# BorrowSanitizer

The Rust programming language is increasingly popular for its static safety guarantees, which rely on strict aliasing rules. However, real-world Rust code frequently uses "unsafe" blocks and C/C++ foreign function interfaces (FFI) that can violate these rules and cause undefined behavior. While interpreters like Miri can detect such violations, their high overhead and limited FFI compatibility hinder testing in the multilanguage legacy systems where Rust is rapidly being adopted. We present *BorrowSanitizer (BSan)*, an LLVM-based sanitizer that detects Tree Borrows violations via native instrumentation. BSan lowers minimal Rust type information into LLVM IR, inserts instrumentation to track pointer provenance, and prunes allocation metadata at runtime using hybrid deferred reference counting. In evaluations comparing BSan with Miri, MiriLLI, and CapsLock, BSan is 7× faster than Miri in wall-clock time with 78% less memory—reproducing 96% of the original Tree Borrows violations—and over 2× faster than CapsLock with 44% less memory, reproducing every cross-language bug found by CapsLock and MiriLLI. On real-world code, BSan uncovered 120 previously unreported aliasing violations, including new cross-language bugs in Hurl, CXX, Bun, and AWS's LibCrypto—demonstrating that native instrumentation scales rich, allocation-level aliasing models to production systems.

## 1 Introduction

Modern compilers rely on the principle that a pointer is more than just a location in memory. Pointers also have *provenance*: an implicit capability to access memory [18]. This is a prescriptive idea; language standards require that developers maintain a variety of invariants associated with provenance, and in exchange, implementations can use these invariants to support static optimizations [18]. This relationship has existed in some form for most mainstream systems programming languages, but provenance itself has been relatively underspecified and undervalidated. Developers tend to adopt standards written in ambiguous natural language, and the formal models that can resolve this ambiguity rarely translate into tools for validating production software.

The Rust programming language has broken from this trend by using a static, substructural type system to restrict aliasing and mutability as a means to ensure that pointers are only ever used according to their provenance. However, these restrictions are inevitably conservative, so developers can bypass them using a subset of "unsafe" features [22]. These features support the core of Rust's standard library and language ecosystem, and they are necessary to interoperate with other systems languages. However, it is easy to use unsafe code incorrectly, which can reintroduce the kinds of undefined behavior that Rust is trying to avoid.

This is not a fundamentally new problem—other languages provide similar escape hatches in their type systems—but the Rust community has responded to it in a novel way. The latest formal model of aliasing in Rust, *Tree Borrows* [58], has been implemented as a dynamic analysis within Miri, a Rust interpreter [21]. Executable semantic models exist for other languages [13, 15, 35], but Miri stands out for its tight integration with Rust's compiler. Miri uses the same "backend" as Rust's constant evaluator, and it is broadly recognized as the most accessible, representative model of Rust's operational semantics. It provides a shared source of truth between the Rust practitioners who use unsafe code and the compiler developers who implement its semantics.

However, the complexity of Rust's semantics also brings new challenges for implementing dynamic analysis tools. As an interpreter, Miri has access to rich type information and a high level of control over its execution environment, which makes it easy to implement arbitrary run-time checks. However, high precision has an even higher run-time cost: Miri is typically thousands of times slower than native execution [21]. Interpreters also typically have limited compatibility with operations that fall outside of their target language, like inline assembly and foreign function calls. Architecture-driven techniques, like CHERI [60] and CapsLock [62], can provide better performance

Author's Contact Information:

in theory, but their semantic models are less capable than Miri's, and they rely on exotic hardware platforms (with CapsLock's only available via emulation).

*Sanitizers* leverage compiler support to inject native run-time checks. This sidesteps the need for custom hardware and makes it possible to implement arbitrarily complicated runtime systems. A variety of sanitizers exist, many of which were implemented using LLVM. Many can detect basic provenance-related issues, but none can validate the allocation-level aliasing invariants of Tree Borrows that Miri enforces. Our goal is to design a sanitizer that provides the performance of native instrumentation and multilanguage compatibility, while retaining Miri's ability to detect complex, Rust-specific undefined behavior.

*Contribution.* We explore the viability of this approach with BorrowSanitizer (abbreviated BSan[1]): an LLVM-based dynamic instrumentation tool for finding violations of Tree Borrows. Our approach uses a novel, compact lowering of Rust's high-level type information into LLVM's intermediate representation. This information is consumed by an instrumentation pass that inserts run-time calls into a new concurrent adaptation of Miri's Tree Borrows implementation, with novel techniques for propagating provenance across uninstrumented code, stack lifetimes, and language boundaries. Tree Borrows tracks derived pointers in per-allocation permission trees that grow with every borrow, making subsequent access checks progressively more expensive. Our runtime curbs this overhead using a hybrid, deferred reference-counting garbage collector that includes new techniques for reclaiming unreachable provenance metadata and pruning dead tree nodes. We evaluated BSan on a variety of popular open-source projects and replicated results from multiple prior studies of cross-language aliasing bugs in Rust libraries [33, 62], comparing BSan against existing tools to answer two research questions:

**RQ1** *What is BorrowSanitizer's run time and memory overhead compared to Miri?*

In the most downloaded libraries (§5.1), BSan is more than 5× faster than Miri and more than 2× faster than CapsLock (geometric mean wall-clock time per-library). BSan's peak memory use is 78% less than Miri and 43.5% less than CapsLock.

**RQ2** *How effective is BorrowSanitizer at finding undefined behavior?*

Replicating the evaluations of three bug detection tools (§5.2), BSan finds *all* of the in-scope bugs MiriLLI reported, seven out of eight that CapsLock reported, and 96.36% of the aliasing violations Miri reported in Tree Borrows. It also finds 32 new bugs in MiriLLI's dataset, 72 in CapsLock's, and aliasing violations in 4,086 cases that Miri rejects as unsupported.

These results demonstrate that sanitizers are efficient at enforcing rich, allocation-level aliasing models across language boundaries.

*Outline.* In §2, we provide background on Rust's ownership and provenance models, demonstrate the risks of aliasing violations across foreign function boundaries, and overview Miri alongside existing FFI analysis and sanitizer tools. In §3 and §4, we describe the design and implementation of BSan. In §5, we present the results of evaluation and replication of prior work. We provide an overview of related work in §6, and we conclude and discuss future work in §7.

## 2 Background

Rust is rapidly growing in popularity because it enforces memory safety statically, without the overhead of a managed runtime. Rust achieves this by enforcing a set of ownership rules. Every value has a single owner, and ownership can be transferred through assignment or temporarily

[1]Not to be confused with BinarySanitizer (BSan) [24, 65], a post-compilation dynamic analysis tool for consumer off-the-shelf binaries.

```rust
fn conn_id(easy: &Easy) -> Result<i64, Error> {
  unsafe {
    let conn_id: curl_off_t = 0;
    let rc = curl_sys::curl_easy_getinfo(
        easy.raw(),
        CURLINFO_CONN_ID,
        &conn_id
    );
    cvt(easy, rc)?;
    Ok(conn_id)
  }
}
```

(a) A snippet of Rust code that triggered a cross-language Tree Borrows violation in Hurl [1].

```rust
pub struct Handle<'a, T: 'a> {
  pub(crate) ptr: &'a T,
}

fn example(handle: Handle<'_, *mut JSObject>) {
    let before: *mut JSObject = *handle.ptr;
    js::jsapi::JS_GC();
    let after = *handle.ptr;
    assert_eq!(before, after);
}
```

(b) A snippet illustrating a Tree Borrows violation in MozJS [56]. This bug was fixed before our evaluation, but we replicated it with BSan.

Fig. 1. Minimal examples of cross-language Tree Borrows violations that triggered miscompilation in production software. The lines and expressions that are relevant for triggering each bug have been highlighted.

"borrowed" by creating a reference. References can either be mutable (`&mut T`) or immutable (`&T`); a value can have either a single, unique mutable reference, or many shared, read-only references, but never both during the same lifetime. These rules are enforced via Rust's *borrow checker*, a static analysis pass for Rust's middle intermediate representation (MIR). In addition to security benefits, Rust also uses ownership information to apply optimizations within its LLVM backend. However, these static rules are too restrictive for some valid programming patterns, and the borrow checker cannot reason about low-level operations like inline assembly, foreign function calls, or raw pointer dereferences. Developers can still perform these operations, but only within a context marked with the `unsafe` keyword. When developers use these operations, subtle mistakes can break the compiler's assumptions about aliasing, introducing *undefined behavior*.

BSan helped us find one of these mistakes in Hurl [1]: a popular[2] Rust utility for making HTTP requests. We provide the relevant source code in Fig. 1a. The function `conn_id` returns the unique ID of an HTTP connection by calling the foreign function `curl_easy_getinfo`. This function receives a shared reference as a C variadic argument—bypassing static type checking on the parameter. C then interprets this argument as a mutable pointer and uses it to write the connection ID. However, casting or passing a shared reference across an FFI boundary does not change its provenance: any pointer derived from `&conn_id` remains strictly read-only, so mutating it is undefined behavior. Rust assumes that the value of `conn_id` will never change, so it replaces the additional read access in `Ok(conn_id)` with the value zero. As a result, Hurl will use the wrong certificate to validate a cached connection, causing it to "hurl" an assertion failure at the user.

A similar miscompilation occurred in MozJS [56], a fork of the SpiderMonkey [38] JavaScript engine that is maintained by the Servo Project [4]. This was found and fixed prior to our evaluation, but we were able to use BSan to replicate finding this bug. In MozJS, JavaScript objects (`JSObject`) are allocated in C++ by SpiderMonkey. Pointers to these objects are "rooted" in a dedicated location so that they can be relocated by the garbage collector (GC). Rooted objects can be accessed in Rust via `Handle` wrappers that preserve this level of indirection. As shown in Fig. 1b, each handle holds a shared reference (`&'a T`) to the location where the object's pointer is rooted. This was intended to allow SpiderMonkey's garbage collector update the rooted pointer in C++ without invalidating the handle. However, because this reference is immutable, the Rust compiler assumes the rooted

[2]At the time of our evaluation, Hurl had 19k stars on GitHub and over six years of active development.

pointer cannot change during the handle's lifetime and optimizes away subsequent reads across GC invocations. Instead of reloading the updated pointer from memory after a GC cycle, the compiled code reuses the stale pointer value read before the GC. For example, in Fig. 1b, the assertion in `example` should fail whenever the GC relocates the object, but it always succeeds in release mode; in practice, dereferencing this stale pointer triggered a crash.

### 2.1 Tree Borrows

The programs in Fig. 1 break the rules of Rust's *Tree Borrows* [58] aliasing model. Tree Borrows is the successor to *Stacked Borrows* [20], the initial aliasing model developed for Rust. While Stacked Borrows organized derived pointers on a stack and required borrows to be strictly well-nested, this restriction proved too rigid in practice, rejecting common idioms and precluding certain compiler optimizations. Tree Borrows relaxes this discipline by organizing derived pointers into a tree rather than a stack. We refer the reader to prior work [58] for a comprehensive overview of Tree Borrows; here, we describe the model in broad strokes, which is sufficient for understanding the kinds of errors that BSan can detect.

In Tree Borrows, every allocation has a tree of nodes, and each node identifies an access permission. A pointer's provenance contains a *borrow tag* that uniquely identifies a node. When a reference is derived from an existing pointer, it is *retagged*, which creates a new *child node* in the tree and updates the pointer's provenance with its tag. Each node also tracks a permission state for every byte of the allocation. The root of a tree is always "Unique," which permits any access. Shared references begin as "Frozen," which is read-only. Mutable references and owned pointers, called *box types* in Rust (e.g. `Box`), begin as "Reserved" and become "Unique" after the first write.

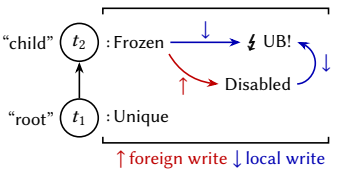



Fig. 2. A subset of the Tree Borrows' state machine that is relevant for the bugs shown in Fig. 1.

When a range of bytes is accessed using a given node, the permissions of every node in the tree are updated for those bytes based on their position relative to the accessed node. An access is considered a *local access* for the node itself and its ancestors; otherwise it is a *foreign access*. With the exception of "Reserved," permissions always decrease monotonically until they reach the "Disabled" state, which does not permit any access. Tree Borrows also uses *protectors* to enforce stronger guarantees for references passed as function arguments. When a function is called, reference arguments undergo a *function-entry retag*. The resulting nodes are *protected* until the function returns, making it undefined behavior for their permissions to transition to "Disabled." This allows downstream optimizations to assume that a reference passed to a function remains valid for the duration of the call.

Each of the examples shown in Fig. 1 involves a similar tree. In Fig. 1a, when the variable `conn_id` is allocated on the stack, its tree begins as a singleton *root* node with a Unique permission. Then, evaluating the expression `&conn_id` executes a retag, creating a pointer with a Frozen permission. Likewise, in Fig. 1b, the rooted allocation begins with a Unique permission in C++, and receives a Frozen child when it is borrowed to create the handle. We illustrate the shape of this tree and relevant state transitions, in Fig. 2. Writing through the child is undefined behavior (Fig. 1a). Writing through the root causes the child to transition to Disabled, breaking any protectors (Fig. 1b) and making future accesses undefined behavior.

### 2.2 Tools for Finding Aliasing Bugs

Miri [21], a Rust interpreter, is the most widely used dynamic analysis tool in the Rust ecosystem. It implements the Tree Borrows and Stacked Borrows aliasing models as dynamic analyses, and

it is where the latest changes to these models are prototyped. However, Miri can only interpret Rust's MIR, so it cannot detect either of the aliasing violations shown in Fig. 1, which require interoperating with C and C++. Miri will stop short of the FFI boundary, terminating with an "unsupported operation" error message. Miri can execute functions from natively compiled shared libraries if the user manually specifies the path to each library. However, only a limited set of argument types are supported, and Miri no longer tracks provenance in this context.

MiriLLI [33] attempted to fill this gap by extending Miri to interoperate with LLI [27], an LLVM interpreter. It replaced each of LLI's core operations for accessing memory and controlling the flow of execution with foreign function calls back into Miri. By extending Miri's existing error detection mechanisms across foreign function boundaries, MiriLLI was able to find new instances of undesirable or undefined behavior from 37 Rust libraries. Notably, this included `flate2`, a compression library maintained by the Rust Project. Although MiriLLI was effective at finding bugs, it had limited test coverage: of the 9,130 test cases used in the evaluation, 61% terminated early due to an unsupported operation.

CapsLock [62] used the Capstone [63] architecture to find cross-language aliasing violations. Capstone is a "tagged architecture" implemented as an emulated extension to RISC-V. It associates pointers with "capabilities" that have revoke-on-use semantics. New capabilities are created at allocation sites and when references are cast into raw pointers. Casts were identified by modifying the Rust compiler to insert additional calls into LLVM. CapsLock identified eight new bugs in eight libraries in its evaluation and was twice as fast as Miri across test cases from 100 popular libraries. However, it relied on emulation of experimental hardware that has yet to materialize, and it implements a more minimal aliasing model than either Stacked or Tree Borrows, without support for protection semantics, or byte-level tracking of interior mutability. CapsLock will miss bugs related to these semantic features.

*Sanitizers* represent a potential middle-ground between these approaches; they instrument a program at compile time to analyze its behavior during native execution. A sanitizer typically consists of a compiler pass, which inserts checks before relevant operations, and a runtime library, which maintains metadata about the program's state and reports errors when a check fails. Most sanitizers store this metadata in *shadow memory*: a region of the address space that maps each byte of application memory to the corresponding metadata. Sanitizers are typically much faster than interpreters, which makes them more practical for running large test suites, fuzzing, and property-based testing. Compile time instrumentation provides broader compatibility with commodity hardware and software platforms.

A variety of sanitizers exist as part of the LLVM toolchain [10, 25, 31, 32, 39, 51, 52, 57].. The only existing sanitizer for finding aliasing-related errors is TypeSanitizer (TySan) [31], which finds violations of the C and C++ standards' strict (type-based) aliasing rules. This algorithm is structurally different from Tree Borrows; TypeSanitizer's descriptor trees are fixed, static objects, while borrow trees are dynamic. There has not been a comparable solution in this category for finding violations of Rust's aliasing properties—until now.

## 3 BorrowSanitizer Overview

BorrowSanitizer creates pointer-level metadata to track provenance at run time, finding violations of Rust's Tree Borrows aliasing model. Users can run a program under BorrowSanitizer using a plugin for Cargo, Rust's build tool (e.g. "`cargo bsan test`"). Our plugin intercepts invocations of the Rust compiler and Clang, configuring each compiler to insert our instrumentation. Programs are statically linked against our runtime, and dynamically linked against instrumented copies of the Rust and C++ standard libraries. We will discuss BSan's implementation in further detail in §4. Here, we provide an overview of how BSan detects the bug in Hurl, which was shown in Fig. 1a.

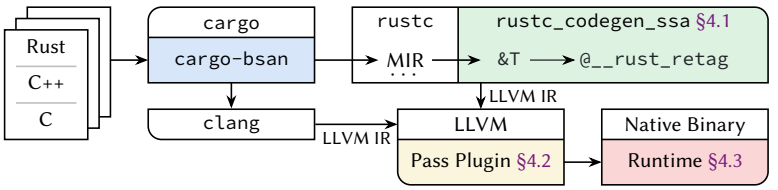



Fig. 3. BorrowSanitizer's architecture.

*Provenance Metadata.* At run time, BSan associates every pointer with a two-word provenance tuple $\langle a, t \rangle$ containing an allocation ID, $a$, and a borrow tag, $t$. The allocation ID is a pointer to an allocation metadata object containing that allocation's borrow tree, while the borrow tag is a 64-bit integer generated from a global atomic counter. Each tag uniquely identifies a node within that tree. For example, in Fig. 1a, when the local variable `conn_id` is allocated on the stack, BSan creates a metadata object $a_1$ whose borrow tree contains a single root node with tag $t_1$ and a Unique permission, giving the variable's stack pointer the provenance $\langle a_1, t_1 \rangle$.

*Retagging.* Next, creating the shared reference `&conn_id` requires executing a retag. However, all of the type information that we need to figure out when to perform a retag is lost when a program is lowered to LLVM IR. BSan preserves this information by emitting function calls that mark at the LLVM IR level when a pointer should be retagged. Our LLVM instrumentation pass translates these calls into concrete run-time checks. In Fig. 1a, we increment the global counter to create a new borrow tag $t_2$, and then use it to initialize a new node as a child of $t_1$. We pass the resulting provenance $\langle a_1, t_2 \rangle$ to the function.

*Propagating Provenance.* We pass provenance between functions using dedicated shadow memory regions. For heap allocations, we create metadata at run time by intercepting calls to system allocation functions (e.g. `malloc`, `calloc`, `realloc`) and associating the newly created provenance with the returned pointer. When a pointer is stored into memory, we store its provenance to corresponding shadow memory regions. In Fig. 1a, passing `&conn_id` as a variadic argument to `curl_easy_getinfo` stores its provenance $\langle a_1, t_2 \rangle$ into shadow memory, where the instrumented C code retrieves it when unpacking the argument into a local pointer. Before a function returns, we invalidate the provenance of every stack allocation and remove protection from each provenance value created by a function-entry retag.

*Validating Memory Accesses.* We insert run-time checks to validate a pointer's provenance before it is used to access memory. When a pointer is used to deallocate memory, we invalidate its metadata object so that all future accesses will be rejected as undefined behavior. The undefined behavior in Fig. 1a is detected when a write access check fails, detecting a write using the Frozen permission of $t_2$. After detecting an error, we terminate the program and report an error message. Error reports have the same level of precision as Miri. They provide a history of state transitions and accesses that span foreign function boundaries.

## 4 Design and Implementation

BSan consists of three core components distributed across the Rust and LLVM toolchains, illustrated by Fig. 3). Within the Rust compiler, we encode Miri's implicit retags as explicit function calls during code generation (§4.1). Next, at the LLVM level, our instrumentation pass consumes these retags, inserts access checks, and propagates provenance metadata through shadow memory alongside every pointer (§4.2). Finally, our runtime library (§4.3) validates these checks against a concurrent

adaptation of Miri's Tree Borrows engine and periodically "prunes" unreachable tree nodes to keep memory and access-checking overhead low.

### 4.1 Retagging in LLVM IR

In Tree Borrows, *retags* create new nodes in an allocation's borrow tree to establish the aliasing permissions of derived references and boxes. However, Rust's middle intermediate representation (MIR) does not contain explicit `Retag` statements. Instead, a retag must occur whenever a reference is created, or whenever a value containing a reference or box is copied from an *operand* (typically a scalar value) into a destination *place* in memory via a *typed copy* [19]. During code generation, BSan identifies these events and recursively walks the layout of the copied type, emitting function calls into LLVM IR to retag every reference and box it encounters.

We refer to these emitted function calls as *retag intrinsics*, and there are two kinds. The first kind is a *register retag*, which is used to retag a pointer that has already been loaded from memory. It returns a pointer with the same address but different provenance. The second kind is a *memory retag*, which is used to retag a pointer in-place at another memory location. It receives a pointer to the location, and it is equivalent to loading the pointer from memory, retagging it, and then storing it back again.[3]

*Register Retag*
```
ptr __rust_retag_reg(ptr, ..)
```

*Memory Retag*
```
void __rust_retag_mem(ptr, ..)
```

Each retag intrinsic has four additional parameters that determine the permission it creates. The first parameter is a series of four bitflags, which indicate whether the permission created by the retag should be protected, whether the pointer is mutable, whether the permission is for a box, and whether the pointee type is *interior mutable*—meaning it allows mutation through a shared reference. The second parameter is the size of the pointee type, which indicates the range of bytes within the allocation that will be covered by the new permission. At run time, we iterate through this range and paint each sub-range with its permission.

Certain fields require relaxed aliasing rules due to interior mutability. We identify these fields using the third and fourth parameters, which specify an optional *interior mutability layout*. When we emit a retag, we capture its interior mutability by visiting the layout of the pointee type and recording the offset and width of each of its interior mutable fields, which tells the runtime precisely which sub-ranges require relaxed permissions. Following Miri, we do not recurse into the variants of enumerations or the fields of unions; because these types involve overlapping storage or dynamic runtime variants, static field tracking would be unsound. Instead, we conservatively treat enums and unions as entirely interior mutable if they contain at least one such field.

An interior mutability layout has a static component and a dynamic component. The static component describes fixed-size types and consists of two parts: either an explicit list of offset and width pairs for interior-mutable fields, or a repeat command. We use repeat commands specifically for array types; without them, large arrays of partially interior-mutable types would generate massive static data structures in the binary. To keep this representation compact, adjacent offset-width pairs are merged, and nested array iterations are flattened when possible.

The dynamic component handles Rust's dynamically sized types (DSTs)—specifically slices and dynamic trait objects, whose exact sizes and layouts are only known at runtime. For slices, we determine the layout of a single element and repeat that layout across every element in the collection. For dynamic trait objects, we treat all remaining bytes up to the dynamic size as interior

[3] Memory retags are necessary to support LLVM's `readonly` annotation and to allow retagging vectors of references. See the following discussion [link removed for double-blind review] for more details.

mutable if any implementation of the trait permits interior mutability.[4] Finally, we encode this interior mutability layout using a compressed format similar to a Lempel–Ziv encoding [66], and we provide a descriptive grammar in the appendix.

*4.1.1 Emitting Retags.* We emit retag intrinsics in Rust's backend-agnostic code generation module wherever Tree Borrows requires a retag—such as reference creation, function entry, and typed copies. When a value requires retagging, we traverse its type layout to construct a *retag plan*: a pruned sub-layout retaining only fields and variants that contain references or boxes. Guided by this plan, we project into the value and emit a memory retag if the pointer resides in a memory place, or a register retag if it is already loaded as an operand.

```
pub enum Outer<'a> {
    P(&'a u32),
    Q(Inner<'a>),
}

pub enum Inner<'a> {
    A(&'a u8),
    B(&'a u16),
    C(u128),
}
```

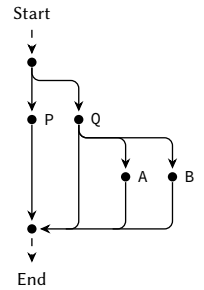



Fig. 4. The definition of an enum `Outer<'a>` and a diagram of the CFG emitted for retagging it.

Unlike structs and tuples, which allow static field projections, arrays and enums require generating control flow. For an array `[&T; N]` passed by value, we emit a loop over its indices to project and retag each element individually. For an enum whose retag plan spans multiple variants, we switch on its discriminant and emit separate retagging blocks per variant, as shown in Figure 4.

*4.1.2 Exposing Provenance.* Rust uses an exposed-and-wildcard provenance model [3, 35] for pointer-integer conversions. When a pointer's provenance is exposed, its allocation is recorded in a global table; pointers subsequently cast from integers receive *wildcard* provenance and are optimistically validated against compatible exposed allocations. Because Rust lowers both exposing and non-exposing casts to LLVM's "`ptrtoint`" instruction, instrumenting every `ptrtoint` would over-expose provenance and cause false negatives. Instead, we emit a "`__rust_expose`" intrinsic at explicit exposure sites to mark the argument pointer's provenance as exposed at run time.

## 4.2 Provenance in LLVM

After lowering a program to LLVM IR, we run an instrumentation pass that inserts our run-time checks. BSan represents provenance as a two-word pair consisting of an *allocation ID* and a *borrow tag*. Alongside concrete tags—which identify a specific permission for an allocation—and the "Wildcard" tag introduced above, BSan adds two new tags (highlighted):

$$\begin{aligned} \mathit{AllocID} &\triangleq \mathbb{N} \\ \mathit{BorTag} &\triangleq \mathrm{Concrete}(\mathrm{tag} : \mathbb{N}) \mid \mathrm{Wildcard} \mid \mathrm{Invalid} \mid \mathrm{Omnivalid} \\ \mathit{Provenance} &\triangleq \mathit{AllocID} \times \mathit{BorTag} \end{aligned}$$

The "Invalid" tag is assigned to stack allocations outside their active lifetimes; any access using it is reported as undefined behavior. Conversely, the "Omnivalid" tag is assigned whenever provenance tracking is omitted or unavailable—such as for constants, stack-safe allocations, and values crossing uninstrumented boundaries—causing all accesses to be accepted. Whereas Miri prioritizes soundness and halts on unsupported operations, BSan trades soundness for coverage by treating missing provenance as valid, allowing it to find downstream bugs that Miri cannot reach.

[4] Miri recurses into the vtable to inspect this, whereas we conservatively treat the remaining bytes as interior mutable to avoid vtable inspection complexity.

*What has provenance?* Although only pointers conceptually carry provenance [18, 45], in practice, integer types for raw memory copies and ABI type-punning—for example, lowering a pair of pointers `{ptr, ptr}` passed across `extern "C"` to `[2 x i64]`. Propagating provenance solely through `ptr` values would lose precision at FFI boundaries. LLVM has introduced a dedicated byte type (`bN`) for this purpose, but it is still unstable. Instead, BSan allows word-sized integers to carry provenance across loads and stores (but not arithmetic or pointer derivation), assigns Wildcard provenance to integers cast to pointers, and assigns Omnivalid provenance to constants, arithmetic results, and all other non-pointer values.

*Shadow Memory.* BSan stores provenance in two word-aligned *shadow memory* regions: one for borrow tags and one for allocation IDs. Each application address is mapped to its shadow addresses via a single XOR and constant offset. Each word-sized unit of shadow memory, or "*granule*," tracks the provenance of one word of application memory. We copy shadow in word-aligned chunks without splitting provenance across sub-granule bytes (§4.4), and we skip writing the allocation ID whenever a tag is non-concrete to avoid unnecessarily paging in shadow memory.

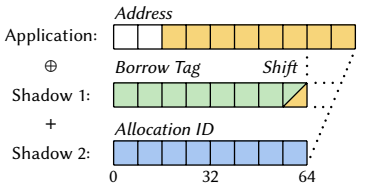



Fig. 5. The layout of a shadow granule.

Pointers can be stored at unaligned addresses (e.g., inside packed structs), causing them to straddle two granules. To handle this, BSan rounds the address down to store provenance in the lower granule and reserves the lower three bits of the borrow tag to record the pointer's byte offset within that granule. By default, we trust LLVM's alignment annotations (though BSan can conservatively treat every store as unaligned). Word-aligned, word-multiple stores write directly to their corresponding granules. For unaligned stores, we encode the destination offset into the tag and write to the lower granule. If the store crosses into the next granule, we check its stored offset and clear any overwritten shadow. When copying shadow between differently aligned addresses, we adjust these offset bits accordingly. An aligned store may partially overwrite an unaligned pointer without clearing its lower-granule shadow. This remains reasonable because the corrupted pointer retains its original provenance. Any corrupted address shifted outside the allocation is caught as out-of-bounds (or faults), while in-bounds accesses are still validated against the allocation's Tree Borrows permissions.

*Stack Lifetimes.* To track stack allocations accurately, BSan models LLVM's "`lifetime.start`" and "`lifetime.end`" intrinsics, which delimit when a stack slot is live so the compiler can merge non-overlapping slots. If a stack allocation (`alloca`) has no explicit lifetime intrinsics, we initialize its provenance immediately upon allocation. Otherwise, it starts with the Invalid tag, receives a fresh concrete tag at each `lifetime.start`, and reverts to Invalid at each `lifetime.end`.

Lifetime bounds also affect LLVM's "`getelementptr`" (GEP) instruction, which computes a new pointer by adding an offset to a base pointer. Normally this would eagerly inherit the base pointer's provenance. However, compiler optimizations—such as hoisting a loop-invariant address calculation out of a loop—can move a GEP of an `alloca` before its `lifetime.start` [30]. At that point, the allocation's provenance is still Invalid; eagerly copying provenance at the GEP would stamp the offset pointer as Invalid for every subsequent use inside the loop. Instead, we lazily assign the provenance of an `alloca`-derived GEP at each use, using the allocation's current lifetime tag.

We query LLVM's *Stack Safety* analysis [29] to skip instrumenting trivial stack allocations. The analysis classifies an allocation as safe only when all accesses to it are statically proven in-bounds within its lifetime and its pointer never escapes to unknown calls. Although in-bounds access alone does not rule out Tree Borrows violations, LLVM treats BSan's retag intrinsics as unknown

calls, automatically marking any retagged allocation unsafe. Only in-bounds allocations that are never retagged remain classified as safe, allowing us to omit metadata for them and assign them Omnivalid provenance.

*Shadow Frames.* When a function returns, BSan must invalidate its stack allocations, remove protectors from function-entry retags, and pass return-value provenance to its caller. To track this metadata across calls, each thread maintains a dedicated shadow stack, and instrumented functions allocate a *shadow frame*—a per-call metadata record—upon entry. Each shadow frame stores provenance for static stack allocations (`alloca`), a dynamic list of protected provenance values from function-entry retags (whose count varies when branching on enum discriminants, Fig. 4), *roots* (loaded or retagged provenance recorded in memory so register-held values remain visible during garbage collection, §4.3.1), and space for return-value provenance. Before returning, the callee writes its return provenance to the frame and calls a runtime routine to invalidate stack provenance and remove protectors; the caller then retrieves the return provenance and pops the frame.

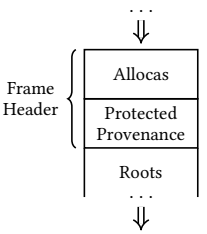



Fig. 6. The layout of a function's shadow frame.

*Boundary Validation.* When instrumented code calls uninstrumented code (or vice versa), return values or parameters lack valid provenance, which would trigger false positives. To prevent this, our pass identifies calls to external functions (which may be uninstrumented) and functions with external linkage or taken addresses (which may have uninstrumented callers). Before calling a potentially uninstrumented function, the caller writes the callee's address into a thread-local *boundary marker*. Upon entry, an instrumented callee checks whether the marker matches its own address: if so, the caller was instrumented and the callee clears the marker to null; otherwise, the caller was uninstrumented and the callee assigns its parameters Omnivalid provenance. On return, the caller checks whether the marker was cleared to null—trusting the return provenance if it was, and otherwise resetting the shadow frame's return slot to Omnivalid.

*Safepoints.* To support stop-the-world tree pruning (§4.3.1), BSan must pause threads at *safepoints* where all live provenance is visible in shadow memory and shadow-frame roots. During IR instrumentation, we run LLVM's safepoint pass, which inserts calls to a stub declaration, `gc.safepoint.poll`, at function entries, exits, and loop backedges [26]. BSan supplies the stub's implementation: each poll performs a cheap load of a global atomic flag set by the pruning thread and, if active, takes a slow path with a stronger atomic load to suspend execution.

### 4.3 Runtime

BSan's runtime links a C++ "wrapper" and a Rust "core" into a single static library. Built on LLVM's compiler runtime libraries, the wrapper manages shadow memory, function interception, global state, and error reporting. The core implements wildcard provenance, allocation metadata, and Tree Borrows validation, building on Miri's borrow-tree state machine. Below, we describe how BSan manages concurrent allocation metadata and tracks provenance reachability before detailing how unreachable nodes are reclaimed.

*Allocation Metadata.* Each allocation is represented by a metadata object whose core state—size, liveness, and borrow tree—is guarded by a per-allocation mutex. Because any allocation may be accessed concurrently, validating a memory access acquires this lock, checks liveness and bounds, and updates the borrow tree's permissions. To reduce overhead, BSan defers instantiating the full

borrow tree until the allocation's first retag. This *lazy allocation* strategy avoids constructing tree structures for the vast majority of allocations that never alias and thus have only a single root node.

To reclaim unreachable nodes and metadata without tracing the entire heap, BSan maintains reference counts for both the allocation metadata object and each initialized tree node. Whenever instrumented stores or intercepted memory intrinsics modify provenance in heap or global shadow memory, the runtime updates the allocation's reference count and—if its tree is initialized—the node's reference count, performing non-atomic updates under the allocation's mutex. Separating the allocation-level reference count allows a stack slot's metadata object to be reused across multiple lifetimes: reinitializing the tree resets permissions while preserving the allocation's reference count to account for dangling pointers from prior lifetimes.

Allocations and nodes start with a reference count of zero. When new provenance is created (via allocation or retag) or a reference count drops to zero, its tag is added to a thread-local *zero count table* (ZCT) [11]—a map from allocation IDs to sets of borrow tags—which is merged into a global ZCT when the thread exits. Heap deallocation eagerly clears the freed region's shadow memory to decrement reference counts of stored pointers. By contrast, stack shadow is not reference-counted, and tags are not eagerly removed from ZCTs when their counts become nonzero; instead, stack shadows are cleared and ZCT entries are filtered during tree pruning (§4.3.1). Reference count updates are eager, for the moment. Switching to a mutation buffer [6] over a ZCT could be a useful optimization for future work.

*4.3.1 Tree Pruning.* Each live allocation maintains a borrow tree that tracks pointer derivations and per-byte access permissions. Every memory access traverses this tree to validate the accessed pointer and update permissions. A node must remain in its tree as long as any live pointer carries its tag—both for soundness (a node disabled on one byte range may remain valid on another) and to preserve access histories for error diagnostics. Left unchecked, trees can grow to hundreds of thousands of nodes, slowing down every subsequent access. Whereas Miri periodically traces the entire heap to find live provenance, BSan pairs deferred reference counting [11] (via ZCTs) with periodic stop-the-world *tree pruning* to reclaim unreachable nodes and metadata without scanning the heap.

*Visit-based Pruning Intervals.* BSan schedules pruning based on actual tree-traversal work rather than control-flow metrics like Miri's fixed 10,000-basic-block interval, which correlates poorly with aliasing activity. Each memory access traverses an allocation's tree once per affected *location* (a contiguous byte range with uniform permissions), applying local transitions to the accessed node and its ancestors and foreign transitions elsewhere (skipping unaffected subtrees). Unpruned dead nodes compound this traversal cost on every access. Each thread counts visited nodes in a thread-local counter. The first thread to reach 150,000 visits triggers a collection across all allocations and resets all counters. Programs with heavy aliasing prune frequently, whereas programs that rarely retag incur minimal collection overhead.

*Stopping the World.* A thread reaching the visit threshold acquires a sequence lock to become the sole *pruning thread*, treating all other threads as *mutators*. Following asymmetric cooperative suspension protocols [12, 37], the pruning thread sets a global safepoint trigger and issues a process-wide memory barrier, ensuring mutators observe the trigger using a cheap relaxed load at safepoints. Newly created threads start in a *safe* state, whereas threads executing instrumented code or runtime checks are *unsafe*. During boundary validation (§4.2), a mutator calling an uninstrumented function transitions itself to the safe state before the call and transitions back to unsafe—polling the trigger—upon return, allowing pruning to proceed without waiting for uninstrumented code (such as

blocking system calls) to finish. Mutators reaching a safepoint while the trigger is active announce a *waiting* state and park on a futex; once every mutator is safe or waiting, pruning begins.

*Collecting Unreachable Provenance.* First, the pruning thread identifies all provenance reachable from stacks and registers by scanning each thread's stack shadow and the *roots* in its shadow frames (where functions record loaded and retagged provenance so register-held tags remain visible). During this scan, it also clears stale shadow between the current stack pointer and its high-water mark since the last prune, and resets each thread's visit counter. Next, it drains unreachable tags from every thread's ZCT into a set of candidates whose reference counts may be zero, and partitions this set by allocation ID. For each affected tree, the pruning thread prunes dead nodes. Afterward, if the tree's root node was pruned, its allocation-level reference count is zero, and its provenance is absent from all roots and stack shadows, BSan frees the allocation's metadata object. If the root node remains reachable with a zero reference count (common for stack allocations), its provenance is inserted into the global ZCT so it will be revisited if it never enters heap memory; if its reference count is nonzero, future drops will return it to the ZCT automatically.

*Pruning.* For each candidate tag in a tree's pending set, if the corresponding node's reference count is zero, we add it to a dead-node list sorted in descending tag order. Because tags are allocated monotonically, descending order processes leaves before interior nodes. Leaf nodes are removed directly, whereas an unreachable interior node $p$ can only be *compacted*—removed and replaced by its children—when doing so preserves Tree Borrows semantics.

Specifically, compaction is sound only if removing $p$ cannot suppress undefined behavior reports that $p$ would otherwise have triggered—for example, when a Disabled or Frozen parent restricts accesses through a more permissive child. To guarantee this, every child $c$ must *simulate* $p$ at every byte location: for any access, $c$ must reject at least every access that $p$ would reject, and if both step to new states, $c$'s new state must again simulate $p$'s. Because Tree Borrows defines a finite set of per-location permission states, this simulation relation is verified exhaustively over state pairs and used as a fast lookup check during pruning.

Whereas Miri compacts only single-child nodes, BSan generalizes compaction to multi-child nodes by reparenting all children onto $p$'s grandparent (Fig. 7). Multi-child compaction is subtler because siblings observe accesses differently from their parent: a write through one child is local to $p$ but foreign to its siblings, and an interior-mutable reserved permission ("ReservedIM") ignores foreign writes. We therefore enforce a stronger simulation relation and extend the exhaustive state-pair check so that $p$ and each child may observe each access under updated relatedness conditions. To prevent flattening deep trees into overly wide nodes, compaction is bounded so that $p$'s parent retains at most 16 children—empirically, this had the best run time result.

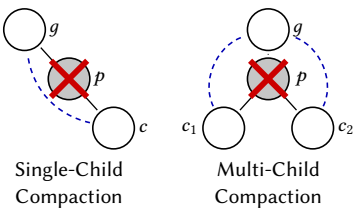



Fig. 7. Tree pruning configurations, showing the reparenting of $c_i$ onto $g$.

### 4.4 Limitations

BSan currently leaves global variables, thread-local variables, and `repr(simd)` vectors uninstrumented, assigning them Omnivalid provenance. Similarly, atomic pointer stores propagate provenance to shadow memory, whereas atomic pointer loads return Omnivalid provenance. Because Omnivalid bypasses Tree Borrows checks, aliasing violations involving pointers loaded atomically or stored in globals, thread-locals, or SIMD types can go undetected (false negatives). In addition, BSan omits reference-count decrements when an atomic store overwrites a shadow

entry or when a write updates another thread's stack shadow; the overwritten tags therefore retain nonzero reference counts, preventing tree pruning from reclaiming their nodes and metadata. Future versions can close these gaps by initializing global and thread-local shadow via module constructors, extending shadow propagation to SIMD lanes, and synchronizing shadow updates for atomic accesses and cross-thread stack writes.

Uninstrumented code—such as external libraries, inline assembly, or system calls—can instead cause false positives. While BSan intercepts common system calls and clears shadow memory touched by kernel writes, it does not propagate provenance across uninstrumented operations. If uninstrumented code copies a pointer in memory without updating its shadow, subsequent instrumented reads may pair the pointer with stale provenance and report spurious violations. Extending boundary validation to mark allocations exposed to uninstrumented code would allow BSan to selectively suppress these warnings.

Finally, BSan tracks provenance at word-sized granule granularity rather than per byte, diverging from Rust and LLVM's byte-level provenance model. In practice, this has little impact on bug detection because programs almost always copy pointers as intact words; moreover, bugs that misassemble individual pointer bytes typically trigger immediate crashes or segmentation faults rather than subtle aliasing violations. Where byte-level precision is required, BSan could be extended to store per-byte shadow entries at the cost of higher memory and instrumentation overhead.

## 5 Evaluation

We evaluated BSan on its performance and bug-finding capabilities. In §5.1, we compare BSan's time and memory use against CapsLock and Miri on versions of the top 100 most-downloaded libraries from `crates.io` used within the original CapsLock evaluation. In §5.2, we replicated aliasing bugs that Miri found in the original evaluation of Tree Borrows [58], as well as a variety of cross-language aliasing bugs from CapsLock and MiriLLI. In §5.3, we describe the qualitative attributes of new aliasing bugs that we discovered during this process, and from independently use.

### 5.1 Performance

*Experimental Setup.* We replicated CapsLock's performance evaluation, adding BSan for comparison. Both BSan and Miri were built from the same tip-of-tree Rust toolchain.[5] For a fair comparison, we made two adjustments to CapsLock to match BSan and Miri: (1) enabling capability instrumentation for `hashbrown`, `addr2line`, and `gimli`, which CapsLock skips by default,[6] and (2) disabling terminal-information parsing in Rust's test harness to match BSan and Miri. Note that we are comparing against the QEMU-based version of CapsLock—no physical hardware platform exists. We configured Miri with Tree Borrows enabled, auxiliary checks (e.g., data race detection) and host isolation disabled, and deterministic concurrency enabled, invoking the interpreter binary directly to exclude Cargo overhead. To match each baseline's default configuration, we evaluated BSan with an optimized standard library against CapsLock and an unoptimized standard library against Miri.

Following Yu et al. [62], we ran all unit and integration tests from the top 100 most downloaded `crates.io` libraries (as of October 24, 2024) using the locked dependencies from their replication package [64]. Retaining tests that pass within a two-minute timeout yields 5,447 tests shared between BSan and Miri, and 5,554 shared between BSan and CapsLock, spanning 72 libraries (1–1,755 tests each; median 33.5). All experiments ran in Ubuntu 22.04.5 or 24.04.5 LTS Docker containers on a 36-core Intel Xeon w9 workstation with 128 GB of memory.

[5] 1.101.0, 64ac935

[6] We will be reaching out to the authors—this may be an oversight in their evaluation

Table 1. Test outcomes with a two-minute timeout (T/O), and the median, mean, and max test execution time, wall-clock time, and peak memory over the "common" tests. BSan* uses an unoptimized standard library. Shaded columns show the number of common tests on which the tool outperforms its counterpart.

| | Outcome | | | Test execution time (s) | | | | Wall-clock time (s) | | | | Peak memory (MiB) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Tool | Pass | Fail | T/O | Best | Med. | Mean | Max | Best | Med. | Mean | Max | Best | Med. | Mean | Max |
| BSan* | 5,597 | 5 | 191 | 5,443 | 0.015 | 0.406 | 112.9 | 5,445 | 0.180 | 0.573 | 113.1 | 5,446 | 47.3 | 49.9 | 757.5 |
| Miri | 5,450 | 99 | 244 | 4 | 0.193 | 1.67 | 116.9 | 2 | 1.39 | 2.85 | 117.7 | 1 | 206.4 | 236.9 | 373.3 |
| BSan | 5,613 | 5 | 175 | 5,439 | 0.004 | 0.496 | 91.5 | 5,439 | 0.059 | 0.552 | 91.6 | 5,551 | 22.0 | 24.8 | 146.1 |
| CapsLock | 5,636 | 61 | 96 | 115 | 0.069 | 0.647 | 112.3 | 115 | 0.158 | 0.732 | 112.4 | 3 | 36.3 | 80.3 | 3,138.2 |

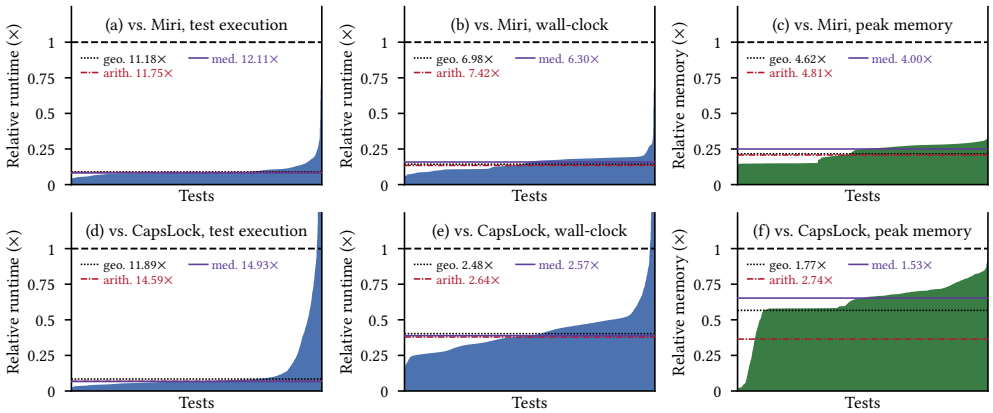



Fig. 8. Relative overhead of BSan per test compared with Miri (top) and CapsLock (bottom) in test execution time, wall-clock time, and peak memory. The dashed line ($y = 1$) is Miri's or CapsLock's value. The horizontal lines mark the geometric mean (geo.), arithmetic mean (arith.), and median (med.) of BSan's speedup or memory reduction. Tests are sorted by ratio; ratios above 1.25 (where BSan is slower) are clipped.

We evaluate runtime using two metrics: (i) *test execution time*, recorded by Rust's test harness from test start to finish excluding setup (matching CapsLock's evaluation [62]); and (ii) *wall-clock time*, covering full process execution including startup and tool initialization, measured via Hyperfine [44] as the median of three runs after one warm-up run. In a separate single pass with identical configurations, we record *peak memory* as the peak resident set size across all processes spawned by each test.

*Results.* As shown in Table 1, BSan is faster than Miri in both test execution and wall-clock time for 99.9% of tests, and it is faster than CapsLock for 97.9% of tests. BSan uses less memory than Miri and CapsLock on more than 99.9% of tests. Fig. 8 shows BSan's run time and memory use relative to Miri and CapsLock, with the geometric mean, arithmetic mean, and median. BSan is 11.18× (geomean) faster than Miri in test execution time, and 6.98× faster in wall-clock time, and Miri uses 4.62× as much memory. When aggregating these results per-library, BSan is 10.35× faster in test execution time and 5.55× faster in wall-clock time, and Miri uses 3.85× as much memory. Compared with CapsLock, BSan is 11.89× (geomean) faster in test execution time and 2.48× faster in wall-clock time, and CapsLock uses 1.77× as much memory. Per library, BSan is 9.54× faster in test execution time and 2.12× faster in wall-clock time, and CapsLock uses 1.73× as much memory.

*Discussion.* BSan is faster than Miri and CapsLock on most tests, in both metrics. Speedups in wall-clock time are smaller because each process has a fixed cost, which is greater than the execution time for most tests.

The speedups are smaller in wall-clock time because each process has a fixed cost that is added to every test and most tests are fast. For Miri, each run compiles the program to MIR and interprets the program's start-up before the tests start. CapsLock and BSan load the test binary and libraries and initialize the tool (e.g., QEMU's translation of the start-up code). BSan is slower than CapsLock on 115 tests that are long-running. Miri is also slow on them, so their cost is likely to be caused by tracking borrows under Tree Borrows. For example, tests in `aho-corasick` call small functions with references into the same few large vectors many times, so they perform many retags on allocations whose trees are large. But, a shorter garbage-collection interval can reduce overhead in those cases: in one test, BSan takes 5.4 seconds with its default interval and 2.6 seconds when it collects ten times as often. Conversely, BSan is slower than Miri on four tests in which many threads contend for a lock or a shared value. Miri runs one thread at a time, whereas under BSan the threads run in parallel.

BSan uses less memory than Miri and CapsLock on most tests. Miri has a certain disadvantage here because it tracks additional state related to each allocation, including initialization. Miri also keeps in memory the compiler's data structures for the whole program including dependencies, such as the MIR of every function that it may interpret. Miri uses less memory on one test, which keeps 1,000 threads alive at once; under BSan, each is an operating-system thread with its own stack, whereas Miri simulates threads within the interpreter. CapsLock's memory includes the code that QEMU translates for the test, the capability metadata, and the nodes of CapsLock's revocation tree. The largest peak (3,138.2 MiB) occurs under CapsLock in a `chrono` test, where two CapsLock processes, each with its own revocation tree, run at the same time. Each tree's memory grows with every new node until its pool of $2^{24}$ 88-byte nodes is exhausted (1,408 MiB); only then does it reuse the slots of dead nodes. In contrast, BSan periodically prunes nodes that are no longer reachable in memory (§4.3.1); its peak memory on this test is 59.5 MiB.

### 5.2 Bug and Aliasing Violation Detection

We used BSan to replicate the evaluations of MiriLLI [33], CapsLock [62], and the evaluation of Miri's implementation of Tree Borrows [58].

*Running Datasets.* Each of the datasets were constructed by downloading versioned libraries from the lists provided by MiriLLI [34], CapsLock [64], and Tree Borrows [59] evaluation's replication package. All three evaluations consist of the testbenches of libraries downloaded from `crates.io`, where published versions are immutable. This guarantees that the versions we download are identical, even if they have been "yanked." However, libraries can still be deleted, which prevented us from downloading a small portion of each dataset. For MiriLLI and CapsLock, we also downloaded the latest version of each library if it had been updated since the original evaluation.

We executed this evaluation on the nodes of two HPC clusters; we ran MiriLLI and CapsLock's datasets on Intel Xeon 8468 CPUs and Miri's Tree Borrows on AMD EPYC 7742 CPUs for greater parallelization. We ran each library's test case sequentially, giving each 1 core and 8GB of memory. We initially set a one hour timeout for MiriLLI's dataset, but after observing each UB report return in under 30 minutes, we lowered the timeout to 30 minutes for CapsLock's Dataset. For Miri's Tree Borrows dataset, we replicated their evaluation's timeout of 150s and included feature-gated tests.

*Identifying Bugs.* While the Tree Borrows evaluation only counted Miri's UB reports, the MiriLLI and CapsLock evaluations counted unreported bugs each tool discovered. Therefore, we classified UB reports that BSan produced from MiriLLI and CapsLock's datasets into unique bugs.

First, we indexed all reports into one dataset, tagging each library with the datasets it came from. Because multiple test cases can trigger the same bug, including across libraries that share a dependency, we deduplicated these reports by the library and source file where the error was triggered, abstracting versions and line numbers. Reports from the same file were not merged if they came from the same version or reported different types of UB. This ensured that we would recognize the same bug across different versions of a library, even when its location within the file changed. We further split aliasing violations into the three categories used in MiriLLI's evaluation, A local read or write to a Disabled node is classified as using an *invalid* permission. A local write with a Frozen node is classified as using an *insufficient* permission (Fig. 1a). A foreign access to a protected node is *protected* (Fig. 1b). For these violations, we grouped reports by the location where the offending tag was created. This merges violations that share a root cause but are triggered at different program points. This automated grouping is still conservative; for instance, it does not recognize a bug that reappears in a fork of a library. We manually reviewed each candidate bug to determine whether it was a duplicate, a false positive, or a previously fixed or reported bug. As a post-filtering step, we ran Miri with Tree Borrows on every test for which BSan reported UB. If Miri reported UB at the same location or a validation error before it could reach the aliasing violation, we did not consider the report to indicate a new bug in this evaluation.

#### 5.2.1 *Bug Identification Results.*

*Replicating Bugs.* We replicated all relevant bugs from MiriLLI [33] and CapsLock [62]. Each tool provides a public artifact via Zenodo. The MiriLLI dataset [34] includes 46 bugs, of which 24 involve aliasing violations or accesses out-of-bounds. We reproduced each of MiriLLI's bugs. The CapsLock dataset [64] includes eight bugs. We could reproduce seven. BSan misses a bug that CapsLock found in the library `ring` because it was a Stacked Borrows violation that is accepted under Tree Borrows. We confirmed this by running the relevant test in Miri.

We replicated bugs by locating relevant test cases from the public reports and commit hashes provided by each dataset. CapsLock's evaluation did not report which test produced the bug, in which case we executed the entire testbench. MiriLLI's replication required patching libraries to fix errors that were once warnings in earlier versions of Rust, or to replace broken versions of dependencies. One bug from CapsLock required additional intervention: the library `corgi` replaces Rust's global allocator with `mimalloc`, which is not intercepted by BSan's runtime, so we had to revert back to the default allocator to detect its bug. CapsLock found a bug in the library `sxd-document` by running a test in `cargo-wix`, one of its dependents. The test in `cargo-wix` triggers a Stacked Borrows violation, so BSan misses the bug in this context, but we were able to find a Tree Borrows violation with the same root cause by running a test in `sxd-document`.

Table 2. Libraries built and tests run from each evaluation dataset, including updated libraries. CapsLock's set contained no pre-filter for runnable testbenches.

| | MiriLLI | *Updated* | CapsLock | *Updated* | Distinct |
|---|---|---|---|---|---|
| Libraries | 1,645 | 718 | 17,317 | 6,278 | 25,352 |
| Built | 1,160 | 558 | 4,962 | 2,606 | 8,816 |
| Tests run | 26,107 | 46,903 | 186,081 | 210,170 | 440,736 |

*Finding New Bugs.* To find bugs that other tool evaluations missed, we ran BSan on the datasets used to evaluate MiriLLI and CapsLock. MiriLLI was evaluated in September 2023 on 9,130 tests that Miri rejected for unsupported FFI, drawn from 23,116 tests across 1,645 libraries [33]. CapsLock was evaluated in October 2024 on 86,051 tests from the 1,000 most downloaded libraries on `crates.io` and 14,000 libraries randomly sampled from a list of 17,317 likely to contain FFI [62]. We ran the entire test suite of every library in MiriLLI's dataset and in CapsLock's full candidate list. Where

a library had been updated since the original evaluation, we also tested its latest version. Only a portion built with BSan on the October 2026 nightly compiler; the distribution of libraries and tests run by BSan are in Table 2. Across both datasets, BSan reported no UB in 336,154 tests and UB in 19,852. Of the rest, 44,669 panicked or crashed, 31,193 ran out of memory, and 8,868 exceeded the timeout. We excluded the 7,549 UB-reporting tests where Miri detects the same error.

After applying our grouping strategy, we organized the remaining reports into 270 distinct bugs. Of these, 70 had already been independently reported or fixed. Another 80 were false positives; although they occurred in many contexts, they stemmed from only a few root causes, such as missing instrumentation for `qsort` and `strtol`. Only one false positive was an aliasing violation; a function used the `longjmp` system call to exit out of a closure before protection was removed, leading to a failed foreign access in the caller. Both `setjmp` and `longjmp` have unstable semantics in Rust and are not supported by Miri [16]. The remaining 120 are new unreported bugs (Table 3). Of these, 83 occur in the original libraries: 32 in MiriLLI's dataset and 72 in CapsLock's, with 21 in both. The other 37 could only be found in the updated versions, 23 from libraries where BSan could not build the original and 14 newly introduced bugs. These bugs originate from 99 distinct libraries. In 83 of these libraries, the library's own tests exposed the bugs, accounting for 99 bugs. The remaining 16 libraries, which account for 21 bugs, were exposed only by the tests of libraries that depend on them. We are in the process of reporting these bugs, the majority of which are in actively-maintained repositories.

Table 3. New bugs (Use After Free, Out of Bounds Access, and Aliasing Violations) identified in the original evaluations' datasets and in their updated versions.

| | | Memory | | Aliasing | | | |
|---|---|---|---|---|---|---|---|
| | Dataset | UAF | OoB | Inv. | Ins. | Pro. | Σ |
| *Original* | MiriLLI | 7 | 3 | 2 | 17 | 3 | 32 |
| | CapsLock | 9 | 7 | 15 | 36 | 5 | 72 |
| | Distinct | 11 | 8 | 16 | 42 | 6 | 83 |
| *Updated* | MiriLLI | 1 | 1 | 1 | 5 | 1 | 9 |
| | CapsLock | 0 | 4 | 11 | 14 | 4 | 33 |
| | Distinct | 1 | 5 | 11 | 15 | 5 | 37 |
| Distinct Total | | 12 | 13 | 27 | 57 | 11 | **120** |

*Discussion.* BSan found 83 new bugs in the same library versions on which MiriLLI and CapsLock were evaluated. This is likely due to the fact that MiriLLI terminates early on non-FFI unsupported operations [33], and that CapsLock's aliasing model is more "relaxed" than Tree Borrows [62]. This count is still conservative: some libraries have since been deleted, many of the original versions no longer build on a current nightly compiler, and we excluded every bug that was reported or fixed even after the original evaluations. However, neither evaluation ran every test in its dataset: MiriLLI ran only the tests that Miri rejected for unsupported FFI, and CapsLock randomly sampled a subset of its candidate libraries. Some of the bugs we report may therefore lie in tests that neither tool was given the chance to run. BSan's relatively low overhead is what made it practical to run every test in every library that built. Finally, 23 of the 37 bugs found only in updated versions come from libraries whose original versions did not build, so we cannot determine whether those bugs were already present when the original evaluations were conducted.

*5.2.2 Aliasing Violation Results.* We ran BSan on the dataset from the Tree Borrows evaluation [59], and compared our results to the evaluation's logs, to check whether it reproduced the aliasing violations that Miri found and to determine if we could find violations that Miri could not reach.

*Replicating Known Aliasing Violations.* Tree Borrows was evaluated against Stacked Borrows on the top 30,000 most downloaded libraries from `crates.io` in September 2024. They reported 3,023 aliasing violations with Tree Borrows. To obtain this figure, they first ran all 674,748 tests in Miri without borrow tracking and discarded tests that timed out, failed, exhibited other undefined behavior, or used unsupported features. This filtering is appropriate for comparing borrow-tracking

models within Miri, but it excludes tests that BSan could still run. Instead, we start from their unfiltered results, in which Miri with Tree Borrows found 5,070 tests with violations.

Of these 5,070 tests, 299 tests originate from 25 libraries that no longer build using BSan's nightly version of Rust. Of the remaining 4,771, BSan reproduced the aliasing violation in 4,507; 122 timed out or ran out of memory, and 142 reported no undefined behavior. However, 94 of those 142 also pass under the latest Miri—these are no longer considered violations of Tree Borrows. Excluding these, BSan reproduced 4,507 of 4,677 (96.36%) of the original violations in libraries that still build.

*Finding New Aliasing Violations.* BSan ran 471,594 tests from this dataset with the October 2026 nightly compiler. Of this set, 439,283 reported no UB and 20,201 timed out or ran out of memory. BSan detected UB in 12,110 tests, 11,417 of which were aliasing violations.

Mirroring our replication, Miri with Tree Borrows reported an aliasing violation in 4,507 of these tests and agreed with BSan on the type in all but 61, because the tools stopped at different stages of the same bug. Miri did not find UB in 4,086 tests, as it halted on an *unsupported operation*. In 2,387 of these tests, across 153 libraries, the operation is a call to a foreign function. The remaining 1,699 across 99 libraries halt on other unsupported operations, such as system calls and inline assembly. In Table 4, we compare these results to Miri's Tree Borrows output. Of tests omitted from the table, Miri failed to build 2,134, which may be a limitation of the original evaluation's setup, not Miri. Miri also reported other UB first in 235, timed out on 150, and passed without UB in 448.

Table 4. A subset of the violations reported by BSan on the Tree Borrows dataset, by Miri's result for the same test. Shaded: Miri halts on an unsupported operation.

| | Miri (TB) | | | | | |
|---|---|---|---|---|---|---|
| | Unsupported | | Aliasing Violation | | | |
| BSan | FFI | Other | Inv. | Ins. | Pro. | Σ |
| Inv. | 756 | 774 | 590 | 0 | 0 | 2,120 |
| Ins. | 1,590 | 890 | 61 | 3,590 | 0 | 6,131 |
| Pro. | 41 | 35 | 0 | 0 | 266 | 342 |
| Σ | **2,387** | **1,699** | 651 | 3,590 | 266 | 8,593 |

*Discussion.* The two tools are complementary. BSan reproduces 96.36% of the aliasing violations Miri found in this dataset, and in §5.1, runs at a fraction of the cost of Miri. This makes it practical in settings where Miri is too slow, such as fuzzing or running every test in continuous integration. Miri remains the more complete checker, as it detects UB that BSan does not track, such as invalid values, data races, and alignment errors [21], so it is still worth running periodically. BSan's main contribution, however, is the violations Miri cannot reach. In §5.2.1, only one aliasing violation bug was identified as a false positive, so we have high confidence that the violations BSan reports on this dataset are real bugs, including tests where Miri passes with no UB. Most of these violations are ones Miri does not report, primarily because it halts on an unsupported operation, usually FFI, before reaching them. These are the programs that BSan is designed to target: Rust that crosses into native code, where an interpreter cannot follow.

### 5.3 Qualitative Findings

We found a variety of new aliasing bugs throughout this process. Aliasing violations rarely have security implications, so we opted to disclose these errors publicly unless we believed that there were security implications or observed differences in behavior caused by an aliasing violation.

*AWS Libcrypto.* AWS maintains Rust bindings for Libcrypto: a cryptography library used internally at Amazon to support TLS. Bindings are available via the library `aws-lc-rs`, which had *117M recent downloads* at the time of writing. This library passed aliasing mutable and immutable references into a foreign function. In C, writing through the pointer derived from the mutable reference invalidated the pointer derived from the shared reference, triggering undefined behavior when it was used to read memory.

*Bun & zlib.* Bun [5] is a JavaScript runtime for the Node.js ecosystem. Bun was originally implemented in Zig, but after the project was acquired by Anthropic, it was translated into Rust with assistance from Claude Fable 5 [55]. However, not all of Bun is written in Rust—it still depends on zlib [14], a popular compression library implemented in C. We equipped an agent with BSan (see our AI use disclosure) and used it to discover a new cross-language aliasing violation in Bun's encapsulation of this library. Bun uses a box to allocate a `zstream` object, which contains the state associated with zlib's compression algorithm. A pointer to this object is derived from a temporary mutable reference, which is implicitly created by dereferencing the box. Then, the pointer is passed into C and copied into the heap, where it becomes disabled by a foreign access. This triggers undefined behavior when it is accessed again in C. The MiriLLI evaluation found similar aliasing violations in `flate2` and `bzip2` that we replicated. We also found several instances of this bug within `zlib-rs`, another popular zlib encapsulation. The bugs in `zlib-rs` had been found by Miri through code paths that did not exercise the FFI.

*CXX.* The Rust ecosystem provides several frameworks for supporting interoperation with C++ from Rust. One of these frameworks, CXX, had 12M recent downloads at the time of writing. It supports implementing a "bridge" module for sharing type definitions between each language. This includes Rust's "`String`," which can be accessed via an equivalent object in C++. It provides an API for accessing the string as a "`const char*`" by allocating space for a null terminator. However, the pointer that Rust returned to C++ from this function was derived from an immutable reference with a Frozen permission, so writing with it triggered undefined behavior.

## 6 Related Work

BorrowSanitizer has notable similarities to MSan [53] and DFSan [25] in that each of these sanitizers implements a dynamic taint analysis. BSan's taint sources are retags and allocation sites, and its taint sinks are memory accesses and deallocation sites. SoftBound and CETS [40, 41, 43] have the closest analogue to our "lock-and-key" allocation-level metadata. BSan is the only native dynamic analysis tool for finding Tree Borrows violations that we are aware of, but it is not the first Rust-specific sanitizer.

*Sanitizing Rust.* RustSan [9] and ERASan [36] are optimized variants of AddressSanitizer. Each emitted taint markers into LLVM IR to identify sources of unsafe operations, and both used SVF's implementation of Andersen's analysis [54] as a foundation for taint tracking. Each tool ran twice as fast as baseline AddressSanitizer by skipping run time checks for untainted objects. This approach is likely to be effective for BSan—much of our current execution time is wasted tracking the state of allocations that are never accessed via unsafe operations, and are therefore never subject to Tree Borrows violations. However, SVF is expensive—compilation was nearly 20× slower under RustSan. LiteRSan [61] implemented a custom dataflow analysis to replace SVF, and it had only a 2× compilation-time overhead. SafeFFI [8] elided sanitizer checks by asserting invariants at casts between raw pointers, skipping checks for the remaining lifetime of the reference.

*Valgrind.* Valgrind [42] is a dynamic binary instrumentation framework, which represents an entirely different category from interpreters like Miri or sanitizers like BSan. It recompiles native code on the fly. Krabcake [23], an extension to Valgrind for detecting Stacked Borrows violations, was proposed and prototyped in 2023, but the project has not been active since. However, BSan's retag intrinsics are available upstream in nightly Rust, which would make it easier to implement this kind of tooling. We expect that such a tool would still be useful to the Rust community because neither BSan nor Miri supports inline assembly or JIT-compiled code.

*Isolation.* Orthogonal research efforts have attempted to mitigate cross-language undefined behavior via isolation. An early example is Fidelius Charm [2], which leveraged support from the Linux Kernel to allow users to selectively protect individual memory locations from unsafe operations by creating "memory compartments," which were read-only or inaccessible. Both Galeed [49] and TRust [7] used Intel's Memory Protection Keys (MPK) to isolate memory regions used by unsafe operations. Notably, Galeed explicitly targeted multilanguage programs by preventing foreign dependencies from accessing Rust-allocated heap memory except through "pseudo-pointers"—32-bit IDs for protected memory objects. Omniglot [50] partitions memory into disjoint regions for the "host" Rust program and its foreign dependencies. The Rust side of the boundary can access foreign memory via a newtype wrapper that dynamically validates an interior mutable reference.

*Ownership in LLVM IR.* Several prior analysis tools have lowered parts of Rust's type information into LLVM IR, but none provide detailed information on interior mutability or emit the necessary control-flow to support function-entry retagging. The closest analogue to BSan's retag intrinsics was used by SeaUrchin [46–48], a bounded model checker for LLVM IR. The authors modified the Rust compiler to emit additional GEP instructions with metadata annotations that indicated whether a pointer was a mutable reference, a shared reference, or an "owning" pointer (e.g. a box). SafeFFI [8] and RustSan [9] relied on metadata nodes and attributes to distinguish references and raw pointers. These approaches do not TRust [7] and ERASan [36] attached metadata to identify operations that fell within unsafe contexts.

## 7 Conclusion & Future Work

We presented BorrowSanitizer (BSan), an LLVM-based sanitizer for detecting violations of Rust's Tree Borrows model. By lowering minimal Rust type information into LLVM IR, BSan instruments pointer operations to propagate provenance through shadow memory and validate accesses against a concurrent borrow-tree runtime. BSan outperforms both Miri and CapsLock while reproducing nearly all aliasing violations from the original Tree Borrows evaluation and every cross-language bug found by CapsLock and MiriLLI, and finding many violations that those tools could not.

Despite these speedups, BSan still incurs higher overhead than traditional sanitizers because it instruments every read, write, and retag without static optimization. Further runtime tuning can help, but closing this performance gap requires statically eliminating redundant checks. Allocations accessed exclusively in safe Rust—or exclusively in C and C++—cannot trigger Rust aliasing violations; developing a sound, lightweight static analysis to elide checks on such allocations would be a promising direction for both BSan and Miri.

Beyond Rust, any language that lowers aliasing guarantees into LLVM faces similar risks of undefined behavior. Although LLVM lacks a canonical aliasing model, a subset of Tree Borrows has been proposed for LLVM [17] and prototyped in LLUBI [28]. In future work, we plan to generalize BSan into *ProvenanceSanitizer*, a configurable framework for validating pointer provenance against the evolving semantics of attributes such as `noalias` and `dereferenceable`.

## AI Use Disclosure

This paper was written by humans. AI tools were used to identify errors and grammatical issues, brainstorm wording, create templates for some of the tables and figures, convert handwritten drawings into a typeset format, and create BibTeX entries for preidentified sources, which were subsequently audited. We used AI tooling throughout our development process and evaluation in a variety of ways—typically for code review, debugging, and pair programming. There has always been a human in the loop. The bug in Bun was detected by directing an agent (usually running Claude Opus 4.8 or 5) to use BSan. An agent pointed the implication of the bug in Hurl.

## Data-Availability Statement

This paper's artifact is a replication package of our evaluation and raw results. It will be submitted for Artifact Evaluation. It will include a multi-architecture container image (`amd64` and `arm64`) with the version of BSan that we used to run the experiments, lists of versioned libraries pulled for each dataset, and the scripts used to run the evaluation. A fully anonymized subset of our raw data is available for reviewers as supplementary material. This includes a spreadsheet of all raw reports of undefined behavior from BSan and the bug categorization spreadsheet, with identifying information redacted. At paper submission time, we will still be in the process of making bug reports. Bugs that require private reporting will be redacted.

## References


[1] 2026. Hurl. https://hurl.dev/.

[2] Hussain M. J. Almohri and David Evans. 2018. Fidelius Charm: Isolating Unsafe Rust Code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy* (Tempe, AZ, USA) *(CODASPY '18)*. Association for Computing Machinery, New York, NY, USA, 248–255. doi:10.1145/3176258.3176330

[3] Roy Ammerschuber. 2026. *Supporting Wildcard Provenance for Tree Borrows in Miri.* Bachelor's thesis. ETH Zürich. https://ethz.ch/content/dam/ethz/special-interest/infk/inst-pls/plf-dam/documents/StudentProjects/BachelorTheses/2025-Roy-Thesis.pdf

[4] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the Servo web browser engine using Rust. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 81–89. doi:10.1145/2889160.2889229

[5] Anthropic, Oven, and Bun contributors. 2026. Bun: A fast all-in-one JavaScript runtime. https://bun.com/.

[6] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. 2001. Java without the coffee breaks: a nonintrusive multiprocessor garbage collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 92–103. doi:10.1145/378795.378819

[7] Inyoung Bang, Martin Kayondo, HyunGon Moon, and Yunheung Paek. 2023. TRust: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 6947–6964. https://www.usenix.org/conference/usenixsecurity23/presentation/bang

[8] Oliver Braunsdorf, Tim Lange, Konrad Hohentanner, Julian Horsch, and Johannes Kinder. 2026. SafeFFI: Efficient Sanitization at the Boundary Between Safe and Unsafe Code in Rust and Mixed-Language Applications. In *35th USENIX Security Symposium (USENIX Security 26)*. USENIX Association, Baltimore, MD, 2843–2861. https://www.usenix.org/conference/usenixsecurity26/presentation/braunsdorf

[9] Kyuwon Cho, Jongyoon Kim, Kha Dinh Duy, Hajeong Lim, and Hojoon Lee. 2024. RustSan: Retrofitting AddressSanitizer for Efficient Sanitization of Rust. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 3729–3746. https://www.usenix.org/conference/usenixsecurity24/presentation/cho-kyuwon

[10] Clement Courbet. 2021. NSan: a floating-point numerical sanitizer. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 83–93. doi:10.1145/3446804.3446848

[11] L. Peter Deutsch and Daniel G. Bobrow. 1976. An efficient, incremental, automatic garbage collector. *Commun. ACM* 19, 9 (Sept. 1976), 522–526. doi:10.1145/360336.360345

[12] Dave Dice, Hui Huang, and Mingyao Yang. 2001. *Asymmetric Dekker Synchronization.* Technical Report SUN040896. Sun Microsystems. https://github.com/davedice/davedice.github.io/blob/main/resources/Asymmetric-Dekker-Synchronization.txt

[13] Chucky Ellison and Grigore Roşu. 2012. An executable formal semantics of C with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 533–544. doi:10.1145/2103656.2103719

[14] Jean-loup Gailly and Mark Adler. 2026. zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library. https://zlib.net/.

[15] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 336–345. doi:10.1145/2737924.2737979

[16] Connor Horman. 2023. Is longjmp through rust code UB. https://github.com/rust-lang/unsafe-code-guidelines/issues/404

[17] Johannes Hostert. 2025. Tree Borrows: A Model of LLVM's noalias? https://jhostert.de/blog/2025/noalias/

[18] Ralf Jung. 2024. Rust Has Provenance. GitHub. https://github.com/rust-lang/rfcs/pull/3559

[19] Ralf Jung. 2026. Make retags an implicit part of typed copies. https://github.com/rust-lang/rust/pull/154341

[20] 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. doi:10.1145/3371109

[21] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. 2026. Miri: Practical Undefined Behavior Detection for Rust. *Proc. ACM Program. Lang.* 10, POPL, Article 48 (Jan. 2026), 29 pages. doi:10.1145/3776690

[22] Steve Klabnik, Carol Nichols, and Chris Krycho. 2026. The Rust Programming Language. https://doc.rust-lang.org/book/. With contributions from the Rust Community.

[23] Felix S. Klock II and Bryan Garza. 2023. Krabcake: A Valgrind-based dynamic checker for Rust's aliasing model. https://github.com/pnkfelix/krabcake.

[24] Yang Li, Yuan Li, Chenyang Li, Xinhui Han, and Chao Zhang. 2026. Poster: BSan: A Non-Intrusive and Comprehensive Binary-Level Memory Sanitizer. In *Network and Distributed System Security Symposium (NDSS) (Poster Session)*. https://www.ndss-symposium.org/wp-content/uploads/ndss26-poster-91.pdf

[25] LLVM Project. 2026. DataFlowSanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html

[26] LLVM Project. 2026. Garbage Collection Safepoints in LLVM. https://llvm.org/docs/Statepoints.html

[27] LLVM Project. 2026. lli – directly execute programs from LLVM bitcode. https://llvm.org/docs/CommandGuide/lli.html.

[28] LLVM Project. 2026. llubi – LLVM UB-aware Interpreter. https://llvm.org/docs/CommandGuide/llubi.html. LLVM Command Guide.

[29] LLVM Project. 2026. Stack Safety Analysis. https://llvm.org/docs/StackSafetyAnalysis.html.

[30] LLVM Project. 2026. *StackColoring.cpp*. https://github.com/llvm/llvm-project/blob/main/llvm/lib/CodeGen/StackColoring.cpp#L347

[31] LLVM Project. 2026. TypeSanitizer. https://clang.llvm.org/docs/TypeSanitizer.html

[32] LLVM Project. 2026. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[33] Ian McCormack, Joshua Sunshine, and Jonathan Aldrich. 2025. A Study of Undefined Behavior Across Foreign Function Boundaries in Rust Libraries. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2075–2086. doi:10.1109/icse55347.2025.00167

[34] Ian McCormack, Joshua Sunshine, and Jonathan Aldrich. 2025. *A Study of Undefined Behavior Across Foreign Function Boundaries in Rust Libraries*. doi:10.5281/zenodo.14867373

[35] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang.* 3, POPL, Article 67 (Jan. 2019), 32 pages. doi:10.1145/3290380

[36] Jiun Min, Dongyeon Yu, Seongyun Jeong, Dokyung Song, and Yuseok Jeon. 2024. ERASan: Efficient Rust Address Sanitizer. In *2024 IEEE Symposium on Security and Privacy (SP)*. 4053–4068. doi:10.1109/SP54263.2024.00258

[37] Mono Project. 2022. Runtime Cooperative Suspend. https://www.mono-project.com/docs/advanced/runtime/docs/coop-suspend/.

[38] Mozilla. 2026. SpiderMonkey: Mozilla's JavaScript and WebAssembly Engine. https://spidermonkey.dev/.

[39] Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. 2019. Analyzing control flow integrity with LLVM-CFI. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 584–597. doi:10.1145/3359789.3359806

[40] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 245–258. doi:10.1145/1542476.1542504

[41] 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. doi:10.1145/1806651.1806657

[42] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 89–100. doi:10.1145/1250734.1250746

[43] Benjamin Orthen, Oliver Braunsdorf, Philipp Zieris, and Julian Horsch. 2024. SoftBound+CETS Revisited: More Than a Decade Later. In *Proceedings of the 17th European Workshop on Systems Security* (Athens, Greece) *(EuroSec '24)*.

Association for Computing Machinery, New York, NY, USA, 22–28. doi:10.1145/3642974.3652285
[44] David Peter. 2026. *Hyperfine*. https://github.com/sharkdp/hyperfine
[45] Nikita Popov. 2025. [RFC] A Provenance Model for LLVM. https://hackmd.io/@nikic/SJBt4mFCll. Draft RFC.
[46] Siddharth Priya. 2026. *From Mock Environments to Ownership-Aware Compilation: Practical Advances in Low-Level Program Reasoning*. Ph.D. thesis. University of Waterloo, Waterloo, Ontario, Canada. Department of Electrical and Computer Engineering. https://dspacemainprd01.lib.uwaterloo.ca/server/api/core/bitstreams/7ee31cee-c81e-4bfd-ada4-ef098d7c1929/content
[47] Siddharth Priya and Arie Gurfinkel. 2024. Ownership in Low-Level Intermediate Representation. In *Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024*, Nina Narodytska and Philipp Rümmer (Eds.). 292–300. doi:10.34727/2024/ISBN.978-3-85448-065-5_35
[48] Siddharth Priya, Joseph Tafese, Giuliano Losa, and Arie Gurfinkel. 2025. Bounded Model Checking for Rust. Talk at the Fifth Rust Verification Workshop (RW 2025), co-located with ETAPS 2025. Slides: https://drive.google.com/file/d/1Eyc6a7YiPHgTt0Ytyq7btENUu9SMQicE/view. Workshop: https://sites.google.com/view/rustverify2025.
[49] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping Safe Rust Safe with Galeed. In *Proceedings of the 37th Annual Computer Security Applications Conference* (Virtual Event, USA) *(ACSAC '21)*. Association for Computing Machinery, New York, NY, USA, 824–836. doi:10.1145/3485832.3485903
[50] Leon Schuermann, Jack Toubes, Tyler Potyondy, Pat Pannuto, Mae Milano, and Amit Levy. 2025. Building Bridges: Safe Interactions with Foreign Languages through Omniglot. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. USENIX Association, Boston, MA, 595–613. https://www.usenix.org/conference/osdi25/presentation/schuermann
[51] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC '12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany
[52] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71. doi:10.1145/1791194.1791203
[53] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 46–55. doi:10.1109/CGO.2015.7054186
[54] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC '16)*. Association for Computing Machinery, New York, NY, USA, 265–266. doi:10.1145/2892208.2892235
[55] Jarred Sumner. 2026. Rewriting Bun in Rust. https://bun.com/blog/bun-in-rust
[56] The Servo Project Developers. 2026. mozjs: Rust Bindings for SpiderMonkey. https://github.com/servo/mozjs.
[57] David Trevelyan, Ali Barker, and Chris Apple. 2024. RealtimeSanitizer (RTSan). https://github.com/realtime-sanitizer/rtsan.
[58] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. 2025. Tree Borrows. *Proc. ACM Program. Lang.* 9, PLDI, Article 188 (June 2025), 24 pages. doi:10.1145/3735592
[59] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. 2025. *Tree Borrows*. Artifact. doi:10.5281/zenodo.15002703
[60] Robert N M Watson, Peter G Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A Theodore Markettos, Simon W Moore, Steven J Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2023. *Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 9)*. Technical Report. SRI International and the University of Cambridge.
[61] Tianrou Xia, Kaiming Huang, Dongyeon Yu, Yuseok Jeon, Jie Zhou, Dinghao Wu, and Taegyu Kim. 2025. LiteRSan: Lightweight Memory Safety Via Rust-specific Program Analysis and Selective Instrumentation. arXiv:2509.16389 [cs.CR]
[62] Jason Z. Yu, Fangqi Han, Kaustab Choudhury, Trevor E. Carlson, and Prateek Saxena. 2025. Securing Mixed Rust with Hardware Capabilities. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25)*. Association for Computing Machinery, 1544–1558. doi:10.1145/3719027.3744861
[63] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, and Prateek Saxena. 2023. Capstone: A Capability-based Foundation for Trustless Secure Memory Access. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 787–804. https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jason
[64] Zhijingcheng Yu, Fangqi Han, Kaustab Choudhury, Trevor E. Carlson, and Prateek Saxena. 2025. *Artifacts of "Securing Mixed Rust with Hardware Capabilities"*. doi:10.5281/zenodo.14625328

[65] Wen Zhang, Botang Xiao, Qingchen Kong, Le Guan, and Wenwen Wang. 2025. BSan: A Powerful Identifier-Based Hardware-Independent Memory Error Detector for COTS Binaries. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 3085–3096. doi:10.1109/ICSE55347.2025.00020

[66] J. Ziv and A. Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 24, 5 (1978), 530–536. doi:10.1109/TIT.1978.1055934

## Appendix

A descriptive grammar of the additional parameters given to the retag intrinsic and a K-map diagram of the boolean function mapping bitflag attributes to the initial permission states of Tree Borrows.

$$
\begin{aligned}
\mathit{RetagInfo} &\triangleq \{\mathsf{perms} : \mathbb{B}^4,\ \mathsf{size} : \mathbb{N},\ \mathsf{im} : \mathit{Layout}_{\perp}\ \mathsf{elems} : \mathbb{N},\} \\
\mathit{Layout} &\triangleq \mathit{List}(\mathit{Chunk}) \times \mathit{Tail}_{\perp} \\
\mathit{Chunk} &\triangleq \mathit{List}(\{\mathsf{offset} : \mathbb{N},\ \mathsf{width} : \mathbb{N}^+\}) \\
&\quad \mid \mathsf{Iter}\{\mathsf{stride} : \mathbb{N},\ \mathsf{count} : \mathbb{N}^+, \mathsf{chunks} : \mathit{List}(\mathit{Chunk})\} \\
\mathit{Tail} &\triangleq \mathsf{Fill} \mid \mathsf{DynIter}\{\mathsf{stride} : \mathbb{N},\ \mathsf{chunks} : \mathit{List}(\mathit{Chunk})\}
\end{aligned}
$$

