Auditing Rust Crates Effectively

Lydia Zoghbi[1]*, David Thien[1], Ranjit Jhala[1], Deian Stefan[1], and Caleb Stanford[2]

[1] University of California, San Diego, USA
{lzoghbi,dthien,rjhala,dstefan}@ucsd.edu
[2] University of California, Davis, USA
cdstanford@ucdavis.edu

Abstract. We introduce Cargo Scan, the first interactive program analysis tool designed to help developers audit third-party Rust code. Real systems written in Rust rely on thousands of transitive dependencies. These dependencies are as dangerous in Rust as they are in other languages (e.g., C or JavaScript) — and auditing these dependencies today means manually inspecting every line of code. Unlike for most industrial languages, though, we can take advantage of Rust's type and module system to minimize the amount of code that developers need to inspect to the code that is potentially dangerous. Cargo Scan models such potentially dangerous code as effects and performs a side-effects analysis, tailored to Rust, to identify effects and track them across crate and module boundaries. In most cases (69.2%) developers can inspect flagged effects and decide whether the code is potentially dangerous locally. In some cases, however, the safety of an effect depends on the calling context—how a function is called, potentially by a crate the developer imports later. Hence, Cargo Scan tracks context-dependent information using a call-graph, and collects audit results into composable and reusable audit files. In this paper, we describe our experience auditing Rust crates with Cargo Scan. In particular, we audit the popular client and server HTTP crate, `hyper`, and all of its dependencies; our experience shows that Cargo Scan can reduce the auditing burden of potentially dangerous code to a median of 0.2% of lines of code when compared to auditing whole crates. Looking at the Rust ecosystem more broadly, we find that Cargo Scan can automatically classify ∼3.5K of the top 10K crates on `crates.io` as safe; of the crates that *do* require manual inspection, we find that most of the potentially dangerous side-effects are concentrated in roughly 3% of these crates.

1 Introduction

Rust is a compelling alternative to C and C++ for systems programming as it provides memory safety by default through its strong static typing, module-level encapsulation, array bound checks and compositional error handling. However, modern software is predominantly *other* people's code, and arbitrary, third-party Rust can be just as dangerous as C or C++. First, Rust provides an `unsafe`

© The Author(s) 2026
R. Krebbers (Ed.): ESOP 2026, LNCS 16502, pp. 424–444, 2026.
https://doi.org/10.1007/978-3-032-22723-2_15

keyword which allows developers to bypass its usual safety guarantees, meaning that third party code must be trusted to uphold Rust's complex memory safety invariants [10,1]. Second, memory safety is not the only concern; memory *safe* third-party Rust code can invoke the operating system to read and write files, connect to network endpoints, and execute arbitrary binaries—all of which can potentially be dangerous.

Problem: Vetting Third-Party Rust. Currently, Rust teams *audit* third-party code (crates) to mitigate the risk of supply chain attacks. This is a manual, painstaking process as real-world systems depend (transitively) on hundreds to thousands of crates. Efforts like Mozilla's Cargo Vet [34] streamline auditing by allowing Rust teams to record their results and sign off on a crate's safety, enabling others to *trust* these results and lower their workload. In practice, however, this trust-based approach does not really simplify the review process (§2). Further, as Cargo Vet's audit results are just approvals with unstructured comments, even audited crates can require a look over, since a crate's publicly exposed function might be safe only in specific contexts. Finally, identifying what code needs auditing—even within an `unsafe` block [35]—and then determining its actual safety remains challenging.

Auditing Goals. We identify two key goals for a tool-assisted auditing process.

1. *Audit the code that matters.* Developers should not have to inspect every line of code; they should only audit code that is indeed potentially dangerous. Most widely-used languages make this near impossible (e.g., C/C++'s memory unsafety, Java's reflections and class loading, and most of JavaScript's features make it hard to look at a piece of code and be confident it's safe); Our goal is to leverage Rust's type and module system to get global safety properties by auditing only a small fraction of the code.
2. *Track context-dependent information.* Developers should not have to audit already-audited code. Currently, they must as some functions are only safe *sometimes*, i.e., depending upon the calling context. Instead, audits should be *composable* and track such information in the audit report itself; to inform developers when they use such potentially dangerous functions, thereby propagating the context-sensitivity to any callers of this function.

Auditing Rust Effectively. We built Cargo Scan, an interactive Rust auditing tool, with these goals in mind. Cargo Scan is founded on central insight borrowed from functional programming: we can view potentially dangerous operations as *effects*, which are flexible enough to model a wide variety of potentially dangerous operations, from low-level memory operations (e.g., raw pointer dereference) to high-level operating system calls (e.g., file reads and writes). However, rather than tracking effects in the type system (e.g., as in Haskell), we track them via an off-the-shelf static analysis that lets developers focus on the code that matters while tracking and propagating effects context-sensitively. In this paper, we present the design and implementation of Cargo Scan and report on our experience auditing crates with Cargo Scan. Our contributions are threefold.

Fig. 1: Interactive auditing with Cargo Scan in VSCode.

1. Effect Analysis. Our first contribution addresses the first auditing goal via a static analysis pass that identifies *potential side effects* in the source code (§3.2). Our effect model (§3.1) and analysis builds on the long line of work on effects analysis, but crucially exploits Rust's type system and our problem domain to make auditing practical. First, we observe that the borrow checker in Rust, combined with module-level abstraction and the explicit safe/unsafe scoping, already makes mutation and aliasing tractable, contrary to other languages. This allows us to focus on a *function-level* analysis (by associating side effects with each function call) rather than a traditional *data-level* analysis that is necessary in languages like Java [49,30,50,18,38,40,25,33,46,55,48]. Moreover, this analysis lets us ensure that code that does not have potentially dangerous side effects is not presented to the user at all. Second, we observe that complex language features—closures, function pointers, traits and generics—which make analysis hard in the general case don't need to make auditing hard(er). Our insight is to "lift" these features into the effect model and ask developers to audit, for example, code creating closures much like they would audit code opening files.

2. Interactive Auditing. Our second contribution addresses the second goal via an interactive auditing process that prompts the user with potentially dangerous effect sites (§3). The user then marks each site with an annotation that indicates it's (un)safe or that its safety is context-sensitive. Cargo Scan captures this information in the audit file, hence, unlike Cargo Vet, developers don't need to trust a library in whole, nor reaudit it: they can read the audit report (using Cargo Scan) and just inspect the calling context of context-dependent functions.

We implement the Cargo Scan interactive process as a VSCode extension. This lets developers easily assess the safety of any particular effect and annotate it inline: they can inspect the surrounding code and take advantage of integrations like `rust-analyzer` to inspect types, definitions, and otherwise navigate the call graph (and codebase). Our experience auditing the `hyper` networking library and all of its dependencies highlights that this is especially useful for

complex codebases and codebases where many effects are context-sensitive (and navigating between call sites and definitions is crucial). Anecdotally, this extends to commercial use cases; the authors have experimented with Cargo Scan for over a year in production to audit Rust crates for a security critical product, and have found that it reduces the time to complete code reviews for new crates in pull requests from multiple days to a few hours.

3. Evaluation. Our third contribution is an evaluation of Cargo Scan which shows that tool-assisted auditing reduces the auditing burden and identifies cases of context-sensitive safety (§4). We first show that out of the top 10K most used crates, 3434 of them are *pure*, i.e., have no dangerous 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 3% of crates. This means Cargo Scan can vet most crates with little to no auditor effort. We then report on our experience auditing the `hyper` networking library and all of its dependencies. Our audit shows that 30.8% of functions' safety is context-sensitive, and 5.2% of functions' safety is context-sensitive across package boundaries; this means auditing code in practice crucially relies on tracking effects across functions (and packages). Overall, though, this does not impose a huge burden: we find that tool-assisted auditing still only requires auditing a median of 0.2% of lines of code (relative to auditing the code without Cargo Scan). Finally, if the top 10K crates are representative of the broader Rust ecosystem, this means we can use Cargo Scan to raise the security bar of Rust as a whole with a modest manual effort.

2 Motivation

Organizations like Mozilla, Fastly, and Google manually audit third-party code before using it in security critical systems like Firefox, Wasmtime, and Chrome. Until recently, much of this effort was duplicated across organizations, in the absence of a common platform to share results, and simply trusting a library because one organization depends on it doesn't provide any inherent guarantees.

Mozilla built Cargo Vet to tackle precisely this problem [34]. With Cargo Vet, developers audit (or vet) packages against shared 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; a team can skip auditing packages by *importing* audits of a trusted entity (e.g., Chrome's audit records), or by marking authors as trusted—especially if say Mozilla and Google trust them.

2.1 Manual Auditing Does Not Scale

In practice, importing external audits and marking prolific developers as trusted, still leaves us 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.

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 406 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. [3]" Unfortunately, this notion of safety is not that meaningful in practice; `hyper` and its transitive dependencies *do* have `unsafe` code. Moreover, code performing potentially dangerous operations (e.g., complex code that accesses the filesystem, network, etc.)—although considered safe by Rust's standards—is still present in these packages. 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 160K lines of code, it means giving up on the audit, or accidentally missing potentially dangerous code. This is particularly unfortunate because—as we show in §4.1—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 example, this code from `hyper`, *looks* safe:

```
pub fn bind(addr: &SocketAddr) -> Builder<AddrIncoming> {
    let incoming = AddrIncoming::new(addr).unwrap_or_else(|e| {
        panic!("error binding to {}: {}", addr, e);
    });
    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. Higher-order functions and polymorphic code makes things harder. Rust's traits in particular, which are similar to interfaces and abstract classes in other languages, 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`:

```
pub trait Subscriber {
  unsafe fn downcast_raw(&self, id: TypeId) -> Option<*const ()> {
    if id == TypeId::of::<Self>() {
      Some(self as *const Self as *const ())
    } else {
```

```
      None
    }
  }
}

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 _))
      }
    }
  }
}
```

The default implementation of `downcast_raw` invoked in `downcast_ref` is safe, as it ensures to only return valid pointers, and is easy to audit. The danger lies when other implementations of this trait don't satisfy the safety requirements. Hence, `downcast_ref` is not actually safe in all contexts—and auditing this trait alone is insufficient; we need to audit all its implementations too.

4: We need to track effects across packages. Reasoning about potentially dangerous code within a single crate is hard and non-local. Consider CVE-2021-45712 [36] in the `rust-embed` project. `rust-embed` is a Rust custom derive macro—a type of procedural macro that automates the implementation of traits for structs, enums, or unions. It takes a user-defined directory and either (1) embeds files into the binary in release mode or (2) reads files directly from the file system 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 [43] and CVE-2021-45712 [36] describe in detail).

This was easy to miss in an audit given many intertwined crates were involved. The `rust-embed` crate defines the `RustEmbed` trait for the procedural macro, implemented in the `rust-embed-impl` crate, which calls into the function `read_file_from_fs` from `rust-embed-util` to access the file system. In debug mode, insufficient filepath sanitization in the buggy trait implementation allows unrestricted access to files beyond the configured directory. While `read_file_from_fs` could have enforced access checks itself, instead of transfering this responsibility to its callers, there is no mechanism for reviewers to indicate this while auditing. This function is not fundamentally dangerous; its safety is context-sensitive and depends on how it is invoked.

Spotting such bugs during auditing requires reasoning across multiple crates, trait implementations, and runtime configurations, and existing tools provide little assistance: (1) simple pattern searches (i.e., with `grep`) fail as there is no `unsafe` code or effectful standard library call involved in the bug's location, and (2) tools like Cargo Vet focus on auditing packages in isolation—and fundamentally this breaks down on many real-world packages.

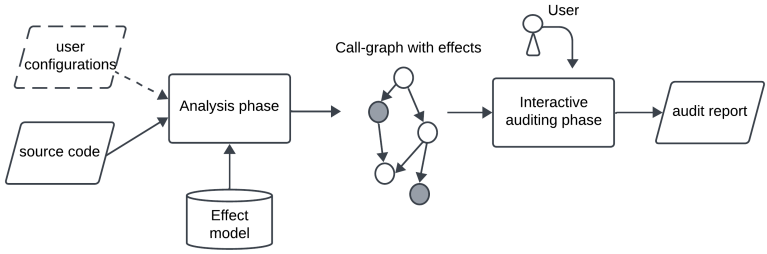


Fig. 2: Cargo Scan overview.

2.2 Our Solution

Thus, while Cargo Vet provides clear benefit to users, it doesn't help auditors and developers in the process of actually reviewing code. We developed Cargo Scan to assist in the lengthy and tedious auditing process. Our goal is to give developers an intermediate step in the space between purely manual inspection during auditing and static analysis-based bug-finding tools used by other ecosystems beyond Rust. Our semi-automated architecture is summarized in Figure 2.

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.

3.1 Effect Model

An *effect* is any function behavior beyond what appears in its signature (input/output types), occurring during execution or afterwards. As Rust permits mutation via explicit mutable reference types like `&mut T` and mutable cells like `Cell<T>`, we do not consider mutation itself as an effect by default. Although interior mutability could be considered an effect, mutation inside interior mutable abstractions, like `Cell`, `UnsafeCell`, and `Mutex`, cannot alone lead to memory unsafety or other system side effects. Such abstractions ultimately get flagged when using unsafe trait functions or other potentially dangerous operations.

Our effect model consists of three categories of effects, as summarized in Table 1: (i) *unsafe effects*, (ii) *system effects*, and (iii) *higher-order effects*. We conservatively overapproximate the sink patterns for our analysis by comprehensively inspecting the standard library's modules to find which of them invoke syscalls. These patterns are (in effect) extended further whenever context-sensitive effects from dependencies are propagated to the target package. Additionally, modelling closures and function pointers as effects at creation time

Table 1: Cargo Scan's effect model.

Effect Model	
Unsafe Effects	FFICall, FFIDecl, StaticExt, StaticMut, UnsafeCall, UnionField, RawPointer
System Effects (SinkCall)	std::fs, std::io, std::os, std::ffi, std::net, std::env, std::arch, std::path, std::mem, std::simd, std::panic, std::process, std::backtrace, std::intrinsics, libc, winapi
Higher-order Effects	FnPtrCreation, ClosureCreation

makes auditing easier; it's easier to audit a closure passed to higher-order functions like `map` than it is to audit every application of it. Indeed, in our experience this design choice greatly simplifies auditing, as, in Rust, many closures are purely functional code. Modelling them as effects at use time would require auditing all locations that perform a map or filter over a generic function, including marking `map` and `filter` in the standard library as effectful, leading to many false positives. In sum, the effect model is sufficient for the fundamental types of effects that can break Rust's safety guarantees and introduce security risks, like memory manipulation, system interactions, or communication with other languages. We describe in detail these above categories in the Appendix of our arXiv version [58].

3.2 Effect Analysis

Our analysis operates directly on a *target package*'s source code in a single, mostly syntactic pass. Cargo Scan parses the input package into an AST and performs a lightweight, pattern-matching traversal to identify effects. For example, each call or expression node is compared against a predefined set of canonical library paths (e.g., `std::fs`) or operation categories (e.g., `RawPointer`) in our effect model. Performing our analysis on AST instead of a lower representation isn't fundamental, but offers some useful advantages. It facilitates the direct mapping of effects back to source code and the precise calculation of audited lines of code before desugaring. Most effects can be sufficiently identified via pattern-matching, but when necessary, Cargo Scan leverages semantic information from `rust-analyzer` [11], which operates on both HIR and MIR. This reduces false positives by resolving types, names, and disambiguating constructs such as raw pointer dereferences or identically named functions and trait methods.

Cargo Scan also builds a call graph for the input package and associates each node with a set of effects inline. We implement a variation of taint and effect analysis, where the identified effects can be considered as the tainted sources and the publicly exposed functions of a crate are the sink locations. Rust's type system makes mutation and aliasing tractable, and thus, allows us to perform a scalable and lightweight intraprocedural analysis without pointer, alias, and data-flow analyses, and capture a wider variety of dangerous behaviors.

3.3 Safety Annotations

Cargo Scan provides users with three safety annotations to mark effects during an audit and indicate their safety: (i) *safe*, (ii) *unsafe*, or (iii) *caller-checked*.

Safe/Unsafe Annotation. The first two annotations – *safe* and *unsafe* – are applicable when an effect's safety can be determined *locally* i.e., from the surrounding code. In the snippet below, the `get_unchecked` effect is safe, because the vector's length is checked before accessing it.

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

Caller-Checked Annotation. The third annotation – *caller-checked* – is applicable when the code's safety (or lack thereof) cannot be established in isolation and depends on the *calling context*. For example, the function

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

may be called innocuously to read a configuration file, or mischievously to read a user's private keys, depending on the value of the parameter `f`. Such effects are annotated as *caller-checked* and Cargo Scan utilizes the call-graph from the analysis phase to propagate them to all callers of the function (here, `read_file`).

3.4 Interactive Auditing

Cargo Scan facilitates a systematic vetting of the identified effects by allowing the user to interactively audit the code at effectful locations. We initially implemented Cargo Scan as a command-line interface similar to interactive staging for `git add` for the interactive auditing. After using that interface to audit numerous crates, we realized that in practice navigating the source code and effects, as well as tracking our progress during auditing was too restrictive. Thus, we implemented a VSCode extension (Figure 1) to tackle these limitations and leverage the interactive features of an editor for more flexibility.

3.5 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 their 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. Cargo Scan tracks public caller-checked functions from dependencies that are transitively reachable from the top-level package.

Default Audits. Auditing all of a project's dependencies in full produces the most precise accounting of possible effects, but requires significant effort

at scale. Cargo Scan automates this process by generating *default* audits—sound, conservative approximations of manual reviews—by marking every effect as caller-checked and propagating them up the stack. Treating all effects as context-sensitive computes every potential propagation path and captures the upper bound of the code requiring review for all possibly dangerous operations. Consequently, users can restrict their attention in auditing only those public caller-checked functions from dependencies called in their application. This allows Cargo Scan to isolate only relevant code for auditing across dependencies, similarly to how we identify effectful locations within a single package. Crates that yield overly conservative default audits can be audited first, and any library calls deemed safe are then removed from dependent packages' effect sets.

4 Empirical Evaluation

We evaluate Cargo Scan by asking three questions:

- How common are effects in popular packages (§4.1)?
- Is context-sensitive effect propagation valuable (§4.2)?
- Does Cargo Scan make auditing more efficient (§4.3)?

Experimental setup. We ran the experiments of §4.1 on a machine with AMD Ryzen ThreadRipper 3960X with 24 cores (and 48 threads), 128GB RAM, with Ubuntu 24.04 and Rust 1.90. We used custom scripts with 16 parallel threads (due to memory restrictions). We performed the audits of sections §4.2 and §4.3 both on the previous machine and on another with Intel i7-8650U, 16GB RAM, with Ubuntu 24.04 and Rust 1.90. All experiments were run as of October, 2025.

4.1 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 10K most downloaded packages on `crates.io`, the central public Rust crate repository. For our ecosystem analysis, we analyzed each crate individually, without tracking effects across dependencies.

Concentration of effects. Figure 3 shows the distribution of crates by total number of effects, divided by order of magnitude. We find that a large number of crates are entirely or mostly *pure*, i.e., effect-free: in particular, slightly over a third of crates (3434) contain no instances of side effects, and over a quarter (2778) contain under 10 effect instances. Further, most effects (approx. 85%) are concentrated in a small number of crates (approx. 3%).

Frequency of effect types. Figure 4a shows the frequency with which different effects occur in the most popular packages. We use the effect categories described in §3.1, and further distinguish `SinkCall`s 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

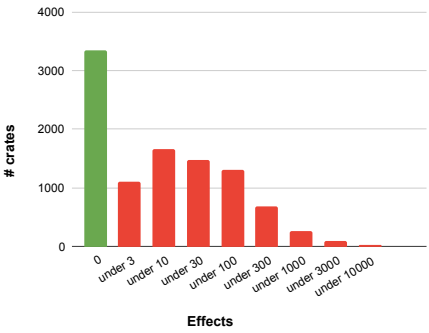


Fig. 3: Distribution of top 10K crates by number of effects.

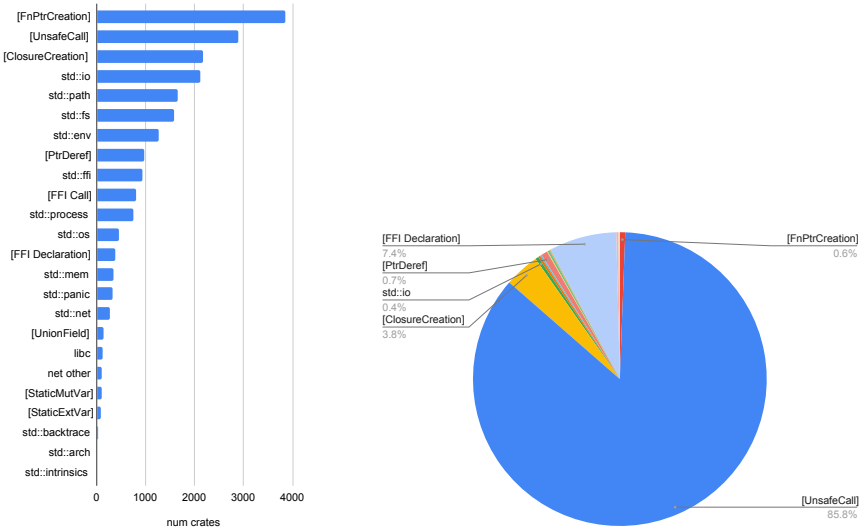
function pointer which may be effectful). The effects which affect the greatest number of *crates* are higher-order effects (function pointers and closures), unsafe function calls, standard I/O operations (sinks), and raw pointer dereferences as shown in Figure 4a. The results are slightly different when broken down by number of *instances* of each effect across all top 10K crates, as demonstrated in Figure 4b; in this case FFI function declarations rise to the top, due to a small number of crates which consist mainly of FFI bindings. To gain a deeper insight, we also filtered out crates particularly dedicated for FFI bindings, commonly known as *sys-* crates. The total number of crates that contain an FFI declaration effect drops from 379 to 190, and the total frequency of this effect type across the top 10K crates is now 1.7% compared to the initial 7.4%.

4.2 Are Context-Sensitive Effects Valuable?

Context-sensitive effects (tracked via the *caller-checked* annotation §3.3) give auditors a more precise model of program safety, but propagating effects to their call-sites may require more auditing effort. To evaluate this we ask:

- **Q1**: What percentage of effects are caller-checked?
- **Q2**: What is the average call-stack we audit for caller-checked effects?
- **Q3**: How many caller-checked effects reach a crate's public API?

For this experiment we audited `hyper`, a widely used HTTP library, and all its dependencies in its default build in a topologically sorted order (seven crates in total), and counted the effects that require context-sensitive reasoning. To answer **Q1**, among the packages we audited, the average percentage of caller-checked effects was 30.8%. Regarding **Q2**, we had to inspect and annotate an arithmetic mean of 3.1 locations per effect including its origin. Overall, this means most effects are only propagated to a small number of locations, i.e., potentially dangerous side effects are well-encapsulated, and reasoning about them requires only modest additional effort. Finally, for **Q3**, fully audited packages (i.e., for which we audited all effects) had on average 5.2% of their public functions marked as



(a) Number of crates with at least one occurrence of the effect in the top 10K crates.

(b) Relative frequency of effects by total instances across all top 10K crates.

Fig. 4: Distribution of side effects by type.

caller-checked. Although caller-checked effects across the crate boundary are less common than caller-checked effects within a crate, this finding still suggests the importance of making audits composable by surfacing such effects in reports.

4.3 Does Cargo Scan make Auditing more Efficient?

As a proxy for measuring Cargo Scan's use in practice, we instead measure auditing efficiency in terms of the total amount of audited code, which directly impacts auditing effort and time. To evaluate Cargo Scan we use the same case study as in the experiment of section §4.2, but instead of using the default build we enable *all* optional features to account for all cases, and generate default audits for all dependencies. This overapproximates all their potential effects and their propagation in parent packages, i.e., caller-checked effects in calls *across* package boundaries (§3.5). Although default audits conservatively result in more audit locations, Cargo Scan only presents those that are reachable from the root crate, in this case `hyper`, so we focus our auditing effort to a small subset of the package set. Specifically, from a total of 30 packages for `hyper` and its normal and optional dependencies, Cargo Scan only presents 17 of them for auditing, as shown in Figure 6. The edge weights represent public functions marked as caller-checked in dependent crates.

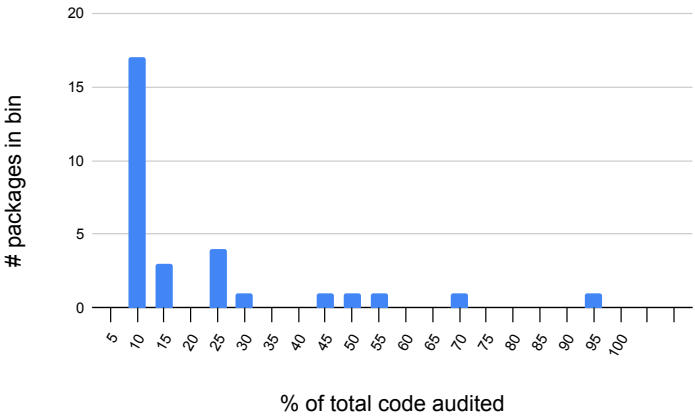


Fig. 5: Probability density function for the percentage of code audited. The bars correspond to the number of packages whose percentage code audited is in the range $(x-5, x]$ (e.g., in the range $(0\%, 5\%]$).

Auditing approach. We audited the crates in their topological order, i.e., we audit a crate's dependencies before the crate itself. In this case, the bottom crates of the dependency graph contain the highest percentage of effects and are "reused" by other packages. When effects could not be resolved locally in a crate, we reviewed call-sites in parent crates, recursively, until we could determine their safety. Cargo Scan automatically updates the set of reachable effects across dependencies and eliminates any corresponding audit locations during auditing. This significantly minimizes the total amount of audited code across the dependency graph, especially when it includes large libraries (e.g., `tokio`) of which only a small fraction of the API is called.

Quantifying auditing effort. We determine auditing effort by measuring how much code we need to review as a percentage of the *total code* of the package as analyzed by Cargo Scan. Measuring "reviewed code" can be tricky as auditing an effect requires understanding its surrounding context—including things like the parameters of a function and how the result of an effect is used. Hence, rather than mark individual lines as relevant, we approximate "reviewed code" by assuming we have to inspect each entire function an effect appears in. To quantify the total amount of audited code in our case study, we collect all functions with effects in `hyper` itself. Since we use default audits as a starting point for each dependency, though, we handle them differently than the top-level crate: we track the effects ultimately marked as *safe* or *unsafe* during our review—indicating we manually audited their containing functions—along with all caller-checked effects still reachable from the root.

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

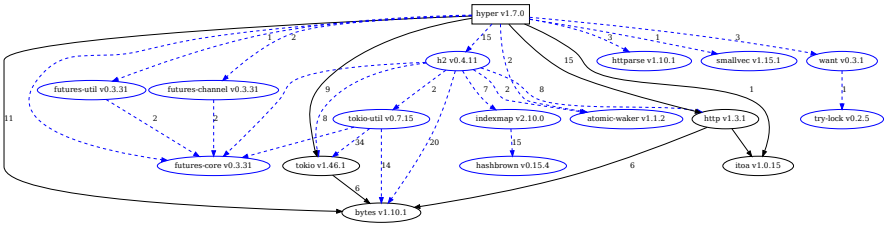


Fig. 6: Subset of the dependencies with effects that flow into `hyper`. Edge weights represent the number of public caller-checked (CC) functions.

effort is concentrated in a few crates with many effects. The arithmetic mean percentage lines of code we audit per package is 13.1%, with a median of 0.4%. The full distribution is in Figure 5. We see a skew here consistent with what we expect from §4.1. Specifically, the packages with a high percentage of audited code are primarily packages which are more commonly reused.

Cross-package audits. Counterintuitively, we find that using Cargo Scan's default audits actually reduces the total amount of code we need to audit; we audited on average 12.4% of lines of code. In contrast, in the experiment of section §4.2, we audited an average of 28.1% for the same packages (Table 2). From this, we conclude that Cargo Scan's default audit generation improves the auditing workflow and reduces the total amount of code that needs inspection across dependencies.

Threats to validity. Auditing is still a manual process and results may be impacted by various factors, like auditors' expertise, complexity of the input crate(s), or their intended threat model. Cargo Scan provides some objective guarantees that may lead to more converging audit results, as it prevents missing relevant code, tracks effects across function calls and crate boundaries, and collects audit results in a composable audit file. However, it cannot further assist users with their final judgement and decision about the safety of each effect.

5 Discussion and Future Work

Cargo Scan helps developers audit Rust code more efficiently. It does so by identifying potentially dangerous side effects through static analysis and tracking context-dependent safety across the call graph. While we found Cargo Scan effective in practice, there are some interesting directions for future improvements. Currently, Cargo Scan does not help developers audit "deltas", i.e., changes between different versions of a crate. Different releases of the same crate usually share great similarities, and extending the tool to focus only on code locations directly or indirectly affected by a change could further reduce the auditing burden. Finally, another direction is to analyze build-time effects. For exam-

Table 2: Audited functions and LoC between default (§4.3) and full (§4.2) audits.

Crate	Default Audits		Full Audits	
	Fns	LoC	Fns	LoC
hyper-1.7.0	30	1453	54	1816
http-1.3.1	9	258	27	583
fnv-1.0.7	0	0	0	0
http-body-1.0.1	0	0	2	14
bytes-1.10.1	68	1425	114	1858
itoa-1.0.15	1	9	1	9
pin-project-lite-0.2.16	0	0	2	12

ple, Rust crates can execute custom build scripts (`build.rs`) that sometimes generate potentially dangerous code.

6 Related work

Empirical analysis and testing of the Rust ecosystem. Rust packages are centrally organized by `crates.io`, with substantial research analyzing `unsafe` code in this ecosystem. Prior work has analyzed `unsafe` blocks usage [10,1], and resulting bugs [39]. In contrast to [10], we also treat closure and function pointer creation as effects, construct a complete crate-level call graph for context-sensitive effects propagation, and capture dangerous behavior beyond unsafe code alone. Similarly, while [1] resembles our effect model and categorizes unsafe code using queries on the compiler's MIR representation, their analysis is limited to unsafe operations. Others research targets specific vulnerabilities, like Rust CVEs [54], yanked crates [29], and semantic version violations [37], whereas we additionally analyze system effects in the ecosystem. Dynamic testing tools, like RULF [20] and SyRust [47], synthesize API call sequences for fuzzing specific bug classes (e.g., unreachable code and out of bounds checks), complementing static analysis approaches and auditing.

Effect analysis. Prior work on side effect analysis—that is, determining how functions or expressions impact a program's state beyond its return value, including memory, I/O, and control flow—focuses on languages like C/C++ [41,7,23] and Java [38,40,25,33,46,55,48], among others. To account for mutation, many of these works aim to compute sets of objects that can be read and modified by each program statement; this relies on computationally expensive points-to, alias, and escape analyses, and produces verbose output. These techniques are not necessary in Rust, where mutation and aliasing are delegated to the type system and are tractable outside of unsafe blocks. Instead, we identify sets of effects at the function level, which in our experience is more useful for auditing.

Static analysis for security audits. There is prior work on static analyses designed specifically for security auditing, for example, in the context of

Java with tools like JoanAudit [49], Andromeda [50], FindBugs [18] and others [12,30,51,27,19]. Most of this work implements taint and information-flow analysis based on preconfigured sets of vulnerable patterns. Although these tools aim to improve the efficiency and scalability of static analysis on Java bytecode, they lack an interactive auditing model that guides users and allows for incremental and composable audit logs.

Supply chain security. Supply chain security spans a wide range of tools targeting different components. In the Rust ecosystem, RustSec [45] tracks vulnerabilities, while tools like Cackle [24], Cargo Audit [44], Cargo Crev [9], JFrog's Pyrsia [6], and Lib.rs [28] provide dependency management, auditing metadata, and API usage checks. In other ecosystems, particularly JavaScript, research focuses on publishing and delivery [57,8,56], though dynamic features, like runtime dependency injection, hinder static analysis. We target manual code auditing.

Cargo Scan's safety annotations can be compared with the "auditing criteria" used by Cargo Vet [34]; in particular, they are most closely aligned with the built-in criteria "safe-to-run" and "safe-to-deploy". We don't support Cargo Vet's other custom criteria, like "crypto-reviewed", targeted for functional correctness, but our effect model could be extended further, for example by treating critical cryptographic functions as effects. For Go, Google's Capslock [16,14,15] performs a similar effect analysis. However, unlike our approach, it is limited to a CLI interface without composable audit logs; and it treats reflection as a runtime effect, while we statically resolve or overapproximate trait-based effects.

Semantics and verification. Much recent effort has gone into formalizing the semantics of Rust's ownership mechanisms, including RustBelt [22], Oxide [53], Stacked Borrows [21], and the Miri interpreter for detecting undefined behavior [42]. 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 [52], Prusti [2], Aeneas [17], Verus [13], Hacspec [32], Flux [26], and RustHornBelt [31]. Most verification tools only apply to safe Rust — with the exception of [31] — as unsafe Rust is difficult to formalize. Verification is labor-intensive and does not scale to the ecosystem, but our techniques can help verifiers target code that is most critical.

Acknowledgments. We thank Bobby Holley for insightful discussions that sparked our effort, the anonymous reviewers for their comments, and Pu Guo for his contribution to the macro analysis. This work was supported in part by the Berkeley Center for Digital Assets, Mozilla, and the National Science Foundation (grant #s 2048262, 2120642, 2155235, 2327336, and 2327338).

References

1. Astrauskas, V., Matheja, C., Poli, F., Müller, P., Summers, A.J.: How do programmers use unsafe rust? Proceedings of the ACM on Programming Languages

4(OOPSLA), 1–27 (2020)
2. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging rust types for modular specification and verification. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–30 (2019)
3. Axum Contributors: crates.io - axum. https://crates.io/crates/axum (2019), [Accessed 2024-02-07]
4. Bae, Y., Kim, Y., Askar, A., Lim, J., Kim, T.: Rudra: finding memory safety bugs in rust at the ecosystem scale. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. pp. 84–99 (2021)
5. Balasubramanian, A., Baranowski, M.S., Burtsev, A., Panda, A., Rakamarić, Z., Ryzhyk, L.: System programming in rust: Beyond safety. In: Proceedings of the 16th Workshop on Hot Topics in Operating Systems. pp. 156–161 (2017)
6. Chin, S.: Closing the Supply Chain Security Loop with Rust @ Rust Nation UK | JFrog — jfrog.com. https://jfrog.com/community/rust/closing-the-supply-chain-security-loop-with-rust-and-pyrsia/ (2023), [Accessed 2023-12]
7. Choi, J.D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 232–245. POPL '93, Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/158511.158639, https://doi.org/10.1145/158511.158639
8. Cogo, F.R., Oliva, G.A., Hassan, A.E.: An empirical study of dependency downgrades in the npm ecosystem. IEEE Transactions on Software Engineering **47**(11), 2457–2470 (2019)
9. Crev development team: GitHub - crev-dev/cargo-crev: A cryptographically verifiable code review system for the cargo (Rust) package manager. — github.com. https://github.com/crev-dev/cargo-crev (2024), [Accessed 14-11-2024]
10. Evans, A.N., Campbell, B., Soffa, M.L.: Is rust used safely by software developers? In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 246–257 (2020)
11. Ferrous Systems: rust-analyzer — Rust Language Server Protocol. https://rust-analyzer.github.io/ (2026), [Accessed 2024-2-8]
12. GitHub: spotbugs/spotbugs: Static analysis tool to find bugs in Java code — github.com. https://github.com/spotbugs/spotbugs
13. GitHub: verus-lang/verus: Verified Rust for low-level systems code — github.com. https://github.com/verus-lang/verus (2022), [Accessed 2023-2-03]
14. Google: Capslock: What is your code really capable of? — security.googleblog.com. https://security.googleblog.com/2023/09/capslock-what-is-your-code-really.html (2023), [Accessed 14-11-2024]
15. Google: GopherCon 2023 — events.bizzabo.com. https://events.bizzabo.com/449995/agenda/session/1160360 (2023), [Accessed 14-11-2024]
16. Google: GitHub - google/capslock — github.com. https://github.com/google/capslock (2025), [Accessed 14-11-2024]
17. Ho, S., Protzenko, J.: Aeneas: Rust verification by functional translation. Proceedings of the ACM on Programming Languages **6**(ICFP), 711–741 (2022)
18. Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Not. **39**(12), 92–106 (Dec 2004). https://doi.org/10.1145/1052883.1052895, https://doi.org/10.1145/1052883.1052895
19. Huang, W., Dong, Y., Milanova, A.: Type-based taint analysis for java web applications. In: Proceedings of the 17th International Conference on Fundamen-

tal Approaches to Software Engineering - Volume 8411. pp. 140–154. Springer-Verlag, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54804-8_10, https://doi.org/10.1007/978-3-642-54804-8_10
20. Jiang, J., Xu, H., Zhou, Y.: RULF: Rust library fuzzing via API dependency graph traversal. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 581–592. IEEE (2021)
21. Jung, R., Dang, H.H., Kang, J., Dreyer, D.: Stacked borrows: An aliasing model for rust. Proc. ACM Program. Lang. **4**(POPL) (dec 2019). https://doi.org/10.1145/3371109, https://doi.org/10.1145/3371109
22. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Rustbelt: Securing the foundations of the rust programming language. Proc. ACM Program. Lang. **2**(POPL) (dec 2017). https://doi.org/10.1145/3158154, https://doi.org/10.1145/3158154
23. Landi, W., Ryder, B.G., Zhang, S.: Interprocedural modification side effect analysis with pointer aliasing. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation. pp. 56–67. PLDI '93, Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/155090.155096, https://doi.org/10.1145/155090.155096
24. Lattimore, D.: Making Rust supply chain attacks harder with Cackle — davidlattimore.github.io. https://davidlattimore.github.io/making-supply-chain-attacks-harder.html (2023), [Accessed 2023-12-06]
25. Le, A., Lhoták, O., Hendren, L.: Using inter-procedural side-effect information in jit optimizations. In: Proceedings of the 14th International Conference on Compiler Construction. pp. 287–304. CC'05, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/11406921_22, https://doi.org/10.1007/11406921_22
26. Lehmann, N., Geller, A.T., Vazou, N., Jhala, R.: Flux: Liquid types for rust. Proceedings of the ACM on Programming Languages **7**(PLDI), 1533–1557 (2023)
27. Lerch, J., Hermann, B., Bodden, E., Mezini, M.: Flowtwist: efficient context-sensitive inside-out taint analysis for large codebases. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 98–108. FSE 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2635868.2635878, https://doi.org/10.1145/2635868.2635878
28. Lesiński, K.: Lib.rs – home for Rust crates. https://lib.rs/ (2024), [Accessed 14-11-2024]
29. Li, H., Cogo, F.R., Bezemer, C.P.: An empirical study of yanked releases in the Rust package registry. IEEE Transactions on Software Engineering **49**(1), 437–449 (2022)
30. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14. p. 18. SSYM'05, USENIX Association, USA (2005)
31. Matsushita, Y., Denis, X., Jourdan, J.H., Dreyer, D.: Rusthornbelt: a semantic foundation for functional verification of rust programs with unsafe code. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 841–856. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523704, https://doi.org/10.1145/3519939.3523704
32. Merigoux, D., Kiefer, F., Bhargavan, K.: Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust. Ph.D. thesis, Inria (2021)

33. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol. **14**(1), 1–41 (Jan 2005). https://doi.org/10.1145/1044834.1044835, https://doi.org/10.1145/1044834.1044835
34. Mozilla: Cargo Vet — mozilla.github.io. https://mozilla.github.io/cargo-vet/ (2024), [Accessed 2023-11-13]
35. das Neves, R.M.: Extending the unsafe keyword — internals.rust-lang.org. https://internals.rust-lang.org/t/extending-the-unsafe-keyword/20638 (2024), [Accessed 26-03-2025]
36. NIST: Cve-2021-45712 — nvd. https://nvd.nist.gov/vuln/detail/CVE-2021-45712 (2021), [Accessed 2024-2-8]
37. Nowak, T., Predrag Gruevski: Semver violations are common, better tooling is the answer. https://predr.ag/blog/semver-violations-are-common-better-tooling-is-the-answer/ (2023), [Accessed 2024-2-8]
38. Pearce, D.J.: Jpure: A modular purity system for java. In: Knoop, J. (ed.) Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6601, pp. 104–123. Springer (2011). https://doi.org/10.1007/978-3-642-19861-8_7, https://doi.org/10.1007/978-3-642-19861-8_7
39. Qin, B., Chen, Y., Yu, Z., Song, L., Zhang, Y.: Understanding memory and thread safety practices and issues in real-world rust programs. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 763–779 (2020)
40. Rountev, A.: Precise identification of side-effect-free methods in java. In: Proceedings of the 20th IEEE International Conference on Software Maintenance. pp. 82–91. ICSM '04, IEEE Computer Society, USA (2004)
41. Rountev, A., Ryder, B.G.: Points-to and side-effect analyses for programs built with precompiled libraries. In: Proceedings of the 10th International Conference on Compiler Construction. pp. 20–36. CC '01, Springer-Verlag, Berlin, Heidelberg (2001)
42. Rust team: GitHub - rust-lang/miri: An interpreter for Rust's mid-level intermediate representation — github.com. https://github.com/rust-lang/miri (2026), [Accessed 2026-01-21]
43. RustSec team: Rustsec-2021-0126 — rustsec.org. https://rustsec.org/advisories/RUSTSEC-2021-0126.html (2021), [Accessed 2024-2-8]
44. RustSec team: rustsec/cargo-audit at main · rustsec/rustsec — github.com. https://github.com/rustsec/rustsec/tree/main/cargo-audit (2024), [Accessed 14-11-2024]
45. RustSec team: Rust Security Advisory Database — rustsec.org. https://rustsec.org/ (2026), [Accessed 2023-11-13]
46. Sălcianu, A., Rinard, M.: Purity and side effect analysis for java programs. In: Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 199–215. VMCAI'05, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_14, https://doi.org/10.1007/978-3-540-30579-8_14
47. Takashima, Y., Martins, R., Jia, L., Păsăreanu, C.S.: Syrust: automatic testing of rust libraries with semantic-aware program synthesis. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 899–913 (2021)

48. Tao, B., Qian, J., Zhou, X.: Side-effect analysis with fast escape filter. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis. pp. 15–20. SOAP '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2259051.2259054, https://doi.org/10.1145/2259051.2259054
49. Thomé, J., Shar, L.K., Bianculli, D., Briand, L.C.: JoanAudit: a tool for auditing common injection vulnerabilities. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 1004–1008. ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3106237.3122822, https://doi.org/10.1145/3106237.3122822
50. Tripp, O., Pistoia, M., Cousot, P., Cousot, R., Guarnieri, S.: Andromeda: Accurate and scalable security analysis of web applications. In: International Conference on Fundamental Approaches to Software Engineering. pp. 210–225. Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_15
51. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: effective taint analysis of web applications. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 87–97. PLDI '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1542476.1542486, https://doi.org/10.1145/1542476.1542486
52. VanHattum, A., Schwartz-Narbonne, D., Chong, N., Sampson, A.: Verifying dynamic trait objects in rust. In: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. pp. 321–330 (2022)
53. Weiss, A., Gierczak, O., Patterson, D., Ahmed, A.: Oxide: The essence of rust. arXiv preprint arXiv:1903.00982 (2019)
54. Xu, H., Chen, Z., Sun, M., Zhou, Y., Lyu, M.R.: Memory-safety challenge considered solved? an in-depth study with all Rust CVEs. ACM Transactions on Software Engineering and Methodology (TOSEM) **31**(1), 1–25 (2021)
55. Yang, J., Hotta, K., Higo, Y., Kusumoto, S.: Revealing purity and side effects on functions for reusing java libraries. In: Schaefer, I., Stamelos, I. (eds.) Software Reuse for Dynamic Systems in the Cloud and Beyond - 14th International Conference on Software Reuse, ICSR 2015, Miami, FL, USA, January 4-6, 2015. Proceedings. Lecture Notes in Computer Science, vol. 8919, pp. 314–329. Springer (2015). https://doi.org/10.1007/978-3-319-14130-5_22, https://doi.org/10.1007/978-3-319-14130-5_22
56. Zahan, N., Zimmermann, T., Godefroid, P., Murphy, B., Maddila, C., Williams, L.: What are weak links in the npm supply chain? In: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. pp. 331–340 (2022)
57. Zimmermann, M., Staicu, C.A., Tenny, C., Pradel, M.: Small world with high risks: A study of security threats in the npm ecosystem. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 995–1010 (2019)
58. Zoghbi, L., Thien, D., Jhala, R., Stefan, D., Stanford, C.: Auditing rust crates effectively (2026), https://arxiv.org/abs/2602.06466

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.