Abstract

Like all languages which interact with the operating system, Rust is vulnerable to supply-chain attacks. The industry standard to prevent supply chain attacks — exemplified by tools like Mozilla’s Cargo Vet — is to manually audit libraries for safety, but this is time-consuming and does little to help with questions of safety that are context-sensitive. To harden the Rust supply chain against this problem, we built Cargo Scan, which uses two key ideas: first, we use side effects analysis to present only relevant information to the user; and second, we track context-sensitive information in the call graph to support composable audits. Our evaluation shows that tool-assisted auditing can reduce the auditing burden to a median of 13.2% of lines of code and can reproduce CVEs that are missed by unassisted human auditors. Cargo Scan contributes to a broader empirical understanding of side effects in the Rust supply chain ecosystem.

1 Introduction

Unlike C and C++, Rust uses modern language design principles — like strong static types, module-level encapsulation, array bounds checks, and compositional error handling — to achieve memory safety by default. However, this notion of safety is precarious, and arbitrary third party Rust code can be quite dangerous to run for two reasons. First, Rust provides an unsafe keyword which turns off its safety guarantees, meaning that third party code must be trusted to uphold Rust’s complex memory safety invariants [2,11]. Second, Rust is not isolated from the outside world; even in safe Rust, third-party code can invoke operating system-level side effects, such as opening file handles, connecting to network endpoints, and executing shell commands or binary payloads which threaten the ecosystem in which the code is run. In fact, even in safe Rust, supply chain attacks have already been found in the wild. In March 2022, the widely used rust_decimal crate was the target of a typo squatting attack: a malicious crate named rustdecimal was uploaded to the Rust package management system, crates.io. If users accidentally imported the malicious crate, they would receive identical code to rust_decimal except for a single malicious side effect: the Decimal::new function would silently install and execute a binary payload on Linux and MacOS.

Currently, Rust developers use a combination of trust and auditing to mitigate the risk of supply chain attacks. The RustSec database (rustsec.org) can be used as an oracle for identifying known vulnerabilities, and crates.io includes other relevant information, including authorship, that organizations use to structure trust. Cargo Vet, a tool developed at Mozilla, lets developers manually audit crates and sign off on their safety, and manage which packages and versions have been audited [27]. However, manual auditing is a substantial burden (§2), and it makes it difficult to track context-sensitive information. For example, a function may be safe to call in some contexts, but not in others.

In this paper, we introduce Cargo Scan which provides tool-assisted auditing for Rust packages. Cargo Scan is based on two major goals.

1. Audit the code that matters. Developers should not be presented with all of the code in a library and asked to audit it all at once; instead, developers should be presented only with the minimum amount of relevant information.

2. Track context-dependent information. Some function calls are only sometimes safe—that is, their safety depends on the calling context. This information should be tracked along with audits, and should also be composable with other information during an audit.

We show how Cargo Scan achieves these goals via three concrete contributions.

1. Effect Analysis. Our first contribution addresses the first goal via a static analysis pass which identifies potential side effects in the source code (§4). We define a side effect as either an unsafe operation (such as a raw pointer dereference) or a sink call, which is a call to a Rust standard library function
which interacts with the operating system (i.e., may invoke a system call). Crucially, code that does not have side effects need not be presented to the user at all. We develop a model of side effects in Rust, and we extend it to soundly handle higher-order functions (closures and function pointers) and polymorphic code (traits and generics), while still minimizing the amount of code that the user has to audit directly.

2. Interactive Auditing. Our second contribution addresses the second goal via an interactive auditing process that shows the user potentially dangerous effect sites to systematically resolve whether the code at the site is safe or not (§3). Crucially, many side effects—such as a filesystem write or connection to a network endpoint—are only safe depending on the calling context; we have observed this behavior in practice in our efforts to audit the hyper networking library and all of its dependencies. To address this reality, side effects can be marked by the user as either safe, unsafe or caller-checked. Safe (resp. unsafe) effects are considered benign (resp. dangerous) regardless of context (for example, a call to libc::sysconf to get system information). However, caller-checked effects are considered potentially dangerous for some input arguments, depending on the caller, and Cargo Scan systematically guides the user through the different call-sites, transitively propagating the effect up the call-chain until it can be resolved as safe or unsafe.

3. Evaluation. Our third contribution is an evaluation of Cargo Scan which shows that tool-assisted auditing substantially reduces the auditing burden, identifies cases of context-sensitive safety, and reproduces real-world security vulnerabilities in the wild, including those missed by human auditors (§5). We show that 277 of the top 1000 most used crates are pure, i.e., have no side effects at all, meaning they can be declared safe to use sans any human auditing, and further, that 85% of side effects are contained in just 15% of crates. This means Cargo Scan can vet most crates with little to no auditor effort. We apply our methodology on a full audit of the hyper networking library and all of its dependencies, and we find that tool-assisted auditing reduces the auditing burden to a median of 13.2% of lines audited. We also find that 14.5% of functions’ safety is context-sensitive, and 2.1% of functions’ safety is context-sensitive across package boundaries. We find that Cargo Scan correctly identifies dangerous code corresponding to known CVEs in 14 out of 16 cases of RustSec CVEs which are in-scope, including a bug in the bumpalo package (CVE-2020-35861) that was missed by two different organizations’ auditing processes using Cargo Vet.

Threat Model. We assume that packages in public repositories may be under the control of arbitrary users who can upload whatever they want to a package manager. However, we assume that all code is audited prior to deployment, and that the auditor may make use of publicly available tools at their disposal.

2 Motivation

Organizations like Mozilla, Fastly, and Google manually audit third-party code before using in it security critical systems like Firefox, Wasmtime, and Chrome. Until recently, much of this painstaking effort was even duplicated across organizations. Many ended up auditing a common set of libraries because they did not have a common platform to share results. Simply trusting a library because one organization depends on it doesn’t help; for example, Firefox’s audit criteria might be different from Chrome’s, which in turn might be different from Wasmtime’s.

Mozilla built Cargo Vet to tackle precisely this problem [27]. With Cargo Vet, developers audit (or vet) packages against a common set of criteria (e.g., “safe to run” locally or “safe to deploy” in production). The tool then records their result in a common format, alongside the package version (or the delta between versions) they audited and any (unstructured) notes. This allows organizations to share the audit burden. Specifically, you can import external audits — so, if you trust Chrome’s audit process you can import their audits and skip auditing the packages they already audited. You can similarly skip auditing a package by marking the author of the package as trusted — especially if say Mozilla and Google trusts them. This dramatically reduces the amount of code you need to audit — but still does not scale.

2.1 Manual Auditing Does Not Scale

In practice, importing external audits (and even marking prolific developers as trusted), still leaves you with a lot of code to audit. Here, Cargo Vet doesn’t help: auditing even strongly-typed Rust code is a manual, painstaking, and error-prone process, for several reasons.

1: We need to audit everything or risk missing dangerous code. Consider auditing the hyper HTTP client and server library, one of the most popular libraries on crates.io with over 126 million downloads. The library is core to popular web frameworks like axum, which use “#![forbid(unsafe_code)]” to ensure everything is implemented in 100% safe Rust. [10] Unfortunately, this notion of safety is not that meaningful in practice. hyper and its 29 transitive dependencies do have unsafe code. Code which is safe by Rust’s standards, but still potentially dangerous (e.g., complex code that accesses the filesystem, network, etc.), is sprinkled throughout. We thus need to vet hyper and its dependencies are “safe to deploy” — and today that would mean manually sieving through 160K lines of code that spans the 30 packages. From our and other developers’ experience auditing crates, finding potentially dangerous code — beyond the obvious unsafe blocks — using off-the-shelf tools like grep is hard. In practice this doesn’t mean sifting through 160 thousand lines of code, it means giving up on the audit, or skipping potentially dangerous code. This is particularly un-
fortunate because—as we show in §5.1—large fractions of many packages can be safely vetted completely automatically.

2: We need to reason about calling contexts. Within hyper itself, there are lots of calls into the standard library—to access the filesystem and network—but unlike unsafe, developers don’t have to explicitly annotate this code as potentially dangerous for it to compile. This means we can’t reason about code locally, in isolation. For example, this code from hyper, looks safe:

```rust
pub fn bind(addr: &SocketAddr) -> Builder<AddrIncoming> {
    let incoming = AddrIncoming::new(addr).unwrap_or_else(|e| { // panic("error binding to {}: {}", addr, e);
        None
    });
    Server::builder(incoming)
}
```

The function is just constructing a new AddrIncoming and is easy to skip—both on manual inspection and with tools like (sem)grep. However, it is only by inspecting AddrIncoming::new that we see that it, and in turn, bind, are potentially dangerous—the call binds and listens to the socket address. For this same reason, we would need to inspect the calling context where bind is used—and potentially the calling context where that function is used.

3: We need to reason about complex types. High-order functions and polymorphic code makes things harder. Rust’s traits, which are similar interfaces and abstract classes in other language, in particular make it hard to reason about and analyze (polymorphic) code. Consider the Subscriber trait definition from the tracing_core package, which provides the core functionalities for application-level tracing to packages like hyper:

```rust
pub trait Subscribe where {
    unsafe
    fn downcast_raw(&self, id: TypeId) -> Option<&const ()> {
        if id == TypeId::of::<Self>() {
            Some(self as &const Self as &const _)
        } else {
            None
        }
    }
}
```

Also note that the `&` is a Rust custom derive macro that looks safe:

```rust
impl dyn Subscriber {
    pub fn downcast_ref<T: Any>(&self) -> Option<&T> {
        unsafe {
            let raw = self.downcast_raw(TypeId::of::<T>());
            if raw.is_null() {
                None
            } else {
                Some(*raw as &const _)
            }
        }
    }
}
```

This implementation overrides `downcast_raw` to always return a pointer—even if the pointer is not valid. Now calling the default `downcast_ref`, which is not marked unsafe, leads to undefined behavior. In other words, `downcast_ref` is not actually safe in all contexts—and auditing the trait in isolation is not enough to determine its safety.

4: We need to track effects across packages. Reasoning about potentially dangerous code within a single crate is hard and non-local. In practice, code spans multiple crates and might depend on build/runtime configurations—and we can’t declare a crate safe if, for example, it exposes a function whose safety is context-sensitive. In other words, we need to reason across package boundaries (and runtime configurations).

Consider CVE-2021-45712 [28] in the rust-embed project. rust-embed is a Rust custom derive macro that takes a user-defined directory and either (1) embeds files into the binary in release mode or (2) reads files directly from the filesystem in debug mode. By design, the rust-embed crate and its related packages (try to) enforce filesystem isolation, i.e., the derive macro is constrained to only read files in the user-defined directory. In debug mode, however, the implementation did not properly sanitize input paths—and broke this isolation guarantee (as RUSTSEC-2021-0126 [31] and CVE-2021-45712 [28] describe in detail).

This was easy to miss in an audit given the many inter-twined crates. First, the RustEmbed trait for the procedural macro is defined in the rust-embed package:

```rust
pub trait RustEmbed {
    fn get(file_path: &str) -> Option<EmbeddedFile>;
    fn iter() -> Iteration;
}
```

Second, the rust-embed-impl crate implements the trait, and in particular the buggy implementation of `get`:

```rust
pub fn get(file_path: &str) -> Option<EmbeddedFile> {
    let rel_file_path = file_path.replace("\", "/");
    let file_path = std::path::Path::new(folder_path)
        .join(rel_file_path);
    rust_embed::utils::read_file_from_fs(file_path).ok()
}
```
Finally, this function calls into `read_file_from_fs`, defined in `rust-embed-util`, which returns a file’s contents in a copy-on-write wrapper along with some associated metadata.

```rust
pub fn read_file_from_fs(file_path: &Path) -> io::Result<EmbeddedFile> {
    let data = fs::read(file_path)?;
    let data = Cow::from(data);
    ...
    let data;
    let metadata: Metadata { hash, last_modified },
)
```

Suppose the user defines a `struct` named `Asset`, and derives `RustEmbed` by providing a directory containing the files to be embedded.

```rust
#[derive(RustEmbed)]
#[folder = "path/to/directory"]
struct Asset;
```

During execution, `Asset::get("path/to/file")` is called to retrieve files from the given directory. In debug mode, the lack of proper filepath sanitization in `get` allows the `read_file_from_fs` function to access files that escape the user-configured directory, potentially reading from anywhere in the file system (e.g., via `Asset::get("/../../../etc/passwd")`). In this case, whether the file is safe to read depends on what path the `read_file_from_fs` function is provided, i.e., that it is properly sanitized. `read_file_from_fs` could have guarded file access itself instead of requiring callers to do so, but there is no mechanism that allows reviewers to indicate this while auditing. Furthermore, although the `read_file_from_fs` function ultimately enabled this vulnerability, it is not fundamentally dangerous. Instead, the function’s safety is context-sensitive: it depends on the manner in which it was called.

Bugs like this are incredibly difficult for auditors to spot because auditing `rust-embed` requires reasoning across multiple crates, trait implementations, and runtime configurations. The current state of affairs provides very little assistance in whether they are in fact safe by marking the effects with a safety annotation. These annotations are then collected to build an audit file for the entire package.

![Figure 1: Cargo Scan overview.](image)

### 3 Cargo Scan Overview

Cargo Scan addresses the challenges laid out in §2 by pinpointing (only) the locations where potentially dangerous code is executed, and guiding users through the function call-stack to make sure that the code is only used in safe contexts.

**Effects.** We call such potentially dangerous operations `effects` and group them into two categories. The first category is `unsafe` code (such as reading or writing raw pointers) which operates outside Rust’s safe subset, and which can cause undefined behavior. The second category is `system` effects, corresponding to system calls like file access, which can expose vulnerabilities (e.g., accessing unauthorized files). Neither category is inherently unsafe; programs routinely use raw pointers or access files without issue. However, both are `potential` origins of dangerous code that Cargo Scan helps users systematically audit in two stages, as shown in Figure 1.

**Stage 1: Effect Analysis.** In the first stage, the auditor picks a target package, and Cargo Scan analyzes the code of the package to determine the subset of code locations that execute potentially dangerous operations that require careful manual audit.

**Stage 2: Interactive Auditing.** The second stage is an auditing process in which Cargo Scan shows the auditor the potentially dangerous effect sites, and asks them to resolve whether they are in fact safe by marking the effects with a `safety annotation`. These annotations are then collected to build an audit file for the entire package.

In this section, we discuss the implementation of the analysis that takes as input a set of effects to be tracked (detailed in §4), and determines which code locations should be the starting point of human audits (§3.1). Next, we explain the different kinds of safety annotations that the auditor can label effects with (§3.2). We then walk through the interactive auditing process, using an example package with effects to illustrate how the auditor uses the different annotations (§3.3). Finally, we discuss how Cargo Scan’s audit files can be used to perform audits across multiple packages (§3.4).

### 3.1 Effect Analysis

When an auditor uses Cargo Scan to audit a package, the first step is performing a scan to find all the code locations...
which contain effects. This is done in a single-pass, parsing the package into an AST, then pattern matching on effects. The scan also builds a call-graph, which is used during the interactive auditing process.

Cargo Scan identifies the effectful locations mostly syntactically (e.g. by matching all function calls). However, several effects, require more information to avoid false positives. For instance, all dereferences are syntactically identical in Rust, but only dereferencing a raw pointer is actually potentially dangerous. Cargo Scan disambiguates such dereferences by using the type of the dereferenced variable. Similarly, Cargo Scan requires name resolution to differentiate between identically named functions, or to identify which trait method is being invoked at a given call-site. Cargo Scan gets this information from rust-analyzer, a tool which implements the language server protocol for Rust [33].

### 3.2 Safety Annotations

The effects collected by Cargo Scan in the first phase help to isolate the small subset of code locations that the auditor needs to pay close attention to. Of course, the presence of an effect does not immediately indicate the code is dubious. For instance, a raw pointer dereference in Rust is potentially dangerous behavior as it goes outside Rust’s type safety guarantees, and so, Cargo Scan flags all such dereferences as effects. However, at this point, the human auditor is required to closely investigate the surrounding context to determine if the dereference upholds Rust’s memory requirements or not, and consequently, marking each effect with one of three safety annotations: (i) safe, (ii) unsafe, or (iii) caller-checked.

**Safe/Unsafe Annotation.** The first two annotations – safe and unsafe – are applicable in those situations where the safety of the code can be determined locally i.e., from the code immediately surrounding an effect’s location. For example, in the snippet

```rust
if v.len() >= 3 {
    unsafe { v.getUnchecked(2) }
}
```

we can tell the `getUnchecked` effect is safe, because the code check’s the length of the vector immediately before the vector access. The auditor can annotate these effects as safe (or unsafe), after which Cargo Scan will proceed to show them the next effect locations.

**Caller-Checked Annotation.** The third annotation – caller-checked – is applicable in the cases where the code’s safety (or lack thereof) cannot be established locally within the code’s method. For example, in the function

```rust
fn read_file(f: &std::path::Path) -> String {
    std::fs::read_to_string(f).unwrap()
}
```

the auditor cannot know which exact path that will be opened. The function may be called innocuously to read a configuration file, or mischievously to read a user’s private keys. Thus, in this case, the code’s safety depends on the calling-context, which determines the value of the parameter `f`. Such effects are annotated as caller-checked, which indicates to Cargo Scan that each call site of the function (here, `read_file`) should have its safety checked for this effect. That is, the caller-checked annotation propagates the effect to each of the call-sites. At each call-site, the user determines whether the input passed into `read_file` is in fact safe or unsafe. Cargo Scan will determine the effect to be safe if all call-sites are safe, unsafe if some call-site is unsafe, or will mark the call-site itself as caller-checked, which transitively propagates the effect up the call-graph.

### 3.3 Interactive Auditing

After identifying all the effects in the target Rust package, Cargo Scan facilitates a systematic vetting of the effects by allowing the user to interactively audit the code at effectful locations via a command-line interface similar to interactive staging for `git add`. In this interface, the user acts as an oracle that determines the final safety of an effect, via a safety annotation. Figure 2 shows Cargo Scan in action, prompting a user to determine if the effect is safe. Line 8 highlights the effect, here, a call to `read_to_end`, with the surrounding context. Underneath that line, a label indicates which kind of effect the user has encountered. In this case, a “sink call” indicates a part of the standard library that is effectful. Finally, the user is prompted to indicate the safety of the effect with a safety annotation. If the user marks the effect caller-checked, then every call site of `read_file` is marked as another location the user has to audit. “Expand context” prints out more of the surrounding code.

**Incremental Auditing.** Cargo Scan simplifies the auditing process via an incremental mode which allows users to look over part of the code, and then come back later, or pass it off
to someone with more expertise. Having a systematic way of precisely tracking effects means users do not have to worry about accidentally missing a file or line of code during the auditing process.

**Audit Logs.** After a user has audited part of the code, the tool produces an audit log, which contains metadata about the effects present in the package and the auditor’s safety annotations. These audit logs can be quickly read to determine if, for example, there are public functions in the package marked caller-checked, indicating effects that span across packages, as for example, std::fs package which has a caller-checked effect on read_to_end.

### 3.4 Cross-Package Audits

To fully audit a (client) package, all its dependencies must first be audited to see which public methods are caller-checked (which would then necessitate auditing the call-sites in the client package. As shown in the rust-embed example in §2, vulnerabilities that originate from caller-checked public functions in other packages happen in practice. To detect these kinds of vulnerabilities, Cargo Scan provides functionality to import public caller-checked functions from other dependencies. These effects manifest as SinkCall effects, just like effectful calls into the standard library.

**Default Audits.** Auditing all a project’s dependencies produces the most precise accounting of possible effects, but requires significant effort as the number of dependencies grows. Cargo Scan enables the automatic generation of default audits, without the need for manual code review. Our effect model permits this sound, conservative approximation of an audit, by marking every effect as caller-checked. Consequently, when creating a default audit for a package, every public function which has a potential effect during its execution is marked as caller-checked. Each of those functions can thus be treated as a new sink for any code which uses this package as a dependency.

This conservative default makes starting audits in a new codebase significantly more efficient, as the user need not audit every dependency before they are able to conclude something about their application. They can instead only restrict their attention to those sites that invoke caller-checked public functions in the dependency. Indeed, as we show in §5.2, many library calls are completely safe—i.e., do not transitively call into any effectful code—and so the auditor can quickly conclude that they are safe to call. Further, as we discuss in §5.2, while performing a full audit for the hyper package, we used the default audit on several dependencies. Hyper only used a few functions from each of these packages, so an overapproximation of which functions might cause effects was sufficient, and saved us from having to audit the entire dependency.

In the case that the default audit is too conservative on some crates, those crates can be audited first, and any library calls which are now deemed safe rather than caller-checked can be removed from the effects of dependent packages.

### 4 Effect Model

Cargo Scan is parameterized by the set of effects that describe potentially dangerous code, and are the starting points of the auditor’s investigation. An effect is any function behavior that goes beyond what is visible in the function signature (input and output types). The behavior may occur during the function execution, or at any point in the future after the function executes. As Rust permits mutation via explicit mutable reference types like &mut T and mutable cells like Cell<T>, we do not consider mutation itself to be an effect.

Cargo Scan groups potentially dangerous operations into three categories of effects. First, unsafe code that operates outside Rust’s type safety guarantees, and hence can trigger undefined behavior. Second, system effects like file accesses which can expose vulnerabilities (e.g., reading or writing unauthorized files). Third, higher-order effects which includes the creation of function pointers and closures as effects, since they may cause side effects when the closure is invoked in the future. Next, we describe each of the above in detail.

### 4.1 Unsafe and System Effects

Figure 3 illustrates the complete set of effects that we support. The unsafe effects capture all possible operations outside of Rust’s safe subset, which are constrained to unsafe blocks. The FFICall effect indicates the declaration of a foreign function. Variables in this function might not respect Rust’s ownership rules if they are used by non-Rust code. This effect type is complementary to FFICall, for packages that only publicly expose foreign functions, without calling them. The system effects originate from packages like the standard library or libc, and are treated as SinkCalls in our model. ClosureCreation indicates the existence of effects.
in closures, which are executed after their creation. Similarly, FnPtrCreation is used when function pointers might point to functions containing other effects (see §4.2).

4.2 Higher-Order Control Flow

Higher-order functions and polymorphic features are ubiquitous in both the standard library and Rust application code. High-order functions take function pointers or closures, which themselves flow through the program as values. It is difficult, in general, to determine if such a function pointer or closure is safe to execute, as it requires a higher-order control flow analysis to determine all possible origins of that value. Similarly, Rust supports a form of polymorphism using traits over generic types or with dynamic trait objects. Analyzing such code also requires special handling, as to properly analyze invocations of generic or dynamically dispatched methods requires tracking all possible implementations that could be called in the respective cases. Cargo Scan handles these two different forms of higher-order control-flow — (i) function pointers and closures, and (ii) trait polymorphism – via methods tailored to each.

Closures and function pointers. In the following example, the closure at line 2 takes a file path as an argument and reads from that file.

```rust
fn danger_reader() -> impl Fn(&Path) -> String {
    |p: &Path| read_to_string(p).unwrap()
}

fn exec_reader {
    reader: impl Fn(&Path) -> String,
    file: &Path
} {
    let out = reader(file);
    ...
}

pub fn read_file(p: &Path) {
    let reader = danger_reader();
    exec_reader(reader, p);
}
```

In this instance, the particular file being read depends on the argument to the public function read_file. Because read_file is a public function, any package calling this library can use it to read any file on the system, and the onus is on callers to make sure paths are properly sanitized. If the function that returns the closure were instead written

```rust
fn safe_reader {
    root_path: PathBuf
} -> impl Fn(&Path) -> String {
    move |p: &Path| if in_root(p, &root_path) {
        read_to_string(p).unwrap()
    } else {
        "Error: invalid path".to_string()
    }
}
```

then we could see the closure itself will properly sanitize any input paths, and thus only access allowed files.

Rather than mark every possible call-site of an effectful function pointer or closure as an effect itself, we instead mark their creation as effects (with ClosureCreation and FnPtrCreation). Developers typically avoid writing closures or creating function pointers to functions if that code has complex preconditions, so safety can almost always be determined at creation. These functions are only created once, but may be called several times, so we observe that, in practice, this approach results in less work for an auditor.

Closures are also frequently used with iterators in Rust to process a collection of items. They rarely entail dangerous behavior, so naively creating effects upon their creation will unnecessarily increase the audit locations. To avoid false positives and further optimize the auditing process, Cargo Scan scans the body of a closure and marks its creation as an effect only if it contains operations that can be maliciously exploited.

Traits. When a call to a trait method can be resolved at compile time, Cargo Scan will scan that specific implementation for potential effects.

```rust
fn func() {
    let s = Struct;
    let obj: Box<dyn Trait> = Box::new(s);
    s.meth();
    obj.meth();
    ...
}
```

In our example, imagine a trait Trait with a method meth and Struct implementing that trait. Since the call to meth (line 5) is statically dispatched, Cargo Scan will point to Struct’s method implementation and look for any dangerous behavior.

However, it is not always possible to resolve the concrete (instance) implementation that will be invoked at a generic trait method call site. For example, Cargo Scan cannot determine the concrete type in the analysis phase for method calls that are in a generic context or resolved during runtime (e.g. line 6). To address this situation, Cargo Scan discovers all available implementations of the trait in the package. The call will resolve to the abstract trait method and any discovered effects for the implementations will be associated with the generic trait method by Cargo Scan. Based on the set of the potential effects propagated among all the implementations, the user will decide the trait’s safety in the current scope. When a trait is publicly exposed, it is critical to conclude if it can be safely used in any context, including outside of the given package.

4.3 Limitations

Our effect model is only meant to capture things going wrong with program execution and unintended system effects. In particular, we do not capture risks related to functional correctness, execution time, or other side channels. Some functional correctness bugs do result in a system effect which our tool would capture. For instance, we investigated a bug in the
warp library where filepaths were improperly sanitized, allowing users to read files anywhere on Windows filesystems. Although we don’t flag the function that validated the supplied path was an allowed path, we did flag the underlying filesystem call as caller-checked, which requires an auditor to make sure the arguments supplied are properly validated. In other cases, Cargo Scan would not catch such bugs. We also investigated a bug in regex where the complexity of parsing user-supplied regular expressions could lead to runtime blowup (i.e., a regex denial-of-service attack). In such cases, there is no particular problematic system effect that is directly visible from the source code.

Cargo Scan also currently does not analyze macros or build-time effects (e.g., build.rs files). In principle, we believe most effects that occur in either of these circumstances can be captured by a multi-pass model, by running on compile-time code first, then expanding macros prior to the analysis.

5 Empirical Evaluation

We evaluate Cargo Scan by asking four questions:

- How common are effects in popular packages (§5.1)?
- Is context-sensitive effect propagation valuable (§5.2)?
- Does Cargo Scan make auditing more efficient (§5.3)?
- Does Cargo Scan help catch real vulnerabilities (§5.4)?

5.1 How Common are Effects?

We first look at how common effects are in Rust by using Cargo Scan to analyze the effects found in the top 1000 most downloaded packages on crates.io, the central public Rust crate repository.

Concentration of effects. Our analysis is summarized in Figure 4 which plots the effects distribution in top crates. Figure 4a shows that a large number of crates are entirely, or mostly, pure, i.e., effect-free. Over a quarter of crates (277) contain no instances of side effects, and over half (551) contain under 10 effect instances. Further, Figure 4b demonstrates that most effects (approx. 85%) are concentrated in a small number of crates (approx. 15%).

Frequency of effect types. Figure 5 shows the frequency with which different effects occur in the most popular packages. We use the effect categories described in §4.1, and further distinguish SinkCalls by the different libraries the calls originate from. In total, we include 24 effect patterns: Rust unsafety (pointer dereferences, unsafe function calls, FFI, etc.); standard library effects (IO, file system, network, and syscalls); and higher-order effects (creation of a closure or function pointer which may be effectful). The effects which affect the greatest number of crates are higher-order effects (function pointers and closures), unsafe function calls (unsafe

5.2 Are Context-Sensitive Effects Valuable?

Context-sensitive effects (tracked via the caller-checked annotation §3.2) give Cargo Scan a more precise model of program safety, but propagating effects to their call-sites can potentially require more effort on the auditor’s part. To determine if this propagation is effective, we conducted two large case-studies to investigate:

- whether caller-checked effects are necessary to model program safety, and
- how much additional effort is required to audit caller-checked effects.

Figure 4: Distribution of effects in the top 1000 crates.

and FFI), and standard I/O operations (sinks) as shown in Figure 6. The results are slightly different when broken down by number of instances of each effect; in this case FFI function declarations rise to the top, due to a small number of crates which consist mainly of FFI bindings.
Case study: hyper. We used Cargo Scan to audit hyper—a widely used HTTP library—and all of hyper’s dependencies, as our first case study to investigate the necessity and overhead of caller-checked effect propagation. hyper contains 29 transitive dependencies in its standard build (for this audit we ignore optional crate features and the development build). We audit the dependencies in topological order to ensure all a crates dependencies were audited before the crate itself.

Minimizing effort with default audits. Most packages were audited in their entirety using Cargo Scan, but several packages consist of large libraries with only a small percentage of their functionality used. Rather than audit these packages in their entirety, we create a sound approximation of their effectful behavior using Cargo Scan by assuming each effect requires each call site to be transitively checked to determine safety (see §3.4 for details on default policies). The packages we create a default audit for are futures-task, futures-util, tokio, and httparse, and are excluded from the summary statistics, as they were audited using a different process.

Additional auditing effort. In our case-study, we found that though reasoning about context-sensitive safety is necessary, it requires very little additional auditing effort, as caller-checked effects occur rarely, and are only propagated to a small number of contexts, as summarized, per package, in Figure 7. Among packages which we audit in their entirety, we find that the arithmetic mean percentage of effects which required caller-checked safety annotations was 14.5%. Furthermore, effects which we mark as caller-checked tend to require reasoning about few calling contexts. Among effects marked caller-checked, we only had to annotate an arithmetic mean of 2.4 locations per effect including the location where the caller-checked effect originates.

Effects across packages boundaries. Caller-checked effects are also important in calls across package boundaries. We find that although this case is rare, it does happen in practice. Fully-audited packages have an arithmetic mean of 2.1% of their public functions marked caller-checked. If an auditor approximates the potentially unsafe behavior of a package by generating a default audit, this number of functions which must be checked across the package boundary goes up significantly. Among packages we generate the default caller-checked audit for, the arithmetic mean of public functions marked caller-checked was 35.6%. This approximation allows auditors to conclude the many remaining (64.4%) public functions in dependencies are safe to call, even without hand-auditing the package. Thus, we conclude that, allowing more expressiveness through caller-checked effects does not add significant extra auditing work.

Case study: configuration packages. As a second case study, we also examine the necessity and additional auditing effort of caller-checked effects on five of the most popular system configuration packages from crates.io, namely config, rust-ini, figment, configparser, and dotenvy. For the purpose of this case study, we audit only the top-level packages, to assess how common caller-checked system effects are in a different domain.

The details of the audits are demonstrated in Figure 8.

Figure 5: How many crates have at least one occurrence of the effect in the top 1000 crates.

Figure 6: Relative frequency of effects by the total instances of that effect across all top 1000 crates.
These packages do not include unsafe code and they mainly consist of system effects related to filesystem accesses and environment variables, and a small amount of high-order effects. Among these system effect-heavy configuration packages, the caller-checked annotation requires somewhat more effort and is more often necessary for a sound audit. The arithmetic mean percentage of effects marked caller-checked is 25.07%, with a mean of 4.9 locations to review (including the base effect).

5.3 Does Cargo Scan make Auditing more Efficient?

We evaluate whether Cargo Scan makes the auditing process more efficient by measuring the amount of code the user has to look at to audit a package, relative to the size of the entire package. For this evaluation, we use our full audit of hyper and exclude the 4 packages we created the default audit on from our summary statistics. Those packages are large libraries like tokio, and since only a small portion of their functionality is used in hyper, we can get meaningful assurance about their safety by using their automatically generated default audits.

Quantifying auditing effort. We determine auditing effort by calculating the reviewed code i.e., how much code auditors review as a percentage of the total code of the package as analyzed by Cargo Scan. The “total code” excludes any conditionally compiled code which is disabled, as well as macros which aren’t support by Cargo Scan yet. Determining “reviewed code” can be tricky as auditing an effect requires understanding its surrounding context—including things like the parameters a function is called with, or how the result of an effect is used. Rather than mark which individual lines are relevant to the user, we conservatively approximate the “reviewed code” by assuming they have to inspect the entire function an effect appears in.

Distribution of audited effects. We find that Cargo Scan significantly reduces the amount of code an auditor has to look at, and that most of the auditing effort is concentrated in a few crates with many effects. The arithmetic mean percentage lines of code we audit per package is 23.7%, with a median of 13.2%, and the full distribution is shown at Figure 9. We see a skew here consistent with what we expect from §5.1. Specifically, the packages with a high percentage of audited code are primarily low-level packages, which are more commonly reused by other projects.

Effects in macro expansions. Cargo Scan does not yet support macros in its analysis (see §4.3). However, to give a rough sense, we also measure the impact of macros on auditing. In contrast to the above optimistic estimates, which are valid if all macros produce pure code, we can consider a pessimistic estimate assuming that all macros emit dangerous code. In this case, we get a more modest arithmetic mean 49.6% of code presented to the user, with a median of 42.9%. We believe that the reality is closer to the optimistic case than the pessimistic one; in a manual inspection of the macros in hyper and its dependencies, we find that most macros do not produce unsafe code or have other side effects.

5.4 Does Cargo Scan find Real Vulnerabilities?

Finally, to check if our analysis can provide assistance in catching real bugs, we will look at a sample of vulnerabilities from the RustSec database [13]. We look at packages which have been previously audited in Cargo Vet repositories, and mark any package versions which have a bug associated with them in the RustSec database. We examine if Cargo Scan presents auditors with security-relevant effects pertaining to the vulnerable lines of code.

Retroactive auditing process. We audit the buggy versions of those packages and check whether Cargo Scan presents to the user the origin of the bug, or relevant code that flows from or into the bug. For instance, if a function call is the source of the vulnerability, we check if Cargo Scan flags as an effect either (1) the function call itself, or (2) any argument to that function call. Note that this does not necessarily mean an auditor would have caught the vulnerability; merely that our tool would aid in finding it. This gives us a list of 20 security-related bugs that were missed by human auditors. From this list, we exclude bugs related to functional correctness, performance, and side channels, which are currently out-of-scope (see §4.3). This excludes four of the 20 bugs, of which two are algorithmic complexity related, one is denial of service attack, and one is a functional correctness bug where random number generation is not initialized with enough entropy.

Results. Out of 16 bugs that are in scope and missed by human auditors, Cargo Scan highlights code related to 14 out of 16. Of the bugs we don’t flag, one results in malformed data during a panic, and the other is related to trait bounds. For the full table of effects we catch, including which ones require the caller-checked annotation, see Table 1.

Case Study: CVE in bumpalo. To explore the implications of this finding in more detail, consider CVE-2020-35861, which is a bug in the bumpalo package, an arena allocator with 58,000,000 downloads. The vulnerable version of this code was audited by two different organizations in Cargo Vet audits. This version permits a read-out-of-bounds bug as a result of copying the wrong number of bytes over when reallocating. On line 11, the code in question copies data of size old_size rather than new_size.

```rust
    ...  
    if self.is_last_allocation(ptr) {  
        // Try to allocate the delta size  
        // within this block so we can reuse  
        // the currently allocated space.  
        let delta = new_size - old_size;
```
### Figure 7

<table>
<thead>
<tr>
<th>Name</th>
<th>Effects</th>
<th>Mean CC Size</th>
<th>Public Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>CC</td>
<td>Total</td>
</tr>
<tr>
<td>bytes-1.4.0</td>
<td>141</td>
<td>41</td>
<td>153</td>
</tr>
<tr>
<td>cfg-if-1.0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fnv-1.0.7</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>futures-channel-0.3.28</td>
<td>21</td>
<td>6</td>
<td>38</td>
</tr>
<tr>
<td>futures-core-0.3.28</td>
<td>4</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>http-0.2.9</td>
<td>44</td>
<td>5</td>
<td>193</td>
</tr>
<tr>
<td>http-body-0.4.5</td>
<td>8</td>
<td>0</td>
<td>29</td>
</tr>
<tr>
<td>httpdate-1.0.2</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>hyper-0.14.27</td>
<td>195</td>
<td>23</td>
<td>267</td>
</tr>
<tr>
<td>itoa-1.0.6</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>libc-0.2.146</td>
<td>140</td>
<td>140</td>
<td>5958</td>
</tr>
<tr>
<td>memchr-2.5.0</td>
<td>874</td>
<td>0</td>
<td>86</td>
</tr>
<tr>
<td>num_cpus-1.15.0</td>
<td>16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>once_cell-1.18.0</td>
<td>52</td>
<td>2</td>
<td>54</td>
</tr>
<tr>
<td>pin-project-lite-0.2.9</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>pin-utils-0.1.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>proc-macro2-1.0.60</td>
<td>13</td>
<td>1</td>
<td>127</td>
</tr>
<tr>
<td>quote-1.0.28</td>
<td>1</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>syn-2.0.18</td>
<td>338</td>
<td>4</td>
<td>1285</td>
</tr>
<tr>
<td>tokio-macros-2.1.0</td>
<td>6</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>tower-service-0.3.2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tracing-0.1.37</td>
<td>5</td>
<td>0</td>
<td>48</td>
</tr>
<tr>
<td>tracing-core-0.1.31</td>
<td>21</td>
<td>4</td>
<td>134</td>
</tr>
<tr>
<td>try-lock-0.2.4</td>
<td>5</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>unicode-ident-1.0.9</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>want-0.3.1</td>
<td>2</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>futures-task-0.3.28 (*)</td>
<td>23</td>
<td>23</td>
<td>14</td>
</tr>
<tr>
<td>futures-util-0.3.28 (*)</td>
<td>241</td>
<td>241</td>
<td>310</td>
</tr>
<tr>
<td>httparse-1.8.0 (*)</td>
<td>123</td>
<td>123</td>
<td>39</td>
</tr>
<tr>
<td>tokio-1.28.2 (*)</td>
<td>2032</td>
<td>2032</td>
<td>803</td>
</tr>
</tbody>
</table>

Figure 7: The caller-checked (CC) statistics for each package in the full hyper audit. * indicates the package was audited with the default caller-checked audit.

### Figure 8

<table>
<thead>
<tr>
<th>Name</th>
<th>Effects</th>
<th>Mean CC Size</th>
<th>Public Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>CC</td>
<td>Total</td>
</tr>
<tr>
<td>config-0.13.4</td>
<td>37</td>
<td>2</td>
<td>10.50</td>
</tr>
<tr>
<td>dotenv-0.15.7</td>
<td>31</td>
<td>11</td>
<td>5.55</td>
</tr>
<tr>
<td>figment-0.10.12</td>
<td>87</td>
<td>4</td>
<td>6.00</td>
</tr>
<tr>
<td>rust-init-0.20.0</td>
<td>22</td>
<td>5</td>
<td>3.60</td>
</tr>
<tr>
<td>configparser-3.0.4</td>
<td>7</td>
<td>4</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Figure 8: The caller-checked (CC) statistics for each configuration package.
7  if let Some(p) = self.try_alloc_layout_fast(
8       layout_from_size_align(delta, layout.align())
9   )
10   {
11       ptr::copy(ptr.as_ptr(), p.as_ptr(), old_size);
12       return Ok(p);
13   }
14 ...

Rather than present the entire codebase to the user during an audit and require them to understand all possible effectful locations, Cargo Scan identifies the buggy line of code on 11 and presents it to the user.

In this example, the buggy line of code is itself an effect. However, some vulnerabilities require investigation through multiple function calls before it becomes clear some dangerous effect is taking place. For many of these cases, using Cargo Scan means the dangerous effect itself is marked callercHECKed (because its safety depends on how it is being called) and the buggy line of code is eventually presented to the user when prompted with call sites. Cargo Scan finds several bugs specifically because of this functionality.

6 Related work

Empirical analysis and testing of the Rust ecosystem. Rust packages are centrally organized by crates.io and have been significant work in analyzing unsafe code in this ecosystem. Several authors have studied how Rust's unsafe blocks are used in practice [2, 11], and whether and how they cause bugs in existing code bases [30]. Others authors have focused on specific vulnerabilities, investigating Rust CVEs [39], yanked crates [23], and semantic version violations [29]. Our work additionally analyzes system effects in the ecosystem.

In addition to analysis, many tools also target dynamic testing techniques in the Rust ecosystem. RULF [16] and SyRust [34] automatically synthesize a sequence of API calls which can be used as a fuzzing target to look for bugs. These techniques are effective at finding specific classes of bugs in real-world Rust code (e.g., unreachable code and out of bounds checks), and are complementary to static analysis and auditing.

Supply chain security. Supply chain security is a broad field with many tools to focus on specific components. Industrial tools such as Cargo Vet [27] can be used to manage audited Rust dependencies. JFrog [8] and the RustSec database [13] are solutions to identify and mitigate supply chain risks by tracking vulnerabilities. Cackle [21] uses an access control list to determine whether specific API(s) are used by any transitive dependencies of a crate. We target specifically the aspect of manual code auditing.

Beyond Rust, the problem of supply chain security has been especially well studied in the JavaScript ecosystem. Most work for JavaScript focuses on vulnerabilities in other aspects of supply chain security, such as publishing and delivery [9, 40, 41]. However, the JavaScript language admits patterns like runtime dependency injection which make JavaScript highly dynamic, and difficult to analyze statically.

Language support for sandboxing and isolation. Language support for isolation goes back to early work in systems programming such as Modula-3 [7, 15, 32] and more recently in capability-based systems such as SHILL [26]. The Safe Haskell project [35] demonstrated that sufficiently strong static type safety can be used to provide assurance against supply chain attacks. These techniques do not directly apply in the context of Rust, which adopts a much more laissez-faire attitude towards supply chain security, and allows developers to bypass the type system and other safety mechanisms at their own risk. Among these works, we are most adjacent to Safe Haskell, as it was able to retrofit safety in an existing supply chain system through labeling crates that avoided the use of dangerous features as unsafe to import. We aim to generalize this approach to arbitrary system side effects and memory safety as present in Rust.

Several recent papers have explored isolation in the context of Rust [1, 6, 19, 20, 24], typically by combining OS and language mechanisms to isolate memory that can be accessed within unsafe code blocks from the rest of the system. This method imposes significant runtime overheads and does not constrain the damage that can be done within the unsafe blocks. It also does not constrain system side effects, and as a result Rust code can continue to pervasively invoke external processes (std::process) and invoke C/C++ code through the FFI.

Semantics and verification. Much recent effort has gone into formalizing the semantics of Rust’s ownership mechanisms, including RustBelt [18], Oxide [38], Stacked Bor-
rows [17], and the Miri interpreter for detecting undefined behavior [36]. Other projects like Rudra [4] analyze the interaction between unsafe Rust, stack unwinding, and the Rust trait system. These techniques can improve automation for detecting problems in unsafe Rust code, but, like the existing ecosystem work, do not consider system side effects. There is also work on verification tools for Rust, including Smack [5], Kani [37], Prusti [3], Aeneas [14], Verus [12], Hacspec [25], and Flux [22]. Most verification tools only apply to safe Rust, as unsafe Rust is difficult to formalize (see the semantics work above). Verification is labor-intensive and does not scale to the ecosystem, but our techniques can help narrow down code that would be the most critical for verifiers to target.

7 Conclusion and Future Work

Cargo Scan helps audit Rust code more efficiently. It does so by identifying side effects in a static analysis pass, then tracking context-dependent safety throughout the call graph in an interactive auditing process. Our evaluation is promising: we demonstrate that using side effects substantially reduces the amount of code that needs to be audited by at least a factor of two, that context-sensitive effects really occur in practice in the Rust ecosystem, and that Cargo Scan can reproduce known RustSec vulnerabilities, even those that are missed by human auditors.

Among the limitations of the current tool, one direction is to use more precise static analysis techniques—for example by tracking information flow and by reducing the number of false positives, especially related to unsafe code and traits. Another direction is a systematic treatment of build-time effects; Rust employs a powerful macro system in which procedural macros can execute arbitrary code, which is executed in a bottom-up fashion, and which opens up an interesting attack vector. Rust code can also execute a custom build script (build.rs), and these scripts sometimes insert dangerous code and have nondeterministic, platform-specific behavior at compile time. Finally, we would like to extend the auditing support to manage cross-package and cross-language effects, through a versioning system which supports audit updates when some versions of some packages are updated, without having to redo an audit of the entire dependency tree.

References


A Appendix

Table 1 details our analysis of 20 vulnerabilities from the RustSec database as described in §5.4.
<table>
<thead>
<tr>
<th>Package</th>
<th>Vulnerability Description</th>
<th>RustSec ID</th>
<th>Cargo Scan</th>
<th>Effect Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>mio-0.7.5</td>
<td>Invalidly assumes the memory layout of std::net::SocketAddr</td>
<td>RUSTSEC-2020-0081</td>
<td>✓</td>
<td>Direct</td>
</tr>
<tr>
<td>nix-0.20.0</td>
<td>Out-of-bounds write in nix::unistd::getgrouplist</td>
<td>RUSTSEC-2021-0119</td>
<td>✓</td>
<td>Direct</td>
</tr>
<tr>
<td>tokio-1.2.0</td>
<td>tokio::io::ReadHalf&lt;T&gt;::unspli violates the Pin contract</td>
<td>RUSTSEC-2023-0005</td>
<td>✓</td>
<td>Caller-checked</td>
</tr>
<tr>
<td></td>
<td>Data race when sending and receiving after closing a Oneshot channel</td>
<td>RUSTSEC-2021-0124</td>
<td>✓</td>
<td>Caller-checked</td>
</tr>
<tr>
<td></td>
<td>Task dropped in wrong thread when aborting LocalSet task</td>
<td>RUSTSEC-2021-0072</td>
<td>✓</td>
<td>Caller-checked</td>
</tr>
<tr>
<td>bumpalo-3.0.0</td>
<td>Flaw in bumpalo::realloc allows reading unknown memory</td>
<td>RUSTSEC-2020-0006</td>
<td>✓</td>
<td>Direct</td>
</tr>
<tr>
<td></td>
<td>Use-after-free due to a lifetime error in Vec::into_iter</td>
<td>RUSTSEC-2022-0078</td>
<td>✓</td>
<td>Caller-checked</td>
</tr>
<tr>
<td>thread_local-1.1.3</td>
<td>Data race due to weaker memory ordering</td>
<td>RUSTSEC-2022-0006</td>
<td>✓</td>
<td>Direct</td>
</tr>
<tr>
<td>vm-memory-0.12.1</td>
<td>Incorrect implementations of VolatileMemory::get_slice can lead to out-of-bounds memory access</td>
<td>RUSTSEC-2023-0056</td>
<td>✓</td>
<td>Caller-checked</td>
</tr>
<tr>
<td>warp-0.3.2</td>
<td>Improper validation of Windows paths could lead to directory traversal attack</td>
<td>RUSTSEC-2022-0082</td>
<td>✓</td>
<td>Caller-checked</td>
</tr>
<tr>
<td>lock_api-0.4.1</td>
<td>Some lock_api lock guard objects can cause data races</td>
<td>RUSTSEC-2020-0070</td>
<td>×</td>
<td>-</td>
</tr>
<tr>
<td>spin-0.9.4</td>
<td>Initialisation failure in Once::try_call_once can lead to undefined behaviour for other initialisers</td>
<td>RUSTSEC-2023-0031</td>
<td>✓</td>
<td>Caller-checked</td>
</tr>
<tr>
<td>smallvec-0.6.12</td>
<td>Buffer overflow in SmallVec::insert_many</td>
<td>RUSTSEC-2021-0003</td>
<td>✓</td>
<td>Caller-checked</td>
</tr>
<tr>
<td></td>
<td>Creates uninitialized value of any type</td>
<td>RUSTSEC-2018-0018</td>
<td>✓</td>
<td>Caller-checked</td>
</tr>
<tr>
<td>time-0.1.45</td>
<td>Potential segfault due to dereferencing a dangling pointer in specific circumstances</td>
<td>RUSTSEC-2020-0071</td>
<td>✓</td>
<td>Direct</td>
</tr>
<tr>
<td>ordered-float-2.0.0</td>
<td>ordered_float::NotNaN may contain NaN after panic in assignment operators</td>
<td>RUSTSEC-2020-0082</td>
<td>×</td>
<td>-</td>
</tr>
<tr>
<td>h2-0.3.14</td>
<td>Resource exhaustion vulnerability may lead to Denial of Service</td>
<td>RUSTSEC-2023-0034</td>
<td>(∗)</td>
<td>-</td>
</tr>
<tr>
<td>rustls-webpki-0.100.1</td>
<td>CPU denial of service in certificate path building</td>
<td>RUSTSEC-2023-0053</td>
<td>(∗)</td>
<td>-</td>
</tr>
<tr>
<td>rand_core-0.6.1</td>
<td>Incorrect check on buffer length when seeding RNGs</td>
<td>RUSTSEC-2021-0023</td>
<td>(∗)</td>
<td>-</td>
</tr>
<tr>
<td>regex-1.5.4</td>
<td>Regexes with large repetitions on empty sub-expressions take a very long time to parse</td>
<td>RUSTSEC-2022-0013</td>
<td>(∗)</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: List of packages with known vulnerabilities. * indicates vulnerabilities that are out of scope