We Need to Talk About the Actual Guarantees of Rust-based Systems

Corinn Tiffany Artem Agvanian Ziyun Song Isabella Szabo Kinan Dak Albab[†]
Will Crichton Philip Levis[‡] Akshay Narayan Deepti Raghavan Malte Schwarzkopf

Brown University [†]*Boston University* [‡]*Stanford University and Google*

Abstract

Over the past decade, systems research has leveraged Rust to build systems that enforce valuable security and isolation guarantees over user-provided code in *extensions*. This paper identifies a common weakness in these Rust-based systems: extensions, even those that may be considered "safe," may fail to uphold the language properties on which these systems rely. We present case studies where this mismatch breaks system guarantees and "safe" extensions corrupt kernel state in RedLeaf [35] and leak private data in Sesame [13]. As a path forward, we propose tooling to detect possible violations of system invariants in extension code. An early prototype static analysis tool suggests that this approach detects guarantee-breaking code with low false-positive rates.

1 Introduction

A longstanding goal in systems research is to enforce security properties over user-provided applications and extensions. Traditionally, researchers achieve this by building mechanisms with substantial complexity or runtime overhead. Rust offers a promising alternative: extensions written in Rust have comparable performance to C/C++ code, but also offers attractive language properties like memory safety. From these properties, systems can bootstrap simpler enforcement mechanisms with lower overhead [3]. Systems use this approach to isolate kernel components [35, 37, 32, 7] or network and serverless functions without virtualization [36, 8], build eBPF alternatives [20, 21], and perform information flow analysis [11] and control [13, 30, 5]. We share this excitement, having designed several such systems ourselves.

A careful investigation of this trend, however, reveals a flaw. A system achieves its security goals when extensions maintain its corresponding *system invariants*. However, we observe that these invariants are upheld only in a restrictive subset of Rust, rather than universally holding in real Rust code. This means that Rust-based systems achieve their goals in that restrictive subset, but cannot confidently extend their guarantees to much practical Rust code. For example, RedLeaf [35] is an extensible Rust-based OS whose goal is to isolate kernel subsystems. RedLeaf's isolation mechanism restricts how data is shared between subsystems: a subsystem either takes exclusive ownership of data or receives read-only access via an immutable reference. The invariant at the heart of this mechanism is that a subsystem cannot modify data it only has access to via an immutable reference. This invariant holds in a subset of the Rust language, but practical Rust code routinely violates it, e.g., in `RefCell`, `Mutex`, or third-party libraries. A subsystem that idiomatically uses such primitives may inadvertently violate RedLeaf's invariant, break its isolation mechanism, and cause unrecoverable corruption of critical kernel state. We show that many Rust-based systems have similar invariants, and that the mechanisms they provide to check these invariants, e.g., lints and marker traits, are simultaneously too restrictive and too brittle.

The root cause of this flaw is a mismatch between what research systems need and what Rust actually promises in practice. While this mismatch is connected to unsafe Rust code, it is distinct from the widely-studied challenge of finding unsafe code that violates memory safety or causes data races [25, 31, 26, 29, 14, 2, 24, 41]. It is distinct because the mismatch also manifests in memory-safe code that meets all of Rust's requirements; such code is the focus of this paper. We call on the research community to build tools that reason about this mismatch directly. Our preliminary results show a proof of concept with one such tool, demonstrating that the community can build Rust-based systems and extend their guarantees to practical Rust code with confidence.

2 The Gap Between Rust and Systems

Our focus is Rust-based systems that enforce security guarantees over *extensions*. An extension consists of user-provided Rust code that the system loads and executes, e.g., a kernel module, network/serverless function, or application HTTP endpoint. Extensions often use dependencies like Rust's standard library, or third-party serialization or logging libraries.

2.1 What Rust-Based Systems Need

A system achieves its guarantees by relying on *system invariants* that must hold over all extension code. Rust-based

PLOS '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2877-8/26/09
https://doi.org/10.1145/3831586.3838150

systems rely on the Rust language and compiler to help them enforce these invariants, and derive their novelty from this.

For example, RedLeaf [35] isolates kernel subsystems so that the kernel can recover when a subsystem crashes. It does so by restricting subsystems to only sharing data that is allocated on a designated shared heap. RedLeaf relies on a crucial invariant: when a subsystem passes an immutable reference to shared heap data, the receiving subsystem may only read *but not modify* that data. This invariant allows RedLeaf to recover from a crash in a subsystem, since the crash can only have corrupted the crashed subsystem's own data. Similarly, several authors of this paper created Sesame [13], a system that enforces privacy policies in Rust web applications by wrapping sensitive data in policy containers (`PCons`). Sesame's invariant is that application code can never directly access the data inside a `PCon`, such that applications can only manipulate that data via Sesame's API.

These systems bootstrap their invariants from properties that the Rust compiler statically checks. RedLeaf depends on the Rust compiler to reject code that mutates data on the shared heap via an immutable reference; Sesame depends on it to enforce the encapsulation of the private members of the `PCon` structure. When these invariants hold throughout extension code, both systems meet their higher-level goals.

However, the compiler relaxes some of its usual checks for code marked with the `unsafe` keyword, to the extent that system invariants are no longer guaranteed in such code. A natural approach many systems take is to put unsafe code out of scope. For example, kernel extensions in RedLeaf are "restricted to the safe subset of Rust" [35]. The problem is that such limitations are ambiguous and may restrict the permissible code so strictly that the system becomes impractical.

2.2 What Rust Actually Promises

Rust's central (and only) promise is that valid Rust programs never exhibit undefined behavior [6]. This is a promise that implies memory safety and the absence of data races. Rust achieves this via a two-tier approach. The Rust compiler statically enforces the absence of undefined behavior for the bulk of a Rust program, but it suspends this enforcement in `unsafe` blocks. There, the responsibility to ensure that there is no undefined behavior shifts from the compiler to the developer—possibly assisted by Miri [25], Kani [26], or other tools [31, 24]. This balances safety and expressivity.

Developers may enclose `unsafe` blocks without undefined behavior behind a *safe interface*. From the perspective of a caller, safe interfaces that contain `unsafe` blocks and Rust functions that lack unsafe code are indistinguishable without reviewing their source code. This is critical for modularity.

Following standard Rust terminology, we use *unsafe* to refer to the syntactic presence of the `unsafe` keyword, *unsound* to refer to code that may exhibit undefined behavior, and use *safe* and *sound* as their respective counterparts.

	Soundness	**Incidental Properties**
safe	Compiler-enforced	
unsafe	Developer must manually enforce	Developer allowed to violate

Figure 1. The compiler enforces soundness and incidental properties in safe code. Rust obligates developers to manually uphold soundness in unsafe code, e.g., using Miri [25]. But Rust allows idiomatic, widely-accepted unsafe code to intentionally violate incidental properties (shaded; bottom right).

Incidental Properties. Safe Rust programs uphold soundness, but also additional *incidental properties* (our term, rather than standard Rust terminology). We define a property as incidental if it is upheld in entirely safe code, but violating it does not cause undefined behavior. Three examples are:

1. **Encapsulation:** private fields of a structure can only be accessed from inside the defining module.
2. **Lifetimes as aliases:** aliases of a reference of type `&'a T` can be soundly approximated from the lifetime `'a`.
3. **Immutability of references:** given an immutable reference of type `&'a T`, all data accessible through the reference will never mutate during the lifetime `'a`.

In safe Rust, the compiler directly enforces encapsulation; the other two properties follow from the compiler's borrow checker rules, e.g., aliasing XOR mutability [28]. Incidental properties imply a variety of useful system invariants, making them an attractive foundation for Rust-based systems.

Sound unsafe code can violate incidental properties to provide useful functionality or improve performance. For example, Rust mutexes use `unsafe` behind a safe interface and intentionally violate the immutability of references, as their purpose is to allow mutating aliased data referenced by multiple threads. Mutexes are sound, since they use atomic operations and are carefully designed to prevent undefined behavior and data races. Unsafe code that does not introduce undefined behavior is sound, regardless of whether or not it upholds any incidental properties (Figure 1).

2.3 Rust and Rust-Based Systems are Mismatched

Soundness and incidental properties have one thing in common: the compiler enforces both in globally safe code. However, **while Rust only mandates that unsafe code maintains soundness, Rust-based systems often depend on incidental properties that sound Rust code is under no obligation to uphold.** For Rust-based systems, this makes safe interfaces particularly dangerous: while a safe interface is expected to be just as sound as its compiler-checked counterpart, it may not necessarily uphold the same incidental properties. Consequently, systems cannot assume that an extension that invokes library code will uphold incidental properties or any system invariants built on top of them.

Between a rock and an unsafe place. Given that sound unsafe code can violate incidental properties, systems today have two unsatisfactory options for reasoning about unsafe code in extensions.

The first option is to entirely prohibit an extension from invoking any `unsafe` blocks, either directly or in its dependency tree. We refer to this as *globally safe* code. Rust-based systems can always achieve their goals over a globally safe extension, since the compiler enforces all incidental properties. In practice, however, globally safe Rust code is extremely rare, since the vast majority of Rust programs invoke `unsafe` code in their dependencies.[1] Thus, a system built upon this assumption may be so restrictive as to be impractical.

The second option is to allow unsafe code, but require that it must be reasoned about separately, i.e., outsource the responsibility. Many systems adopt this approach: some assume `unsafe` "is confined to trusted libraries," [3, 8] and others use lints to forbid unsafe code in the top-level extension, but not in library code [36, 20]. We refer to this as *locally safe* code. §3 shows that this approach is too brittle: locally safe extensions may nonetheless violate important incidental properties and inadvertently break the guarantees systems seek to provide, even when the extensions are sound, idiomatic, and only use widely-trusted libraries such as `std`.

The Gap. Developers have access to a variety of conventions, e.g., safety comments [41], and tools, e.g., Miri, to ensure their unsafe code is sound, and soundness is the criterion that determines whether a function may be a safe interface. By contrast, there are no "incidental property" interfaces, and developers have no conventions or tools to reason about them. In order to bridge the gap between Rust and systems, we propose developing tools that reason about incidental properties directly (§4).

3 Case Studies

We now show examples of how sound code breaks the guarantees of Rust-based systems, including in our own work.

Sesame: Leaking Private Data. Sesame is an end-to-end privacy policy enforcement system for Rust web applications, developed by some of this paper's authors. Application developers use Sesame's API to define privacy policies and associate them with data, e.g., access control or k-anonymity for sensitive data or aggregates. Sesame ensures policies remain attached to data throughout execution and checks policies before allowing the application to externalize data. Sesame's central abstraction is the *policy container* (`PCon<T,P>`), which holds two private members: sensitive data of type T and a policy of type P. If encapsulation (an incidental property) holds, applications can never directly access `PCon`-protected data.

[1] A study of the 500 most-used Rust crates found that more than half call into unsafe code, even after excluding calls into the standard library [16].

```
1  #[derive(Serialize)]
2  pub struct SubmitForm {
3      username: String,
4      #[serialize_any(String, NoLoggingPolicy)]
5      answer: PCon<String, NoLoggingPolicy>,
6  }
7  // Simplified WebSubmit endpoint.
8  fn submit_homework(request: SubmitForm) {
9    // Logging sensitive data violates privacy policy.
10   println!("{}", serde_json::serialize(&request));
11   SesameDB.insert(request.username, request.answer);
12 }
```

Figure 2. Application code that inadvertently leaks sensitive data despite its Sesame policy (line 10). The library macro (line 4) uses `serde-remote` and `mem::transmute` to serialize `answer`, violating encapsulation of the `PCon` type.

Encapsulation is merely an incidental property, and a locally-safe application may unwittingly break it. This could happen via unsafe code hidden in a serialization or logging library, causing an inadvertent data leak. Consider an application that wants to serialize instances of a type defined by a third-party dependency. If the third-party developers fail to implement the `Debug` or `Serialize` trait for that type, the application code cannot log or serialize it directly [27]. To work around this common pain point, `serde` provides the *remote* feature [43]: application developers write a custom "remote type" that tells `serde` how to access the fields of the third-party type, acting as a "surrogate" for serialization. The remote type usually invokes the third-party type's public getters. But using unsafe code it can also access fields in types without getters, e.g., `PCon`. We built such a library using `serde-remote` and `std::mem::transmute` to serialize any third-party type regardless of the visibility of its fields.

Figure 2 shows how a developer may use this library to serialize a `SubmitForm` struct containing a `PCon` using the `serialize_any` macro (line 4), causing a data leak. A developer may do this out of confusion or oversight, exactly the kind of mistake Sesame aims to protect against. Once `SubmitForm` implements `Serialize`, the leak of sensitive data (line 10) looks like a standard serialization call.

We confirmed that `serialize_any` allows Sesame's WebSubmit example application to log sensitive data, and we confirmed the code is sound with Miri. We also replicated the leak using another serialization library [34], and we believe other libraries may implement similar patterns e.g., for stack inspection or rewinding after panics.

When we designed Sesame, we—fairly late in the game—realized the risk of unsafe code in dependencies and included a protection mechanism: rather than storing the protected data itself, `PCons` store a pointer to the sensitive data XORed with a private constant. With this mechanism in place, serialization shows only the obfuscated address. However, the

System	Goal	System Scope	Incidental Property	Violation
Sesame [13]	policy enforcement	no "malicious" `unsafe` code	encapsulation	data leaks, policy violations
RedLeaf [35]	kernel subsystem isolation	"restricted to safe Rust"	immutability of references	no isolation between subsystems, kernel cannot recover after crashes
Flowistry [11]	analyzing information flow	"safe subset of Rust programs"	lifetimes + immutability	misses existing flows (unsound)
Netbricks [36]	packet isolation between NFs	no "`unsafe` pointer arithmetic"	immutability of references	packet modified after NF sent, memory bugs
Cocoon [30]	information flow control	"absence of `unsafe` Rust"	encapsulation + immutability	data leakage
Boucher et al. [8]	serverless function isolation	"safe Rust"	"language isolation"	data leaks between serverless functions
Rex [20]	safe kernel extensions	"safe Rust with selected features"	lifetimes + "resource safety"	kernel corruptions and crashes

Figure 3. Example Rust-based systems with their incidental properties and confirmed violations with *locally* safe extensions.

pointer obfuscation comes at a 2× overhead in Sesame's microbenchmarks, and fails to guard against other guarantee-breaking patterns, e.g., code that swaps the policy with a more permissive one. Tooling that ensures the absence of encapsulation-breaking operations on `PCons` would let Sesame avoid this overhead and also guard against other violations that circumvent `PCon` protections.

RedLeaf: Violating Kernel Subsystem Isolation. RedLeaf is an operating system that enforces crash isolation between kernel subsystems. RedLeaf relies on *immutability of references* when rolling back crashed kernel subsystems. RedLeaf only allows kernel subsystems to exchange data through a special shared heap via immutable references or by transferring ownership. When a subsystem crashes, the rest of the kernel only needs to clean up data the crashed subsystem owns, since that is the only data it could have modified.

However, immutability of references is an incidental property, and sound Rust may violate it. RedLeaf only has partial protections against this: it uses an auto trait, `RRefable`, to restrict the types that can be shared between subsystems using the shared heap. Specifically, `RRefable` disallows sharing raw pointers, but does not exclude interior mutable types like `RefCell`. Thus, shared heap data can contain an interior mutable type, and a subsystem that holds an immutable reference to that data can mutate it. If the subsystem crashes after such a mutation, RedLeaf cannot guarantee fault isolation.

We implemented a locally-safe, pluggable scheduler as a RedLeaf kernel subsystem. RedLeaf cannot guarantee isolation of that scheduler even though it meets RedLeaf's shared heap restrictions. The scheduler interface, which selects the next thread to run, takes the list of metadata for the queued threads as input. Thread metadata is shared between the scheduler and the kernel because the base kernel must also be able to modify it, e.g., to create and destroy threads. We allocate it on the shared heap, and we wrap it in a `RefCell`, allowing both the kernel and scheduler to modify it. RedLeaf's original scheduler uses `RefCell` in a similar manner, although that scheduler is built into the core kernel, part of the trusted code base, rather than as an isolated subsystem. If the scheduler crashes after modifying the metadata via interior mutation—e.g., removing the thread metadata from the queue—the kernel fails to roll back the change. This leaves the kernel in an inconsistent state. In the example, it permanently loses the dequeued thread's metadata, so that thread never runs again.

At the heart of the issue is Rust's core interior mutability primitive, `UnsafeCell`. Its `get()` and `raw_get()` methods are the only sound ways to interior mutate a type [46]. RedLeaf could disallow allocating `UnsafeCell` on the shared heap, which would prohibit `RefCell` and all other interior mutable types. But this would restrict expressivity: developers could no longer use common data structures that contain interior mutable types, e.g., doubly-linked lists or ring buffers, even when subsystems never actually interior mutate them. A less restrictive approach would permit storing `UnsafeCell` on the shared heap, but disallow invoking the `get()` and `raw_get()` methods within a subsystem. This requires tooling beyond Rust's type system and auto traits.

Other Systems. We confirmed that sound, locally-safe extensions can break the guarantees of the other systems listed in Figure 3; we only outline these violations. Code that violates encapsulation or immutability of references may leak high-security data to low-security sinks in Cocoon [30]. Similarly, code that extends lifetimes, e.g., using `mem::transmute`, may cause Flowistry [11] to miss information flows. Netbricks [36] and Boucher et al. [8] aim to isolate network and serverless functions, but merely locally-safe

code may leak references or reuse addresses across function boundaries, leaking packets or sensitive data. Finally, Rex [20] is a Rust-based eBPF alternative. To prevent resource leaks, Rex uses a lint to forbid `unsafe` code and leak-prone functions like `mem::forget`. However, the lint only checks the top-level extension, so dependencies may still invoke `mem::forget` and silently leak resources.

4 Rust-Based Systems Need Better Tooling

Practical systems can neither globally disallow nor ignore unsafe code. We advocate for better tools that confront unsafe code directly and reason about system invariants in its presence.

Property Preservation. Relative to a given incidental property, a piece of Rust code must be one of the following:

- **Unsound:** Has undefined behavior and no guarantees.
- **Sound but not property-preserving:** Has no undefined behavior, but violates the incidental property in question.
- **Property-preserving:** Has no undefined behavior and upholds the incidental property in question.

The Rust compiler ensures that globally safe code preserves all incidental properties, but even unsafe code may still preserve a given property in practice. Each incidental property can only be violated by a subset of sound unsafe patterns. For example, `UnsafeCell::get` violates immutability of references but not encapsulation, while `transmute` and pointer arithmetic may violate the latter but not the former.

This bodes well for Rust-based systems: a system can still provide guarantees over an extension that invokes unsafe code. For example, a kernel subsystem that is sound and respects immutability of references meets RedLeaf's requirements, regardless of whether it invokes other unsafe code.

From Incidental Properties to System Invariants. The precise invariants of many Rust-based systems can be expressed as a *refinement* of an incidental property. This refinement scopes the property to specific entities in the extension, rather than requiring it universally. For example, Sesame only requires encapsulation over `PCons` and is unaffected by applications that break encapsulation of other unrelated types. Similarly, RedLeaf requires immutability of references for data on the shared heap and is compatible with a subsystem that interior mutates its own local data, e.g., because it contains a doubly-linked list. Furthermore, we observe that many Rust-based systems reuse similar incidental properties (Figure 3). So, tools can provide a "library" of incidental properties, each mapped to its corresponding unsafe patterns, that systems can refer to when describing their invariants.

Threat Model. Most Rust-based systems fall under two threat models. The first targets *cooperative extensions*, where the system trusts extension developers to apply steps to ensure their extension meets the system invariants. This includes running compilers or tools that check or transform their extensions and auditing their outputs diligently. This model aligns with Sesame, RedLeaf, and other systems [20, 30].

The second targets *adversarial extensions*, where the system itself must ensure that extensions from untrusted developers meet the system invariants, e.g., before loading or running an extension. Here, the system needs tooling with near-perfect precision, as it cannot rely on humans to resolve ambiguities. The system may also need to detect undefined behavior with orthogonal tooling. A smaller subset of Rust-based systems requires this model, e.g., when extensions are provided by untrusted tenants [36, 8].

Paths Forward. Existing techniques are insufficient, but provide a foundation for further research. *Ad-hoc* mechanisms, e.g., lints [36, 20] or runtime mechanisms like Sesame's obfuscated pointers, have serious weaknesses (§3). Future systems may design more reliable custom mechanisms by leaning on techniques that require explicitly naming the incidental properties that a system depends on. *Marker traits* stand out: the unstable `Freeze` [45, 40] marker trait is a leading contender in the Rust community for dealing with interior mutability [38, 15]. `Freeze` is only implemented for types that do not contain `UnsafeCell`, so systems can use `Freeze` trait bounds to prohibit interior mutability. But `Freeze` restricts types, rather than behavior. It prohibits passing a type containing an `UnsafeCell` even if the code never performs interior mutation, motivating more precise alternatives.

Some existing techniques reason about the soundness of unsafe code, and novel approaches may extend them to system invariants. *Static analysis* and *auditing* tools [2, 47] help developers find unsound code, including in their dependencies. We propose adapting this approach to help developers find violations of incidental properties while filtering out the bulk of irrelevant code to reduce review burden. *Formal methods* [31, 26, 1, 14, 24] enable reasoning about the soundness of unsafe code with strong guarantees, but require significant developer labor and novel research to reason about incidental properties and how they interact with system invariants. For example, Verus [31] supports reasoning about `UnsafeCell` via ghost state, but not `transmute`. Finally, *runtime mechanisms*, e.g., hardware assistance or SFI techniques, can quarantine unsafe code [4], protect safe objects from it [33], and recover from [17] or prevent [42, 39] undefined behavior. With further research, it may be possible to aim similar mechanisms at incidental property preservation.

To be practical, any approach must be lightweight without being too restrictive. This requires reasoning at the granularity of system invariants and incorporating system-specific information to refine and scope the incidental properties.

5 Static Analysis for System Invariants

We investigate one approach, *invariant-targeted* static analysis, to help extension developers audit their code for violations of system invariants. Targeted auditing benefits from the flexibility of human inspection, but reduces auditor effort by filtering the code that requires review. To give developers confidence that an extension maintains the system's guarantees, the auditing tool must be exhaustive, surfacing all potential violations of the invariant. The challenge is to manage audit burden by distinguishing between unsafe code that is benign and unsafe code that violates the system invariant.

We designed a prototype invariant-targeted auditing tool, Sniffer, and sketch how system and extension developers use Sniffer to surface guarantee-breaking code.

Invariant Specifications. Sniffer leverages the fact that only a subset of sound unsafe code patterns violate each incidental property. We encoded checks for encapsulation and immutability of references into our prototype via all unsafe patterns that may violate them. System developers specify their system invariants as a combination of what incidental properties they care about, and if applicable, the system-provided types over which the property must hold.

Sniffer approximates the code base of an extension as a call graph, and checks these invariants as assertions over the reachable function bodies. It identifies reachable paths from annotated entities to any unsafe operations that may violate the specified incidental property. The absence of such paths indicates that the extension upholds the system invariant.

Using Sniffer. System developers first provide an invariant specification for their system. For example, they specify that all instances of Sesame's `PCon` type must remain encapsulated, or that data placed on RedLeaf's shared heap must not be interior mutated. Systems reuse this specification for all extensions, e.g., via a build script that runs Sniffer when compiling an extension. They may also provide extension developers with instructions to annotate specific top-level functions to use as entry points for analysis, e.g., the definitions of the subsystem interface in RedLeaf.

Next, extension developers run Sniffer over their extension. Sniffer flags call chains that lead to potential violations of the invariant specification. In our examples, Sniffer flags the invocation of the `serialize_any` macro as leading to a call to `mem::transmute` with a `PCon` argument, and the call in the scheduler subsystem to `RefCell::borrow_mut`, which invokes `UnsafeCell::get`.

Finally, extension developers audit each flagged location and call chain. If the developers confirm it is a genuine violation of the system invariant, they revise their code to avoid it. Otherwise, they exempt the flagged code, e.g., if the reported flow is spurious or the code is benign in context.

Implementation. Sniffer is a compiler plugin; it performs analysis on the mid-level intermediate representation (MIR) that the Rust compiler generates. The first step in a Sniffer analysis pass is to construct a *monomorphized call graph* (MCG) of the input extension and its dependency tree. The MCG instantiates generic functions and it soundly over-approximates dynamic dispatch. To build the MCG, Sniffer extends the Rust compiler's built-in reachability analysis to record, at each call site, the metadata needed to soundly resolve dynamic dispatch, e.g., the trait and method signatures of `dyn` calls, and the function signatures of function pointers.

Sniffer then searches the MCG for possible violations of the invariant specification. Sniffer walks each reachable function body with a custom MIR visitor that encodes the patterns of unsafe code that are property-violating. To filter the analysis results with additional system-specific context, Sniffer can inspect the type of the arguments passed to the flagged operation, and limit its results to the call graphs rooted at developer-annotated entry points.

Sniffer is sound: any reachable code that actually violates the property is flagged. However, Sniffer may also flag code that does not violate the property (a *false positive*) due to either the MCG's over-approximation of dynamic dispatch or the heuristic nature of the per-property patterns, e.g., `mem::transmute` on a `PCon` that does not result in a leak.

Future Work. Our prototype supports annotating types and top-level functions with the incidental properties that should apply to them. Future work on Sniffer will support more expressive annotation patterns, e.g., specific variables or all implementations of a particular trait, and integrate information flow analysis at the granularity of variables [11]. This would provide system developers more control over the analysis of extension code, and enable Sniffer to filter out additional false positives, e.g., by differentiating between a `RefCell` that is internal to a RedLeaf subsystem versus passed to the subsystem via the shared heap.

5.1 Preliminary Evaluation

We evaluate our prototype invariant-auditing system with respect to three questions: *(Q1)* Does Sniffer detect guarantee-breaking code? *(Q2)* How much effort does Sniffer impose on extension developers? *(Q3)* How effectively does Sniffer's design reduce false positives compared to existing auditing approaches? A good result for Sniffer would flag the guarantee-breaking code in our examples with only a modest audit burden (i.e., false positives) for the extension developer.

Setup. We apply Sniffer to Sesame's full WebSubmit homework submission endpoint, modified with the call to the logging library described in §3. The RedLeaf code base is incompatible with the Rust compiler versions Sniffer supports, so we extract small compatible excerpts from our RedLeaf scheduler case study and apply Sniffer to them. These excerpts place a `RefCell` on the shared heap directly and nested in other structs, and mutate it directly, in transitive helpers, and after storing and retrieving from collections.

Q1: Detection. Sniffer successfully detects both guarantee-breaking code examples: the `mem::transmute` call inside the logging library that leaks `PCon`-protected data in Sesame, and the `UnsafeCell::get` that allows interior mutation of shared-heap data in RedLeaf.

Q2: Developer Burden. We focus on the Sesame case study.[2] Sniffer flags twelve calls as potential violations. One is the genuine encapsulation-breaking `transmute` described in §3, which occurs inside a macro expansion at the interface of the remote type with `serde`. A developer would likely miss this in an unguided, manual audit, unless they explicitly instructed their IDE or the compiler to expand all macros. The remaining eleven are false positives. Five involve pointer arithmetic, but are harmless in context, e.g., iterators that use pointer arithmetic to advance through elements of a `Vec<PCon>`. The other six false positives are due to over-approximation of dynamic dispatch in Sniffer's call graph. To dismiss them, the developer can confirm that no `PCon` object is ever passed to the flagged function via a mechanical check on the call site.

Q3: Design Effectiveness. To evaluate the impact of Sniffer's design choices, we implement two simpler alternatives and compare false-positive rates. A purely syntactic approach [9, 10] that flags any `unsafe` block in the code base or its dependencies flags 33,735 regions for Sesame's WebSubmit application. Only counting unsafe code that is *reachable* from this endpoint yields 9,982 regions. Combining reachability with Sniffer's invariant-targeted analysis, e.g., only flows from a `PCon` to pointer arithmetic and `mem::transmute`, further reduces this to twelve regions. This validates that Sniffer's invariant-targeting design reduces false positives compared to syntactic and reachability-only baselines.

We also compare to `cargo-scan` [47], an interactive auditing tool that identifies potentially dangerous code (e.g., `unsafe` operations and system calls) as effects and tracks them across crate boundaries. It reports a total of 43,761 items to audit when we applied it to each dependency individually. Its more precise "cross-package audit" mode cannot be meaningfully compared to Sniffer: its AST-level call graph approximation is unsound as it does not resolve trait implementations and dynamic dispatch into other crates.

6 Discussion

Complementary Mechanisms. In the adversarial threat model (§4), the system itself must check the extensions before loading them, e.g., as in NetBricks [36] and Boucher et al. [8]. We envision combining Sniffer with runtime mechanism and formal methods approaches in order to compensate for the absence of a trusted auditor. Sniffer highlights the select regions that endanger system guarantees, e.g., twelve rather than 9,982 `unsafe` blocks in the Sesame example (§5.1). Sniffer could guide formal approaches by emitting property-preservation obligations about these individual `unsafe` regions or flows, turning whole-extension verification into a set of local proofs. Sniffer could also employ runtime mechanisms, e.g., Omniglot-style type checks [42], software fault isolation, or hardware isolation as a fallback for specific flows that Sniffer cannot dismiss statically. Sniffer could apply the cheapest appropriate mechanism based on the property specification and the flagged region. For example, Sniffer could instrument a runtime check before a flagged `transmute` to reject calls only when the argument contains a `PCon`, or use MPK [19] to protect shared-heap data in RedLeaf as read-only for non-owner extensions, trapping flagged `UnsafeCell::get` calls.

Beyond Incidental Properties. The incidental nature of the properties systems rely on and its consequences for Rust-based systems is an instance of a broader issue: namely, what invariants are reasonable to rely on for achieving soundness [18] and beyond. This is an active discussion in the Rust community under many names (e.g., "language" and "library" invariants [22]). Future work could adopt Sniffer-like approaches to identify violations of such invariants, e.g., safety invariants for custom types [23], or to even high-level application invariants such as panic freedom. Current approaches to panic freedom rely on either syntactic reasoning [44] or complex formal verification techniques [12, 31]. Sniffer can provide a middle ground that considers semantic information but requires less effort to set up and use.

7 Conclusion

There is a serious mismatch between the properties that Rust-based systems assume hold in sound Rust code, and the reality that Rust code often violates these *incidental properties*. We define more precise vocabulary to describe the property-preservation requirements of a system, and we demonstrate how a sound extension that fails to meet these requirements will violate system guarantees. We propose static analysis tooling to aid invariant-targeted auditing as a flexible, expressive means to maintain the guarantees of Rust-based systems in the presence of unsafe code.

Acknowledgments

We thank Olivia Brode-Roger, Nicole Pauda, Leon Schuermann, and Carolyn Zech for their helpful feedback on earlier drafts. Annika Singh investigated possible property violations in Rex as part of her master's project. We also thank the anonymous reviewers from HotOS 2025 and PLOS 2025 and 2026, whose feedback helped us much improve this paper.

This research was supported by NSF awards CNS-2045170 and DGE-233562, by two Google ML and Systems Junior Faculty Awards, a Microsoft Grant for Customer Experience Innovation, and an Amazon Research Award.

[2] Sniffer produces no false positives on the RedLeaf case study, but these isolated excerpts do not capture the full complexity of a practical subsystem.

References

[1] Vytautas Astrauskas, Aurel Bílỳ, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J Summers. "The Prusti Project: Formal Verification for Rust". In: *NASA Formal Methods Symposium*. Springer. 2022, pp. 88–108. URL: https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5 (cit. on p. 5).

[2] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. "Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 84–99. ISBN: 9781450387095. DOI: 10.1145/3477132.3483570. URL: https://doi.org/10.1145/3477132.3483570 (cit. on pp. 1, 5).

[3] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. "System Programming in Rust: Beyond Safety". In: *SIGOPS Oper. Syst. Rev.* 51.1 (Sept. 2017), pp. 94–99. ISSN: 0163-5980. DOI: 10.1145/3139645.3139660. URL: https://doi.org/10.1145/3139645.3139660 (cit. on pp. 1, 3).

[4] Inyoung Bang, Martin Kayondo, Hyungon Moon, and Yunheung Paek. "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, 2023, pp. 6947–6964. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/bang (cit. on p. 5).

[5] Vincent Beardsley, Chris Xiong, Ada Lamba, and Michael D Bond. "Carapace: Static–Dynamic Information Flow Control in Rust". In: *Proceedings of the ACM on Programming Languages* 9.OOPSLA1 (2025), pp. 364–392 (cit. on p. 1).

[6] *Behavior considered undefined*. 2025. URL: https://doc.rust-lang.org/reference/behavior-considered-undefined.html (cit. on p. 2).

[7] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. "Theseus: an Experiment in Operating System Structure and State Management". In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Nov. 2020, pp. 1–19. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/boos (cit. on p. 1).

[8] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. "Putting the "Micro" Back in Microservice". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 645–650. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/boucher (cit. on pp. 1, 3–5, 7).

[9] *cargo-geiger*. Accessed: 2026-05-23. URL: https://github.com/geiger-rs/cargo-geiger (cit. on p. 7).

[10] *count-unsafe*. Accessed: 2026-07-31. URL: https://crates.io/crates/count-unsafe (cit. on p. 7).

[11] Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. "Modular Information Flow through Ownership". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 1–14. ISBN: 9781450392655. DOI: 10.1145/3519939.3523445. URL: https://doi.org/10.1145/3519939.3523445 (cit. on pp. 1, 4, 6).

[12] Cryspen. *Panic Freedom*. https://hax.cryspen.com/manual/lean/tutorial/panic-freedom/. hax tutorial documentation, accessed 2026-05-14. 2026 (cit. on p. 7).

[13] Kinan Dak Albab, Artem Agvanian, Allen Aby, Corinn Tiffany, Alexander Portland, Sarah Ridley, and Malte Schwarzkopf. "Sesame: Practical End-to-End Privacy Compliance with Policy Containers and Privacy Regions". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 2024, pp. 709–725 (cit. on pp. 1, 2, 4).

[14] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. "Creusot: a foundry for the deductive verification of rust programs". In: *International Conference on Formal Engineering Methods*. Springer. 2022, pp. 90–105 (cit. on pp. 1, 5).

[15] *E0492: borrow of an interior mutable value may end up in the final value during const eval when no inner mutability is involved*. Accessed: Jan. 2025. URL: https://github.com/rust-lang/rust/issues/121250 (cit. on p. 5).

[16] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. "Is rust used safely by software developers?" In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. ACM, June 2020. DOI: 10.1145/3377811.3380413. URL: http://dx.doi.org/10.1145/3377811.3380413 (cit. on p. 3).

[17] Merve Gülmez, Thomas Nyman, Christoph Baumann, and Jan Tobias Mühlberg. "Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust". In: *2023 IEEE Secure Development Conference (SecDev)*. IEEE, 2023, pp. 54–66. DOI: 10.1109/SecDev56634.2023.00020. URL: https://doi.org/10.1109/SecDev56634.2023.00020 (cit. on p. 5).

[18] ia0 and Rust Internals contributors. *Conditions for Unsafe Code to Rely on Correctness*. https://internals.rust-lang.org/t/conditions-for-unsafe-code-to-rely-on-correctness/23995. Rust Internals forum discussion, accessed 2026-06-06. Feb. 2026 (cit. on p. 7).

[19] Intel Corporation. *Intel(R) 64 and IA-32 Architectures Software Developer's Manual*. 2016. URL: https://software.intel.com/en-us/articles/intel-sdm (cit. on p. 7).

[20] Jinghao Jia, Ruowen Qin, Milo Craun, Egor Lukiyanov, Ayush Bansal, Minh Phan, Michael V Le, Hubertus Franke, Hani Jamjoom, Tianyin Xu, et al. "Rex: Closing the language-verifier gap with safe and usable kernel extensions". In: *Proceedings of the 2025 USENIX Annual Technical Conference (ATC)*. July 2025, pp. 325–342 (cit. on pp. 1, 3–5).

[21] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V. Le, and Tianyin Xu. "Kernel extension verification is untenable". In: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. HotOS '23. Providence, RI, USA: Association for Computing Machinery, 2023, pp. 150–157. ISBN: 9798400701955. DOI: 10.1145/3593856.3595892. URL: https://doi.org/10.1145/3593856.3595892 (cit. on p. 1).

[22] Ralf Jung. *Stabilize Having the Concept of "Validity Invariant" and "Safety Invariant"? Under Which Name?* GitHub issue #539 in the Rust Unsafe Code Guidelines repository, accessed 2026-06-05. Oct. 2024. URL: https://github.com/rust-lang/unsafe-code-guidelines/issues/539 (cit. on p. 7).

[23] Ralf Jung. *Two Kinds of Invariants: Safety and Validity*. Accessed: 2026-06-05. Aug. 2018. URL: https://www.ralfj.de/blog/2018/08/22/two-kinds-of-invariants.html (cit. on p. 7).

[24] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the foundations of the Rust programming language". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–34 (cit. on pp. 1, 2, 5).

[25] Ralf Jung, Benjamin Kimock, Christian Poveda, Eduardo Sánchez Muñoz, Oli Scherer, and Qian Wang. "Miri: Practical Undefined Behavior Detection for Rust". In: *Proc. ACM Program. Lang.* 10.POPL (Jan. 2026). DOI: 10.1145/3776690. URL: https://doi.org/10.1145/3776690 (cit. on pp. 1, 2).

[26] *Kani*. Accessed: Jan. 2025. URL: https://github.com/model-checking/kani (cit. on pp. 1, 2, 5).

[27] Steve Klabnik, Carol Nichols, and Chris Krycho. *The Rust Programming Language: Implementing a Trait on a Type*. Accessed: June 2, 2026. URL: %5Curl%7Bhttps://doc.rust-lang.org/book/ch10-02-traits.html#implementing-a-trait-on-a-type%7D (cit. on p. 3).

[28] Steve Klabnik, Carol Nichols, and Chris Krycho. *The Rust Programming Language: References and Borrowing*. Accessed: May 13, 2026. URL: %5Curl%7Bhttps://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html%7D (cit. on p. 2).

[29] Rahul Kumar, Celina Val, Felipe Monteiro, Michael Tautschnig, Zyad Hassan, Qinheping Hu, Adrian Palacios, Remi Delmas, Jaisurya Nanduri, Felix Klock, Justus Adam, Carolyn Zech, and Artem Agvanian. "Verifying the Rust Standard Library". In: *16th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*. Prague, Czech Republic, Oct. 2024. URL: https://www.soundandcomplete.org/vstte2024/vstte2024-invited.pdf (cit. on p. 1).

[30] Ada Lamba, Max Taylor, Vincent Beardsley, Jacob Bambeck, Michael D. Bond, and Zhiqiang Lin. "Cocoon: Static Information Flow Control in Rust". In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (Apr. 2024). DOI: 10.1145/3649817. URL: https://doi.org/10.1145/3649817 (cit. on pp. 1, 4, 5).

[31] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. "Verus: Verifying rust programs using linear ghost types". In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA1 (2023), pp. 286–315 (cit. on pp. 1, 2, 5, 7).

[32] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. "Multiprogramming a 64kB Computer Safely and Efficiently". In: *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, 2017, pp. 234–251. ISBN: 9781450350853. DOI: 10.1145/3132747.3132786. URL: https://doi.org/10.1145/3132747.3132786 (cit. on p. 1).

[33] Shaowen Li and Hiroyuiki Sato. "SBD: Securing Safe Rust Automatically From Unsafe Rust". In: *Science of Computer Programming* 243 (2025), p. 103281. ISSN: 0167-6423. DOI: 10.1016/j.scico.2025.103281. URL: https://doi.org/10.1016/j.scico.2025.103281 (cit. on p. 5).

[34] Frank McSherry. *Abomonation: A high performance and very unsafe serialization library*. Accessed: Jan. 2025. URL: https://crates.io/crates/abomonation (cit. on p. 3).

[35] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. "RedLeaf: Isolation and Communication in a Safe Operating System". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 21–39. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram (cit. on pp. 1, 2, 4).

[36] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. "NetBricks: Taking the V out of NFV". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 203–216. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda (cit. on pp. 1, 3–5, 7).

[37] Yuke Peng, Hongliang Tian, Zhang Junyang, Ruihan Li, Chengjun Chen, Jianfeng Jiang, Jinyi Xian, Xiaolin

Wang, Chenren Xu, Diyu Zhou, et al. "Asterinas: A Linux ABI-Compatible, Rust-Based Framekernel OS with a Small and Sound TCB". In: *Proceedings of the 2025 USENIX Annual Technical Conference (ATC)*. July 2025 (cit. on p. 1).

[38] Reddit r/rust contributors. *Why immutability is important, and why many don't care about it.* https://www.reddit.com/r/rust/comments/25ul91/why_immutability_is_important_and_why_many_dont/. Accessed: 2026-05-24. 2014 (cit. on p. 5).

[39] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. "Keeping Safe Rust Safe with Galeed". In: *Proceedings of the 37th Annual Computer Security Applications Conference*. ACSAC '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 824–836. ISBN: 9781450385794. DOI: 10.1145/3485832.3485903. URL: https://doi.org/10.1145/3485832.3485903 (cit. on p. 5).

[40] Rust Lang Team. *Design meeting 2024-07-24: Freeze in bounds.* https://hackmd.io/@rust-lang-team/SyRlhj0u0. Accessed: 2026-05-24. 2024 (cit. on p. 5).

[41] *Safety comments policy - Standard library developers Guide.* Accessed: Jan. 2025. URL: https://std-dev-guide.rust-lang.org/policy/safety-comments.html (cit. on pp. 1, 3).

[42] Leon Schuermann, Jack Toubes, Tyler Potyondy, Pat Pannuto, Mae Milano, and Amit Levy. "Building bridges: Safe interactions with foreign languages through omniglot". In: *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 2025, pp. 595–613 (cit. on pp. 5, 7).

[43] The Serde Developers. *Deriving De/Serialize for type in a different crate.* Accessed: 2025-08-01. URL: https://serde.rs/remote-derive.html (cit. on p. 3).

[44] David Tolnay. *no_panic.* https://docs.rs/no-panic/latest/no_panic/. Accessed: 2026-05-14. 2026 (cit. on p. 7).

[45] *Trait std::marker::Freeze.* Accessed: Jan. 2025. URL: https://doc.rust-lang.org/std/marker/trait.Freeze.html (cit. on p. 5).

[46] *UnsafeCell in std::cell - Rust.* Accessed: June 2026. URL: https://doc.rust-lang.org/std/cell/struct.UnsafeCell.html#:~:text=UnsafeCell%20opts%2Dout,UnsafeCell%20to%20wrap%20their%20data. (cit. on p. 4).

[47] Lydia Zoghbi, David Thien, Ranjit Jhala, Deian Stefan, and Caleb Stanford. "Auditing Rust Crates Effectively". (Preprint). 2024. URL: https://web.cs.ucdavis.edu/~cdstanford/doc/2024/CargoScan-draft.pdf (cit. on pp. 5, 7).