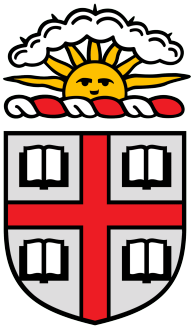
Sniff-Test: Lightweight, ergonomic property checking

Alex Portland

Advisor: Will Crichton

Reader: Shriram Krishnamurthi



Department of Computer Science

Brown University

Providence, RI

April 28, 2026

Contents

1 Introduction

Developers care about a variety of code properties in the software they create. For instance, most developers care about undefined behavior, as it can introduce everything from subtle functionality bugs & misbehaviors to full program crashes by rendering a program's semantics meaningless. Others who are creating safety- or mission-critical software might especially care about the absence of panics at runtime which could kill their process and have disastrous consequences.

As a consequence of this, developers of third-party libraries also care about ensuring that these properties hold for the APIs they implement or, at the very least, documenting the way their code can violate them to keep end-user developers informed and able to uphold the properties they care about with the provided APIs.

Such developers may use programming languages which provide certain guarantees about such properties. For instance, Rust is a popular programming language whose strong type system and compiler can statically prevent undefined behavior at compile time. However, this comes with the caveat of only being true for a safe subset of the programming language, beyond which the compiler's analysis makes no guarantee. In practice, most production codebases use more than the safe subset for performance, foreign function interoperability, or low-level hardware access [1]. Furthermore, there are properties (like the aforementioned panic-freedom) that the compiler's analyses do not check.

1.1 The Problem

This begs the question of how developers can best ensure that these desired properties are upheld in their code.

Existing code analysis approaches largely exist along a spectrum from heavy- to light-weight analyses based on both the thoroughness with which they aim to check for properties and, thus, the complexity of their analysis techniques.

On the heavy-weight side of the spectrum, verification techniques aim to formally prove that code cannot panic at runtime, cannot exhibit undefined behavior, or cannot be functionally incorrect. These techniques are impressive and very useful, but run into at least one of two fundamental issues. First, these tools have to translate your code into a logical model that can then be verified, which often leads to them not being able to represent a large portion of complex data structures and language features that are ubiquitous in production code. Secondly, they typically require high developer effort in writing some form of proofs or specifications in order to make verification more tractable and help the tool scale more effectively.

So, in practice, many organizations will just invest in verifying the core functionality of their codebases and rely on lighter-weight methods for the rest. The most common light-

weight methods in practice are code lints, tools which aim to consolidate code quality by enforcing consistent formatting, denying problematic operations or suggesting more idiomatic expressions of common code patterns. To these ends, the local analyses they employ tend to be effective as they are both fast and scalable. But when applied to the more complex task of checking code properties that can inherently be propagated between multiple layers of function calls, local linting analysis tends to miss many issues that would be caught by more exhaustive methods.

Thus, current developers that wish to check that their libraries or binaries exhibit certain properties wind up stuck choosing between thorough but impractical verification tools and lints that are ineffective in finding actual issues.

1.2 Proposed Solution

This thesis proposes that there should be a class of analysis tool in the middle ground, aiming to be both practical and expressive enough for use in production codebases. We propose `Sniff-Test`, a lightweight, inter-procedural analysis tool for Rust that checks whether developers have correctly documented and propagated the conditions under which key code properties hold.

`Sniff-Test`'s approach is heavily informed by the existing documentation convention in the Rust ecosystem. For instance, it is considered best practice to add a `# Safety` section to any `unsafe` function's doc comment, detailing the conditions that the function's callers must uphold to avoid undefined behavior. Similarly, functions that can panic under certain conditions are expected to have a `# Panics` section detailing them. And, at the call sites to such functions, thoughtful developers will already write `// SAFETY: ...` or `// PANICS: ...` to record their reasoning as to why the callee's conditions have been properly upheld at that call site. `Sniff-Test` builds an automatic checking framework around these existing documentation practices.

Critically, `Sniff-Test` does not aim to prove the absence of undefined behavior or panics. Instead, it aims to ensure that developers have thought about, documented and propagated the conditions under which such properties hold reporting issues where they have not been. Although this is a weaker guarantee than that provided by formal verification techniques, we posit that lack of proper consideration is the cause of a large number of property violations, rather than considering them incorrectly. Furthermore, these techniques can be applied to the full breadth of a codebase rather than a carefully selected core with minimal proof writing or specification beyond the documentation that developers should already be writing.

This thesis contributes:

1. A novel documentation-based checking model that ties into existing documentation convention in the Rust ecosystem, allowing `Sniff-Test` to check code properties with little-to-no code modifications.

2. An inter-procedural call graph analysis algorithm, including over-approximate strategies for practically handling higher-order functions and closures — two constructs that are common in Rust code but complex for property analysis.
3. A prototype implementation of `Sniff-Test` as a Rust compiler plugin.
4. Case studies evaluating the use of our prototype on two production Rust crates — `indexical` (1,608 LoC) & `zerocopy` (11,207 LoC, maintained by Google) — demonstrating that `Sniff-Test` is fast and practical while surfacing documentation gaps that are missed by similarly practical alternatives.

2 Background & Related Work

2.1 Rust Safety Properties

Rust is a systems programming language which is designed with safety as a first-class principle [2]. Its type system and borrow checker's static analyses allow the compiler to guarantee memory safety and the absence of undefined behavior for code that is written in the *safe* subset of the Rust language [1]. However, many production codebases require using the `unsafe` portion of the language for a variety of reasons, including performance, operating with other pieces of code, and more [1]. When using `unsafe` code, the compiler cannot statically guarantee the absence of undefined behavior, and it's on the developer to ensure their code is UB-free.

Furthermore, there are a significant number of properties which are of importance to developers, but that the Rust compiler cannot check for, even in the *safe* subset of the language. For instance, many developers care about panic-freedom, as it allows them to ensure a program cannot abort execution at runtime. Panics typically result in the executing thread being killed and, although Rust does allow you to specify custom panic handlers, they are very difficult to recover from. This makes them especially dangerous for mission-critical contexts like embedded systems where there typically aren't even the resources to have a panic handler [3]. Other properties like blocking-freedom (ensuring async `Futures` do not call blocking functions, which would stall the async runtime's threads) and cancellation safety (ensuring `Futures` can be safely dropped mid-execution without breaking program correctness) are similarly unchecked by the compiler, but important to developers for the correctness and liveness of their code. Current work on `Sniff-Test` focuses on UB-freedom and panic-freedom, but blocking-freedom and cancellation-safety represent natural extensions to `Sniff-Test`'s current work.

2.2 Dynamic Methods

Dynamic analysis techniques find property violations by running code and observing its behavior in motion. Miri [4] is a Rust interpreter that executes programs at the MIR level and

detects undefined behavior on the executed paths, including invalid pointer dereferences and use-after-free. Fuzz testing tools automatically generate large numbers of inputs and check for crashes or panics, and have proven effective at finding bugs in practice. However, both approaches are fundamentally limited to the executions they observe. For library APIs with complex preconditions or very specific failure conditions, the input space is often too large to fully explore.

2.3 Formal Verification

In terms of static analysis, formal verification tools aim to exhaustively prove that a program satisfies a given property for all possible inputs and executions. Bounded model checkers like Kani [5] use symbolic execution to implicitly enumerate this search space, conclusively proving properties like UB-freedom, panic-freedom and even the functional correctness of your code. Tools like Verus [6] offer similar guarantees by checking developer-written proofs and verifying their correctness.

However, formal verification runs into a pair of limitations that restrict its practical use on large codebases. First of all, these tools must inherently translate Rust code into a logical model that can be verified. Fundamental gaps between their simplified logical model and the operations that a complex language like Rust provides in practice mean that they're often unable to represent large portions of complex data structures and language features that are ubiquitous in production code. For instance, Kani cannot support common Rust data structures like `HashMap` and cannot handle any `async` code because of a lack of concurrency within its logical model.

Second, they typically require developers to write proofs or specifications alongside their code, something that requires a significant investment of time and expertise. In practice, organizations tend to wind up selectively using formal verification only on their most critical functionality, relying on lighter-weight methods for the rest of their large codebases.

2.4 Linting and Auditing Tools

Code lints represent the lightest-weight approach to property checking. Clippy, the standard Rust linting tool, includes checks for some property-relevant issues: `clippy::missing_safety_doc` flags `unsafe` functions without a `# Safety` section, `clippy::missing_panics_doc` flags functions that can directly panic without a `# Panics` section, and `clippy::undocumented_unsafe_blocks` [7]. These checks are fast and require minimal developer effort. However, Clippy's analyses are strictly local, examining each function in isolation and not finding issues that propagate through multiple layers of function calls. Furthermore, these lints don't generalize to detecting issues besides the small number of operations that are hardcoded to be problematic for UB-freedom and panic-freedom respsectively.

3 Design

We now describe the core design of `Sniff-Test`'s analysis. Section 3.1 details the full documentation model for describing the conditions under which properties hold, largely a systemization of existing convention in the ecosystem. Section 3.2 describes the fundamental operations the tool knows to be problematic for certain properties, and 3.3 explains how those properties are then propagated throughout a codebase, resulting in a full analysis. Section 3.4 explains how `Sniff-Test` builds its call graph for that propagation, along with som interesting challenges with building a call graph for the specific purpose of tracking property propagation.

3.1 Documentation as Specification

The Rust ecosystem has already converged on existing documentation conventions that work well in practice, as they allow developers to express why given conditions on the operation were satisfied inline in a way that is easy for themselves and others to understand. The core insight behind `Sniff-Test` is that these closely match the property information that a property-checking tool needs so, rather than introducing a new specification language or formalize, the tool codifies existing conventions into a checking model.

Consider a Rust function that just dereferences a raw pointer:

```
unsafe fn deref(ptr: *const u32) -> u32 {
    *ptr
}
```

This function will have UB if the pointer is null (or if it is misaligned or has invalid provenance, but let's focus on the pointer being non-null for now). The Rust ecosystem's existing convention is for the `deref` function to have a `# Safety` section in its doc comment, specifying the conditions that callers must uphold to avoid undefined behavior. In `Sniff-Test`, we often call these conditions the function's *obligations* as they're conditions the callers are obligated to uphold.

```
/// # Safety
/// `ptr` must be non-null.
unsafe fn deref(ptr: *const u32) -> u32 {
    *ptr
}
```

At call sites to unsafe functions, it is similarly convention to write a `// SAFETY: ...` comment on the line preceding the call to explain why the documented conditions are satisfied for that specific call. For instance, in the below example, we're checking that `ptr` is non-null and then have documented the call to `deref` with our reasoning that, since we just checked that the pointer isn't null, it can't be null! We call these comments on call

sites *justifications*, as they are justifying why the obligations of a given function have been satisfied and thus why the desired property should hold.

```
if !ptr.is_null() {
    // SAFETY: we just checked that ptr is non-null.
    let _ = unsafe { deref(ptr) };
}
```

This example may seem painfully obvious, but the obligations placed on callers can be much more complicated and less straightforward to check in practice (as even the previous example would be if we're fully considering alignment and pointer provenance).

Similarly, for functions that can panic on certain inputs, it's convention to place a `# Panics` section in the function's doc comment detailing the conditions under which it will execute panic free. Then, similarly, developers will place `// PANIC: ...` comments on call sites to such functions explaining why that call site ensures the non-panicking conditions are met.

Here, it is important to reiterate that the goal of our analysis is to detect *missing documentation* that fails to consider a property's obligations, rather than incorrect documentation that is simply reasoning about those obligations incorrectly. Thus, the base version of this analysis does nothing to reason about the contents of either justifications or obligations, although extensions to this are discussed in 7.

3.2 Base Case Obligations

`Sniff-Test`'s analysis is built on a set of base case obligations: primitive operations that the tool knows to be the potential source of a property violation. For example, the tool might know that a call to Rust's `core::panic()` function will unconditionally panic (and thus the obligation to ensure panic-freedom is to ensure that it is never called in practice), or that array indexes can panic if the index is out of bounds (resulting in the obligation that the provided index must always be in bounds).

The core of `Sniff-Test`'s analysis is to track as these base case obligations propagate up through a codebase, and it's job is to ensure that they are documented or discharged at each step using the documentation patterns described in 3.1.

As such, the tool is largely agnostic to the base case axioms used during analysis, allowing them to be modularly adjusted to support new language features or properties. In the current version of the tool, we use a simple set of one axiom for undefined behavior and three for panics, as detailed in 6. This seems to work well in practice, but the set could be expanded as the tool matures.

3.3 Propagation Rules

Given a set of base case obligations that the tool knows for primitive operations, `Sniff-Test` propagates those obligations upwards through the call graph using a small number of rules. `Sniff-Test`'s analysis works on a single property at a time, combining multiple disjoint passes to check for different properties.

The analysis begins from a set of entry points designated by the user (as explained in section 3.4.2) and walks the call graph of reachable functions in a depth first search ordering. At each function body, it first finds all the obligations within a function's body. We say the function originates obligations wherever it uses an operation that the tool knows to have one of the base case obligations from section 3.2 (e.g. doing a raw pointer dereference). And the function also inherits obligations whenever it calls a function that itself has propagated obligations to its callers, something determined by checking the callee's doc comment for whichever `# PROPERTY` section corresponds to the given property.

If the function has obligations from either of these sources, it must satisfy one of two following conditions:

- **Declared**: The function itself is documented as having obligations on its callers. In this case, since we do not analyze the content of documentation and just its existence, we trust that the obligations have been properly propagated from all sources when documenting this function.
- **Discharged**: The function has placed an inline justification comment at the source of every obligation it contains.

The tool emits an error for each function only if neither of these conditions are met, pointing out the unjustified obligations that either need to be declared or discharged at their sources.

3.4 Call Graph Construction

The key part of `Sniff-Test`'s analysis that differentiates it from the local analyses of a lint is the fact that it builds a call graph of the target code to detect where properties have not been properly considered even multiple function calls deep.

3.4.1 Entrypoints

We start building the analysis' call graph from a number of entry points, as specified by the user. This is similar to how users can opt in to specific lints on a per-function or per-workspace granularity for existing Clippy-style lints, with the notable difference that annotating a function as a `Sniff-Test` entry point also implicitly will analyze all of its callees as well. This means the actual granularity of `Sniff-Test` is at the level of call trees, rather than simply functions.

Scoping the analysis to just the functions reachable from these entry points serves two key purposes. First, it makes `Sniff-Test`'s analysis more tractable by focusing it on slices of the codebase that the developer using the tool cares about, rather than all function bodies that exist. Entire call trees may just be used for testing infrastructure or otherwise not be exported to a library's users, so it would not make sense to analyze all of them. Additionally, the configurability of specific entry points makes it significantly easier to port an existing tool to use `Sniff-Test`. For large codebases that haven't already been following this documentation convention, it is much more efficient to enable `Sniff-Test`'s analysis incrementally, fixing reported errors as they appear, rather than being forced into a binary of full analysis across the entire codebase.

```
// Enable panic-freedom checking for all public functions in
// the crate and their callees.
#![sniff_test::check_panics_pub]

// Enable panic-freedom checking just for this function and
// its callees.
#[sniff_test::check_panics]
pub fn into_chunks(data: &[usize], chunk_sz: usize)
    -> Vec<&[usize]> {
        // ...
}
```

This is made possible by annotations that the `Sniff-Test` user can add to their code to check for a given property in just a single function and those it calls, or all public functions within the codebase.

3.4.2 Detecting Call Edges

`Sniff-Test` will then do a depth first search analysis starting from these entry points to keep adding new call edges it detects within reachable functions to its call graph, recursively analyzing newly reachable functions until all have been analyzed for reachability.

The key question in a reachability analysis like this is how call edges are detected, as this will inform the tradeoffs and effectiveness of the analysis.

For the most part, `Sniff-Test` takes a very simple approach to call graph construction. It looks at the monomorphized bodies of all reachable functions, and adds an edge to the call graph for each call to a function that it finds where the call is already statically resolved. This approach is very simplistic and, with certain code patterns, can clearly miss call edges, mistakenly ignoring functions that are actually reachable from a given entry point. However, as will be discussed in our evaluation, we've found that this approach works well in practice on production codebases (where complex dispatch patterns exist, but are not the common case) for `Sniff-Test`'s goals.

Some analyses will try to resolve or over-approximate dynamic dispatch to build a conservative call graph of everything that is possibly reachable from a given function. This is good for ensuring soundness of the resulting analysis, but will add edges to the call graph that might never be taken in practice. `Sniff-Test` does not aim to have any guarantees of soundness in its analysis, it simply aims to fulfill its specific role of checking for missing documentation effectively and quickly. Because of this, trying to resolve dynamic dispatch or doing value tracking to fully resolve some of the more complex dispatch patterns that can arise in Rust code is a non-starter for the tool. Doing so would heavily complicate and slow down our analysis to an unreasonable degree, given our analysis isn't deeply concerned with soundness or the additional granularity they can provide. Thus, the current iteration of the tool simply warns its user when encountering function pointers or dynamic dispatch that it can't reason about.

3.4.3 Higher-order Functions

That being said, there exists code patterns along these lines which are ubiquitous in the Rust ecosystem, and important to any analysis of property violation.

Consider the following code snippet, which takes in a Rust iterator of strings and returns a Rust iterator of numbers by applying a parsing closure to each element. Within the closure, we're calling `.unwrap()` on the `Option` returned by the parsing, as some strings won't validly correspond to a number once parsed. This call to `unwrap` will panic if the string was unable to be parsed and, thus, the closure can definitively panic.

```
#[sniff_test::check_panics]
pub fn parse_string_iter(iter: impl Iterator<Item = String>)
    -> impl Iterator<Item = usize> {
        iter.map(|str| str.parse::<usize>().unwrap())
}
```

However, how should this be handled within `Sniff-Test`? Since iterators in Rust are lazily evaluated, this call to `Iterator::map` never actually calls the closure. Instead, it just stores the closure away within the iterator so that it can be applied to each element once the iterator is eventually used. In order to track that the iterator returned by this closure can now panic, we'd likely need to do some sort of value tracking–recording that the closure stored with this iterator can panic, so that subsequent calls to `Iterator::next` will recognize that a panicking closure is called, and thus the call to `next` could panic. For the reasons stated above, doing so would heavily complicate our analysis and likely compromise its scalability, but being unable to detect this use of a panicking closure could lead to many missed issues in practice.

To resolve this tension, we lean on an over-approximation to make analysis more feasible. We make the observation that it is incredibly rare for a closure that's passed into a function to not

be eventually called. Thus, we assume that, regardless of the body of a function, it can call any closures that are passed in to it. Thus, in this case, we add an additional over-approximate call edge from the call to `Iterator::map` to the parsing closure. Now, this causes the call to `Iterator::map` inherit the obligations of the closure, so the user must either a) add an inline justification above that call justifying why the closure cannot panic in practice or b) propagate the obligation to callers of `parse_string_iter` (namely that all elements of the input iterator must be valid to parse as a `usize`).

In Rust, closures each have a unique type that implements one of Rust's function traits (`Fn`, `FnMut`, or `FnOnce`) so, in order to take in different closures as arguments, a function like `Iterator::map` must be generic over implementors of one of those function traits. Thus, detecting the closure that's passed in to a given function is possible by looking for closure types within that function's generics. And, since each closure type is unique, once you have the type of the closure, you can find where it's defined for analysis.

However, closures that don't capture any of their environment can be coerced from their unique closure types into a function pointer type and closures can also be cast into a dynamic trait object for the function trait (for example through a `Box<dyn Fn()>`). In both of these cases, we would need some sophisticated value analysis to determine which closure could be referred to by a specific function pointer or trait object as the type information has been lost. `Sniff-Test` again decides not to handle this and simply warns about the dispatch it cannot fully determine. That being said, closures will remain as their unique type unless coerced, so higher-order functions that directly take in a closure (like above) will be fully analyzable with this method, and that seems to be the most prevalent pattern in practice.

4 Implementation

We've developed a prototype implementation of `Sniff-Test`—a Rust compiler plugin implemented in 2.2k significant lines of Rust code. The prototype is open source and available at `https://github.com/cognitive-engineering-lab/Sniff-Test`.

4.1 Call Graph Construction (MIR)

When running on a target crate, `Sniff-Test`'s implementation of its analysis is roughly structured into two phases: constructing the call graph (this section!), checking each of the found reachable functions (section 4.2), and reporting consistency issues that were found (section 4.3).

When building the call graph, our prototype uses Rust's optimized mid-level intermediate representation (MIR), as it represents a post-optimization and post-monomorphization byte code that is the state of a program right before LLVM codegen begins.

The structure of Rust's MIR as a byte code-like intermediate representation actually makes this call graph construction easier than if it was done at a higher level. The MIR is broken down into basic blocks, with each basic block having a set of statements that will execute sequentially with no control flow between them. Control flow can only happen at the end of a basic block, with its terminator. This means that, to detect direct calls, we just need to look through the terminators of each basic block within a function, filtering for just terminators that will result in a function or method call. Since this stage of the compiler is post-optimization, some calls that the compiler was able to statically determine are unreachable (e.g. through a constant propagation / dead code elimination pass) will have already been eliminated, removing some false positives. This is also the stage at which we look for generics that are of a closure type, and add the corresponding conservative call edge as discussed in section 3.4.3.

4.2 Checking Reachable Functions (HIR)

Once the call graph has been constructed, we then use that data structure to check all reachable functions. Doing so requires access to the user's documentation, both for their function definitions (e.g. for finding the obligations in a `# Panics` section) and inline on expressions (e.g. for justifications like `// PANICS: ...`). Doc comments on function definitions are maintained fairly deep into Rust's compilation pipeline, and they are still available for retrieval even once code has been lowered to the MIR level. However, inline comments on expressions are lost in the early parsing stages of the pipeline.

Thus, the current iteration of the tool requires inline comments to be changed to doc comments by adding an additional slash, as doc comments are maintained until the high-level intermediate representation (HIR) level of the compiler, which is an IR one stage before the MIR we use for reachability. The structure of the HIR is much more similar to an AST than the MIR's byte code, so our prototype then uses the Rust compiler's internal `Span`s (its representation of locations in source code) to map every call edge it detected back to the HIR node that would have generated it. This allows us to finally up the HIR tree to robustly find documentation that is placed on function calls or operations with base case obligations, even if the documentation is placed on a surrounding statement or block rather than exactly on the call itself.

This approach allows documentation to be properly detected in the first two of the following examples, which is a ubiquitous pattern in the existing documentation ecosystem.

```
/// PANICS: ...
let result = some_panicking_fn();
// ^^ Sniff-Test detects this

/// PANICS: ...
let result = {
```

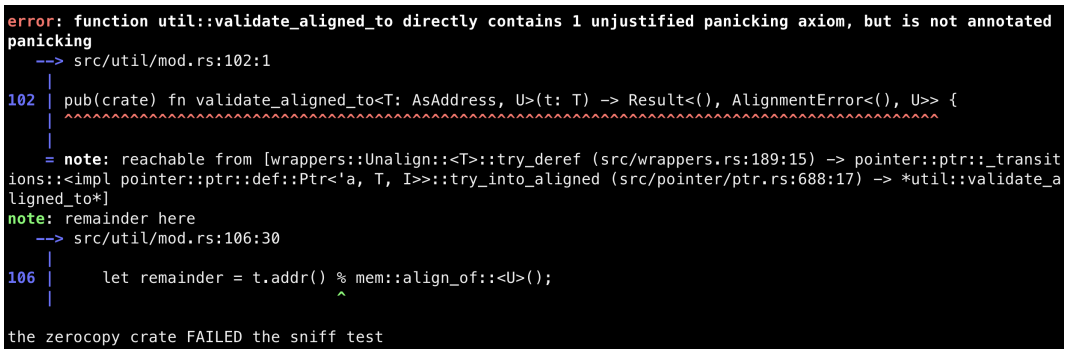


Figure 1: A sample `Sniff-Test` error message

```
    some_panicking_fn()
};
// ^^ Sniff-Test also detects this

let result = /* PANICS: ... */ some_panicking_fn();
// ^^ Which is important because this is not what developers
// typically write in practice
```

Making inline comments into doc comments is a fairly non-intrusive and automate-able change (just changing a `// PANICS: ...` into a `/// PANICS: ...`), but an improved version of this tool could hook directly into the parser to avoid this additional user work.

4.3 Reporting Errors

A key advantage of `Sniff-Test` being implemented as a Rust compiler plugin is that, once issues have been found, the prototype can leverage the compiler's robust diagnostics facilities to convey informative error messages to the user. Not only does this allow for our errors to highlight the relevant places in the source code where obligations are coming from, but it also is in the same format that Rust developers are already familiar with (from both the compiler and Clippy), making it much easier to read and digest quickly. A sample error message from our prototype is displayed in figure 1, but with more time and effort, the output could be better optimized for user understanding.

4.4 Limitations

The current `Sniff-Test` prototype has two main limitations.

First of all, our call graph construction is not fully robust to the complex patterns that can be seen in practice. Although it is able to run on some large production codebases, current techniques we use to resolve paths aren't able to fully support the kinds of paths you'll see in production code. Fixing this is just a matter of running on more and more crates to see where the assumptions we've implicitly made in our implementation are breaking down.

With every fix, our analysis should become more generalizable and better able to support the full spectrum of possible Rust code.

Additionally, the current iteration of the tool provides only options for checking your own local crate, or checking all crates in your dependency tree individually. This is not ideal for users, as an issue three layers deep in their dependency tree is good to know, but it's not very feasible to open a PR for every documentation issue for every crate you want to use. On the other hand, some checking of dependencies is important, as (assuming the dependency's developers don't themselves use `Sniff-Test`) their public API could itself have undocumented obligations that may end up causing a property violation in your own code if unfulfilled. We're working on a middle-ground for dependency checking which would warn users about issues in their dependencies, but only at call sites where the user is calling a problematically undocumented API within a dependency. This would give users full information about how undocumented obligations could be affecting their own code through those call sites, but without them having to eagerly fix all documentation issues within all of their dependencies. Implementation for this is still ongoing, but we are confident that this will result in a more effective tool when complete, with minimal additional false positives.

5 Case Studies

To evaluate `Sniff-Test`'s effectiveness and practicality, we ran the tool on two real-world production crates from the Rust ecosystem. These crates were selected to provide a representative sample of the kinds of code patterns `Sniff-Test` will encounter in practice, despite the small sample size. One crate is primarily focused on panic-freedom, was initially developed without explicit regard for documenting code properties, and uses limited unsafe code, while the other is more focused on UB-freedom, with complex unsafe code invariants and a very strong existing documentation culture. Together, they span an order of magnitude in codebase size, both properties our prototype targets, and two very different relationships developers can have with the documentation conventions `Sniff-Test` builds on.

Below, we discuss the codebases, what makes them a good fit as tests for our tool, and the small modifications required to make them suitable for `Sniff-Test`'s analysis. Ideally, the tool would be able to run on codebases without modification (beyond adding annotations to mark analysis entrypoints), but non-inherent limitations of our prototype do require some minor changes on some codebases.

5.1 `indexical`

`indexical` is a Rust library providing efficient bit-set data structures. It is open source at `https://github.com/willcrichton/indexical`. At 1,608 LoC, it is the smaller of our two case studies. The crate makes heavy use of arithmetic and indexing operations, with minimal unsafe code use, so its developers were most concerned about how those

operations could violate the panic-freedom property of the API provided to their users. Unlike `zerocopy`, `indexical` had relatively little existing panic documentation, making it a good test of `Sniff-Test`'s ability to find undocumented obligations in a codebase that had not previously explicitly prioritized this convention.

5.1.1 Porting `indexical`

In order to get `Sniff-Test` to run on `indexical`, we had to add the two lines to mark all public functions as entrypoints for analysis for both the panic-freedom and UB-freedom properties. Then, all we had to do was change one existing inline `// SAFETY: ...` comment to be a doc comment. This was not difficult, but does indicate the lack of existing inline documentation in the crate, especially when contrasted with the number of occurrences that required changing in `zerocopy`, even when taking into account differences in codebase size.

5.2 `zerocopy`

`zerocopy` is a library developed and maintained at Google for its byte manipulation that is both fast and safe, something so commonly useful that it has become a core part of the Rust ecosystem. It is also open source at `https://github.com/google/zerocopy`. At 11,207 LoC, it is substantially larger than `indexical`, and its ethos as a "safe" implementation of byte manipulation that so inherently relies on unsafe code causes its developers to be very concerned about the UB-freedom property. Additionally, `zerocopy` already had a very strong culture of safety documentation, with much of its documentation on function definitions reading like specifications or requirements, while inline comments often read like formal proofs those requirements have been met. This makes it a complimentary case study to `indexical` — rather than testing whether `Sniff-Test` can bootstrap documentation from scratch, it tests whether the tool can find gaps even in a mature codebase that takes safety documentation very seriously.

5.2.1 Porting `zerocopy`

Because of its size, getting `Sniff-Test` to run on `zerocopy` was more difficult than getting it to run on `indexical`, but the effort was still very reasonable. This required us to again add the two lines of `Sniff-Test` annotations to mark all public functions as entrypoints for analysis for both panic-freedom and UB-freedom. We then had to change 2.7k lines of comments from inline comments using '//' to doc comments with '///' that would remain present for our prototype to pick up during later compiler stages. This incredible amount of documentation (and only counting inline comments) is another strong signifier of the impressive documentation culture of this crate's developers, and their desire to get these properties right. Conversion of those comments was completed using a Python script in <10 minutes of total work but, as mentioned in section 4.2, this user effort could be avoided by an improved prototype with access to the Rust parser.

We then noticed a number of `# Safety` sections in documentation on function definitions which, although discussing safety in general, were not referring to specific safety obligations that should be placed on callers of the function. Most commonly, this would be used to tell callers of the function invariants that the function would uphold, allowing the caller to rely on them for their own safety. We had to replace 23 instances of these sections with a new `# Safety Invariants` section to delineate that they were referring to invariants a function was telling callers **it would uphold**, rather than obligations that a function was telling callers **they must uphold**. After this, `Sniff-Test` could run on the crate, the results of which are analyzed in section 6.

6 Evaluation

We evaluate the issues our `Sniff-Test` prototype is able to find on the case studies in section 5, hoping to find additional instances of missing property documentation when compared with existing lints, with an acceptable false positive rate.

6.1 Setup

For this evaluation, we use a small set of four operations (listed below) which we have told the tool have base case obligations. This set could be expanded as the tool matures but, for now, the goal was to show how, even with a small number of base case obligations, those obligations could be effectively propagated through a codebase to find missing documentation.

Table 1: The `Sniff-Test` Prototype's Base Case Obligations

Operation	**Property**	**Obligation(s)**
Array indexing	panic-freedom	The index must be in bounds.
Division & remainder	panic-freedom	The divisor must not be zero.
`panic!()` macro	panic-freedom	Unconditional violation (This cannot be called).
Raw pointer dereference	UB-freedom	The pointer must be non-null, aligned, and have valid provenance.

When run on each crate, the `Sniff-Test` prototype reported a number of errors. We then manually classified them into three categories based on their importance to developers, the cause of the error, and whether it was propagated to the crate's public API:

- **False positives** (due to tool limitations) → Places where a lack of documentation was flagged, but only because of a limitation in our prototype. For example, if a panic is flagged as reachable just because of call-graph construction compromises that could

be fixed with additional effort. These 'errors' could be safely ignored by developers, as they are not possible in practice.

- **Internal documentation improvements** → Places where developers were implicitly considering the obligations of called functions, but were not explicit in recording this through documentation. These errors should be considered by thorough library developers who wish to fully document the internal logic of their code with regard to the desired properties, but they have no effect on the library's users as they don't propagate to the public API.
- **User-facing documentation bug** → Places where a lack of documentation was flagged and, upon review, the property's obligation should have been propagated all the way up to the users of the library through its public API. These errors should be fixed by developers, as they represent undocumented property violations that are possible from the library's public API.

Note that the tool could theoretically also find concrete property violations if run on an application's main function rather than on a library. However, since our test crates were both libraries, having the documentation propagate all the way up to the public API is the most severe property issue possible (beyond causing a property violation directly from within your library's code regardless of user input).

6.2 Results

When `Sniff-Test`'s prototype is run on both of our case studies for both UB-freedom and panic-freedom, and we classify output issues as detailed above, we get the following results. The number in parenthesis is the number of `Sniff-Test`'s issues that were also found by Clippy, a leading Rust linter. Clippy caught no issues that were not also caught by our `Sniff-Test` prototype.

Table 2: Classification of Flagged Issues Across Analyzed Crates

Category	`indexical`	`zerocopy`
Tool limitation false positives	0	31
Internal documentation improvements	6 (4)	75 (0)
User-facing documentation bugs	37 (2)	1 (0)

Clippy is able to find a small number of safety issues within `indexical`, as it has lints to find `unsafe` blocks that are missing an inline justification to explain why they avoid UB in practice. However, its local analysis finds no issues for panics, for which there is no such keyword or language feature to easily lint for. `Sniff-Test`, on the other hand, finds a variety of issues across both properties in both crates.

As could be expected, `indexical` had many more user-facing documentation bugs, as documentation was not as core a part of the library's development ethos as it was for `zerocopy`. That being said, `Sniff-Test` does find 16 internal safety issues within `zerocopy`, and 62 panic issues. `Sniff-Test` caught all of the known undocumented unsafe blocks that the `zerocopy` developers had recorded as part of an effort to document every unsafe block (see `zerocopy` issue #429). But, more than that, it also caught additional cases that weren't tracked by the effort due to the fact that they came from macro expansions not tracked by Clippy lints. Macro expansions also notably inflate the number of issues `Sniff-Test` reports. For instance, 14 of the found safety issues all come from the same macro body, meaning that documenting the body of the macro itself would fix all of them.

Our prototype also finished running on `indexical` in 5.76s (analyzing 212 reachable functions) and `zerocopy` in 9.02s (2,898 reachable functions), a reasonably fast result that could likely be significantly improved if optimization effort was placed into the prototype.

This shows that `Sniff-Test` is both relatively fast and effective at finding property violation issues when compared to leading lint alternatives.

7 Discussion

There are a number of interesting extensions that could be done to `Sniff-Test`'s analysis. One direction promising direction is trying to check the contents of property documentation, rather than simply checking for their existence. Section 7.1 details early work we've done to introduce named obligations in a new structured format and extend `Sniff-Test`'s analysis to ensure that each of those obligations is considered by name, rather than just that some obligations are considered. Section 7.2 details potential extensions to the tool which would allow LLMs to guide audits of property documentation, and potentially find bugs in the content of property documentation.

7.1 Fine-Grained Obligations

Throughout the process of building our `Sniff-Test` prototype, we've experimented with different forms that property obligations on function definitions should take. When looking at what thorough developers were doing in the `zerocopy` codebase, we noted that their `# Safety` sections often came in the form of a bulleted list of different obligations that callers to the function must individually uphold. Based on this, we introduced a new documentation format we call fine-grained obligations, which allow a function's developers to give each of its obligations a concrete name.

Then, when we detect a call site to such a function, we can check not only that some inline justification comment exists, but also that the content of the justification mentions each of the obligations by name. We also use a thin synonym layer to allow common obligation names

to be dismissed in a variety of ways (e.g. “aligned” could also be dismissed by mentioning “alignment”). We also employ a small number of operators to allow for more flexibility in giving names. For instance, when the tool sees a hyphen in an obligation’s name, it will ensure that both sides of the hyphen independently appear in a justification’s content in any order. Within the current prototype of the tool, this functionality is opt-in per function: if a function has fine-grained documentation, the content of it’s callers’ justifications will be checked accordingly, otherwise, it will just be checked that justifications exist. In practice, no existing doc comments for properties contain these fine-grained comments in the exact format we expect them so this is rare when directly running on production code.

An example of the style of bulleted `# Safety` sections found in `zerocopy`.

```
/// # Safety
/// - `ptr` must be non-null.
/// - `ptr` must be validly aligned for a `u32`.
unsafe fn deref(ptr: *const u32) -> u32 {
    *ptr
}
```

A codified format for fine-grained obligations

```
/// # Safety
/// - non-null: `ptr` must be non-null.
/// - aligned: `ptr` must be validly aligned for a `u32`.
unsafe fn deref(ptr: *const u32) -> u32 {
    *ptr
}
```

Like with all aspects of this system, it is still possible to mention the name of an obligation in your inline justification but consider the obligation incorrectly, but it adds more fine-grained endurance that not only has the caller considered your property in general, but they’ve considered each obligation individually. After we ported all of `zerocopy`’s safety comments to this format, and found at least four additional documentation improvements. This system is notably sensitive to both obligation names and the synonym layer, so looking into how to make a system like this effective at surfacing issues while remaining ergonomic is an interesting future direction.

7.2 LLM-guided audits

When thinking about what a tool like `Sniff-Test` might want from a structured feature for auditing the content of a codebase’s property documentation, it seems that the use of large language models would align well with our goals of building a tool that is not exhaustive or provably correct, but works well and is scalable in practice. If prompted with specific obligation, justification pairs and the necessary code for context, LLMs would

likely perform well in checking if the content of a justification comment is correctly and exhaustively discharging the listed obligation. This could be especially useful in that it could tie the content of property documentation back to the underlying code. `Sniff-Test` in its current form does nothing to ensure that the code surrounding an inline justification actually does what the justification says, but giving an LLM the obligations for each inline justification, along with a snipped of code to give context to the justification could allow them to heuristically check that the code seems correct in doing what the justification describes, and that the justification is correctly enforcing the callee's obligations.

8 Conclusion

We propose `Sniff-Test`, a light-weight analysis system that checks for missing documentation of vital code properties. `Sniff-Test`'s key contribution is the idea that the Rust community's existing convention and proven call graph construction techniques can be leveraged to make such an analysis feasible, scalable and effective. Our evaluation and experience using our `Sniff-Test` prototype on two Rust crates shows that the idea has promise and finds significantly more documentation issues than a leading lint baseline, all with comparably minimal developer effort to start using the tool. Although improvements must be made to make our analysis robust to the full set of code that exists in the ecosystem, this work hopes to show the practicality of a new class of code quality tools that aim to be light-weight and ergonomic like lints, but also have some inter-procedural analysis to check for properties more thoroughly than existing tooling.

References

[1] Meet safe and unsafe. `https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html`.

[2] The Rust Core Team. Announcing rust 1.0. `https://blog.rust-lang.org/2015/05/15/Rust-1.0/`, 2015.

[3] The embedded rust book: Panicking. `https://docs.rust-embedded.org/book/start/panicking.html`, 2025.

[4] Miri: An interpreter for rust's mid-level intermediate representation. `https://github.com/rust-lang/miri`, 2025.

[5] Kani rust verifier. `https://github.com/model-checking/kani`, 2025.

[6] Verus: Verified rust for low-level systems code. `https://github.com/verus-lang/verus`, 2025.

[7] Clippy lints. `https://rust-lang.github.io/rust-clippy/master/index.html`, 2025.