arXiv:2602.06466v1 [cs.PL] 6 Feb 2026

高效审计 Rust crate

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

[1] 美国加利福尼亚大学圣迭戈分校, {lzoghbi,dthien,rjhala,dstefan}@ucsd.edu[2] 美国加利福尼亚大学戴维斯分校，cdstanford@ucdavis.edu

摘要。 我们介绍了 Cargo Scan，这是首个旨在帮助开发者审计第三方 Rust 代码的交互式程序分析工具。以 Rust 编写的真实系统依赖成千上万个传递依赖。这些依赖在 Rust 中和在其他语言（如 C 或 JavaScript）中一样危险，而如今审计这些依赖意味着必须手动检查每一行代码。不过，与大多数工业语言不同，我们可以利用 Rust 的类型系统和模块系统，将开发者需要检查的代码范围缩小到可能存在危险的代码。Cargo Scan 将这类潜在危险代码建模为效应，并针对 Rust 执行副作用分析，以识别效应并追踪其跨 crate 和模块边界的传播。在大多数情况下（69.2%），开发者可以检查标记出的效应，并在本地判断代码是否存在潜在危险。然而在某些情况下，效应是否安全取决于调用上下文——即函数如何被调用，而调用方可能是开发者之后才导入的 crate。因此，Cargo Scan 使用调用图追踪依赖上下文的信息，并将审计结果收集到可组合、可复用的审计文件中。本文介绍了我们使用 Cargo Scan 审计 Rust crate 的经验。具体而言，我们审计了广受欢迎的 HTTP 客户端和服务器 crate hyper 及其所有依赖项；我们的经验表明，与审计整个 crate 相比，Cargo Scan 可将审计潜在危险代码的负担降低至代码行数中位数的 0.2%。从更广泛的 Rust 生态系统来看，我们发现 Cargo Scan 可以自动将 crates.io 上排名前一万的 Cargo 包中 ∼3.5K 个判定为安全；对于需要人工检查的 crate，我们发现大多数潜在危险的副作用集中在其中约 3% 的 crate 中。

1 引言

Rust 是 C 和 C++在系统编程领域的一个极具吸引力的替代方案，因为它通过强大的静态类型、模块级封装、数组边界检查和组合式错误处理，默认提供内存安全。然而，现代软件主要由他人编写，而任意第三方 Rust 代码也可能像 C 或 C++一样危险。首先，Rust 提供了一个**不安全**关键字，允许开发者绕过其通常的安全保证，这意味着

必须信任第三方代码能够维护 Rust 复杂的内存安全不变量 [10,1]。其次，内存安全并非唯一需要考虑的问题；内存安全的第三方 Rust 代码可以调用操作系统来读写文件、连接网络端点并执行任意二进制程序——所有这些行为都可能带来危险。

问题：审查第三方 Rust。 目前，Rust 团队审计第三方代码（crate），以降低供应链攻击的风险。这是一个人工且费力的过程，因为现实世界的系统（传递地）依赖于数百至数千个 crate。Mozilla 的 Cargo Vet 等工作 [34] 通过允许 Rust 团队记录其结果并对 crate 的安全性予以签核来简化审计，从而使其他人能够信任这些结果并降低其工作量。然而在实践中，这种基于信任的方法并不能真正简化审查过程（§2）。此外，由于 Cargo Vet 的审计结果只是带有非结构化注释的批准，即使已审计的 crate 也可能需要再看一遍，因为某个 crate 公开暴露的函数可能仅在特定上下文中才是安全的。最后，识别哪些代码需要审计——即使是在一个**不安全**块 [35]内——然后判定其实际安全性，仍然具有挑战性。

审计目标。 我们为工具辅助的审计流程确定了两个关键目标。

1. **审计真正重要的代码。** 开发者不应该逐行检查所有代码；他们只需审计确实可能存在危险的代码。大多数广泛使用的语言几乎无法做到这一点（例如，C/C++ 的内存不安全、Java 的反射和类加载，以及 JavaScript 的大多数特性，都让人难以查看一段代码后确信它是安全的）；我们的目标是利用 Rust 的类型系统和模块系统，只审计一小部分代码，就能获得全局安全属性。

2. **追踪依赖上下文的信息。** 开发者不应该审计已经审计过的代码。目前他们却不得不这么做，因为有些函数有时才安全，也就是说，是否安全取决于调用上下文。审计过程应当具有**可组合性**，并在审计报告中追踪这类信息；当开发者使用这些可能存在危险的函数时，报告应提醒他们，从而将上下文敏感性传递给该函数的任何调用方。

高效审计 Rust。 基于这些目标，我们构建了 Cargo Scan，这是一款交互式 Rust 审计工具。Cargo Scan 的核心理念借鉴自函数式编程：我们可以将可能存在危险的操作视为效应，它足够灵活，能够描述各种可能存在危险的操作，从底层的内存操作（例如原始指针解引用）到高层的操作系统调用（例如文件读写）。不过，我们并不在类型系统中追踪效应（例如 Haskell 的做法），而是通过现成的静态分析来追踪效应，使开发者能够专注于重要的代码，同时以上下文敏感的方式追踪和传播效应。本文介绍 Cargo Scan 的设计与实现，并报告我们使用 Cargo Scan 审计 crate 的经验。我们的贡献主要有三点。

1. 效应分析。我们的首项贡献通过一项静态分析，解决了第一个审计目标：识别源代码中的潜在副作用（§3.2）。



图 1：在 VSCode 中使用 Cargo Scan 进行交互式审计。

我们的效应模型（§3.1）和分析建立在长期以来效应分析领域的研究基础之上，但关键在于，它利用 Rust 的类型系统和我们的问题领域，让审计变得切实可行。首先，我们发现，Rust 的借用检查器结合模块级抽象和明确的安全/不安全作用域，使修改和别名问题变得易于处理，这与其他语言截然不同。因此，我们可以专注于函数级分析（将副作用与每次函数调用关联），而不是像 Java 等语言那样需要采用传统的数据级分析 [49,30,50,18,38,40,25,33,46,55,48]。此外，这项分析使我们能够确保不会向用户展示没有潜在危险副作用的代码。其次，我们发现，闭包、函数指针、特征和泛型等复杂语言特性，虽然会使一般情况下的分析变得困难，却不必让审计变得更困难。我们的见解是，将这些特性“提升”到效应模型中，并要求开发者审计相关代码；例如，审计创建闭包的代码，就像审计打开文件的代码一样。

2. 交互式审计。我们的第二项贡献通过交互式审计流程实现了第二个目标，该流程会提示用户关注可能存在危险的效应位置（§3）。随后，用户为每个位置添加注释，指出其是否安全，或者其安全性是否依赖于上下文。Cargo Scan 会将这些信息记录在审计文件中。因此，与 Cargo Vet 不同，开发者无需完全信任某个库，也无需重新审计它：他们可以阅读审计报告（使用 Cargo Scan），只需检查依赖上下文的函数的调用上下文即可。

我们将 Cargo Scan 的交互式流程实现为 VSCode 扩展。这样，开发者便能轻松评估任意特定效应的安全性，并在代码中直接添加注释：他们可以检查周边代码，并利用 rust-analyzer 等集成工具检查类型和定义，或浏览调用图（以及代码库）。我们审计 hyper 网络库及其所有依赖项的经验表明，这对于复杂代码库，以及许多效应具有上下文敏感性（在调用点与定义之间导航至关重要）的代码库尤其有用。根据我们的经验，这也适

用于商业场景；作者已在生产环境中使用 Cargo Scan 审计安全关键产品所用的 Rust crate，历时一年多，并发现它能将拉取请求中新 crate 的代码审查时间从数天缩短至数小时。

3. 评估。我们的第三项贡献是对 Cargo Scan 的评估。评估结果表明，借助工具进行审计可以减轻审计负担，并识别出上下文敏感安全性的情况（§4）。我们首先发现，在使用最广泛的前一万名 Cargo 包中，有 3434 个是纯粹的，也就是说完全没有危险副作用，因此无需任何人工审计即可声明为安全地使用；此外，85% 的副作用仅存在于 3% 的 crate 中。这意味着 Cargo Scan 能以极少甚至无需审计人员投入的方式审查大多数 crate。接下来，我们报告了审计 `hyper`网络库及其所有依赖项的经验。我们的审计表明，30.8% 的函数的安全性具有上下文敏感性，5.2% 的函数的安全性在包边界之间具有上下文敏感性；这意味着，实际审计代码时，跟踪跨函数（及软件包）的效应至关重要。不过，总体而言，这并未造成沉重负担：我们发现，借助工具进行审计所需审查的代码行中位数仅为 0.2%（相对于不使用 Cargo Scan 审计代码而言）。最后，如果排名前一万的 Cargo 包能够代表更广泛的 Rust 生态系统，这就意味着我们可以借助 Cargo Scan，以适度的人工投入提升整个 Rust 的安全门槛。

2 动机

Mozilla、Fastly 和 Google 等组织会在将第三方代码用于 Firefox、Wasmtime 和 Chrome 等安全关键系统之前，进行人工审计。直到最近，由于缺乏共享审计结果的通用平台，各组织的大量审计工作都在重复进行；而仅仅因为某个组织依赖某个库就信任它，并不能提供任何固有保障。

Mozilla 开发了 Cargo Vet，专门解决这一问题 [34]。借助 Cargo Vet，开发者可以依据共享标准审计（或审查）软件包（例如，“可在本地安全运行”或“可安全部署到生产环境”）。随后，该工具会以通用格式记录审计结果，以及受审计的软件包版本（或版本之间的差异）和任何（非结构化）备注。这样一来，各组织便可分担审计负担：团队可以通过导入 受信任实体（例如 Chrome 的审计记录）的审计结果，或将作者标记为可信作者，来跳过对软件包的审计——尤其是在 Mozilla 和 Google 等组织都信任这些作者的情况下。

2.1 人工审计无法扩展

实际上，即使导入外部审计结果并将经验丰富的开发者标记为受信任者，我们仍然需要审计大量代码。在这方面，Cargo Vet 无能为力：即使是类型安全性很强的 Rust 代码，审计起来也仍是一个需要手动完成、耗时费力且容易出错的过程。

1：我们需要审计所有代码，否则就可能漏掉危险代码。考虑审计 hyper HTTP 客户端和服务器库。它是 crates.io 上最受欢迎的库之一，下载量超过 4.06 亿次。该库是

诸如 axum 这样的热门 Web 框架的核心；这些框架使用“#![forbid(unsafe_code)]”来确保所有代码都以 100% 安全 Rust 实现。 [3]”然而，这种安全观念在实践中意义不大；hyper 及其传递依赖确实包含 **不安全**代码。此外，这些软件包中仍包含执行潜在危险操作的代码（例如访问文件系统、网络等的复杂代码），尽管按照 Rust 的标准，这些代码仍被视为安全。因此，我们需要审查 hyper 及其依赖项是否“可以安全部署”——而如今，这意味着要手动筛查横跨 30 个软件包的 16 万行代码。根据我们和其他开发者审计 crate 的经验，使用 grep 等现成工具，很难找到潜在危险代码——显而易见的 **不安全**代码块除外。实际上，这并不意味着要筛查 16 万行代码，而是意味着放弃审计，或意外漏掉潜在危险代码。尤其遗憾的是，正如我们在 §4.1所展示的，许多软件包中的大部分代码都可以完全自动地进行安全审查。

2：我们需要推理调用上下文。 在 hyper 自身内部，有大量调用标准库的代码——用于访问文件系统和网络——但不同于 **不安全**代码，开发者不必明确标注这些代码可能存在危险。例如，下面这段来自 hyper 的代码， 看起来 是安全的：

```
公开函数 bind（addr: & SocketAddr）-> Builder <AddrIncoming >{令 incoming =
AddrIncoming::new(addr).unwrap_否则_else（|e|{ panic!（"绑定 {} 时出错: {}",
addr, e);});Server::builder(incoming)}
```

该函数只是构造一个新的 AddrIncoming，无论是手动检查，还是使用（sem）grep 之类的工具，都很容易跳过它。然而，只有检查 AddrIncoming::new 后，我们才能发现它以及后续的 bind 可能存在危险——该调用会绑定并监听套接字地址。出于同样的原因，我们还需要检查使用 bind 的调用上下文，以及可能使用该函数的调用上下文。

3：我们需要推理复杂类型。高阶函数和多态代码让事情变得更加困难。Rust 的特征尤其如此，它们类似于其他语言中的接口和抽象类，因此很难推理和分析（多态）代码。以 tracing_core 软件包中的 Subscriber 特征定义为例，它为 hyper 等软件包提供应用级跟踪的核心功能：

```
公开 trait Subscriber{不安全 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{公开 fn downcast_ref <T: Any >(&self) ->
Option <&T> {不安全 {let raw = self.downcast_raw(TypeId::of::<T >
())?;
```

```
    如果 raw.is_null(){None}否则 {Some
(&*(raw as *const _))}}}}
```

---

downcast_ref 中调用的 downcast_raw 默认实现是安全的，因为它确保只返回有效指针，且易于审计。危险在于该特征的其他实现可能不满足安全要求。因此，downcast_ref 并非在所有情境下都安全——仅审计该特征并不足够；我们还需要审计它的所有实现。

4：我们需要跟踪跨软件包的效应。在单个 crate 中推理潜在危险代码既困难又非局部。以 CVE- 2021-45712 [36] 中的 `rust-embed` 项目为例。`rust-embed` 是一个 Rust 自定义派生宏——一种过程宏，可自动为结构体、枚举或联合体实现特征。它接收用户定义的目录，然后（1）在发布模式下将文件嵌入二进制文件，或（2）在调试模式下直接从文件系统读取文件。按设计，`rust-embed` crate 及其相关软件包（试图）强制实施文件系统隔离，也就是说，派生宏只能读取用户定义目录中的文件。然而，在调试模式下，该实现未能正确清理输入路径，从而破坏了这一隔离保证（详见 RUSTSEC-2021-0126 [43] 和 CVE-2021-45712 [36]的详细说明）。

由于涉及许多相互交织的 crate，因此审计时很容易忽略这一点。rust-embed crate 为过程宏定义了 RustEmbed trait，该 trait 在 rust-embed-impl crate 中实现；后者调用 rust-embed-util 中的 read_file_from_fs 函数来访问文件系统。在调试模式下，有缺陷的 trait 实现未能充分净化文件路径，因此可以不受限制地访问配置目录之外的文件。read_file_from_fs 本可以自行实施访问检查，而不是把这一职责转交给调用方，但审计时没有机制让审查者指出这一点。这个函数本身并非根本上具有危险性；它是否安全取决于上下文，以及它是如何被调用的。

审计时发现此类漏洞，需要跨多个 crate、trait 实现和运行时配置进行推理，而现有工具几乎无法提供帮助：(1) 简单的模式搜索（即使用 `grep`）会失效，因为漏洞所在位置既没有不安全 代码，也没有涉及效应的标准库调用；(2) Cargo Vet 等工具侧重于单独审计各个软件包——而从根本上说，这种做法无法应对许多真实世界中的软件包。

2.2 我们的解决方案

因此，尽管 Cargo Vet 能为用户带来明显好处，却无法帮助审计人员和开发者实际进行代码审查。我们开发了 Cargo Scan，旨在协助完成耗时且繁琐的审计工作。我们的目标是在纯手动检查与

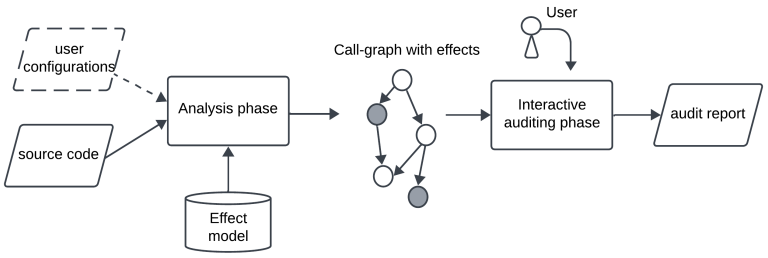


图 2：Cargo Scan 概览。

审计期间使用的静态分析式漏洞查找工具之间，为开发者提供一个中间步骤；其他生态系统也采用了这类工具，而 Rust 生态系统之外同样如此。我们的半自动化架构总结见 图 2。

3 Cargo Scan 概览

Cargo Scan 通过精准定位（仅限）潜在危险代码的执行位置，并引导用户遍历函数调用栈，确保这些代码仅在安全上下文中使用，从而应对第 §2 节中列出的挑战。

3.1 效应模型

效应是指函数签名（输入/输出类型）中未体现的任何函数行为，这些行为可能在执行期间或执行之后发生。由于 Rust 允许通过显式的可变引用类型（如 `&mut T`）以及可变单元（如 `Cell<T>`）进行修改，因此默认情况下，我们不将修改本身视为效应。尽管内部可变性可以被视为一种效应，但内部可变抽象（如 `Cell`、`UnsafeCell` 和 `Mutex`）中的修改本身无法导致内存不安全或其他系统副作用。使用不安全特征函数或其他潜在危险操作时，这些抽象最终会被标记。

我们的效应模型包含三类效应，如表 1所总结：(i)不安全效应、(ii) 系统效应和 (iii) 高阶效应。我们通过全面检查标准库的模块，找出哪些模块会调用系统调用，从而对分析中的汇点模式进行保守的过度近似。每当依赖项中的上下文敏感效应传播到目标包时，这些模式实际上还会进一步扩展。此外，在闭包和函数指针创建时将其建模为效应，可以让审计更容易；审计传递给`map` 等高阶函数的闭包，比审计它的每次调用容易得多。事实上，根据我们的经验，这种设计选择大大简化了审计，因为 Rust 中许多闭包都是纯函数式代码。如果在使用时才将它们建模为效应，就需要审计所有对泛型函数执行 map 或 filter 的位置，包括将标准库中的 `map`和 `filter` 标记为具有效应，这会导致大量

表 1：Cargo Scan 的效应模型。

效应模型	
不安全效应	FFICall、FFIDecl、静态外部项、静态可变项、不安全调用，联合体字段，RawPointer
系统效应（汇调用）	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
高阶效应	函数指针创建、闭包创建

误报。总之，该效应模型足以涵盖可能破坏 Rust 安全保证并引入安全风险的基本效应类型，例如内存操作、系统交互或与其他语言通信。我们在附录 A 中详细介绍上述类别。

3.2 效应分析

我们的分析直接在目标包的源代码上运行，采用单次、以语法为主的遍历。Cargo Scan 将输入包解析为抽象语法树，并执行轻量级的模式匹配遍历以识别效应。例如，每个调用或表达式节点都会与我们效应模型中预定义的一组规范库路径（例如 std::fs）或操作类别（例如 RawPointer）进行比较。在抽象语法树而非更低层表示上进行分析并非根本要求，但带来了一些有用的优势。它便于将效应直接映射回源代码，并在脱糖之前精确计算已审计的代码行。大多数效应可通过模式匹配充分识别，但在必要时，Cargo Scan 会利用 rust-analyzer [11], 的语义信息，后者同时基于 HIR 和 MIR 运行。这通过解析类型、名称，并对原始指针解引用或同名函数与 trait方法等构造进行消歧，从而减少误报。

Cargo Scan 还会为输入包构建调用图，并为每个节点关联一组内联效应。我们实现了一种污点与效应分析的变体，其中识别出的效应可视为污点源，而 crate 的公开暴露函数则是汇点。Rust 的类型系统使修改和别名问题变得易于处理，因此，我们无需进行指针分析、别名分析和数据流分析，即可执行可扩展且轻量级的过程内分析，并捕获更多种类的危险行为。

3.3 安全注解

Cargo Scan 为用户提供三种安全注解，用于在审计期间标记效应并指明其安全性：(i) 安全、(ii) 不安全，或 (iii) 调用方检查。

安全/不安全注解。 前两种注解——安全 和 不安全 ——适用于能够在本地 判定效应安全性的情况，也就是说，可以根据周围的代码来判断。在下面的代码片段中，get_unchecked 效应是安全的，因为在访问向量之前会先检查其长度。

```
如果 v.len() >= 3{不安全 { v.get_
unchecked(2) }}
```

调用方检查注解。 第三种注释——调用方检查 ——适用于无法孤立地确定代码是否安全，而是取决于调用上下文的情况。例如，该函数

```
函数 读取_文件 (f: &std :: path :: Path ) -> 字符串 {std::fs::
read_转_字符串 (f). unwrap () }
```

根据参数 f 的值，它可能被无害地调用来读取配置文件，也可能被恶意地调用来读取用户的私钥。这类效应会标注为 调用方检查 ，而 Cargo Scan 会利用分析阶段生成的调用图，将这些效应传播到该函数（此处为 read_file）的所有调用方。

3.4 交互式审计

Cargo Scan 通过允许用户交互式审计产生作用的位置上的代码，帮助系统地审查已识别的效应。我们最初将 Cargo Scan 实现为命令行界面，类似于 `git add` 的交互式暂存功能。使用该界面审计了许多 crate 后，我们意识到，实际操作中，浏览源代码和效应以及跟踪审计进度的能力都受到过多限制。因此，我们实现了一个 VSCode 扩展（图 1），以解决这些限制，并利用编辑器的交互功能，提供更大的灵活性。

3.5 跨包审计

要全面审计一个（客户端）包，必须先审计它的所有依赖项，查看其中哪些公有方法经过调用方检查（这意味着随后还需审计它们在客户端包中的调用点）。如 `rust-embed` 示例（见 §2）所示，实际中确实存在源自其他包中经调用方检查的公共函数的漏洞。Cargo Scan 会跟踪从顶层包可传递到达的依赖项中的公共且经过调用方检查的函数。

默认审计。 完整审计项目的所有依赖项，能够最精确地统计可能产生的效应，但在大规模情况下需要付出大量工作。Cargo Scan 通过生成默认审计来自动完成这一过程——这些审计是对人工审查进行可靠且保守的近似：将每个效应标记为调用方检查，并沿调用栈向上传播。将所有效应视为上下文敏感，可以计算出每条可能的传播路径，并确定所有可能涉及危险操作、需要审查的代码范围上限。

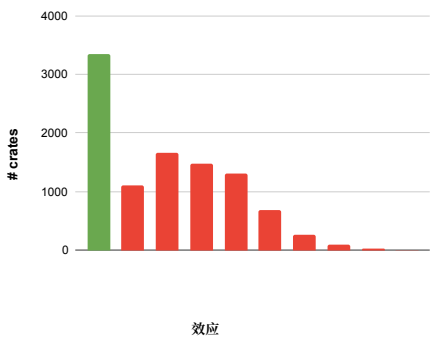


图 3：排名前一万的 Cargo 包按效应数量的分布。

因此，用户只需在审计时关注应用程序所调用的依赖项中那些公开的、由调用方检查的函数。这样，Cargo Scan 就能像识别单个包中产生作用的位置一样，从依赖项中筛选出仅与审计相关的代码。对于默认审计结果过于保守的 crate，可以优先审计；随后，将判定为安全的所有库调用从依赖包的效应集合中移除。

4 实证评估

我们通过提出三个问题来评估 Cargo Scan：

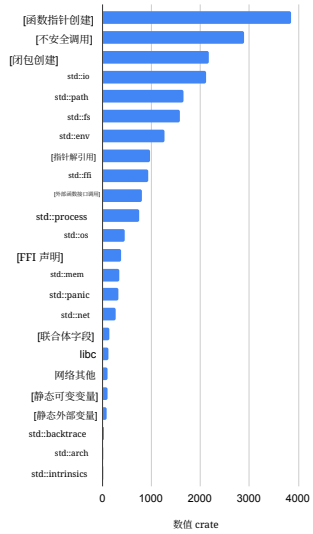
► 热门软件包中效应有多常见（§4.1）？► 上下文敏感的效应传播是否有价值（§4.2）？► Cargo Scan 是否能提高审计效率（§4.3）？

实验设置。我们在一台配备 AMD Ryzen ThreadRipper 3960X（24 核、48 线程）、128GB 内存，运行 Ubuntu 24.04 和 Rust 1.90 的机器上进行了 §4.1节的实验。由于内存限制，我们使用了自定义脚本，开启 16 个并行线程。我们在之前那台机器以及另一台配备 Intel i7-8650U、16GB 内存，运行 Ubuntu 24.04 和 Rust 1.90 的机器上，分别进行了 §4.2 节和 §4.3节的审计。所有实验均于 2025年10月进行。

4.1 效应有多常见?

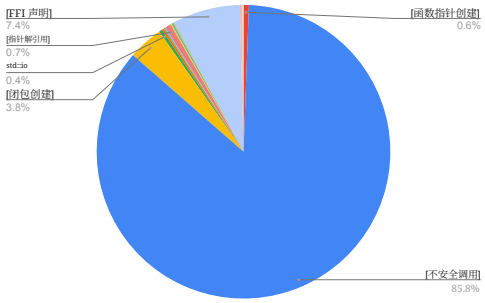
我们首先使用 Cargo Scan 分析 crates.io（Rust 的中央公共 crate 仓库）上下载量最高的 1 万个软件包中发现的效应，以了解效应在 Rust 中有多常见。在进行生态系统分析时，我们逐一分析了每个 crate，没有追踪跨依赖项的效应。

效应的集中程度。图 3 按效应总数的数量级划分，展示了各 crate 的分布情况。我们发现，大量



（a）至少包含一次该效应的 crate 数量

至少一次 ef- 出现。
排名前一万的 Cargo 包中的 crate。



（b）按效应总数计算的相对频率
所有排名前一万的 Cargo 包中的立场。

图 4：按类型划分的副作用分布。

crate 完全或大多是纯的，也就是说没有效应：具体而言，略多于三分之一的 crate（3434 个）不含任何副作用实例，超过四分之一的 crate（2778 个）含有不到 10 个效应实例。此外，大多数效应（约 85%）集中在少数 crate（约 3%）中。

效应类型的频率。图 4a展示了不同效应在最受欢迎的软件包中的出现频率。我们采用 §3.1中描述的效应类别，并进一步根据汇调用的来源库对其进行区分。我们总共纳入了 24 种效应模式：Rust 不安全性（指针解引用、不安全函数调用、外部函数接口等）、标准库效应（输入输出、文件系统、网络和系统调用），以及高阶效应（创建可能产生效应的闭包或函数指针）。如汇调用图 4a所示，影响最多 *crate* 的效应包括高阶效应（函数指针和闭包）、不安全函数调用、标准输入输出操作（汇）以及原始指针解引用。图 4a。若按排名前一万的 Cargo 包中各类效应的实例总数统计，结果则略有不同，如图 4b所示；此时，外部函数接口函数声明跃居首位，这是因为少数 crate 主要由外部函数接口绑定构成。为了深入了解情况，我们还过滤掉了专门用于外部函数接口绑定的 crate，这类 crate 通常称为系统*crate*。包含外部函数接口声明效应的 crate 总数从 379 个降至 190 个；在排名前一万的 Cargo 包中，此类效应的总频率如今为 1.7%，而最初为 7.4%。

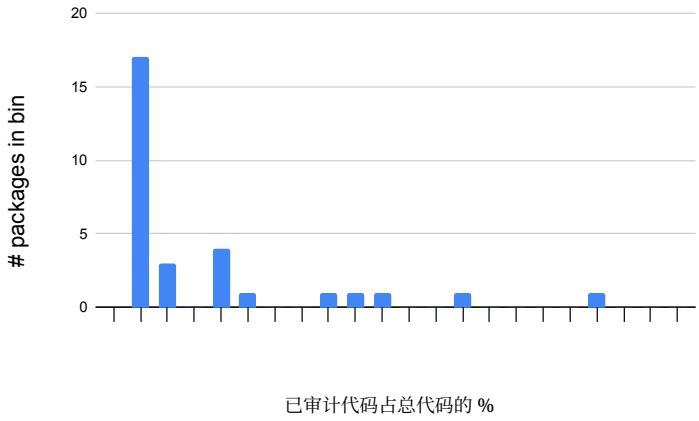


图 5：已审计代码百分比的概率密度函数。条形对应已审计代码百分比处于区间 $(x-5, x]$（例如区间 $(0\%, 5\%]$）内的软件包数量。

4.2 上下文敏感效应是否有价值?

上下文敏感效应（通过调用方检查注释 §3.3进行跟踪）能为审计人员提供更精确的程序安全性模型，但将效应传播到调用点可能需要更多审计工作量。为评估这一点，我们提出以下问题：

► **Q1**：效应中有多大比例是调用方检查的? ► **Q2**：对于调用方检查的效应，我们审计的平均调用栈是多少? ► **Q3**：有多少调用方检查的效应会到达 crate 的公共 API?

在本实验中，我们按拓扑排序顺序审计了广泛使用的 HTTP 库 hyper 及其默认构建中的全部依赖项（共计七个 crate），并统计了需要上下文敏感推理的效应。为回答**Q1**，在我们审计的软件包中，调用方检查的效应的平均百分比为 30.8%。关于 **Q2**，我们必须检查并标注每个效应（包括其起源）的位置，其算术平均值为 3.1 个。总体而言，这意味着大多数效应仅传播到少量位置，即潜在危险的副作用得到了良好封装，对它们进行推理只需适度的额外工作。最后，对于 **Q3**，完全审计的软件包（即我们审计了其全部效应的软件包）平均有 5.2% 的公共函数被标记为调用方检查。尽管跨越 crate 边界的调用方检查的效应不如 crate 内部的调用方检查的效应常见，但这一发现仍表明，通过在报告中呈现此类效应以使审计可组合具有重要意义。

4.3 Cargo Scan 能让审计更高效吗?

作为衡量 Cargo Scan 在实际中的使用情况的一种替代指标，我们改为根据已审计代码的总量来衡量审计效率，这直接

影响审计工作量和时间。为了评估 Cargo Scan，我们采用与第 §4.2 节实验相同的案例研究，但不使用默认构建，而是启用 所有可选功能，以涵盖所有情况，并为所有依赖项生成默认审计。这会对它们所有潜在的效应及其在父包中的传播进行过度近似，即跨越 包边界 (§3.5) 的调用中由调用方检查的效应。尽管默认审计会保守地导致更多审计位置，Cargo Scan 只会展示从根 crate 可达的位置；在本例中，根 crate 是 `hyper`，因此我们将审计工作量集中在软件包集合中的一个小子集上。具体而言，在 `hyper`及其常规依赖项和可选依赖项共 30 个软件包中，Cargo Scan 只展示了其中 17 个供审计，如图 6所示。边的权重表示依赖 crate 中标记为调用方检查的公共函数数量。

审计方法。我们按照拓扑顺序审计各个 crate，也就是说，先审计某个 crate 的依赖项，再审计该 crate 本身。在这种情况下，依赖图底层的 crate 含有比例最高的效应，并且会被其他软件包“复用”。如果某个 crate 中的效应无法在本地解析，我们就递归地检查父级 crate 中的调用点，直到能够确定其安全性。Cargo Scan 会在审计过程中自动更新各依赖项中的可达效应集合，并移除相应的审计位置。这大幅减少了依赖图中需要审计的代码总量，尤其是在依赖图中包含大型库（例如 tokio），而这些库只有一小部分 API 会被调用时。

量化审计工作量。我们通过衡量需要审查的代码占 Cargo Scan 分析所得的软件包总代码量的百分比，来确定审计工作量。衡量“已审查代码”可能比较棘手，因为审计某项效应需要理解其上下文，包括函数的参数以及效应的结果如何被使用等。因此，我们不标记单独的相关代码行，而是假设必须检查效应所在的整个函数，以此估算“已审查代码”。为了量化案例研究中审计代码的总量，我们收集 hyper 本身中包含效应的所有函数。不过，由于我们以默认审计作为每个依赖项的起点，因此处理依赖项的方式与顶层 crate 不同：我们会跟踪审查期间最终被标记为安全或不安全的效应——这表明我们手动审计了其所在函数——以及所有从根节点仍可到达的调用方检查的效应。

已审计效应的分布。 我们发现，Cargo Scan 显著减少了审计人员需要查看的代码量，而且大部分审计工作量集中在少数效应较多的 crate 中。每个包中我们审计的代码行占比的算术平均值为 13.1%，中位数为 0.4%。完整分布见 图 5。我们看到的偏斜情况与 §4.1中的预期一致。具体而言，已审计代码占比较高的软件包主要是那些更常被复用的软件包。

跨包审计。 与直觉相反，我们发现，使用 Cargo Scan 的默认审计实际上减少了我们需要审计的代码总量；我们平均审计了 12.4% 的代码行。相比之下，在

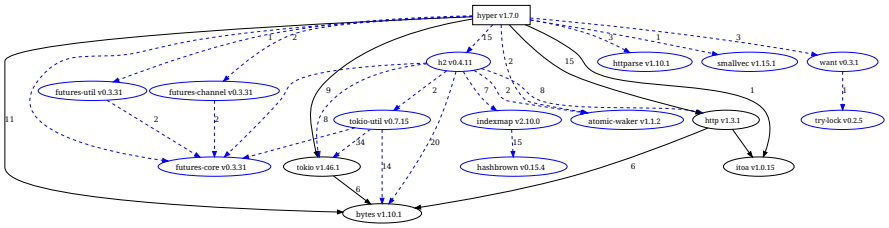


图6：效应流入 hyper 的部分依赖项。边的权重表示公开调用者检查（CC）函数的数量。

表2：默认审计（§4.3）与完整审计（§4.2）中的已审计函数和代码行数。

crate	默认审计		完整审计	
	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

章节 §4.2中，对于相同的软件包，我们平均审计了 28.1%（表2）。由此我们得出结论：Cargo Scan 的默认审计生成方式改善了审计工作流程，并减少了检查依赖项所需审查的代码总量。

有效性威胁。 审计仍然是一个人工流程，结果可能受到多种因素影响，例如审计人员的专业水平、输入 crate 的复杂性，或其预期的威胁模型。Cargo Scan 提供了一些客观保障，有助于使审计结果更加趋同，因为它能够避免遗漏相关代码、跟踪函数调用和 crate 边界间的效应，并将审计结果收集到可组合的审计文件中。不过，它无法进一步帮助用户对每种效应的安全性做出最终判断和决定。

5 讨论与未来工作

Cargo Scan 帮助开发者更高效地审计 Rust 代码。它通过静态分析识别潜在危险的副作用，并沿调用图追踪依赖上下文的安全性。虽然我们发现 Cargo Scan 在实践中效果良好，但未来仍有一些值得探索的改进方向。

目前，Cargo Scan 无法帮助开发者审计“差异”，也就是同一 crate 不同版本之间的变更。同一 crate 的不同版本通常高度相似；若将工具扩展为只关注直接或间接受变更影响的代码位置，就能进一步减轻审计负担。最后，另一个方向是分析构建时效应。例如，Rust crate 可以执行自定义构建脚本（build.rs），有时会生成潜在危险的代码。

6 相关工作

对 Rust 生态系统的实证分析与测试。Rust 软件包由 `crates.io` 集中管理，已有大量研究分析该生态系统中的**不安全**代码。先前的工作分析了不安全代码块的使用情况 [10,1],及由此产生的漏洞 [39]。与 [10], 不同，我们还将闭包和函数指针的创建视为效应，构建完整的 crate 级调用图以传播上下文敏感的效应，并捕获仅靠不安全代码无法涵盖的危险行为。同样，尽管 [1] 与我们的效应模型相似，并通过查询编译器的 MIR 表示对不安全代码进行分类，但其分析仅限于不安全操作。其他研究则针对特定漏洞，例如 Rust CVE [54], 撤回的 crate [29], 和语义化版本违规 [37], ；而我们还分析了生态系统中的系统效应。RULF [20] 和 SyRust [47], 等动态测试工具会合成 API 调用序列，以对特定类别的错误（例如不可达代码和越界检查）进行模糊测试，从而补充静态分析方法和审计工作。

效应分析。此前关于副作用分析的研究，即确定函数或表达式如何影响程序状态（除返回值之外），包括内存、输入输出和控制流，主要关注 C/C++ [41,7,23]和 Java [38,40,25,33,46,55,48],等语言。为考虑修改，许多此类研究旨在计算每条程序语句可能读取和修改的对象集合；这依赖于计算开销高昂的指向分析、别名分析和逃逸分析，并会产生冗长的输出。在 Rust 中，这些技术并非必要，因为修改和别名由类型系统管理，在不安全代码块之外是可处理的。相反，我们在函数层面识别效应集合；根据我们的经验，这对审计更有用。

面向安全审计的静态分析。已有一些专门为安全审计设计的静态分析研究，例如针对 Java 的 JoanAudit [49], 、Andromeda [50],、FindBugs [18]等工具 [12,30,51,27,19]。其中大多数基于预先配置的易受攻击模式集合，实现污点分析和信息流分析。尽管这些工具旨在提升针对 Java 字节码的静态分析效率和可扩展性，但它们缺乏交互式审计模型，无法引导用户，也不支持渐进式、可组合的审计日志。

供应链安全。 供应链安全涵盖了针对不同组件的各种工具。在 Rust 生态系统中，RustSec [45]跟踪漏洞，而 Cackle [24],、Cargo Audit [44],、Cargo Crev [9], 和 JFrog 的

Pyrsia [6], 和 Lib.rs [28] 提供依赖管理、审计元数据和 API 使用检查。在其他生态系统中，尤其是 JavaScript，研究重点是发布与交付 [57,8,56],，不过运行时依赖注入等动态特性会妨碍静态分析。我们的目标是手动代码审计。

CargoScan 的安全注解可以与 Cargo Vet [34] 使用的“审计标准”进行比较；尤其是，它们与内置标准“可安全运行”和“可安全部署”最为接近。我们不支持 Cargo Vet 的其他自定义标准，例如面向功能正确性的“已通过密码学审查”，但可以进一步扩展我们的效应模型，例如将关键密码学函数视为效应。对于 Go，Google 的 Capslock [16,14,15] 会执行类似的效应分析。不过，与我们的方法不同，它仅限于命令行界面，且不支持可组合的审计日志；此外，它将反射视为运行时效应，而我们会静态解析或过近似基于 trait 的效应。

语义与验证。近期有大量工作致力于形式化 Rust 所有权机制的语义，包括 RustBelt [22], 、Oxide [53],、Stacked Borrows [21], ，以及用于检测未定义行为的 Miri 解释器 [42]。其他项目（如 Rudra [4] ）则分析不安全 Rust、栈展开与 Rust trait 系统之间的交互。这些技术可以提高自动检测不安全 Rust 代码中问题的能力，但与现有的生态系统研究一样，并未考虑系统副作用。此外，还有一些面向 Rust 的验证工具，包括 Smack [5],、Kani [52],、Prusti [2],、Aeneas [17],、Verus [13], 、Hacspec [32], 、Flux [26], 和 RustHornBelt [31]。大多数验证工具只适用于安全 Rust—— [31] 除外——因为不安全 Rust 很难形式化。验证工作耗费大量人力，且无法扩展到整个生态系统；而我们的技术可以帮助验证人员将目标聚焦于最关键的代码。

致谢。 我们感谢 Bobby Holley 富有见地的讨论，这些讨论启发了我们的研究；感谢匿名审稿人提出的意见；也感谢 Pu Guo 对宏分析的贡献。本研究部分得到了伯克利数字资产中心、Mozilla 和美国国家科学基金会的资助（项目编号：2048262、2120642、2155235、2327336 和 2327338）。

参考资料

1. Astrauskas, V., Matheja, C., Poli, F., Müller, P., Summers, A.J.: 程序员如何使用不安全 Rust？ ACM 编程语言学报4(OOPSLA), 1–27 (2020) 2. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: 利用 Rust 类型进行模块化规范与验证。ACM 编程语言学报3(OOPSLA), 1–30 (2019) 3. Axum 贡献者：crates.io - axum。 https://crates.io/crates/axum (2019), [2024 02 07] 访问日期 - - 。

Bae, Y., Kim, Y., Askar, A., Lim, J., Kim, T.: Rudra：在生态系统规模上发现 Rust 内存安全漏洞。载于：ACM SIGOPS 第 28 届操作系统原理研讨会论文集。第 84–99 页（2021）

5. Balasubramanian, A., Baranowski, M.S., Burtsev, A., Panda, A., Rakamarić, Z., Ryzhyk, L.: Rust 系统编程：超越安全性。载于：第 16 届操作系统热点研讨会论文集。第 156–161 页（2017）6. Chin, S.: 借助 Rust 闭合供应链安全闭环 @ Rust Nation UK| JFrog — jfrog.com。https://jfrog.com/community/rust/closing-the-supply-chain-security-loop-with-rust-and-pyrsia/ (2023 [2023 12|), 访问日期 - 。
7
Choi, J.D., Burke, M., Carini, P.: 高效、流敏感的过程间指针引起的别名与副作用计算。载于：第 20 届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集。第 232–245 页。POPL '93，计算机协会，纽约，美国（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.: npm 生态系统中依赖项降级的实证研究。IEEE 软件工程汇刊 47(11)，2457–2470（2019）9. Crev 开发团队：GitHub - crev-dev/cargo-crev：适用于 Cargo（Rust）包管理器的可加密验证代码审查系统。— github.com。https://github.com/crev-dev/cargo-crev (2024 [14 11 2024]), 访问日期 - - 。
10
Evans, A.N., Campbell, B., Soffa, M.L.: 软件开发者是否安全地使用 Rust? 载于：ACM/IEEE 第 42 届国际软件工程会议论文集。第 246–257 页（2020）11. Ferrous Systems：rust-analyzer — Rust 语言服务器协议。https://rust- analyzer.github.io/ (2026 [2024 2 8|), 访问日期 - - 。GitHub：
12
spotbugs/spotbugs：用于发现 Java 代码错误的静态分析工具 — github.com。https://github.com/spotbugs/spotbugs 13. GitHub：verus-lang/verus：适用于底层系统代码的经过验证的 Rust — github.com。https://github.com/verus-lang/verus (2022 [2023 2 03|), 访问日期 - - 。
14
Google：Capslock：你的代码真正能做什么? — security.googleblog.com。https://security.googleblog.com/2023/09/capslock-what-is-your-code- really.html (2023 [14 11 2024]), 访问日期 - - 。Google：GopherCon — 2023
15
events.bizzabo.com。https://events.bizzabo.com/ 449995 1160360 2023 [14 11 2024] /agenda/session/ ( ), 访问
16
日期 - - 。Google：GitHub - google/capslock — github.com。https://github.com/google/capslock (2025 [14 11 2024]), 访问日期 - - 。Ho, S., Protzenko, J.:
17
Aeneas：通过函数式转换验证 Rust。ACM 编程语言学报 6(ICFP)，711–741（2022）18. Hovemeyer, D., Pugh, W.: 发现错误很容易。SIGPLAN Not. 39(12)，92–106（2004 年 12 月）。https://doi.org/10.1145/1052883.1052895，https://doi.org/ 10.1145/1052883.1052895 19. Huang, W., Dong, Y., Milanova, A.: 面向 Java Web 应用的基于类型的污点分析。载于：第 17 届软件工程基础方法国际会议论文集——第 8411 卷。第 140–154 页。Springer-Verlag，柏林，海德堡（2014）。https://doi.org/10.1007/978-3-642-54804-8_10 978 3 642 54804 8_10 , https://doi.org/10.1007/ - - - - 。
20
Jiang, J., Xu, H., Zhou, Y.: RULF：通过 API 依赖图遍历进行 Rust 库模糊测试。载于：2021 年第 36 届 IEEE/ACM 自动化软件工程国际会议（ASE）。第 581–592 页。IEEE（2021）21. Jung, R., Dang, H.H., Kang, J., Dreyer, D.: Stacked Borrows：Rust 的别名模型。ACM 编程语言学报 4(POPL)（2019 年 12 月）。https://doi.org/10.1145/ 3371109，https://doi.org/10.1145/3371109

22. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt：保障Rust 编程语言的基础安全。Proc. ACM Program. Lang. 2(POPL)（2017年12月 3158154 23 3158154, ）。https://doi.org/10.1145/ https://doi.org/10.1145/ 。Landi, W., Ryder, B.G., Zhang, S.: 结合指针别名的过程间修改副作用分析。载于：ACM SIGPLAN 1993年编程语言设计与实现会议论文集。第56–67页。PLDI '93，计算机协会，纽约，NY，美国（1993年）。https: //doi. 155090.155096 24 10.1145 155090.155096 。10.1145 org/ / ，https://doi org/ / 。Lattimore, D.: 借助Cackle提高Rust 供应链攻击的难度 — david- lattimore.github.io。https://davidlattimore.github.io/making-supply-chain- attacks-harder.html (2023 [2023 12 06])，访问于 - - 。Le, A., Lhoták, O., 25 Hendren, L.: 在即时编译优化中使用过程间副作用信息。载于：第14届编译器构造国际会议论文集。第287–304页。CC'05，Springer-Verlag，柏林，海德堡（2005年）。https://doi.org/10.1007/ 11406921_22 11406921_22 , 26 https://doi.org/10.1007/ 。Lehmann, N., Geller, A.T., Vazou, N., Jhala, R.: Flux：面向Rust的液体类型。ACM编程语言论文集 7(PLDI)，1533–1557（2023年）27. Lerch, J., Hermann, B., Bodden, E., Mezini, M.: Flowtwist：面向大型代码库的高效上下文敏感由内向外污点分析。载于：第22届ACM SIGSOFT软件工程基础国际研讨会论文集。第98–108页。FSE 2014，计算机协会，纽约，NY，美国（2014年）。https://doi.org/10.1145/2635868.2635878，https://doi.org/ 10.1145/2635868.2635878 28. Lesiński, K.: Lib.rs – Rust crate之家。https://lib.rs/（2024年），[访问于 14 11 2024] - - 。Li, H., Cogo, F.R., Bezemer, C.P.: 对Rust 软件包注册表中撤回版本的实 29 证研究。IEEE 软件工程汇刊 49(1)，437–449（2022年）30. Livshits, V.B., Lam, M.S.: 利用静态分析发现Java 应用程序中的安全漏洞。载于：第14届USENIX安全研讨会论文集 - 第14卷。第18页。SSYM'05，USENIX协会，美国（2005年）31. Matsushita, Y., Denis, X., Jo urdan, J.H., Dreyer, D.: RustHornBelt：面向含不安全代码的Rust 程序函数式验证的语义基础。载于：第43届ACM SIGPLAN编程语言设计与实现国际会议论文集。第841–856页。PLDI 2022，计算机协会，纽约，NY，美国（2022年）。https://doi.org/10.1145/3519939.3523704 3519939.3523704 32 。10.1145 ，https://doi org/ / 。Merigoux, D., Kiefer, F., Bhargavan, K.: Hacspec：嵌入Rust的高保证密码学简明、可执行、可验证规范。博士论文，Inria（2021年）33. Milanova, A., Rountev, A., Ryder, B.G.: 面向Java指向分析的参数化对象敏感性。ACM Trans. Softw. Eng. Methodol. 14(1)，1– 41（2005年1月）。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 [2023 11 13])，访问于 - - 。das Neves, R.M.: 扩展unsafe 关键字 — 35 internals.rust-lang.org。https://internals.rust-lang.org/t/extending-the-unsafe-keyword/20638 (2024 [26 03 2025])，访问于 - - 。美国国家标准与技术研究院：Cve- - — 36 2021 45712 NVD。https://nvd.nist.gov/vuln/detail/CVE-2021- 45712（2021年），[访问于 2024-2-8]

37. Nowak, T., Predrag Gruevski：语义化版本违规很常见，更好的工具才是答案。https://predr.ag/blog/semver-violations-are-common-better- tooling-is-the-answer/ (2023 [2024 2 8])，访问于 - - 。Pearce, D.J.:
38 Jpure：面向Java的模块化纯度系统。载于：Knoop, J.（编）编译器构造 - 第20届国际会议，CC 2011，作为欧洲软件理论与实践联合会议ETAPS 2011的一部分召开，萨尔布吕肯，德国，2011年3月26日至4月3日。论文集。Lecture Notes in Computer Science，第6601卷，第104–123页。Springer（2011年）。https://doi.org/10.1007/ 978 3 642 19861 8_7 978 3 642 19861 8_7 - - - - ,
39 https://doi.org/10.1007/ - - - - 。Qin, B., Chen, Y., Yu, Z., Song, L., Zhang, Y.: 理解真实世界 Rust 程序中的内存与线程安全实践及问题。载于：第41届ACM SIGPLAN编程语言设计与实现会议论文集。第763–779页（2020年）40. Rountev, A.: 精确识别Java中的无副作用方法。载于：第20届IEEE软件维护国际会议论文集。第82–91页。ICSM '04，IEEE计算机学会，美国（2004年）41. Rountev, A., Ryder, B.G.: 针对使用预编译库构建的程序的指向分析与副作用分析。载于：第10届编译器构造国际会议论文集。第20–36页。CC '01，Springer-Verlag，柏林，海德堡（2001年）42. Rust团队：GitHub - rust-lang/miri：Rust中层中间表示的解释器 — github.com。https://github.com/rust-lang/miri（2026年），
43 [2026 01 21] 访问于 - - 。RustSec团队：Rustsec-2021-0126 — rustsec.org。https://rustsec.org/ advisories/RUSTSEC-
44 2021 2021 [2024 2 8] -0126.html（），访问于 - - 。RustSec团队：rustsec/cargo-audit at main · rustsec/rustsec — github.com。https: //github.com/rustsec/rustsec/tree/main/cargo-audit（2024年），[访问于 14-11 2024] - 。RustSec团队：Rust安全公告数据库 — rustsec.org。https:// rustsec.org/
45 (2026 [2023 11 13])，访问于 - - 。Sălcianu, A., Rinard,
46 M.: 面向Java程序的纯度与副作用分析。载于：第6届验证、模型检测与抽象解释国际会议论文集。第199–215页。VMCAI'05，Springer-Verlag，柏林，海德堡（2005年）。https://doi.org/10.1007/978-3-540-30579-8_14，https:// doi.org/10.1007/ 978 3 540 30579 8_14 - - - - 。Takashima, Y., Martins, R., Jia, L.,
47 Păsăreanu, C.S.: SyRust：利用语义感知程序合成对Rust库进行自动测试。载于：第42届ACM SIGPLAN编程语言设计与实现国际会议论文集。第899–913页（2021年）48. Tao, B., Qian, J., Zhou, X.: 采用快速逃逸过滤器的副作用分析。载于：ACM SIGPLAN Java程序分析最新进展国际研讨会论文集。第15–20页。SOAP '12，计算机协会，纽约，NY，美国（2012年）。https://doi.org/10.1145/2259051.2259054，https://doi. 2259051.2259054 49 10.1145org/ / 。Thomé, J., Shar, L.K., Bianculli, D., Briand, L.C.: JoanAudit：用于审计常见注入漏洞的工具。载于：2017年第11届软件工程基础联合会议论文集。第1004–1008页。ESEC/FSE 2017，计算机协会，纽约，NY，美国（2017年）。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：Web 应用程序的准确且可扩展的安全分析。载于：国际软件工程基础方法会议。第 210–225 页。Springer (2013)。https://doi.org/10.1007/ 978 3 642 37057 1_15 - - - - 。Tripp, O., Pistoia, 51 M., Fink, S.J., Sridharan, M., Weisman, O.：Taj：Web 应用程序的有效污点分析。载于：第 30 届 ACM SIGPLAN 编程语言设计与实现会议论文集。第 87–97 页。PLDI '09，计算机协会，纽约，纽约州，美国 (2009)。https:// doi. 1542476.1542486 52 10.1145 1542476.1542486 。10.1145 org/ / ， https://doi org/ / 。VanHattum, A., Schwartz-Narbonne, D., Chong, N., Sampson, A.：验证 Rust 中的动态 trait 对象。载于：第 44 届国际软件工程会议：软件工程实践论文集。第 321–330 页 (2022) 53. Weiss, A., Gierczak, O., Patterson, D., Ahmed, A.：Oxide：Rust 的本质。arXiv 预印本 arXiv:1903.00982 (2019) 54. Xu, H., Chen, Z., Sun, M., Zhou, Y., Lyu, M.R.：内存安全问题是否已被视为解决？对所有 Rust CVE 的深入研究。ACM 软件工程与方法学汇刊 (TOSEM) 31(1), 1–25 (2021) 55. Yang, J., Hotta, K., Higo, Y., Kusumoto, S.：揭示用于复用 Java 库的函数纯度和副作用。载于：Schaefer, I., Stamelos, I. (编) 云及其他动态系统的软件复用——第 14 届国际软件复用会议，ICSR 2015，美国佛罗里达州迈阿密，2015 年 1 月 4–6 日。论文集。计算机科学讲义，第 8919 卷，第 314–329 页。Springer (2015)。https://doi.org/10.1007/978-3-319-14130-5_22, https: // doi.org/10.1007/ 978 3 319 14130 5_22 - - - - 。Zahan, N., 56 Zimmermann, T., Godefroid, P., Murphy, B., Maddila, C., Williams, L.：npm 供应链中的薄弱环节是什么？载于：第 44 届国际软件工程会议：软件工程实践论文集。第 331–340 页 (2022) 57. Zimmermann, M., Staicu, C.A., Tenny, C., Pradel, M.：高风险的小世界：对 npm 生态系统安全威胁的研究。载于：第 28 届 USENIX 安全研讨会 (USENIX Security 19)。第 995–1010 页 (2019)

A 效应类别

A.1 不安全和系统效应

表 1 展示了我们支持的完整效应集合。不安全效应涵盖了 Rust 安全子集之外的所有可能操作，而这些操作仅限于 不安全代码块中。`FFIDecl` 效应表示外部函数的声明。如果这些函数中的变量被非 Rust 代码使用，它们可能不遵循 Rust 的所有权规则。对于只公开外部函数、但不调用这些函数的软件包，此效应类型与`FFICall`互为补充。系统效应源自标准库或`libc`等软件包，在我们的模型中被视为汇调用。Rust 中的系统效应可能会在内部调用 `libc` 库，这些调用完全可以归入第一类效应。不过，将它们建模为单独的类别，可以让用户审计其调用上下文，而非其实现。例如，审计文件访问时，用户应考虑所访问的文件是否

适合该 crate 的上下文。因此，例如在审计文件读写操作时，向用户展示原始的不安全效应可能并不实际，因为它无法提供有关文件本身的有意义信息。闭包创建 表示闭包中存在效应，这些效应会在闭包创建后执行。同样，函数指针创建用于表示函数指针可能指向包含其他效应的函数（见 §A.2）。

A.2 高阶控制流

高阶函数和多态特性在标准库与 Rust 应用代码中都无处不在。高阶函数接受函数指针或闭包，而这些函数指针或闭包本身又作为值在程序中流动；若不借助高阶控制流分析来确定所有这些值的可能来源，就很难判定其安全性。同样，Rust 基于 trait 的多态（通过泛型或动态分派）要求追踪所有可能的实现，才能健全地分析方法调用。Cargo Scan 针对这两种不同形式的高阶控制流——（i）函数指针和闭包，以及（ii）trait 多态——分别采用专门的方法进行处理。

闭包和函数指针。在以下示例中，第2行的闭包接受一个文件路径作为参数，并从该文件读取内容。

```
1 fn danger_reader () -> impl Fn(& Path) -> String { 2 |p: &Path | read_to_
string (p). unwrap () 3 } 4 5 fn exec_reader (reader: impl Fn(& Path) -> String , file: &
Path) { 6 let out = reader(file); 7 .. 8 } 9 10 pub fn read_file (p: Path) { 11 let reader =
danger_reader (); 12 exec_reader (reader , p); 13 }
```

在这个例子中，具体读取哪个文件取决于公共函数 read_file 的参数。由于 read_file 是一个公共函数，任何调用此库的包都可以用它读取系统上的任意文件，因此调用方有责任确保路径经过适当的清理。如果返回闭包的函数改为写成

```
1 fn 安全_reader ( root_path : PathBuf ) -> impl Fn(& Path) -> String { 2 move |p: &Path |
if in_root (p, & root_path ) { 3 read_to_string (p).unwrap () 4 } else { 5 "错误：无效路径". to_
string () 6 } 7 }
```

那么我们就会发现，闭包本身会妥善清理所有输入路径，因此只能访问允许访问的文件。

我们不将效应函数指针或闭包的每个潜在调用点都标记为效应，而是将它们的创建标记为效应（使用

闭包创建和函数指针创建）。这些函数只会创建一次，但可能会被调用多次，因此我们发现，实际上这种方法能减少审计人员的工作量。

此外，闭包通常与迭代器搭配使用来处理集合，而且通常是纯的，因此若不加区分地将它们全部标记为有副作用，就会不必要地增加审计位置。为避免这种情况，Cargo Scan 会扫描闭包的主体，仅当其中包含可能被恶意利用的操作时，才将闭包的创建标记为一种效应。

特征。当 trait 方法调用采用静态解析时，Cargo Scan 会扫描相应的具体实现，以查找潜在效应。

```
1 函数 func() {2 令 s = 结构体;3 令 obj: Box <动态 特征
> =Box::new(s); 4 s.方法 (); 5 obj.方法 ();6 }
```

在我们的示例中，假设有一个特征 `Trait`，其中包含一个方法 `meth`，以及一个实现该特征的 `Struct`。由于对`meth` 的调用（第 4 行）采用静态派发，Cargo Scan 会检查`Struct` 的方法实现。然而，对于泛型调用或动态派发的调用，Cargo Scan 在分析时无法解析具体类型（例如第 5 行）。此时，它会发现该包中该特征的所有可用实现，并将它们的效应与抽象trait方法关联起来。随后，用户会根据组合效应判断该方法是否安全。