines affected by a bug in CrowdStrike [8].

A major impedance to Miri's usability is its runtime. To put it bluntly, Miri is slow. Users of Miri have reported overheads of up to 7,000× native execution [2]. Targeting the testbench of the popular Rust crate `aho-corasick`, we saw runtimes of over 35,000×. Such extreme overheads can severely limit exploration of different execution paths with Miri, making combining it with other bug-finding methodologies, such as fuzzing, virtually impossible.

Another Rust sanitization tool, BorrowSanitizer, is currently under development with the goal of detecting aliasing violations faster than Miri and on more applications of unsafe code [5]. As an interpreter, Miri cannot support FFI and thus is limited to analysis within the bounds of the Rust language. BorrowSanitizer, on the other hand, operates at the LLVM intermediate representation level. It can continue to analyze program execution through calls to C and C++ functions, ensuring the safety of programs that interop with legacy unsafe code, for example.

```rust
let mut root = 42;
let ref1 = &mut root;
let ref2 = &mut *ref1;
let ref3 = &mut root;
```

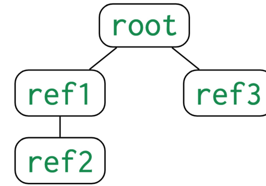

Figure 1: A code snippet and its corresponding tree representation, as seen in Villani et al. [7]

While BorrowSanitizer is not fully functional yet, it uses the same Tree Borrows model to track aliasing. Therefore, optimizations made to Miri's implementation of Tree Borrows can be translatable to BorrowSanitizer in future work.

Through simple **lazy allocation** of trees in Tree Borrows, we sped up Miri's runtime by an average of **3.8%** and up to **6.7%** among crates with over 100× overhead from `crates.io`. In the process, we address the following research questions:

RQ1: What is a major overhead of Tree Borrows in Miri?

RQ2: How can we optimize Miri's Tree Borrows implementation?

2 Background

To understand our profiling and optimization methods, some background is needed on the domain-specific constraints of Tree Borrows. Designed by Villani et al. [7], Tree Borrows is a model for tracking the relationship of references in Rust to detect UB in the form of aliasing violations. Tree Borrows tracks an allocation as a root and references as child nodes in a tree data structure, as demonstrated in Figure 1. Each node has a corresponding tag, thus creating a new node in the tree is known as a *retag*.

Each tag also corresponds to a permission tracking state machine. Depending on the current state and origin of access, reads and writes can cause a state transition. If the access is from the node itself or one of its children, it is identified as *local*. If the access is from a parent or cousin node, it is *foreign*. UB has occurred if it reaches the final state depicted in Figure 2. For example, a local write to some immutable reference (initialized at the Frozen state) would result in UB.

By default, Miri uses Stacked Borrows as an aliasing model. Tree Borrows is a newer experimental implementation that can be enabled with `-Zmiri-tree-borrows` that results in 54% fewer false positives [7]. Miri implements Tree Borrows very conservatively. It creates trees for every allocation, regardless if the allocation is involved in an unsafe operation or if any references are created. This leaves room for optimization in limiting tree creation. In the scope of this paper, we aim to preserve soundness in all cases. However, in future work, we intend to trade soundness for efficiency in cases where there is a lot of repetition and redundancy (e.g. fuzzing).

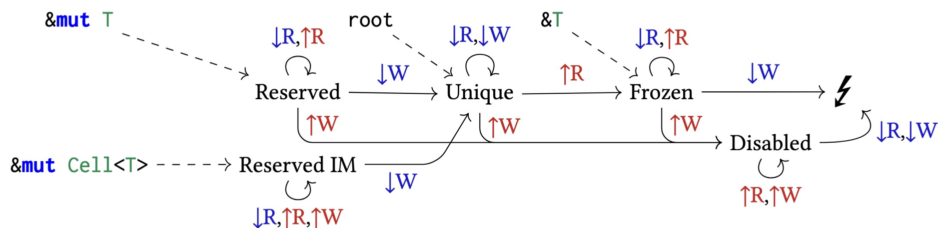

Figure 2: Default state machine of permissions, where reaching the lighting bolt means UB has occurred. Accesses can be ↑(foreign) or ↓(local). [7]

3 Profiling

The main methods of identifying overheads were found through collecting end-to-end runtimes of popular crates and extending Miri's internal tracing library to log Tree Borrows-specific metrics.

3.1 End-to-end runtimes

We first identified the top 200 crates by number of downloads on `crates.io`, the official Rust package registry. Using the package `cargo-geiger` on this corpus, we identified 137 crates using unsafe code to some capacity.

Recording end-to-end runtimes is fairly simple. To minimize impact from Miri's other runtime checks, we disabled alignment checks, data race detection, validation, and leak detection. Then, we ran three executions across the entire test suite of each crate: once without Miri, once with Miri and borrow tracking disabled, and once with Miri and Tree Borrows enabled. From there, we identified 12 target high-overhead crates as those with a Tree Borrows overhead of over 100×, disregarding `aho-corasick` as a target for now, as it would impede iterative testing. Focusing optimization on these exceptionally bad cases can improve runtime when it matters most. The results can be found in Table 1.

3.2 Tree-specific metrics

We also forked Miri to add additional tracing to Tree Borrows operations. We track tree events (allocations, retags, reads, writes, visits, prunes) and state machine transitions by inserting additional logging messages as they occur. Modifying Miri requires rebuilding the entire Rust standard library alongside it, ensuring compatibility between Miri and the toolchain that executes it.

This metric-tracking Miri was run on 15 of the top 200 crates. Three failed to execute their crate's entire testbench, and two did not complete within eight hours. From the modest sample of 10, one metric stood out: per crate, **73.83%** of trees created contain just one node (σ = 12.08%). This was identified by tracking how many retag operations occur in each tree, of which there are none for single-node trees. Creating these trees is not necessary to preserve the soundness Miri's analysis: there can be no aliasing violations if an allocation is

Crate	Native (s)	Miri, no TB (s)	OH (x)	Miri, TB (s)	OH (x)
rand-0.10.0	1.0	59.2	60.5	114.2	116.5
typenum-1.19.0	0.2	52.8	278.0	136.9	720.5
smallvec-2.0.0	0.1	90.5	753.9	181.7	1,514.2
indexmap-2.13.0	0.8	98.7	121.8	189.4	233.8
bytes-1.11.1	1.1	126.9	112.3	389.0	344.3
encoding_rs-0.8.35	0.6	85.9	145.5	378.7	641.9
clap_builder-4.5.59	0.2	245.6	1,535.3	429.9	2,686.9
unicode-norm...0.1.25	0.8	312.6	395.6	518.6	656.5
zerocopy-0.8.31	1.5	161.1	108.1	695.8	626.9
chacha20-0.10.0	0.8	168.3	224.4	705.3	940.4
gimli-0.33.0	0.6	292.75	487.92	748.4	1,247.3
itertools-0.14.0	5.1	276.2	54.0	807.7	157.8

Table 1: End-to-End Runtimes and Overheads (OH) of High OH Crates

never aliased! The next question becomes "How do we prevent creating single-node trees in the first place?"

4 Optimization

We implemented a simple optimization based on the principle of laziness. Tree Borrows should not create a tree for a given memory allocation until an alias is created, thus introducing the possibility of an aliasing violation.

4.1 Design

The optimization was designed and tested in two phases: restricting the creation of a tree until any tree-related event occurs, then expanding the restriction until a new node is created.

We first create a state wrapper for the `Tree` data structure. The state may either be `Initialized` with the `Tree` object or `Uninitialized` with the parameters to initialize it later. The state wraps every `Tree` method as well, including the creation of a new tree, in which case it instead creates an `Uninitialized` state. In Lazy v1, every method checks whether the state is `Initialized`. If not, we initialize it before continuing.

In Lazy v2, we only initialize the state when the new child method is called. In all other methods, if the state is `Uninitialized`, we may do nothing or return the empty environment, depending on the return type. This optimization is still a valid Tree Borrows implementation as the state machine of the root node cannot transition unless a reference is created. Additionally, all visits, reads, writes, and prunes to the tree are irrelevant until the first alias.

We validated the analysis of Lazy v1 and Lazy v2 Miri on Miri's CI testbench. They both pass 1,116 of the 1,143 testcases, ignoring the remaining 27 due to in-test comments.

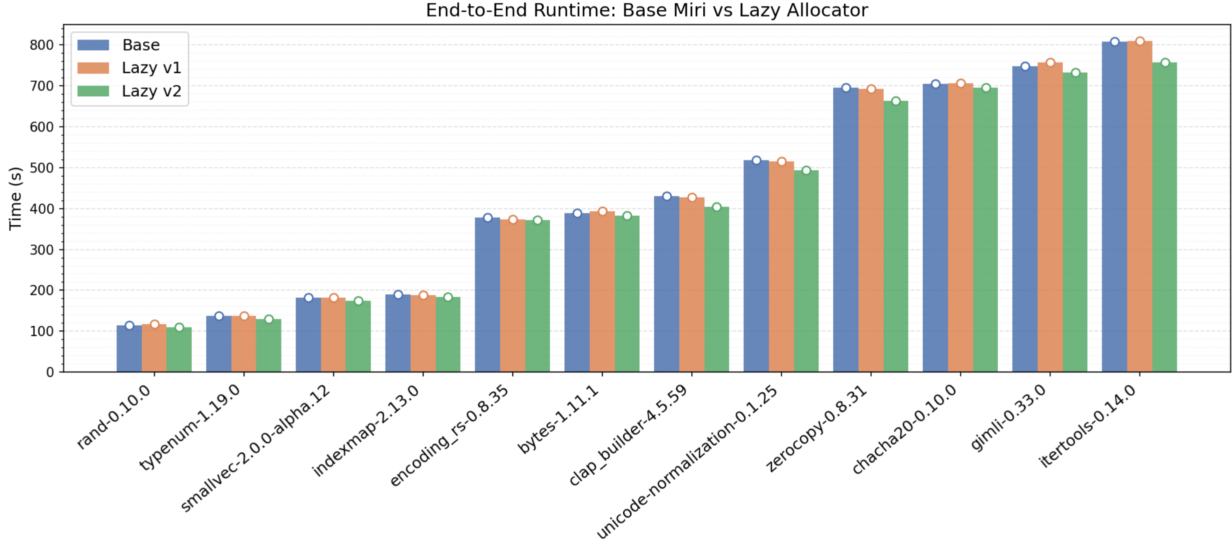

Figure 3: End-to-End Runtimes of High-Overhead Crates with Lazy Optimizations

4.2 Results

We evaluated Lazy v1 and Lazy v2 against the base Miri implementation on the 12 identified high-overhead crates (those with Tree Borrows overheads of over 100×). Runtimes and per-crate speedups are shown in Figure 3.

Lazy v1 provided negligible benefit, only outperforming base Miri in five crates. The best single-crate speedup was 1.013× on `encoding_rs-0.8.35`. This is consistent with expectations: Lazy v1 defers tree initialization only until the first tree event, which occurs almost immediately for most allocations, so the saved work is minimal.

Lazy v2 yielded a more meaningful improvement, achieving a mean speedup of 1.039× (3.8%) across all 13 high-overhead crates, with every crate showing improvement over the base. The maximum speedup was 1.067× (6.7%) on `itertools-0.14.0`, decreasing runtime by 50.4s. The mean Tree Borrows runtime fell from 436.2s (base) to 420.0s (Lazy v2), a reduction of 16.2s on average. The benefit of Lazy v2 is attributable to its stronger defferal condition: single-node trees (comprising 73.83% of all trees created) are never created at all unless a retag occurs.

However, the greatest percent speedup from this optimization was due to the crate with the least overhead. This is likely because the source of the majority of the overhead are extremely large, yet relevant, trees. This warrants future work in other directions to more effectively reduce the runtime of high-overhead cases.

5 Future Work

One identification not mentioned yet in this paper, as it did not directly influence the design of the current optimization, is the concept of a no-op tree. The nodes of these trees contain no state machine transitions that progress it beyond the initial state. Single-node trees are

just a subset of no-op trees, as no-op trees can still have more than one node without state machine progress. Limiting the creation of no-op trees can further reduce runtime overhead.

Further constraining the trees we track to only allocations involved with unsafe code is likely to reduce the overhead. We can safely disregard trees that depend on entirely safe code as they can never exhibit UB. Dynamic taint tracking of unsafe Rust, such as implemented by Zhuang et al. [9], is possible, guided by the MIR's control flow graph.

To optimize Tree Borrows for a fuzzing context, guaranteed soundness may not be crucial. UB that was not caught in one execution may be caught later in an execution which exercises the same path. Therefore, optimization to avoid redundant checks, such as in Kim et al. [4] for *Valg* would speed up each execution, leading to the discovery of more unique paths in a fuzzing loop.

Finally, as evident in Table 1, Miri's baseline interpreter itself contributes to non-trivial overhead. By translating these optimizations to BorrowSanitizer, which operates on the LLVM IR rather than MIR, we can replace and reduce this overhead. However, taint tracking unsafe code at this level may be more difficult as safety information is optimized out [9].

6 Conclusion

This work identifies an overhead to Tree Borrows in Miri: the unnecessary tracking of single-node trees. We solve it through lazily allocating trees only when their root is first aliased, speeding up the end-to-end runtime of every crate we tested. Our analysis scripts and forks of Miri and Rust can be found at `https://github.com/cmu-cornell-for-rust`.

References

[1] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe rust? *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[2] Keaton Brandt. Data-driven performance optimization with rust and miri. *Medium*, December 2022. https://medium.com/source-and-buggy/data-driven-performance-optimization-with-rust-and-miri-70cb6dde0d35.

[3] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. Miri: Practical undefined behavior detection for rust. *Proc. ACM Program. Lang.*, 10(POPL), January 2026.

[4] Shinhae Kim, Saikat Dutta, and Owolabi Legunsen. Valg: A fast reinforcement learning based runtime verification tool for java. *ICSE Demo*, 2026.

[5] Ian McCormack. Borrowsanitizer, 2026. https://borrowsanitizer.com/.

[6] Alex Rebert, Chandler Carruth, Jen Engel, and Andy Qin. Safer with google: Advancing memory safety, October 2024. Google Online Security Blog. https://security.googleblog.com/2024/10/safer-with-google-advancing-memory.html.

[7] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. Tree borrows. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025.

[8] Pradeep Viswanathan. Microsoft finally explains the root cause behind crowdstrike outage, Jul 2024. https://www.neowin.net/news/microsoft-finally-explains-the-root-cause-behind-crowdstrike-outage/.

[9] Zeyang Zhuang, Zilun Wang, Wei Meng, and Michael R. Lyu. Don't panic! finding bugs hidden behind rust runtime safety checks. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, page 186–200, New York, NY, USA, 2025. Association for Computing Machinery.