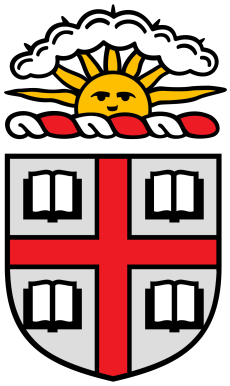
Sniffer: Surfacing Dangerous Unsafe Code in Rust

Corinn Tiffany

Advisor: Malte Schwarzkopf

Reader: Will Crichton



Department of Computer Science

Brown University

Providence, RI

April 2025

Contents

Acknowledgments

To Malte, for your mentorship and friendship. You've made an immeasurable impact on me as a learner, a writer, a problem solver, and a good-science-doer. Though my chapter at Brown has come to a close, what I've learned from you is here to stay.

To Will, for lending your boundless Rusty knowledge, good humor, and crisp writing. The future Rustaceans of Brown are lucky to have you.

To Kinan (Professor-Doctor-Sensei Dak Albab), for adopting me as your karate kid in training. Thank you for teaching me about the finer things in life, e.g., marking functions as pub(crate) and mortar-and-pestle pesto. You'll make a fantastic professor, and I can't wait to see your future students thrive.

To Artem, for coffees in the courtyard and pointing out our matching haircuts. Here's to many walks after class, and more to come in new locations. Sniffer wouldn't be the same without you, and neither would my time at Brown.

To the rest of the Sniffer team: Philip Levis, Akshay Narayan, Deepti Raghavan, Ziyun Song, and Isabella Szabo. I've learned so much from you, and I'm excited to continue this journey.

To my lovely friends, particularly Aijah Garcia, for being my first friend at Brown, and the housemates of 25 East—Aidan Harbison, Susanne Kowalska, Kenny Daici, and Angel Benjamin—for being the best late-night-chatters and snowball-throwers.

To the rest of ETOS, in particular the Sesa-team and the ETOS girlies. You were first my role models, and then my wonderful friends, too.

To my parents and Maquelle, for your endless encouragement, love, and curiosity.

Abstract

Over the past decade, systems researchers have leveraged Rust to build systems with valuable security or isolation properties. In this thesis, we explore how unsafe code in dependencies can cause serious violations of the guarantees these Rusty systems seek to provide—even in the ideal case, where all unsafe code is free of undefined behavior. We present Sniffer, a static analysis tool to help developers audit their systems for guarantee-breaking unsafe code. We evaluate it on real systems and libraries, and we find that Sniffer reduces the audit burden to check that a codebase upholds properties such as the absence of interior mutability, encapsulation of structs, and panic freedom. Sniffer can help developers fulfill the promise of strong guarantees in the presence of unsafe code.

1 Introduction

Rust is an exciting programming language for systems research because it has the performance characteristics of languages like C++, but also offers guarantees about properties such as memory safety. Over the last decade, many systems have used Rust to provide guarantees such as crash safety [28], data integrity [10, 29, 32, 35], and data privacy [15, 26]. Valuable features like memory safety, encapsulation, immutable references, and data race freedom [42] form the foundation for these higher-level guarantees.

However, these properties only hold in Rust without any unsafe code; in unsafe blocks, the compiler doesn't enforce all the same properties that it usually does, so a developer can perform, e.g., raw pointer dereferences. A common refrain among Rusty research systems is caveats for unsafe code that their authors *correctly* establish. Papers often state that their system's guarantees only apply in the "safe subset of Rust," because in unsafe blocks, the compiler will not check the properties upon which these systems depend. However, it is rarely clear what exactly constitutes the "safe subset," e.g., whether libraries must also consist entirely of safe code. In this thesis, we discuss the problems that arise when trying to rely on such properties in real-world programs that contain unsafe code in dependencies. We introduce Sniffer, a static analysis tool to help developers surface information on the properties of their program and maintain guarantees in the face of unsafe code.

1.1 Problem Statement

The ambiguity of "safe" Rust becomes particularly pertinent in the face of what we call *bonus properties*: certain properties, such as encapsulation of private fields and immutability of references, are enforced by the compiler in programs without any unsafe code, but can still be broken in valid unsafe code. This is because the *sole requirement* Rust imposes for unsafe code is to not cause undefined behavior [43].

In practice, most Rust programs use unsafe code abstracted away in library calls [21]. While much of this unsafe code preserves bonus properties, there is neither a guarantee nor an expectation that any given unsafe block does so. This mismatch is the source of the central tension Rusty research systems must navigate: many systems rely on Rust's bonus properties, but neither the Rust compiler nor convention around unsafe code ensures these properties are upheld, even when a system developer never writes an unsafe block themselves.

In this thesis, we seek to bridge the gap between theoretical guarantees that hold in the absence of unsafe code and the reality of Rust programs that cannot avoid it.

1.2 Example Scenario

Imagine the following: a system designer identifies that a bonus property, such as encapsulation, is normally enforced in safe Rust, and builds a system atop this property. Their goal is to provide strong guarantees about the behavior of their system, even when it accepts user-provided *extension code*. For example, Sesame [15] is a a system for privacy compliance in web applications and supports custom HTTP handler code, and RedLeaf [32] is a operating system that seeks to provide lightweight subsystem isolation and supports loadable kernel extensions. The system developer states that their bootstrapped guarantees will hold for extensions written in "safe Rust."

When a user of the system goes to write an extension that interfaces with this system, they read the caveat about unsafe code, and they attempt to ensure that their program meets this restriction. First, they use the `forbid_unsafe` lint [48] to disallow unsafe code in their own crate, that is, their Rust library or package. However, the system user is aware that their program may invoke unsafe code in libraries. Unsafe code is a critical component of the Rust ecosystem; it is frequently used to interact with hardware, to implement low-level performance optimizations, and to improve expressivity when the borrow checker is overly restrictive. In the best case, this unsafe code will be correctly implemented and free of undefined behavior. To confirm that their program does not have memory bugs or undefined behavior, they follow best practices by using tools like Miri [38], Kani [51], and Rudra [6].

However, there is a catch—existing tools that target the absence of undefined behavior will not help the developer if some unsafe code inadvertently breaks the higher-level guarantees of the system. This is because violating bonus properties does not constitute undefined behavior, nor is upholding them relevant to all systems, given that violating bonus properties can be useful to improve expressivity, e.g., interior mutability via `RefCell`. Even in the ideal case, where all unsafe code is free of bugs, unsafe code deep in dependencies may violate bonus properties, and in turn, a system's intended higher-level guarantees.

In addition to guarantees derived from bonus properties, developers often seek to uphold other properties, such as panic freedom in utility libraries [8, 9, 47], or the minimization of unsafe code in a cryptography library [19].

Tools such as Cargo Scan [53] and Cargo Geiger [12] can help a developer audit a whole codebase for unsafe code by finding any unsafe code in a crate or its dependencies. However, this returns thousands of functions to audit, many of which are unreachable, and among the reachable unsafe code, much of it is irrelevant to bonus properties or panics. This limits the utility of these tools for auditing for panics, unsafety, or violations of bonus properties.

Consequently, developers lack practical tooling to navigate the unsafe code their program invokes and to understand the interactions between this unsafe code and the bonus proper-

ties upon which a Rusty system may depend. Sniffer is a new developer-oriented auditing tool that seeks to fill this gap by precisely finding *reachable* unsafe code, and reducing the effort to audit these results by automatically filtering them for a system-specific property. We call this approach *property-targeted auditing*. Sniffer's goals are as follows:

1. Support configurable analyses based on bonus properties and application-specific constructs;
2. Be as precise as possible, i.e., never skip a region of concern and minimize false positives;
3. Be efficient and easy to use.

1.3 Contributions

This thesis makes the following contributions:

1. We propose a more precise vocabulary for talking about safety of Rust code, and survey published research systems whose guarantees are vulnerable even when users write only "safe Rust" (§3).
2. We design and implement Sniffer, a static analysis tool to aid property-targeted auditing (§4).
3. We use Sniffer to audit five crates which seek to provide panic-freedom, minimal unsafe code, and higher-level guarantees bootstrapped from bonus properties (§7).

This work was done in collaboration with others from the Sniffer team: Artem Agvanian, Ziyun Song, Kinan Dak Albab, Will Crichton, Philip Levis, Akshay Narayan, Deepti Raghavan, and Malte Schwarzkopf.

2 Background & Related Work

2.1 Background on Rust

Safety. In C and C++, it is largely the programmer's responsibility to avoid writing code with undefined behavior. Rust shifts a significant portion of that responsibility onto the compiler by limiting code that can have undefined behavior, such as calling into other languages or pointer arithmetic, to explicit unsafe blocks. In Rust, the compiler enforces certain safety properties–including absence of undefined behavior–for safe code, which is the majority of Rust code.

Programmers are responsible only for ensuring unsafe code doesn't cause undefined behavior. For example, if a function uses an unsafe block to perform pointer arithmetic, the programmer is responsible for checking pointer alignment and bounds.

The absence of undefined behavior outside of unsafe blocks is *Rust's only guarantee*. As of May 2025, Rust does not have a formal reference semantics, so today undefined behavior is defined as a list of problem behaviors [7] familiar to systems programmers, such as accessing a dangling pointer, causing a data race, or violating LLVM's pointer aliasing rules. In practice, developers use Miri [38] to detect the presence of undefined behavior in their programs.

Safe Interfaces and Safety Comments. Developers can encapsulate unsafe blocks within safe functions, creating *safe interfaces*. Without reviewing the source code, developers cannot distinguish calls into safe interfaces from those into an exclusively safe Rust implementation. This is deliberate, since it allows for a separation of concerns: library developers can, e.g., use unsafe code to optimize performance, while application developers need not know the details of the unsafe optimization, provided it is implemented correctly. A study of the 500 most-used Rust crates found that more than half call into unsafe code, even when excluding calls into standard library safe interfaces [21].

When a function itself is marked as unsafe, the unsafe label propagates to the caller, and the developer calling the unsafe function must ensure their invocation will not cause undefined behavior in their program. The Rust community has developed a convention of writing *safety comments* for unsafe functions, e.g., in the standard library [39]. These comments document the invariants and preconditions under which the unsafe function is free of undefined behavior. This is meant to help client code use these APIs safely, and to make maintenance of the unsafe code easier.

There is a growing body of sociotechnical work describing how Rust programmers use unsafe code. They find that even though the developers strive to keep unsafe code localized and simple, it is nevertheless used extensively [5, 21]. Studies also indicate that the majority of developers lack confidence in the correctness of their safe interface implemen-

tations [30] and that exposed unsafe library functions often fail to document their required preconditions [5]. This complicates the ideal that unsafe code within safe interfaces upholds the same guarantees as compiler-checked safe code, even when it comes to absence of undefined behavior.

Bonus Properties. When a function f is globally safe, Rust also enforces useful properties that go *beyond* absence of undefined behavior. For example:

- **Encapsulation:** f can never directly read or write private fields of structures outside the module of f.
- **Lifetimes as aliases:** If f contains a reference of type `&'a T`, then it is possible to soundly approximate the reference's aliases from the lifetime `'a`.
- **Immutability of references:** If f contains an immutable reference of type `&'a T`, then all data accessible through the reference will never mutate during the lifetime `'a`.

We call these *bonus properties* because while they often serve as a basis for higher-level system guarantees, valid unsafe Rust can violate them. The optional status of these properties is an intentional choice on the part of the Rust language developers; allowing unsafe code to violate them increases the expressivity of the language. Mutexes, for example, break the immutability property: multiple threads can each hold an immutable reference to share data through the mutex, allowing them to modify the protected data while using the lock to prevent data races.

We have found that Rusty systems often bootstrap guarantees from the properties of encapsulation and immutability of references, and real-world crates seek to uphold panic-freedom. Sniffer can surface violations of these properties, as well as panics and invoked unsafe code, providing developers with the means to check that their code upholds the guarantees they desire.

2.2 Related Work

Sniffer complements a rich body of related work that aims to check for undefined behavior or verify functional correctness. We discuss alternative approaches to propagate information about bonus properties to a system auditor in §2.3.

Detecting Undefined Behavior. Many tools aim to enforce the absence of undefined behavior. We envision that a developer would use existing tooling to check for undefined behavior, as well as Sniffer to check for the particular bonus properties that are critical to their system.

Miri [38] is a Rust interpreter that identifies undefined behavior in unsafe code. Developed alongside the Rust compiler, Miri has become the community standard for defining undefined behavior. Miri detects undefined behavior as it occurs during the execution of a test

suite, so it can only provide guarantees for the code executions that a developer explicitly tests.

Kani [51] is a bounded model checker that can detect a suite of behaviors including UB and panics. Running Kani does not require developers to write specifications, and unlike Miri, it finds error states that exist along any possible execution path. However, it encounters long runtimes on large programs, and it lacks support for common Rust constructs such as vectors and hashmaps.

Rudra [6] is a static analysis tool aimed at detecting a selection of common *memory safety* bugs at the ecosystem scale, rather than violations of bonus properties, as Sniffer does. Both tools conduct a suite of analyses on high-level IR (HIR) and mid-level IR (MIR) of a Rust program and its dependencies.

Functional Correctness. There are many verification frameworks that enable formalizing the behavior of Rust programs. Frameworks such as Prusti [4], Creusot [17], and Aeneas [24] emphasize safe Rust as their target language, taking Rust's memory safety and type system guarantees as a given. Verus [27] provides coverage for verifying the behavior of a subset of unsafe code. This is largely code that fails the borrow checker but semantically follows Rust's ownership rules, e.g., operations on raw pointers and the interior mutability primitive `UnsafeCell` are supported, but not `transmute`. Sniffer instead targets bonus properties and is — in some cases — interested precisely in code that breaks Rust's ownership model.

2.3 Alternative Approaches to Uphold Bonus Properties

We discuss alternative approaches to ensure that extension code upholds the bonus properties that a system depends on, and explain how Sniffer fits into this design space. Each strategy offers a trade-off between the strength of its validation mechanism and the burden it places on the system, extension, and dependency developers.

Safety Comments. Library developers could extend classic safety comments with preconditions under which the unsafe code upholds bonus properties ("*property comments*"). Developers would informally consult these preconditions to increase their confidence that their extensions inherit the guarantees of the bonus properties they rely on.

While safety and property comments offer a lot of flexibility, they provide no enforcement guarantees and face two challenges in practice. First, the documented preconditions may be incorrect or out of date—this risk can be mitigated by formalizing and verifying them. Second, extension developers may not even be aware of these preconditions if the unsafe function is in a deep call chain. Extension developers would benefit from tooling that identifies and surfaces hidden unsafe code, along with its safety and property comments. Sniffer provides the former and could be adapted to do the latter.

Formal Verification. In the presence of unsafe code, the Rust compiler alone does not guarantee that extension code exhibits the bonus properties Rusty systems require. Extensions could use Rust formal verification tools like Verus [27] and Kani [51] to determine whether a system's promised guarantees, e.g., isolation, actually hold, e.g., because the unsafe blocks actually obey the relevant bonus properties. This may be especially important in critical settings, such as kernel extensions in RedLeaf.

With this approach, the extension developer is responsible for verifying their codebase in its entirety, which is too high of a burden for general use, e.g., for web developers using Sesame. Further, support for unsafe constructs is incomplete in many verification tools: e.g., Verus supports `UnsafeCell` via ghost state, but not reasoning about `transmute`.

We believe that verification is best aimed at critical components of the Rust ecosystem, such as the standard library. This requires new research to extend formal contracts [3] with descriptions of when unsafe code upholds bonus properties, and approaches to verify the correctness of these contracts. This complements efforts to crowdsource verification of the standard library for memory safety and absence of undefined behavior [25]. With a carefully scoped contract language, it may be possible to create automated tools that enforce them in extension code [11, 22, 36].

Auto Traits. The compiler automatically implements auto traits [40] for types whose fields also implement the trait. Because the compiler and type system implement and enforce them, they require no opt-in from extension or dependency developers. One example of this is Rust's `Freeze` trait [46], which guarantees that a type does not contain standard interior mutability primitives, e.g., `RefCell`. Using `Freeze`, the Rust developers fixed a critical language bug that allowed mutating data inside static constants [20].

On the other hand, using auto traits to prevent violations of bonus properties can lead to more restrictive APIs. RedLeaf's `RRefable` auto trait [32], which was intended to uphold a "no interior mutability" invariant, is one example. RedLeaf could use `Freeze` to forbid passing `RefCell` to subsystem invocations to prevent the broken guarantees we describe in §3.3. The original design uses an auto trait `RRefable` to a similar end, but did not forbid using standard library interior mutability primitives. While using autotraits would accomplish the goal of disallowing interiorly mutable types from being shared between subsystems, this would have resulted in a restrictive API that forbid passing a `RefCell` in any context – a restriction that might preclude efficient and correct designs and force less efficient ones. In that example, a less restrictive, but correct mechanism would allow passing `RefCell` to the scheduler domain for read-only use, but forbid calling `borrow_mut()`. More broadly, a primary limitation of auto traits as a mechanism to enforce bonus properties is that they are suitable to express properties about the type itself, e.g., disallowing interiorly mutable types, but not how surrounding code operates on that type, e.g., forbidding `transmute` on a type that should uphold encapsulation.

Currently, `Freeze` and custom auto traits are only available on nightly Rust, but stabilizing them would greatly benefit Rusty systems that depend on the absence of interior mutability. Due to concerns about semantic versioning [37], the Rust developers state these features are "unlikely to be" stabilized [18]. We note that the `Send`, `Sync`, and `UnwindSafe` auto traits are stable even though they exhibit the same issues with semantic versioning [34, 45], and that stabilizing `Freeze` yields comparable benefits. One pathway to stabilization is to only allow defining *private* auto traits, such that only the system itself can depend on them. Consequently, minor-version changes to downstream crates would cause compilation errors (and thus break semantic versioning) only when they violate the critical properties of the system, which is desirable. This mirrors stable Rust as of this writing, where `Freeze` is private but causes semantic versioning hazards [37] precisely when critical properties, e.g., immutability of constants, are violated.

Comparison with Sniffer. Sniffer can provide stronger guarantees than safety or property comments alone, especially in the face of deep call chains in dependencies. Auto traits provide strong, compiler-checked guarantees. However, they can only encode whether a type has some property based on its component types, as they solely enforce properties via the type system. This can be overly restrictive in some cases. Sniffer can capture a diverse set of fine-grained properties relating to the functions invoked in a given program.

Formal verification provides stronger guarantees than Sniffer, though at the cost of vastly increased developer effort. Sniffer places no burden on library developers and requires some effort from the extension developer, which is proportional to how much code they wish to audit. Furthermore, Sniffer is tailored to bonus properties, which are only partially supported by current formal verification techniques.

Sniffer fills a gap in the design space: it can precisely express bonus properties and provide strong guarantees with acceptable amounts of developer effort.

3 Safety Challenges for Rusty Systems

3.1 Scoping Safety

The phrases "safe Rust" and "safe subset of Rust" have no formal meaning within the Rust language, but they colloquially refer to "Rust without the `unsafe` keyword." These phrases, however, are ambiguous: they could mean either "source code with no `unsafe` blocks," or "source code which does not invoke any `unsafe` blocks."

To differentiate these meanings, we introduce two new terms. A piece of Rust code is **locally safe** if it syntactically does not contain `unsafe` code. A piece of Rust code is **globally safe** if all reachable functions in its call graph do not contain `unsafe` code. Figure 1 shows an example to illustrate the difference. Using these definitions, we can state more precisely what Rust seeks to guarantee with safety. For any locally safe function f that passes the Rust type checker, f cannot cause undefined behavior if and only if all unsafe code called by f cannot cause undefined behavior. This definition helps us be more precise — the earlier claim that "safe Rust code is guaranteed to be free of [undefined behavior]" is only true if "safe" means "globally safe."

Rusty systems often rely on bonus properties that only hold in globally safe Rust, but Rust programs are often only locally safe. In practice, this mismatch can result in the loss of guarantees. This mismatch might be problematic for standalone systems, but it is even more salient for systems that embed user-provided *extension code* such as the systems in Figure 2. An extension developer (who is distinct from the original system developers) may be unaware that their extension can violate the system's guarantees when they write an extension that is only locally safe.

In the remainder of this section, we investigate case studies of systems that rely on bonus properties and the types of bugs to which they are vulnerable as a result. We describe Sesame [15] and RedLeaf [32] in detail, and give an overview of several other systems which face similar vulnerabilities.

3.2 Sesame: Leaking Private Data

Sesame is an end-to-end privacy compliance system for Rust web applications. Application developers declare privacy policies and associate them with data using Sesame's API, such as access control policies for sensitive data or k-anonymity requirements for aggregates. Sesame ensures policies remain attached to data throughout execution and checks policies before allowing the application to externalize the associated data, e.g., by rendering it to users via HTML.

Sesame's central abstraction is the *Policy Container*: a `PCon<T, P>` encapsulates sensitive data of type T and an associated policy of type P as private members. As a result, applications cannot directly access `PCon`-protected data. As unsafe Rust code is not guaranteed to

```rust
let arr: [i32; 2] = [2, 5];
let mut it: Iter<i32> = arr.iter();
it.next()?;
let n: &i32 = it.next()?;
```

(a) Locally safe Rust.

```rust
let arr: [i32; 2] = [2, 5];
let mut it: *const i32 = &raw const arr[0];
it = unsafe { it.add(1) };
let n: i32 = unsafe { *it };
```

(b) Unsafe Rust.

```rust
let arr: [i32; 2] = [2, 5];
let mut it: usize = 0;
it += 1;
let n: i32 = arr[it];
```

(c) Globally safe Rust.

Figure 1: Three code examples of iterating over an array in Rust: (a) shows idiomatic Rust code using the `Iter` type in the standard library. This code looks safe—we call it *locally safe* because it does not contain any unsafe code on the surface. (b) directly uses unsafe pointer arithmetic to iterate over the array, which is essentially the underlying implementation of (a). This shows how locally safe Rust often relies on unsafe code encapsulated in safe interfaces like `Iter`. (c) is non-idiomatic but *globally safe* Rust code, because it does not use any features which transitively invoke unsafe code. Efficient iterators based on pointer arithmetic cannot be implemented in globally safe Rust.

System	Goal	Scope	Rust Property	Violation
Sesame [15]	privacy enforcement	no "malicious" `unsafe` code	encapsulation	data leakage and privacy violations
RedLeaf [32]	kernel subsystem isolation	"restricted to safe Rust"	immutability	no isolation between subsystems, kernel cannot recover after crashes
Flowistry [13]	analyzing information flow	"safe subset of Rust programs"	lifetimes + immutability	misses existing flows (unsound)
Net-bricks [35]	packet isolation between NFs	no "`unsafe` pointer arithmetic"	immutability of references	packet modified after NF sent, memory bugs
Cocoon [26]	information flow control	"absence of `unsafe` Rust"	encapsulation, immutability of references	data leakage
Boucher et al. [10]	serverless function isolation	"safe Rust"	"language isolation"	data leaks between serverless functions

Figure 2: A list of recent Rusty systems along with their declared scope, the bonus properties they rely on, and example violations that we confirmed are possible with *locally* (but not globally) safe extensions.

respect **encapsulation**, it can break Sesame's guarantees (e.g., by calling Rust's `unsafe mem::transmute` function). Importantly, such code does *not* necessarily cause undefined behavior, but it still violates the bonus property upon which Sesame depends.

Scenario. To demonstrate how applications might inadvertently violate Sesame's properties, we consider the `questions_submit` endpoint in Sesame's WebSubmit homework submission application. This endpoint receives a homework submission, which Sesame wraps inside a `PCon` with an access control policy attached.

We added a logging library to this application that uses the `abomonation` [31] crate to serialize objects before logging them. This library uses unsafe code to re-interpret objects' memory content as bytes and serialize any heap data the object points to recursively. In addition to serialization and logging, developers may implement similar code for other purposes, such as stack inspection for panics. Using this library to log the text of a homework submission breaks the guarantees of a `PCon` implementation that depends on encapsulation, because the policy-protected data gets externalized (logged) without a policy check.

To protect against this pattern of unsafe code, Sesame uses a fallback mechanism: rather than storing the protected data itself, `PCons` store a pointer to the data and obfuscate the pointer with a private constant in Sesame. With this mechanism in place, the logging library prints only the obfuscated pointer. This defends against some unsafe code that violates the bonus property of encapsulation; however, it comes at a 2× runtime overhead in Sesame's microbenchmarks, and it does not prevent arbitrary unsafe code from leaking data, e.g., unsafe code that swaps policies with more permissive ones, or reads Sesame's private constant and de-obfuscates the data pointer stored in a `PCon`.

3.3 RedLeaf: Violating Kernel Subsystem Isolation

RedLeaf is an operating system that implements isolation between kernel subsystems using Rust [32]. Popular kernels today lack such isolation; each subsystem can access the entire memory. When a subsystem crashes, the kernel often cannot recover, as it is unclear what state the crashed subsystem modified.

RedLeaf relies on **immutability of references** when rolling back crashed kernel subsystems. In RedLeaf, kernel subsystems can only 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, interior mutability in Rust allows modifying data behind immutable references, which may be owned by another subsystem, thus endangering the guarantees of RedLeaf.

RedLeaf restricts the types that can be allocated on the shared heap using an auto trait `RRefable`. `RRefable` disallows data types containing raw pointers from being allocated on the shared heap, but not interiorly mutable data types (e.g., `RefCell`). Therefore, a

crashed subsystem that modified the content of a `RefCell` from the shared heap could leave the system in an inconsistent state.

Scenario. To demonstrate how this might happen, we implemented a pluggable scheduler as a kernel subsystem that meets RedLeaf's `RRefable` restrictions but breaks the system's guarantees. The scheduling interface uses a single function to choose the thread that should be scheduled next. That function takes the list of metadata for the queued threads as input, chooses the next thread, removes its metadata from the list, marks the thread as running, and then returns. Thread metadata cannot be private to the scheduler because the base kernel must also be able to modify it when a thread is created or destroyed. We share this metadata between the kernel and the scheduler via the shared heap. To allow both subsystems to modify it, we store thread metadata in a `RefCell`. The original RedLeaf scheduler also follows this design pattern.

However, if the scheduler code crashes after removing the thread metadata from the list but before returning, that thread's metadata is lost. Because the scheduling subsystem does not own the threads list, the rest of the kernel will not attempt to roll it back after the crash. Hence, the crash in the scheduler subsystem will cause the permanent loss of the thread, preventing it from being scheduled again. In this way, interior mutability in the locally safe scheduler code breaks RedLeaf's guarantees.

3.4 Other Systems

Similar problems show up in other systems. Information flow control in Cocoon [26] relies on immutability of references and encapsulation; outside of globally safe Rust, Cocoon can leak high-security data to low-security sinks. Flowistry [13] relies on Rust's aliasing and immutability properties to soundly establish data flow; otherwise, it may miss flows through unsafe code, breaking soundness. A proposal for lightweight isolation in microservices [10] relies on Rust's immutability and encapsulation properties, among others, to provide language-based isolation. Serverless functions that are only locally safe will pass the implementation's "no-unsafe" lint, but may invoke dependencies that use unsafe code to break language isolation. Finally, Netbricks [35] uses Rust's features to enforce isolation between network functions (NFs). Unless NFs are globally safe, they can leak references that allow access to packets already passed off to another NF (an isolation violation).

3.5 Summary

These examples illustrate that valid unsafe code can break the bonus properties upon which published research systems rely, causing serious vulnerabilities in extension code. Because unsafe code is necessary, systems cannot avoid it entirely. Furthermore, because it is often encapsulated in libraries, finding and auditing unsafe code is difficult for extension and system developers, not to mention the effort required to vet unsafe code across

a realistically sized dependency tree. Developers need a better solution that directs their attention to *reachable* and *problematic* unsafe code in dependencies.

4 Sniffer Design

The examples in §3 demonstrate that some Rusty systems rely on bonus properties that are only guaranteed in globally safe code, which is rare in practice [21]. Despite this shortcoming, Rusty systems still present an exciting opportunity. We would like their guarantees to hold across a wide variety of applications.

Property-targeted auditing is our means to extend the guarantees of systems that depend on bonus properties to code that is not globally safe. We present Sniffer, a prototype tool for property-targeted auditing using static analysis. Sniffer seeks to not only provide precise information about what unsafe code is reachable, but also leverage the specific and scoped requirements of Rusty systems, such as "encapsulation of PCons must never be violated" (Sesame) or "shared-heap references passed to a domain invocation must never be mutated" (RedLeaf). We pursue this approach because it fills a gap in the design space: it requires no opt-in from library developers who are agnostic to the concerns of such a system, and it is a strong and expressive validation mechanism.

4.1 Overview

Sniffer analyzes an input crate and its dependencies for adherence to specified properties. Sniffer represents the properties as specific unsafe operations and function calls. Sniffer is a compiler plugin; its analysis is performed on mid- and high-level intermediate representations (MIR & HIR) that the Rust compiler generates. Sniffer approximates each Rust codebase as a call graph, and looks for reachable paths to functions that are flagged as violating the property. The call graph approximation is easy and fast to compute. It quickly rules out many false positives proposed by other tools that return all property-violating code in dependencies, although it is less precise than variable-granularity dependency graphs as in Paralegal [1].

To begin their audit, users annotate functions in their extension code as entry points for analysis. Figure 3 describes the available annotations that users can attach to functions. Sniffer emits compiler diagnostics to indicate calls in the top-level application that invoke code that may violate the specified property. It also produces visualizations of the call chains between the top-level crate and the flagged region. Auditors can use the call chain information and a careful review of the source code to decide if the region truly threatens the guarantees of the system.

4.2 Analysis

The first step in any Sniffer analysis is to construct a monomorphized call graph of all transitively reachable functions using PEAR [2]. The call graph construction conservatively resolves dynamic dispatch. Sniffer implements checks for bonus properties as a visitor over

Example	Description	Parameters
`forbid_unsafe`	Find reachable `unsafe` functions, methods, and blocks.	–
`freeze`	Detect interior mutability via calls to `UnsafeCell`'s `.get()` and `.raw_get()`.	–
`panic_free`	Find panicking bodies.	–
`encapsulation(PCon)`	Find violations of struct encapsulation via `transmute` and raw pointer casts.	Extension-level types that must remain intact.
`allow(std::iter, core)`	Filter out analysis results that are invoked in or by this crate.	Path prefixes to allowlist.

Figure 3: Sniffer API. Auditors attach annotations to functions they wish to use as entry points to analysis. Each annotation triggers a different analysis to detect reachable unsafe code, interior mutability, panics, or violations of encapsulation. Auditors specify allowlists via path prefixes to filter results from crates, modules, or functions (last line).

each function body in the call graph. These analyses identify all regions that *may* violate the property.

Sniffer is sound but not complete, meaning that it may warn the developer of regions that do not actually violate the property (false positives), but it will never miss a region that truly does violate the property. Sniffer has two main sources of false positives: dynamic dispatch resolution, which can mark functions as "possibly reachable," and property heuristics, which may mark innocuous regions as violating the target property.

When Sniffer returns analysis results, the auditor's next step is to determine whether any "possibly reachable" functions would be invoked at runtime. Next, the auditor uses the context of the intended guarantees of a system to determine if the flagged region constitutes a true violation. This step depends on the property the auditor is investigating; the process to follow up on results from various Sniffer analyses is described below.

Reachable Unsafe. Sniffer finds reachable unsafe code via a visitor over the High-Level Intermediate Representation (HIR) of each function body. This visitor finds all unsafe blocks, functions, and implementations of unsafe traits. Unlike most other analyses, Sniffer performs this analysis over the HIR rather than the MIR. This is because unsafety information is stripped away during an unsafety checking pass between the HIR and MIR [50].

Broken Encapsulation. Auditors can specify types in their program whose encapsulation must not be broken. To circumvent the field privacy of a struct, the struct must first be converted to another type, e.g., one with public fields. Therefore, Sniffer alerts the auditor of flows from the specified type to `transmute` or raw pointer cast operations, which can be used to reinterpret the struct as another type. This is implemented via a visitor over the MIR of each reachable function body.

This is a stronger, but easier to check, property than encapsulation alone. To determine if a region Sniffer finds is a violation of encapsulation, the auditor must confirm that the bytes exposed by the `transmute` or raw pointer cast are read afterwards.

Interior mutability. Sniffer's check for interior mutability searches the monomorphized call graph for calls to `.get()` and `.raw_get()` on `UnsafeCell`. The `UnsafeCell` is the core primitive for interior mutability in Rust [49]. These getter methods are the two ways to obtain a raw pointer `*mut T` to its contents. Other interiorly mutable types such as `Cell`, `RefCell`, and `Arc` wrap `UnsafeCell` and must invoke these methods to mutate the contents of an `UnsafeCell` when holding an immutable reference. This check cannot be circumvented by implementing an alternative interiorly mutable struct, because `UnsafeCell` is specially privileged by the Rust compiler to allow its functionality; in other contexts, transmuting a `&T` to a `&mut T` is undefined behavior [44].

When Sniffer alerts an auditor to a call to `UnsafeCell`'s `.get()` or `.raw_get()`, they can look at the call graph to determine in what context the call originated, e.g., an application-

level call to `RefCell`'s `borrow_mut`, and decide whether this is an acceptable instance of interior mutability. This may depend on the type of the data stored in the `UnsafeCell`. For example, in RedLeaf, the `RRef` type represents data shared between kernel subsystems; interior mutability over this type can break subsystem isolation.

Panics. To find calls that may panic, Sniffer searches the call graph for invocations of `core::panicking::panic_fmt`. All the various panicking operations, such as explicit `panic!()`, `.expect()`, and `.unwrap()` on a `Option` or `Result` can be detected by finding instances of `panic_fmt`.

Auditing a possible panic flagged by Sniffer involves reasoning about application-level invariants and determining whether error states are actually reachable. In our case studies, a frequent source of false positives is instances where a constant is passed to a function that will panic only if passed a particular value. The Sniffer-provided call graph visualization is particularly useful to identify these false positives.

Analysis	LoC
Reachable Unsafe	119
Broken Encapsulation	148
Interior Mutability	43
Panic Freedom	24

Figure 4: LoC for each of Sniffer's analyses.

5 Implementation

Sniffer is built on `rustc nightly-2024-01-06` in 1.4k lines of Rust. Figure 4 shows the lines of code for each specific analysis, and the remainder is shared infrastructure. Sniffer is a compiler plug-in using PEAR [2] to build the underlying monomorphized call graphs and rustc_plugin [14] to interface with the Rust compiler. Sniffer requires client code to compile with `nightly-2024-01-06`.

6 Case Studies

We evaluate Sniffer on five case studies. In this section, we describe each case study's purpose, target property, and the ways in which developers sought to uphold this property. §7 discusses the results of our evaluation.

6.1 Sesame

Sesame [15] is described more thoroughly in §3.2. Sesame is a system for end-to-end privacy compliance in Rust web applications. It guarantees that code processes user data in accordance with privacy policies. As data moves around the application, it is stored as a private member in a *privacy container* or `PCon` struct. This controls access to the private data to guarantee it is only operated upon in policy-compliant operations.

This guarantee depends on the bonus property of encapsulation: if a library uses unsafe code to access a private member of a `PCon`, operations on the data will not necessarily be governed by Sesame policies. Due to its prevalence in real-world applications and their dependencies, Sesame cannot place all unsafe code out of scope, but excludes "outright malicious" unsafe code from the threat model. To handle other problematic unsafe code, Sesame includes a runtime mechanism to protect against mistakenly casting away the `PCon` wrapper. This prevents the application from leaking private data in a case like the `transmute` example, but incurs a 1.7–2.1× overhead in microbenchmarks [15]. Using Sniffer to find possible violations of encapsulation at compile time would eliminate this overhead.

6.2 RedLeaf

RedLeaf [32] is described more thoroughly in §3.3. RedLeaf is an operating system that seeks to use Rust to provide lightweight isolation between kernel subsystems. A primary goal of this work is to prevent a crash in any one subsystem from affecting the execution of the others. The rollback mechanism hinges on a design that only requires references to be immutable in order to be shared between subsystems. In the absence of interior mutability, this ensures that each kernel subsystem cannot mutate shared data, only that which it owns, which cleanly delineates the data that can be in an inconsistent state after a crash.

RedLeaf attempts to govern the types that can be shared between subsystems with the `RRefable` autotrait. However, the autotrait implementation still allows interiorly mutable types, e.g., `RefCell`, which mistakenly permits subsystems to modify shared data and leave the system in an inconsistent state if they crash. Using Sniffer to audit the RedLeaf domains for instances of interior mutability would allow the extension developers to confirm that RedLeaf's rollback guarantees truly hold.

Function	Purpose	# Crates Analyzed	# Functions Analyzed	# Unsafe Instances Found
`crypto_kdf_keygen`	Key Generation	4	16	7
`crypto_sign`	Signing	8	148	20
`crypto_box_easy`	Authenticated Asymm. Encryption	8	202	54

Figure 5: Sniffer located 7, 20, and 54 instances of unsafe code in the call graphs of three `dryoc` cryptography functions. All counts are deduplicated, and we exclude `std`, `core`, and `alloc` (which contain substantial unsafe code, but are generally trusted).

6.3 `dryoc`

The cryptography library `dryoc` [19] is a pure Rust re-implementation of libsodium [16]. It aims to be "mostly free of unsafe code", with caveats for "some third-party libraries, such as those with SIMD" [19]. Minimizing unsafe code in a cryptographic context is relevant because unsafe buried in the call chains of the library's functions could leak cryptographic secrets (e.g., via buffer overflows or memory allocations that are not zeroed out after use). When seeking to minimize unsafe code, the developers of `dryoc` likely only considered unsafe code that they directly invoke. Sniffer could help them find unsafe code abstracted away in dependencies in order to audit it.

6.4 `ufmt` & `bitter`

We applied Sniffer to audit two utility crates advertised as panic-free: ufmt [47], and bitter [9]. The crate ufmt is an alternative to `core::fmt`, and the crate bitter is a bit-reading utility. The developers of both crates used the tool no-panic [33] to confirm panic freedom. no-panic prevents programs from linking with the symbol for panic in the standard library. If a crate compiles with all optimizations and no-panic enabled, it is statically proven to not panic.

The tool no-panic is a suitable approach for crates with relatively simple, straight-line code, such as these two case studies. However, it is likely overly restrictive for more complex code where reasoning about program state is necessary to determine if a panic is possible. If the compiler does not optimize away the path to a panic state, the whole codebase fails the check, and no-panic returns the function from which the panic originated without further information about the intermediate call graph [33]. Furthermore, no-panic cannot check individual functions or modules for panic freedom, as it must compile the entire crate. On the other hand, Sniffer allows auditors to specify individual functions to analyze, and it returns the full call graph leading up to a panic to help the auditor complete the audit.

7 Evaluation

To evaluate Sniffer's value as a property-targeted auditing tool, we ask the following questions:

1. Does Sniffer find the problematic code?
2. What is the false positive rate of Sniffer analyses? As discussed in §4, Sniffer can return false positives from dynamic dispatch approximation and conservative property heuristics.
3. How does Sniffer compare to the existing audit tools Cargo Scan [53] and Cargo Geiger [12] in terms of accuracy and audit burden?

We evaluate these questions by applying Sniffer to audit four endpoints in WebSubmit, a real-world application ported to Sesame [15]; a scheduler subsystem for RedLeaf [32]; `dryoc` [19], a cryptography library that seeks to minimize unsafe code, and two crates that are known to be panic-free, `ufmt` [47] and `bitter` [9].

Baseline Audit Tools. The existing audit tools against which we compare are Cargo Scan [53] and Cargo Geiger [12]. Both find unsafe code in an application and its dependencies. Cargo Scan is a program analysis tool to help developers audit a Rust codebase for side effects including unsafe operations and system calls. In this evaluation, we apply Cargo Scan to the crate and its dependencies individually. We omit considering Cargo Scan's optional "cross-package audit" mode, which creates an approximated call graph, because it does not expand macros or resolve trait method calls for implementations in foreign crates, thus failing to report some reachable unsafe code.

7.1 Encapsulation Violations in Sesame

We applied Sniffer to the modified WebSubmit `questions_submit` endpoint described in §3.2. We inserted a call to a logging library where unsafe code breaks encapsulation, leading to a leak of sensitive data. We run Sniffer over the endpoint with the analysis set to detect possible instances of broken encapsulation on the `PCon` type. More specifically, this analysis finds calls to `transmute` or raw pointer casts on an object that is or contains the `PCon` type (see §4.2 for more). A good result would be to locate the known dangerous call to the logging library and few false positives. Sniffer identified 12 calls with potential violations:

1. The call to the logging library, which then calls `abomonation` where an encapsulation-breaking `transmute` is executed.
2. Four calls with pointer casts inside the standard library's `Iterator` and `HashMap`, which we confirmed do not violate encapsulation. If standard library collections are trusted, Sniffer could skip those calls.

Endpoint	Purpose	# Crates Analyzed	# Functions Analyzed	# Encap. Breaking Instances Found
`leclist`	display lecture menu	65	33,666	31
`composed_answers`	display answers	66	35,116	32
`questions`	display questions	66	34,634	31
modified `questions_submit`	submit answer	79	35,768	12

Figure 6: Sniffer located possible instances of broken encapsulation in the call graphs of four WebSubmit application endpoints. This represents the audit burden for each of these endpoints. In the *modified* `submit_answer` endpoint, one of these calls was the call to the encapsulation-breaking logging library. All counts are deduplicated, and we exclude calls through Sesame, which is trusted.

3. One call is a drop, e.g., destructor, on the backing store connection. The auditor would disregard this result because it does not externalize data.
4. Six other calls are false positives due to dynamic dispatch approximation. We manually checked that these functions are irrelevant and dismissed them.

Figure 6 shows the counts of false positive violations of encapsulation on `PCon`s in this modified endpoint and three other unmodified WebSubmit endpoints. These results suggest that the Sniffer encapsulation analysis focuses auditor attention on a manageable number of regions in real-world web application code.

Reduction in Audit Effort. To provide a baseline for the audit burden required to check for breaches of encapsulation with existing tools, we run the experiment described above with Cargo Scan [53] and Cargo Geiger [12]. We applied Cargo Scan and Cargo Geiger to every dependency required by the application. A good result for Sniffer would indicate that it reduces the developer effort required to check the target property as compared to these baselines. Cargo Geiger reports 33,735 potentially "used" unsafe items, while Cargo Scan reports 43,761 items to audit, in line with the developer burden reported in their experiments [53]. This contrasts with the 12 items to audit with Sniffer, and illustrates Sniffer's reduction in audit effort.

Effect of property targeting. To better understand the impact of property targeting, we also used Sniffer to find any unsafe blocks that is reachable from this endpoint. This removes the property targeting layer of Sniffer's analysis, and is thus more comparable to

Cargo Geiger and Cargo Scan's approach. We found that there are 9,982 reachable unsafe blocks. This indicates that property targeting is a critical component of the reduction in audit effort.

7.2 Interior Mutability in RedLeaf

We applied Sniffer to the custom scheduler domain in RedLeaf described in §3.3. Due to compiler versioning differences, in lieu of running Sniffer over all of the RedLeaf OS, we isolated and ported the relevant portion of the RedLeaf code, i.e., shared heap infrastructure and `RRefable` trait definition, to a separate crate, and ran Sniffer over the scheduling domain. A good result would be to highlight the single call to `borrow_mut` on the `RefCell` that is passed to the domain, and few false positives. Sniffer identified two calls with potential violations:

1. The dangerous call to `borrow_mut` on the `RefCell` passed to the scheduler domain.
2. The call to the automatically-derived `drop` implementation for the thread metadata object.

To estimate how many regions Sniffer would flag if we were to use it to audit all of RedLeaf, we manually found calls to `borrow_mut` in RedLeaf's "domains" module. We found 38 instances that Sniffer would highlight, which would be a reasonable audit burden. These calls are false positives because the relevant `RefCell`s wrap data passed *within* a domain, rather than data passed *between* domains. Our Sniffer prototype does not track variable-level data flow, so it would not be able to filter out these false positives. A more precise check for RedLeaf's target property would flag interior mutability of data derived from variables that are shared between domains.

To measure the reduction in audit effort that the property-targeting component of Sniffer's analysis conveys, we also used Sniffer to find any unsafe blocks reachable from the scheduler code. This more general analysis does not differentiate between guarantee-breaking and irrelevant unsafe code; the false positives from this analysis form a subset of those returned by tools that do not filter out unreachable unsafe code, e.g., Cargo Scan [53] and Cargo Geiger [12]. We found 31 reachable unsafe blocks, one of which is the guarantee-breaking instance of interior mutability—the dereference of the raw pointer to the contents of the `UnsafeCell` requires an unsafe block. The reduction from 31 to 2 regions to audit indicates that filtering reachable unsafe code by properties effectively reduces audit effort.

On the whole, these results indicate that Sniffer's interior mutability analysis poses a manageable audit effort and finds guarantee-breaking unsafe code with few false positives.

7.3 Unsafe Code in `dryoc`

This case study demonstrates how a developer concerned with minimizing unsafe code could use Sniffer to strengthen their confidence that any reachable unsafe is trusted. We applied Sniffer to three functions from the cryptography library `dryoc`. Figure 5 shows the results: Sniffer found 7, 20, and 54 instances of unsafe code. Most of these went beyond the uses mentioned in the library's README, which stated that "3rd party libraries used by this crate, such as those with SIMD, may contain unsafe code" [19].

Many of these functions are calls into `generic-array` [23], an array operation library that uses unsafe code for optimizations. Our audit revealed that other retrieved functions originated in the `zeroize` [52] crate to zero out memory, as well as the `subtle` [41] crate, which provides a primitive to prevent timing attacks via compiler optimizations. We inspected the code discovered. For example, the unsafe code reachable from the `crypto_sign` function include libraries for constant-time operations and memory zeroing (via unsafe, volatile operations); an elliptic curve library that uses unsafe code for performant math; and array data structure operations. These are reasonable, and the auditor would accept them after careful inspection.

Reduction in Audit Effort. An audit with Cargo Geiger reports 667 unsafe blocks in `dryoc` and its dependencies. This indicates that Sniffer significantly reduces audit burden by allowing auditors to target specific function and via its use of precise reachability analysis. Sniffer's report gives the developer a clearer idea of potential risks and they can choose to audit or replace these components.

7.4 Panic Freedom in `ufmt` & `bitter`

Both ufmt and bitter were statically confirmed to be panic-free with the tool no-panic [33]. We analyze these crates with Sniffer to evaluate the false positive rate of the panic freedom analysis. A good result for Sniffer would be to return zero or few possible panics.

Our audit confirmed the panic freedom of ufmt with zero false positives. We analyzed three core utilities in `bitter`. Figure 7 shows the results. Two of the functions passed Sniffer's analysis with zero false positives, and `read_n_bytes` returned two false positives. Both of these could be eliminated with interprocedural constant propagation. For example, `core::slice::windows` can panic if passed a window size of 0, but this call receives the constant 2. These results indicate that Sniffer's panic freedom analysis is effective as a tool to audit for panics, and its accuracy would be further improved with constant propagation.

Function	# Crates Analyzed	# Functions Analyzed	# Panics Found	# True Panics
`refill_lookahead_unchecked`	2	29	0	0
`sign_extend`	2	4	0	0
`read_n_bytes`	2	61	2	0

Figure 7: Sniffer located possible panics in a the `bitter` [9] crate, which is statically proven [33] to not panic. Sniffer encountered two false positives on `read_n_bytes`. All counts are deduplicated, and we do not allowlist any crates.

8 Discussion

Developing New Analyses. A strength of Sniffer's design is extensibility. In our experience, implementing and integrating new analyses to Sniffer's suite is low-effort, as analysis components and patterns are reusable between analyses. For example, the pattern of detecting a call to a particular standard library function is the core of both the interior mutability and panic analyses. Beyond similarities in design, additional analyses require a manageable amount of additional development: each analysis in the existing suite consists of 24-148 lines of code.

Checking Properties over Variables. Users of Sniffer auditing their systems for properties over specific variables, e.g., a `dryoc` policy that a secret key does not flow into any unsafe blocks, would benefit from variable-granularity flow analyses. Tracking data flow at the granularity of individual variables could be implemented via alias resolution facilitated by Flowistry [13].

Automated Auditing. Developers of Rusty systems that depend on very specific target properties can also incorporate Sniffer analyses to reject extension code that violates those properties. For example, Sesame could incorporate an encapsulation analysis pass into the build script that invokes the Scrutinizer static analyzer [15], and RedLeaf could replace or supplement the `RRefable` auto trait with an interior mutability analysis pass [32]. This would shift more effort to the system developer, who fully understands the properties upon which their system relies, instead of the extension developer. To make this approach feasible, Sniffer would need a more expressive DSL in order to support very precise policies and would need to further reduce false positives that would reject valid code.

9 Conclusion

Sniffer is a tool to surface dangerous unsafe code. The key challenge of this work is differentiating between banal unsafe code and that which threatens the guarantees of a system. When we investigate the effects of unsafe code that go beyond undefined behavior, what is "dangerous" is specific to a system. This is particularly pertinent for systems that depend on bonus properties that are compiler-checked in globally safe Rust, like encapsulation and immutability of references. In this thesis, we demonstrated how the guarantees of systems that depend on bonus properties can be violated even when an extension developer does not write any unsafe code themselves, and proposed more precise vocabulary to describe the scopes and unsafety status of Rust programs. Finally, we proposed Sniffer as a flexible, expressive means to maintain the guarantees of Rusty systems in the presence of unsafe code. We evaluated it on real systems and libraries, finding that Sniffer is practical and imposes lower audit burden than existing tools intended for similar use cases.

References

[1] Justus Adam et al. *Paralegal*. 2023. URL: https://github.com/brownsys/paralegal (cit. on p. 20).

[2] Artem Agvanian. "PEAR: Practical Interprocedural Analysis in Rust". Honors Thesis. 2025. URL: https://etos.cs.brown.edu/publications/theses/aagvania-honors.pdf (cit. on pp. 20, 24).

[3] Roderick Chapman et al. *Rust Contracts RFC Draft*. Accessed: Jan. 2025. URL: https://hackmd.io/@nG8Ewk1OTDS-qIUxGrXyVw/BJ7N-uRLs (cit. on p. 12).

[4] V. Astrauskas et al. "The Prusti Project: Formal Verification for Rust". In: *NASA Formal Methods (14th International Symposium)*. Springer, 2022, pp. 88–108. URL: https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5 (cit. on p. 11).

[5] Vytautas Astrauskas et al. "How do programmers use unsafe rust?" In: *Proceedings of the ACM on Programming Languages* 4 (Nov. 2020), pp. 1–27. DOI: 10.1145/3428204 (cit. on pp. 9, 10).

[6] Yechan Bae et al. "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. 7, 11).

[7] *Behavior considered undefined*. 2025. URL: https://doc.rust-lang.org/reference/behavior-considered-undefined.html (cit. on p. 9).

[8] *binary-utils*. Accessed: April 2025. URL: https://github.com/NetrexMC/binary-utils (cit. on p. 7).

[9] *bitter*. Accessed: April 2025. URL: https://github.com/nickbabcock/bitter (cit. on pp. 7, 26, 27, 31).

[10] Sol Boucher et al. "Putting the "Micro" Back in Microservice". In: *Proceedings of the 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. 6, 16, 18).

[11] Nick Cameron. *LibHoare - Simple Rust support for design by contract-style assertions*. Accessed: Jan. 2025. URL: https://github.com/nrc/libhoare (cit. on p. 12).

[12] *cargo-geiger*. Accessed: Jan. 2025. URL: https://github.com/geiger-rs/cargo-geiger (cit. on pp. 7, 27–29).

[13] Will Crichton et al. "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. 16, 18, 32).

[14] Will Crichton et al. *rustc_plugin*. https://github.com/cognitive-engineering-lab/rustc_plugin. 2025 (cit. on p. 24).

[15] Kinan Dak Albab et al. "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. 6, 7, 14, 16, 25, 27, 32).

[16] Frank Denis. *libsodium*. Accessed: Jan. 2025. URL: https://github.com/jedisct1/libsodium (cit. on p. 26).

[17] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. "Creusot: a foundry for the deductive verification of rust programs". In: *Proceedings of the International Conference on Formal Engineering Methods*. Springer. 2022, pp. 90–105 (cit. on p. 11).

[18] *Design Notes: Auto traits*. Accessed: Jan. 2025. URL: https://lang-team.rust-lang.org/design_notes/auto_traits.html (cit. on p. 13).

[19] *dryoc: Don't Roll Your Own Crypto*. Accessed: Jan. 2025. URL: https://github.com/brndnmtthws/dryoc (cit. on pp. 7, 26, 27, 30).

[20] *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. 12).

[21] 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 pp. 6, 9, 20).

[22] Robert Bruce Findler and Matthias Felleisen. "Contracts for higher-order functions". In: *SIGPLAN Not.* 37.9 (Sept. 2002), pp. 48–59. ISSN: 0362-1340. DOI: 10.1145/583852.581484. URL: https://doi.org/10.1145/583852.581484 (cit. on p. 12).

[23] *generic-array*. Accessed: June 2025. URL: https://crates.io/crates/generic-array (cit. on p. 30).

[24] Son Ho and Jonathan Protzenko. "Aeneas: Rust verification by functional translation". In: *Proceedings of the ACM on Programming Languages* 6.ICFP (Aug. 2022). DOI: 10.1145/3547647. URL: https://doi.org/10.1145/3547647 (cit. on p. 11).

[25] Rahul Kumar et al. "Verifying the Rust Standard Library". In: *Proceedings of 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. 12).

[26] Ada Lamba et al. "Cocoon: Static Information Flow Control in Rust". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1 (Apr. 2024). DOI: 10.1145/3649817. URL: https://doi.org/10.1145/3649817 (cit. on pp. 6, 16, 18).

[27] Andrea Lattuada et al. “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. 11, 12).

[28] Hayley LeBlanc et al. “SquirrelFS: using the Rust compiler to check file-system crash consistency”. In: *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 387–404. ISBN: 978-1-939133-40-3. URL: https://www.usenix.org/conference/osdi24/presentation/leblanc (cit. on p. 6).

[29] Haoran Ma et al. “DRust: Language-Guided Distributed Shared Memory with Fine Granularity, Full Transparency, and Ultra Efficiency”. In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 97–115. ISBN: 978-1-939133-40-3. URL: https://www.usenix.org/conference/osdi24/presentation/ma-haoran (cit. on p. 6).

[30] Ian McCormack et al. *A Mixed-Methods Study on the Implications of Unsafe Rust for Interoperation, Encapsulation, and Tooling*. 2024. arXiv: 2404.02230 [cs.SE]. URL: https://arxiv.org/abs/2404.02230 (cit. on p. 10).

[31] Frank McSherry. *Abomonation: A high performance and very unsafe serialization library*. Accessed: Jan. 2025. URL: https://crates.io/crates/abomonation (cit. on p. 17).

[32] Vikram Narayanan et al. “RedLeaf: Isolation and Communication in a Safe Operating System”. In: *Proceedings of the 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. 6, 7, 12, 14, 16, 17, 25, 27, 32).

[33] *no-panic*. Accessed: April 2025. URL: https://github.com/dtolnay/no-panic (cit. on pp. 26, 30, 31).

[34] Tomasz Nowak et al. *Semver violations are common, better tooling is the answer*. Accessed: Jan. 2025. Sept. 2023. URL: https://predr.ag/blog/semver-violations-are-common-better-tooling-is-the-answer/ (cit. on p. 13).

[35] Aurojit Panda et al. “NetBricks: Taking the V out of NFV”. In: *Proceedings of the 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. 6, 16, 18).

[36] Matthew M Papi et al. “Practical pluggable types for Java”. In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. 2008, pp. 201–212 (cit. on p. 12).

[37] *RFC: Stabilize Marker Freeze - Drawbacks*. Accessed: Jan. 2025. URL: https://hackmd.io/@rust-lang-team/SyRlhj0u0#Drawbacks (cit. on p. 13).

[38] *rust-lang/miri: An interpreter for Rust's mid-level intermediate representation.* 2025. URL: https://github.com/rust-lang/miri (cit. on pp. 7, 9, 10).

[39] *Safety comments policy - Standard library developers Guide.* Accessed: Jan. 2025. URL: https://std-dev-guide.rust-lang.org/policy/safety-comments.html (cit. on p. 9).

[40] *Special types and traits - The Rust Reference.* Accessed: Jan. 2025. URL: https://doc.rust-lang.org/reference/special-types-and-traits.html#auto-traits (cit. on p. 12).

[41] *subtle.* Accessed: June 2025. URL: https://crates.io/crates/subtle (cit. on p. 30).

[42] The Rustonomicon. *Data Races and Race Conditions.* Accessed: Jan. 2025. URL: https://doc.rust-lang.org/nomicon/races.html (cit. on p. 6).

[43] The Rustonomicon. *How Safe and Unsafe Interact.* Accessed: Jan. 2025. URL: https://doc.rust-lang.org/nomicon/safe-unsafe-meaning.html#how-safe-and-unsafe-interact (cit. on p. 6).

[44] The Rustonomicon. *Transmute.* Accessed: June 2025. URL: https://doc.rust-lang.org/stable/nomicon/transmutes.html (cit. on p. 22).

[45] David Tolnay. *Adding interior mutability into a type is already an API breaking change.* Accessed: Jan. 2025. URL: https://github.com/rust-lang/rust/issues/60715#issuecomment-491533461 (cit. on p. 13).

[46] *Trait std::marker::Freeze.* Accessed: Jan. 2025. URL: https://doc.rust-lang.org/std/marker/trait.Freeze.html (cit. on p. 12).

[47] *ufmt.* Accessed: April 2025. URL: https://github.com/japaric/ufmt (cit. on pp. 7, 26, 27).

[48] *UNSAFE_CODE in rustc_lint::builtin - Rust.* Accessed: Jan. 2025. URL: https://doc.rust-lang.org/beta/nightly-rustc/rustc_lint/builtin/static.UNSAFE_CODE.html (cit. on p. 7).

[49] *UnsafeCell in std::cell - Rust.* Accessed: April 2025. URL: https://doc.rust-lang.org/std/cell/struct.UnsafeCell.html#:~:text=UnsafeCell%20opts%2Dout,UnsafeCell%20to%20wrap%20their%20data. (cit. on p. 22).

[50] *Unsafety Checking - Rust Compiler Development Guide.* Accessed: April 2025. URL: https://rustc-dev-guide.rust-lang.org/unsafety-checking.html (cit. on p. 22).

[51] Alexa VanHattum et al. "Verifying Dynamic Trait Objects in Rust". In: *Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* 2022, pp. 321–330. DOI: 10.1145/3510457.3513031 (cit. on pp. 7, 11, 12).

[52] *zeroize.* Accessed: June 2025. URL: https://crates.io/crates/zeroize (cit. on p. 30).

[53] Lydia Zoghbi et al. “Auditing Rust Crates Effectively”. (Preprint). 2024. URL: https://web.cs.ucdavis.edu/~cdstanford/doc/2024/CargoScan-draft.pdf (cit. on pp. 7, 27–29).