sniff-test: Light-weight, Automated Checking of Code Properties in Rust

Alex Portland

Problem

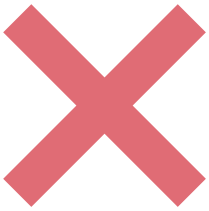
Developers need a way to **automatically reason** about whether their code is UB- or panic-free!

Formal reasoning tools are great, but only cover a *restrictive subset* of most languages & take a *long time* to run

Approach

Use a simple analysis for structured doc comments to make sure all *preconditions* are considered.

```
#[sniff_tool::check_unsafe]
fn main(ptr: *mut u32) -> u32 {
    unsafe { *ptr }
}
```



ERROR: `main` doesn't consider safety obligations of `raw_ptr_deref`: [non-null, aligned, provenance]

```
#[sniff_tool::check_unsafe]
/// # Safety
/// - provenance: ptr must have valid
///     provenance
fn main(ptr: *mut u32) -> u32 {
    if ptr.is_null() || !ptr.is_aligned() {
        return 0;
    }
    /// SAFETY: the above checks that
    /// ptr is non-null and aligned
    unsafe { *ptr }
}
```

This code passes the sniff test!!

Results

Tested on *zerocopy*, a popular, security-critical Rust crate for zero-cost memory manipulation.

Analyzes 2807 reachable functions from the crate's public API in ~4 seconds, finding **5 consistency issues**.

Two were unsafe blocks *known to be undocumented* (which I opened a PR to fix), but the other three are cases where *their safety comments are likely incomplete*.

Sniff-test runs quickly on real crates and can find real issues!

Introduction:

Developers constantly have to reason about complex properties of their code. Often, the first property to reason about is correctness – whether their code faithfully implements their desired functionality or abstract specification. However, the overarching goal of correct code has more dimensions than might be initially apparent.

For instance, is your code still considered ‘correct’ if it has subtle undefined behavior under very specific input conditions? For the meticulous developer, the answer is likely to be “clearly not” (as it could lead to an accidental or indeterminate outcome), but what if the code instead just panics at runtime if those conditions? For some developers, this might be of little consequence, but in high-stress, embedded environments, this could be equally unacceptable.

These different dimensions of correctness are vital to consider, especially in low-level languages and libraries whose users depend on these properties for the correctness of their own systems. However, these properties often involve complex, nested conditions, making them difficult to reason about in practice, and developers lack structured tools to check that they’ve done so correctly.

Sniff-test aims to be a lightweight solution that checks that developers have properly considered such code properties when needed. The system’s core design goal is practicality for use on real codebases, which hinges on analysis being lightweight enough to be scalable and requiring limited modification to existing codebases.

Approach

Sniff-test is conceptually built on the idea that functions often have obligations, or preconditions that must be upheld by their callers to ensure a given property holds. We detect where these obligations occur, whether it’s from language primitives (like a raw pointer dereference), standard library functions (like std::mem::transmute()) or from user-defined functions. We then ensure that a function either a) documents that it has considered & fulfilled each of those obligations where they occur (either at the language primitive or the callsite) or b) propagates those obligations to its own callers through a structured, readable doc comment.

As a whole, this can modularly ensure that developers have, at the very least, considered each of the obligations required for their code to be undefined behavior-, or panic-free, asserting that the most likely failure method is for developers to forget to consider their obligations rather than improperly fulfill them.

Implementation:

Sniff-test is currently implemented for Rust as a plugin to the compiler, which gives it access to the internal data structures needed for its analysis. It currently implements checking for the absence of panics and the absence of undefined behavior (often called ‘safety’ in the Rust ecosystem), but the techniques used could be expanded to other properties.

When run on a given crate, the tool will first build a callgraph of all code reachable from the user-annotated entrypoints. Then, for each of those reachable functions, it sources all documentation pertaining to relevant properties, and checks for consistency between them. If there are any consistency issues, they are presented using compiler-like diagnostics to the user, pointing them to the exact lines of their code that were found to be problematic. To explain the high-level implementation of the tool, we'll briefly walk through how each of these steps is implemented in practice.

User Annotation
To start using Sniff-test, a developer has to do is add a special `#![sniff_tool::check_unsafe]` attribute to the root of their Rust crate. This indicates that they'd like all of the functions in their code to be checked for the safety property.

If they'd like more fine-grained control over which call-trees in their codebase are checked, they can add the `#![sniff_tool::check_unsafe_pub]` attribute instead to only check code reachable from their public API, or put the check_unsafe attribute on a specific function within their code, to only check code reachable from that function. This is useful for modularly porting code to use Sniff-test's formalized checks, as code quality can be tightened-up one subtree at a time. All attributes also have check_panics equivalents which check for the absence of panics rather than the absence of undefined behavior.

Call-graph Analysis
Sniff-test then builds an over-approximate call-graph based on all code that is reachable from those user-annotated entry points. For now, this process is relatively simple compared to many analyses. We walk through Rust's mid level intermediate representation (MIR), looking for all calls to a function or method that we can concretely resolve. We build the set of all functions that are transitively reachable. This has the known limitation of not handling dynamic dispatch on trait objects, which we aim to fix in the future. The goal was to get this primitive version implemented and running first, with refinements to come later.

Our call-graph analysis is robust to handling static dispatch with generics, as we enforce that all implementers of a trait have the same obligations of their callers. Although this is a restriction on the kinds of code patterns that are allowed (and thus we will likely add an option to disable it in the future), for now this allows for modular checking where static dispatch to an associated function for a trait can be determined to have the exact obligations found on that trait definition.

Language Primitive Detection
Similarly, we detect uses of potentially problematic language primitives at the MIR level. For instance, calls to the Rust language items for panics are flagged as unconditionally problematic for panic-freedom, and raw pointer dereferences are flagged as conditionally problematic for UB-freedom (as it's UB if the pointer is unaligned or null). We then use these primitives as a base case for our analysis – any function that uses them must either dismiss the obligations or propagate them to their callers.

Consistency Checking
Now that Sniff-test knows what code we need to analyze, it must actually check that code properties have been considered at all reachable callsites to potentially violate functions.

Sniff-test gets the value of doc comments on function definitions at the MIR level. However, inline doc comments on expressions are lost when lowering to MIR, so it cross-references the span for a found call or language primitive to find the HIR node it corresponds to. Once it has the doc comment for a function's obligations, and a call site's justification, it has to check that the justification is sufficient.

To do this very quickly and be minimally reliant on additional user effort, we introduce a *'buzzword'* heuristic for determining if a callsite has sufficiently considered a given condition. This strategy simply marks a condition as validly considered by a callsite's comment if the name of the condition (the buzzword) occurs in it.

Early testing showed that a realistically-sized Rust codebase with ~30 unsafe function definitions unsurprisingly will have on the order of ~300 unique callsites to those functions. Thus, if we can limit the requirement of adding specific Sniff-test style comments to definitions, rather than all calls, it will greatly reduce the amount of developer effort required to use the tool.

A key downside of this method is that the output of the tool is now dependent on the developer's ability to choose representative condition names that are specific enough to uniquely identify a given condition, but not vague enough such that safety comments would inadvertently include them and pass the check without considering the condition. To address this, Sniff-test uses a light-weight specification grammar that function developers can use to have their condition names be properly recognized where they're actually considered while making it difficult for them to accidentally be dismissed where they're not. For instance, a condition called "bit_validity" must contain both the words "bit" & "validity", which is unlikely in a comment that doesn't properly address the bit validity requirement of the function it's calling.

Furthermore, a naive implementation of buzzword checking is very brittle to the exact wording used in safety comments – a safety comment saying that "'a must outlive 'b" will likely not satisfy a "lifetime" condition, despite obviously referring to it. Sniff-test implements a synonym layer which matches commonly used safety requirements with synonyms phrases for the same concept. This gives added flexibility and, in the above example, the comment would be found to satisfy the condition as "lifetime" is part of the same synonym equivalence class as "outlives".

Although extremely simple, testing shows that this checking technique is both fast and reliable in practice.

Artifact
Development on Sniff-test is ongoing at https://github.com/cognitive-engineering-lab/Sniff-test, with a Sniff-test compatible fork of zerocopy being maintained at https://github.com/AlexanderPortland/zerocopy/tree/Sniff-test.

Evaluation:

The performance of Sniff-test is evaluated in two contexts: on the safety-critical zerocopy crate from within the Rust ecosystem, and on in-house code examples that represent difficult patterns it would encounter in practice. The former aims to ensure that high-quality, safe code can pass 'the sniff test' with minimal code changes, while the latter aims to show that the tool can catch property violation early in development while being ergonomic to develop with.

Zerocopy

Zerocopy is a reputable Rust crate for zero-cost memory manipulation that is both easy and correct. With prolific use in security-critical contexts and the mantra *'We write `unsafe` so you don't have to.'*, they understandably take soundness incredibly seriously. The codebase uses a combination of rigorous requirements for documentation on unsafe code, runtime testing with Miri, and formal verification with Kani. Their rigorous documentation ethos results in a very high-quality codebase, where safety comments are structured like proofs and there's more lines of comments than lines of code.

By all means, **a codebase like this should pass the Sniff-test**, so it's a good benchmark for how difficult it is to get your high-quality code modified to fit the Sniff-test format.

To initially run Sniff-test, two main changes needed to be made to the codebase. First, traditional comments (using // in Rust), had to be changed to doc comments (using /// in Rust), which was a simple, methodical change that was done quickly with any find and replace tool. Then, all function definitions must be lightly modified so that all obligations on callers of the function are explicitly named, to allow us to properly check that callsites have considered them. The codebase already implicitly used this structure, and often even had names they used to refer to properties when justifying unsafe blocks, so changing the documentation for the 38 relevant unsafe functions was not difficult. Since we're currently using buzzword checking (see above), callsites themselves do not require any changes.

When run on the zerocopy codebase for the safety property (@ commit e8eb595 consisting of 11k lines of Rust code), Sniff-test analyzes 2807 functions reachable from its public API in just over 4 seconds. Initially, the tool reported 90 errors, 85 of which were false positives. 54 of those false positives were fixed through renaming 8 of the properties to be more consistent with what they were referred to as in the code. The other 31 were fixed by recategorizing what was thought to be safety requirements on callers as "safety invariants" guaranteed by the callee.

The remaining flagged errors included both of the codebase's remaining undocumented safety blocks that were known to the developers. I opened an upstream PR that attempts to document them (zerocopy#2847). (The codebase also has numerous safety comments they deem subpar, but none of those were detected as they're lacking in citations and references, rather than reasoning or consideration of code properties.)

The final three flagged errors were cases where the zerocopy developers seem to have omitted certain obligations on the function they called. All three cases are very intrinsically tied to logic and structures internal to their crate, so I have an ongoing effort to become more acquainted with the code to determine whether this was due to assuming that the obligations were obviously satisfied, or forgetting them. I suspect the latter, but want to be very thorough before claiming that's the case – either way, the solution would be to upstream a small change adding those considerations to the relevant safety comments.

This shows that, not only is it feasible to run large codebases on Sniff-test, but the tool gives quick, actionable feedback about how properties hold throughout.

In-house Testing

Sniff-test has a set of tests that check for common situations in which developers could forget to document their code properties. Common cases include forgetting to document that an unsafe operation happens with an argument to a function, documenting preconditions but forgetting to use the unsafe keyword, or having documentation in common but difficult places to parse (e.g. on a let statement that assigns to the output of an unsafe block). The current set of tests unfortunately don't test the fine-grained mode that was used to test zerocopy, and only require that there be a safety comment rather than reasoning about their content, so this is a space for improvement in the future.

Despite the potential for improvement in the future, work with these tests does show some promise that it is simple to develop new codebases with Sniff-test, and that it's good at catching common issues developers will make. (There were a few times while writing tests that I thought the tool was spuriously failing only to realize after further review that I had simply been overlooking a safety property.)

Next Steps (threats to validity I'd like to address in the future)

I'd like to continue expanding this project with a better call analysis that can more robustly handle higher-order functions. For example, there's a safe function in zerocopy that calls an unsafe function, but from within the closure of a call to map. The safety of this call was clearly underdocumented (although it has since been fixed in the upstream), but Sniff-test didn't detect this as the call was within a higher order function that may not have been actually called.

I'd also like to add more machinery to check the correctness of dependencies instead of just trusting that their documentation is accurate.