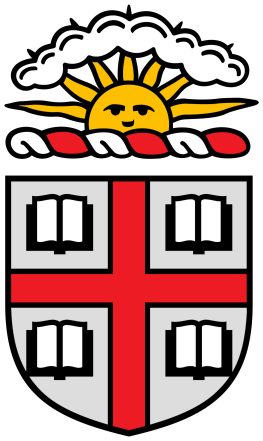
PEAR: Practical Interprocedural Analyses in Rust

Artem Agvanian

Advisor: Malte Schwarzkopf

Reader: Will Crichton



Department of Computer Science

Brown University

Providence, RI

May 2025

Contents

Acknowledgments **4**

Abstract **5**

1 Introduction **6**
1.1 The Problem 6
1.2 Proposed Solution 8

2 Background and Related Work **9**
2.1 Rust Compiler Overview 9
2.2 Rust Compiler Plugins 10
2.3 Static Analysis in Rusty Systems 10
2.4 Interprocedural Analyses in Object-Oriented Programming Languages . . 11

3 Design **13**
3.1 PEAR Overview 13
3.2 PEAR Analysis Walkthrough 14
3.3 Monomorphized Call Graph (MCG) 16
3.4 MCG Construction 17
3.4.1 Reachability Analysis 17
3.4.2 Refinement 18
3.4.3 MCG Construction Soundness 20

4 Implementation **22**
4.1 Integrating with PEAR 22
4.2 Implementation Limitations 22

5 Application Case Studies **23**
5.1 Sesame 23
5.1.1 Requirements for Scrutinizer 23
5.1.2 PEAR-enabled Scrutinizer 23
5.2 Sniffer 23
5.2.1 Requirements for Sniffer 24
5.2.2 PEAR-enabled Sniffer 24

6 Evaluation **25**
6.1 Dynamic Dispatch Resolution Correctness 25
6.2 Accuracy of MCG Construction 25
6.2.1 Sesame 27
6.2.2 Sniffer 29

6.3 Developer Effort 29
6.3.1 Sesame 29
6.3.2 Sniffer 30

7 Discussion **31**
7.1 Compiler Versions 31
7.2 Differences in MIR Optimization Levels 31
7.3 MCG Anti-Patterns 32

8 Conclusion **33**

References **34**

Acknowledgments

A single acknowledgments section is clearly insufficient to give credit to everyone who supported me during my time at Brown and beyond, but here is an attempt.

I extend my foremost gratitude to my advisor, professor, and friend, Malte, for his extensive knowledge in systems and further, as well as for his kindness, encouragement, and unwavering support of my most ambitious undertakings (like writing a static analysis of Rust without knowing Rust). These have been cornerstones of this project's success and have contributed immeasurably to my growth as a researcher, writer, and computer scientist.

I thank my reader, Will, for his support, expertise, and feedback on this project. I am deeply sad I won't be around to take Topics in PL with you next semester.

I am deeply grateful to my mentor, buddy, and amazing labmate Kinan, for his guidance, motivation, lighthearted humor, espresso tonics, and Shiner Bocks. None of the projects I worked on would have been as successful without your mentorship and eagerness to help me debug, brainstorm, and proofread. Your passion for teaching is one of the greatest I've ever seen in a person, and it has profoundly impacted all of ETOS. I am excited to see all the cool things you will accomplish as a professor, and I am sad I won't find you sleeping on the lab couch in the mornings anymore.

I sincerely thank my best friend Corinn, for her warmhearted support during late nights in the lab, tea hangouts at Ceremony, and Flatbread dinners in the Wellness courtyard, and for proofreading numerous drafts of this and other works. I am grateful for the many ways our paths have intersected as TAs, labmates, and friends, among others.

I have been privileged to work with many members of the ETOS group, including many of my coauthors. You all gave me a sense of community and belonging at Brown.

I'm indebted to my professors and teachers, of whom there are many. I can't underestimate your impact on me as a learner.

I thank my friends who have supported me from places close and far.

Finally, I owe my thanks to my family, whose unconditional support has made me who I am.

Abstract

Rust is a popular systems programming language that offers compile-time guarantees via its strong type system. Many recent systems have used Rust to enforce safety properties, such as memory safety or the absence of data races. In addition, developers often write additional static analyses of Rust code to bootstrap higher-level guarantees for their systems, such as data privacy or data integrity.

This thesis presents work to improve the usability of the Rust static analysis infrastructure. Currently, the Rust compiler provides very limited support for custom static analyses spanning multiple function bodies, which we call *interprocedural analyses*. This hinders the development of static analyses that verify global properties of the program, rendering them unsound or imprecise in the real-world setting.

We present PEAR, a framework for Rust static analysis that fills the gap in the lack of tooling for interprocedural analyses in Rust. PEAR APIs allow the developers to compute low-level information for each of the program constructs and integrate it into a new, automatically derived global Rust program representation we call a *monomorphized call graph* (MCG).

We evaluate PEAR on two real-world systems that employ Rust static analysis. We empirically confirm that PEAR MCG construction is sound, as it resolves dynamic dispatch correctly. We show that using PEAR allows expressing interprocedural analyses with less developer effort and higher accuracy compared to prior, handwritten versions of the same analyses.

1 Introduction

Rust is a general-purpose systems programming language that offers static guarantees like memory safety and the absence of data races through its type system. This allows building systems with appealing security properties on top of Rust. Many recent systems have used Rust both as a way to enforce memory safety and to bootstrap *higher-level guarantees* such as crash safety [LTB+24b], data integrity [BKA+18; MQL+24; NHD+20; PHJ+16], and data privacy [DAA+24; LTB+24a]. To catch offending programs, the Rust compiler runs a suite of static analyses such as the borrow checker [Ruse] or the unsafety checker [Rusf], rejecting programs that violate the checks at compile time. In addition to built-in analyses, the Rust compiler provides an interface to write compiler plugins to perform custom static analyses [Rusc]. Multiple systems use this framework to integrate with the compiler and perform additional analyses [ABF+22; AZS+23; CPA+22; DAA+24; GC23; LHC+23; MWD+23; VSC+22].

1.1 The Problem

Writing static analyses of Rust code is easier than doing so for other systems programming languages since the Rust compiler was designed with extensibility in mind [Rusc]. However, the Rust compiler exposes a limited set of analysis primitives for third-party users. The reason behind this is the focus of the Rust compiler developers on the needs of the compiler itself rather than the needs of the compiler plugin developers.

The Rust compiler exposes type analysis queries to compiler plugins and provides analysis frameworks for intraprocedural analyses, such as the visitor framework [Rusb] or the dataflow framework [Rusa]. However, the Rust compiler provides limited support for tools that perform analyses across function boundaries, i.e., *interprocedural analyses*. This limitation hinders the development of interprocedural analyses that verify useful global properties about the programs with high accuracy.

Building an interprocedural analysis requires reasoning about the entire program and its control flow across function boundaries. This is a hard task for languages that support higher-order control flow, like Rust. For instance, the exact values of certain callees in Rust, such as trait objects or closures passed as values throughout the program, can only be determined at runtime. We refer to this situation as *dynamic dispatch*.

Currently, program analyses that aim to analyze global properties of Rust programs choose one of the following approaches:

1. Perform best-effort interprocedural analysis and do not attempt to fully resolve dynamic dispatch [LWS+21; ZTJ+24].
2. Do without interprocedural analysis and perform intraprocedural analysis for every function in isolation [BKA+21; Car15; LTB+24a].

3. Limit interprocedural analysis to simple functions and use dynamic enforcement (e.g., sandboxing) for more complicated scenarios [DAA+24].

4. Translate Rust code into a different representation and dispatch to third-party tools for symbolic execution [ABF+22; VSC+22].

None of these approaches is fully satisfactory. To illustrate this, consider a developer concerned about security of some dependency who wants to audit it to figure out if it ever performs a network-related system call. To perform the audit thoroughly, the developer needs to look into the transitive dependencies of a program as well. However, manually following every call chain of every function in the dependency is labor-intensive and error-prone. A static analysis that traverses the program and finds all reachable network calls would simplify the auditing process. It removes the need for manual traversal and reduces the auditing to looking through a small set of calls.

Now, consider how each of the approaches described above falls short in this situation.

Ignoring dynamic dispatch makes the analysis unsound. The strawman analysis will skip all network calls behind dynamic dispatch, which should be shown to the person performing the auditing. Dynamic dispatch is ubiquitous in the Rust ecosystem [VSC+22], so any analysis run in a real-world setting will have this limitation.

Considering every function in isolation is sound but incomplete. Without the information about calling relationships between functions, the person performing the auditing will not be able to tell which functions can actually be called at runtime and which ones the strawman analysis collected erroneously. Hence, analyzing every available function instead of only the reachable ones could become a problem if it yields too many false positives.

Having a fallback mechanism for whenever static analysis fails might work for some use cases but introduces trade-offs. In our example, the developer could run a best-effort static analysis and run the rest of the code in a sandbox. However, using a sandbox carries a performance cost, which could be prohibitive for some applications.

Translating Rust code into a different representation requires careful reasoning about its semantics. At the same time, using symbolic execution engines is time-consuming for larger functions. In our example, the strawman analysis would need to encode information about which functions correspond to network calls into the target representation and dispatch to the symbolic execution engine. However, running the strawman analysis on some of the larger targets may take a prohibitively long time.

A framework for performing interprocedural analysis that builds directly on Rust compiler primitives could avoid the problems outlined above. This would, in turn, help program analyses using this framework to perform their analysis with better precision and less developer effort.

1.2 Proposed Solution

Interprocedural analyses in Rust benefit from having access to two types of information: *(i)* all reachable function bodies in a monomorphized form (with all generics instantiated) and *(ii)* a sound approximation of calling relationships between them.

We present a call-graph-based representation for Rust programs where all dynamic dispatch is soundly unwrapped, and all functions have type parameters instantiated, which we call a *monomorphized call graph* (MCG).

Hence, the thesis makes the following contributions:

1. A new Rust program representation (MCG), which makes it easier for static analysis developers to express their interprocedural analyses of Rust code (§3.3).
2. An algorithm for deriving an MCG for arbitrary Rust programs, borrowing techniques from Rapid Type Analysis [BS96] (§3.4).
3. A shared library called PEAR, which derives MCGs for arbitrary Rust programs and helps static analysis developers express their interprocedural analyses with less developer effort (§4).
4. Experience with porting static analyses from two real-world systems to PEAR, Sesame [DAA+24] (§5.1) and Sniffer [TAS+25] (§5.2), which shows that PEAR can support the static analysis needs of these systems (§6.2) and that using PEAR allows expressing these analyses with less developer effort (§6.3).

The conceptualization, design, and implementation of PEAR presented in this thesis, as well as the work of porting Scrutinizer, the static analyzer of Sesame [DAA+24], to PEAR was done by myself.

PEAR was developed simultaneously with Sniffer [TAS+25], which is a parallel piece of work complementary to PEAR that uses PEAR as a backbone of its static analysis. PEAR design and APIs were heavily influenced by the use case of Sniffer. The process of porting Sniffer to PEAR was performed by Sniffer's lead author, Corinn Tiffany.

Some of the PEAR implementation borrows directly from the previous work: the serialization and tagging functionality is an extension of the approach that first appeared in Paralegal [AZS+23] and was used in my prior published work on Sesame [DAA+24].

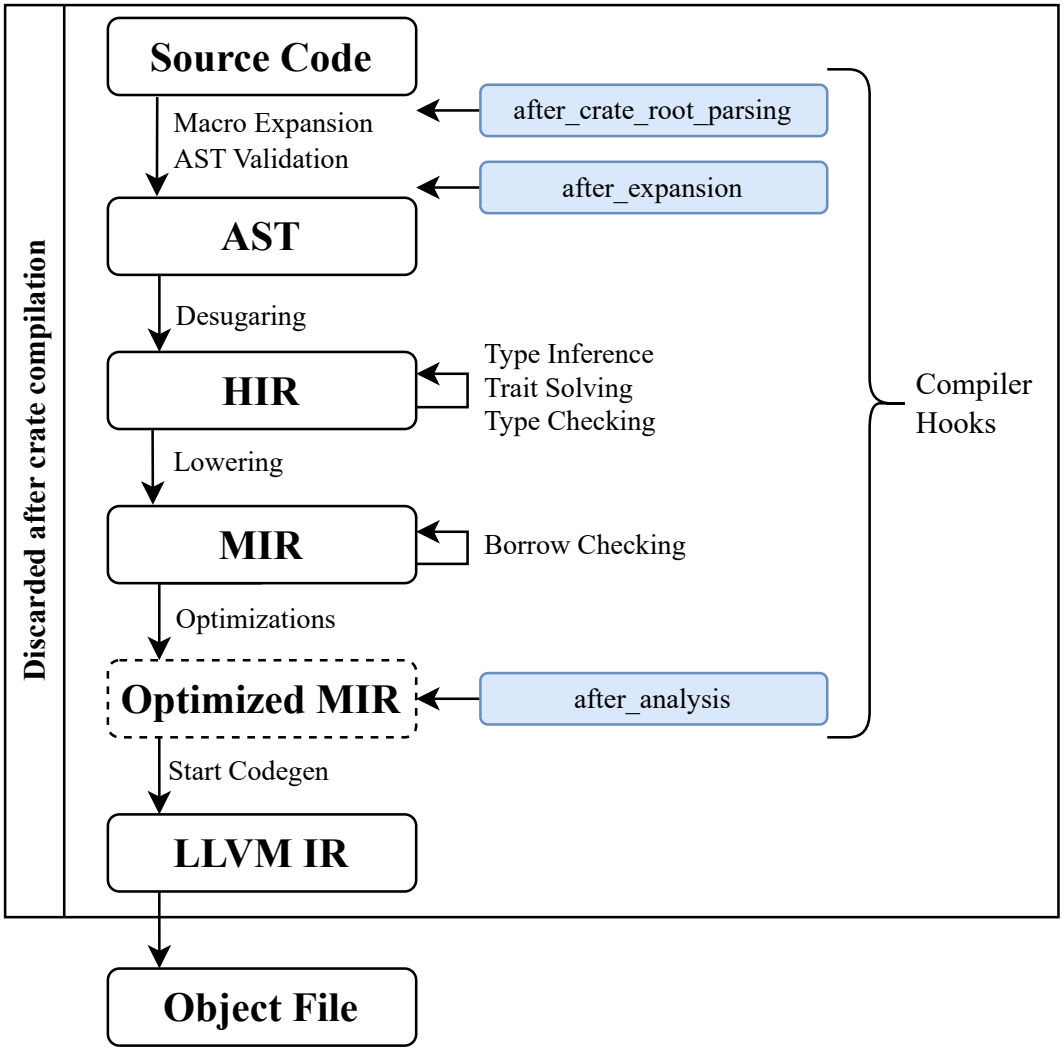


Figure 1: Overview of the Rust compilation process relevant to PEAR. Boxes shaded blue show compiler hooks available to compiler plugin developers.

2 Background and Related Work

2.1 Rust Compiler Overview

Modules in Rust are organized into units of compilation called "crates," which are similar to software packages in other languages. The Rust compiler performs compilation per crate, transforming each crate into an object file.

At a high level, the Rust compiler performs a sequence of transformations to the source code, generating multiple intermediate representations (IRs). During each transformation, the compiler performs analyses, like borrow checking or type checking, to prove properties about the program and reject programs violating these properties. By default, the compiler discards each crate's intermediate representations after its compilation.

Figure 1 shows a simplified diagram of the Rust compilation process relevant to PEAR. First, the Rust compiler performs the source code lexing and parsing, transforming the source code to the abstract syntax tree (AST) representation. After that, the Rust compiler lowers the AST to the High-Level IR (HIR), which it then lowers to the Mid-Level IR (MIR). Once the compiler produces MIR for each item, it lowers it to the LLVM IR. LLVM [LA04] is an off-the-shelf compilation toolchain that performs further optimizations and translates LLVM IR to object files for linking.

Each of the lowering steps performs a transformation that makes certain program analyses easier to perform and brings the program one step closer to the machine code. For example, the compiler performs type inference, trait solving, and type checking on the HIR and some Rust-specific optimizations on the MIR.

2.2 Rust Compiler Plugins

The Rust compiler can invoke third-party tools during the compilation process via the compiler plugin APIs [Rusc]. The compiler plugins can access the Rust IRs and perform additional static analyses over them. Compiler plugin developers interact with the Rust compiler by providing callbacks, which get invoked during the compilation process. The Rust compiler invokes callbacks repeatedly for each of the compiled crates. Notably, a callback invoked during the compilation of one crate does not have access to the compilation results from other crates. Figure 1 shows available callbacks and stages in the compilation process where they get invoked (shaded blue).

The Rust community has created tools to simplify the process of developing compiler plugins. `rustc_plugin` [CAG+23] is a framework for writing Rust compiler plugins that provides infrastructure for integrating with Rust's package manager, Cargo, as well as additional utilities. PEAR uses `rustc_plugin` to integrate with Cargo and run cross-crate analysis. Stable MIR [Sta] is a set of compiler APIs stable across compiler version changes, which aims to simplify the compiler plugin development process. It makes it easier for compiler developers to migrate between compiler versions and provides a simpler API for interacting with a subset of MIR.

2.3 Static Analysis in Rusty Systems

Multiple research systems use static analysis to reason about Rust code. Sesame [DAA+24] includes a Rust static analysis tool called "Scrutinizer." Scrutinizer verifies leakage-freedom of code regions as a part of the larger goal of Sesame to provide practical policy enforcement guarantees for Rust developers. Rudra [BKA+21] is a static analysis tool that finds memory safety bugs in unsafe Rust at scale. Rudra applies multiple heuristics over both HIR and MIR to detect problematic patterns. MirChecker [LWS+21] performs static analysis over MIR using constraint-solving techniques and abstract interpretation to detect potential runtime crashes and memory-safety errors. These systems mention the limita-

tions of their interprocedural analysis or complete lack thereof as an obstacle to system usability.

Automated verification tools for Rust, like Kani [VSC+22] or Prusti [ABF+22], translate Rust code to a different representation (GOTO for Kani and Viper for Prusti). They then use existing verification tooling on the translated code to reason about all possible executions of the program using symbolic execution techniques. While those tools choose a different approach, alternative verification tools developed purely for Rust could benefit from having access to PEAR-style call graphs. For example, Njor and Hilmar [NG21] outlines an abstract interpretation engine for Rust's MIR and mentions the lack of interprocedural analysis as a limitation.

Rupta [LHG+24] integrates with the Rust compiler to perform points-to analysis of Rust programs. Points-to analysis approximates which references point to which heap regions. Results from points-to analysis could serve as a foundation for interprocedural analyses in Rust since points-to analysis can discover the pointees of function pointers or trait objects. While precise, points-to analysis is resource-intensive since it computes much more information than necessary for resolving dynamic dispatch. In comparison, PEAR's call graph construction collects only objects relevant for dynamic dispatch and approximates them without computing precise points-to information. If necessary, static analyses expressed with PEAR can compute points-to information as a part of their analysis.

2.4 Interprocedural Analyses in Object-Oriented Programming Languages

Related work includes several approaches for constructing call graphs and performing interprocedural analyses of object-oriented programming languages.

Reachability analysis (RA) [Sri92] is an inexpensive method for constructing call graphs. When resolving dynamic dispatch, RA considers only function names or type signatures. Hence, RA assumes that *any* function with the matching name or type signature can be callable.

Class hierarchy analysis (CHA) [DGC95] and Rapid Type Analysis (RTA) [BS96] build on RA to also account for inheritance relationships and instantiation information, respectively. When resolving dynamic dispatch, CHA only considers the methods from the associated class and its ancestors. RTA further prunes all methods for classes that have not been instantiated in the program.

Further modifications to RTA, like XTA [TP00], consider more information, such as accessed class fields and method type signatures, to resolve dynamic dispatch more precisely.

Control-flow analyses (CFAs) [Shi91] compute a conservative approximation of the values each expression in the program evaluates to. In comparison to the previous analyses, CFAs compute information for each individual expression rather than the whole program. Hav-

ing this information at hand allows building an approximation of the call graph for a program. This, in turn, allows performing interprocedural analyses since all dynamic dispatch has already been unwrapped. A common way to implement a CFA is via small-step abstract interpretation [Mig07], which is the approach of static analysis in Sesame [DAA+24].

PEAR borrows from the techniques discussed above in multiple ways. PEAR's call graph construction is similar to the approaches that collect information per program rather than per expression. As a result, the PEAR call graph construction approach is more coarse-grained and less resource-intensive than CFAs. Similarly to RTA, PEAR excludes all non-instantiated dynamic dispatch objects from the constructed call graph.

Most of the algorithms above were designed for the object-oriented programming model of Java. Instead of Java classes, Rust uses traits, which are conceptually similar to interfaces, to implement the object-oriented paradigm. This choice introduces some constraints that help simplify the PEAR call graph construction. For example, Rust does not support function overloading, so all functions are uniquely identified by their absolute path. Furthermore, trait casts in Rust are explicit, so the trait of each dynamically dispatched method is always statically known. Hence, the modifications to RTA provided by XTA are unnecessary in Rust. On the other hand, Rust supports dynamic dispatch via function pointers. Since those do not belong to a trait, PEAR disambiguates them by type signatures, similarly to RA. This creates imprecision in some anti-pattern situations, which we discuss in §7.3.

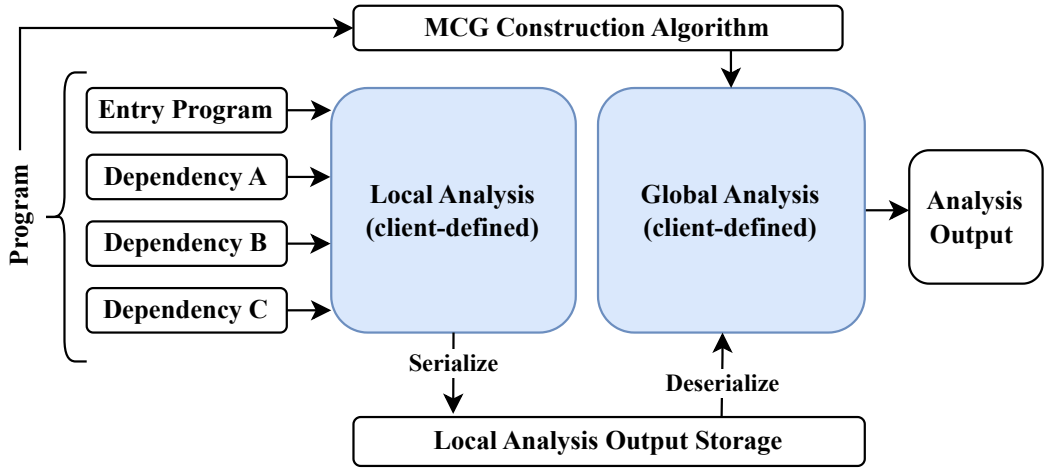


Figure 2: Overview of PEAR architecture. PEAR clients define local and global analyses for PEAR to execute.

3 Design

We now give an overview of PEAR and its APIs (§3.1), describe the MCG structure (§3.3), and the process of MCG construction (§3.4).

3.1 PEAR Overview

PEAR's goals are to make the analyses expressed with PEAR easy to write, customizable, and efficient. In doing so, we face three major challenges. First, PEAR needs to resolve dynamic dispatch correctly to save the static analysis developer from having to resolve it on their own. Second, PEAR needs to collect information from different crates that the Rust compiler discards after compilation to allow the static analysis to use all available information. Third, PEAR needs to filter out the information unnecessary to the analyses early on since preserving all intermediate representations for large programs is prohibitively expensive.

We overcome these challenges in PEAR by structuring PEAR analyses in two parts: *local analysis* and *global analysis*. Local analysis runs first, retrieves relevant details for each crate, and saves them to disk. Global analysis runs after, computes the MCG for the program, and can query the results saved by the local analyses. Both of the analysis types serve a distinct role, since a local analysis cannot construct the MCG as it sees only one crate in isolation, and a global analysis misses low-level details available to the local analysis.

Local analysis provides the means for the static analysis developers to transform and filter relevant crate-local information. This is necessary since serializing all available information for global analysis to consume would be costly, compromising the efficiency goal. Global analysis connects a bird's eye view of the program, represented by the MCG, to-

	Local Analysis	Global Analysis
Invocation place	Each compiled crate	Entry function crate
Available inputs	Everything the Rust compiler has access to	Optimized MIR, local analysis outputs
Analysis output	Serialized to disk to be later retrieved by global analysis	Provided directly to the client
Can access MCG	No	Yes

Figure 3: Comparison between local and global analyses of PEAR. Local and global analyses complement each other: local analysis captures low-level information for each compilation unit, while global analysis integrates it into a program-level data structure (MCG).

gether with the relevant per-function information, obtained from the local analyses. PEAR provides APIs to traverse MCGs and query deserialized information from local analyses to aid the static analysis developers.

Figure 2 illustrates the process of PEAR operation. Static analysis developers provide static analyses for PEAR to execute by implementing a `LocalAnalysis` and a `GlobalAnalysis` traits. PEAR invokes the analyses at the appropriate time during the compilation process.

Figure 3 compares the two types of analysis. Local analyses run as a part of the compilation of each crate. This allows local analyses to access information that the compiler would otherwise discard after the crate compilation is done, like unsafety information or borrow checker facts. At the end of each local analysis, PEAR serializes its results and tags them with a stable identifier, which the global analysis can later query. Global analysis runs after all local analyses complete and can access the results from all local analyses that ran beforehand and the computed MCG for the program. Global analysis runs in the context of the crate of the entry function. It has access to all of the local analysis results and the MCG constructed from the entry function. Most global analyses take the form of a traversal over the MCG. Global analyses use identifiers present in the MCG to query the results from local analyses, which PEAR deserializes and provides to the global analysis.

3.2 PEAR Analysis Walkthrough

Let us take an example of a program analysis that identifies all reachable unsafe blocks in the program and walk through how a static analysis developer uses PEAR APIs to express this analysis.

The code snippet provided in Figure 4 shows the implementation of local and global analyses for this example. PEAR clients write local and global analyses by implementing the corresponding trait: line 3 for the local analysis and line 17 for the global analysis.

```rust
pub struct LocalUnsafety { ... }

impl LocalAnalysis for LocalUnsafety {
  type Output = Vec<CodeLocation>;

  fn execute(&self, compiler: Compiler, function_id: Id) -> Output {
    // Query the compiler to retrieve HIR for the function.
    let function_hir = compiler.get_hir(function_id);
    // Search unsafe locations in the HIR of the function.
    let unsafe_locations = get_unsafe_locations(function_hir);
    // Return the unsafe locations.
    // PEAR serializes those and tags them with the function ID.
    return unsafe_locations;
  }
}

pub struct GlobalUnsafety { ... }

impl GlobalAnalysis for GlobalUnsafety {
  type LocalAnalysis = LocalUnsafety;

  fn execute(&self, compiler: Compiler, mcg: MCG) {
    // Traverse the MCG of the program.
    for node in mcg {
      // Query PEAR for the relevant local analysis results.
      let unsafe_locations = load_local_analysis_results(node);
      // Print locations of unsafe blocks to the user.
      for location in unsafe_locations {
        println!(location);
      }
    }
  }
}
```

Figure 4: PEAR-based implementation of static analysis that finds all reachable unsafe blocks in a given program. PEAR automatically serializes results of the local analysis, loads them during global analysis on demand, and provides automatically constructed MCG for the global analysis.

The compiler performs unsafety checking at the HIR level, so it will discard unsafety information once it is done compiling the crate. Hence, the local analysis, which runs in the context of each crate, needs to retrieve unsafety information for each of the function bodies before the compiler discards it. To do so, the local analysis first needs to register the type of its output (line 4). PEAR needs to know how to serialize the output of each local analysis to save it after PEAR is done executing the local analysis. PEAR synthesizes serialization logic for common types from the Rust compiler and the standard library containers holding those types. Thus, PEAR knows how to serialize `Vec<CodeLocation>`, and no further actions are necessary from the developer. The only task the local analysis has left to complete is to query the appropriate information from the compiler (line 8) and filter the relevant information about unsafe blocks (line 10) using a custom helper function `get_unsafe_locations`, not shown here. PEAR runs the local analysis for each crate, and serializes and stores its outputs on disk.

Global analyses serve as a way to both filter and aggregate the information local analyses provide. For the unsafety analysis example, the global analysis needs to filter out unsafe blocks not reachable by the program. Global analyses make use of the MCG that PEAR constructs for the program. PEAR in Figure 4 provides the global analysis with the MCG (line 22). Then, the global analysis traverses the MCG (line 24), querying the results of the local analysis for each node (line 26). PEAR automatically determines the appropriate output type of the local analysis linked to the currently running global analysis (line 20) and retrieves the data. Once the analysis finishes loading and filtering the information about reachable unsafe blocks, it can output the information to the user (line 29).

3.3 Monomorphized Call Graph (MCG)

Any program in a language with dynamic dispatch could produce different call graphs for different executions, depending on which execution path it takes at runtime. Intuitively, an MCG is a union of all possible call graphs of a program for all possible runtime execution paths.

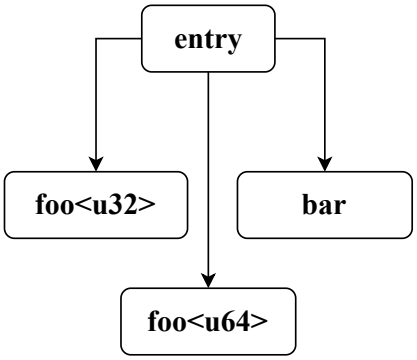
PEAR always builds MCG with respect to some root node, which is the entry point of the program PEAR analyzes. PEAR requires the root node to contain no unresolved generics or arguments containing dynamic dispatch objects (we discuss this limitation in detail in §4.2).

More specifically, an MCG is a program call graph that satisfies the following properties:

- All nodes in an MCG are function instances with all generics instantiated (i.e., monomorphized function instances).
- The outgoing node set of each node contains all functions called by the corresponding monomorphized function instance at some program execution.

```
1 fn foo<T>(input: T) { ... }
2
3 fn bar(input: u32) { ... }
4
5 fn entry(flag: bool, a: u32, b: u64) {
6   let func = if flag { foo } else { bar };
7   func(a); // Call via dynamic dispatch.
8   foo(b); // Static call.
9 }
```

(a) Sample Rust program.



(b) MCG constructed for the program.

Figure 5: Sample Rust program and the corresponding MCG PEAR constructs for the program.

Figure 5 shows a sample Rust program and its MCG. Here, PEAR starts constructing MCG from function `entry` (line 5), so PEAR analyzes its function body first.

Note that the compiler does not statically know the value of `func` because `func` can refer to either `foo` or `bar` (line 6), depending on the value of `flag` at runtime. Hence, PEAR must include both of the instances in the MCG, since `entry` could be calling either at runtime.

It is insufficient to simply include generic `foo<T>` in the MCG, since `foo` could behave differently based on the value of its type parameter `T` at runtime. To address this, PEAR must add the monomorphized version of `foo` to the MCG. Since the type of `a` on line 5 is `u32`, PEAR includes `foo<u32>` in the MCG.

`entry` could also call `foo` with a different value of type parameter `T`. On line 8, `entry` calls `foo` with an argument of a different type. In this case, the type of `T` associated with `foo` is `u64`, so PEAR includes an additional instance of `foo<u64>` in the MCG.

3.4 MCG Construction

PEAR constructs the MCG in two steps: reachability analysis and refinement.

3.4.1 Reachability Analysis

In the reachability analysis step, PEAR computes a set of functions transitively reachable from a given entry point using the reachability algorithm from the Rust compiler [Rusd]. The reachable set contains both the functions called statically and the functions called via dynamic dispatch. For example, the call to `foo` on line 8 of Figure 5a is a static call, since the value of `foo` is unambiguous at compile time. In contrast, for the calls via dynamic dispatch, the compiler does not know the callee at compile time. In the context of Rust, calls via dynamic dispatch are calls to dynamic trait object methods or function pointers.

Metadata Stored for Refinement	Function Call	Function Call Type
Trait `DefId`, trait method `DefId`	Vtable item, indirect drop	Dynamic
Function signature	Static function, function pointer, `Fn` trait item, static closure shim	Dynamic
None	Analysis root, direct call, drop, assert, unwind, inline assembly	Static

Figure 6: PEAR stores additional metadata for each node in the reachability graph. This information is used during refinement for MCG construction.

For example, the call to `func`, on line 7 of Figure 5a is a call via dynamic dispatch, since `func` is a function pointer and the compiler does not know its value at compile time.

Since the target of calls via dynamic dispatch is known only at runtime, the reachability algorithm uses heuristics to collect dynamic dispatch callees and approximate the set of reachable functions. To maintain soundness, the algorithm collects all objects callable via dynamic dispatch at their creation. For example, the algorithm collects all methods attached to dynamic trait objects and functions converted to function pointers whenever a dynamic trait object or a function pointer is created anywhere in the program. For the example in Figure 5a, the algorithm will collect `foo` and `bar` on line 6, once the function pointers are created.

We extend the reachability algorithm by making PEAR annotate each reachable function with the reason for its collection and additional metadata to differentiate between calls via dynamic dispatch. Figure 6 shows the list of call types and additional metadata PEAR stores for each item. For the example in Figure 5a, PEAR will categorize a call to `foo` on line 8 as a static function call and store no extra metadata for it. For the function pointers later assigned to `func`, PEAR will store the signature corresponding to the function pointers at their creation on line 6.

3.4.2 Refinement

After the reachability algorithm, PEAR runs the refinement algorithm to compute the MCG starting from a given entry point. PEAR recursively builds the MCG by visiting all function calls from the current function body and adding the discovered callees to the graph. For static function calls, the callee is known at compile time, so PEAR adds the function to the graph and recurses into the callee. For dynamically dispatched function calls, the callee is not known at compile time, so PEAR uses one of the heuristics, which correspond to

```
1  trait Foo { fn method(&self); }
2  trait Bar { fn method(&self); }
3
4  struct S1 { ... }
5  struct S2 { ... }
6  struct S3 { ... }
7
8  impl Foo for S1 { fn method(&self) { ... } }
9  impl Foo for S2 { fn method(&self) { ... } }
10 impl Bar for S3 { fn method(&self) { ... } }
11
12 fn main() {
13     let s1 = S1 { ... };
14     let s2 = S2 { ... };
15     let s3 = S3 { ... };
16
17     // True positive.
18     // PEAR adds <S1 as Foo>::method to the MCG.
19     // main does call <S1 as Foo>::method.
20     let s1_ref: &dyn Foo = &s1;
21
22     // False positive.
23     // PEAR adds <S2 as Foo>::method to the MCG.
24     // main does not call <S2 as Foo>::method.
25     let s2_ref: &dyn Foo = &s2;
26
27     // True negative.
28     // PEAR does not add <S3 as Bar>::method to the MCG.
29     // main does not call <S3 as Bar>::method.
30     let s3_ref: &dyn Bar = &s3;
31
32     // <S1 as Foo>::method is the only call from main.
33     s1_ref.method();
34 }
```

Figure 7: PEAR uses a heuristic to approximate dynamic dispatch, which introduces false positives to the MCG. PEAR adds `<S2 as Foo>::method` to the MCG (line 25), but `main` never calls `<S2 as Foo>::method`.

the two metadata types stored for dynamically dispatched calls (described in the first two rows of Figure 6).

For trait object method calls (row 1 of Figure 6), PEAR finds all reachable trait object methods that match the trait and method identifier of the target. Any trait object method call happens only after the trait object containing the corresponding function is created, which the reachability algorithm will collect. Hence, the heuristic soundly approximates all trait object method calls that could take place.

Figure 7 illustrates the false positives that could occur during the refinement process for trait object method calls. During reachability analysis, PEAR collects all methods attached to `S1`, `S2`, and `S3` on lines 20, 25, and 30 respectively. During refinement, PEAR only considers the call to `method` on line 33. Since both `<S1 as Foo>::method` and `<S2 as Foo>::method` match the trait identifier (`Foo`) and the method identifier (`method`) of the call on line 33, PEAR adds both to the MCG.

`<S1 as Foo>::method` is called during program execution (line 33), and PEAR added it to the MCG, so it is a true positive. However, since `<S2 as Foo>::method` is never called during program execution but PEAR added it to the MCG, it is a false positive. Since `<S3 as Bar>::method` has a different trait identifier, PEAR does not add it to the MCG and it is never called during program execution, so it is a true negative.

For any of the function-pointer-like targets (row 2 of Figure 6), which includes function pointers, closures, and static functions, PEAR finds all reachable function-pointer-like function calls that match the desired function signature. By the same token, any function-pointer-like function call happens only after the reference to the corresponding function is created, so this heuristic is also sound. Once PEAR is done computing the heuristic, PEAR adds all matching nodes to the MCG, assuming that any of them could be callable at the runtime. PEAR marks the approximated nodes as such to show that the approximation took place to the client. At the end of the refinement phase, PEAR will have constructed an MCG from the set of reachable functions.

3.4.3 MCG Construction Soundness

While a proof of soundness for the MCG construction is out of scope for this thesis, we provide an argument about the soundness of its dynamic dispatch resolution, which captures the intuition for why MCG is a sound approximation.

First, we assume that the compiler reachability algorithm correctly computes a set of all functions that could potentially be called at runtime.

Then, we empirically observe the following facts about Rust:

1. Dynamic dispatch in Rust happens either through trait object methods or through function-pointer-like constructs.

2. Creation of trait objects and function-pointer-like constructs is explicit and captured by the reachability algorithm of the compiler.
3. Calls to dynamically dispatched functions through trait objects and function-pointer-like constructs are explicit and captured by the MIR.
4. Each dynamic dispatch object belongs to a single group, which can be uniquely identified by the metadata from Figure 6 at the creation time and at the call time.

It follows that the creation of all dynamic dispatch objects is captured by the reachability analysis part of the MCG construction algorithm, and the call sites of all dynamic dispatch objects are captured by the refinement part of the algorithm. Furthermore, since the metadata matches at creation and execution, the MCG construction algorithm soundly resolves dynamic dispatch.

4 Implementation

We implemented PEAR in 5.8k sLoC of Rust. PEAR is a compiler plugin, which makes it necessary to pin both PEAR and its dependencies to a specific compiler version. This is required since internal Rust compiler APIs are unstable and change from one nightly version to another. PEAR is pinned to the `nightly-2024-01-06` version of Rust compiler. We provide a modified version of `rustc_plugin` pinned to the same compiler version as a part of the PEAR repository.

We release the implementation of PEAR as an open-source GitHub repository available at `https://github.com/artemagvanian/pear`.

4.1 Integrating with PEAR

Static analyses integrate with PEAR by providing their implementation of local and global analyses to PEAR. PEAR then generates a set of compiler callbacks and invokes the compiler with them. PEAR also configures the Rust compiler with the appropriate arguments to preserve optimized MIR and recompile the Rust standard library. Both of the options are necessary for the soundness of MCG construction. PEAR further modifies `rustc_plugin` to provide the necessary arguments to all compiler invocations needed by PEAR.

4.2 Implementation Limitations

Currently, our PEAR prototype errors when tasked with constructing an MCG from an entry point that has unresolved generics or accepts trait objects and function pointers as arguments. Dynamic dispatch that depends on such uninitialized parameters is unresolvable in full, since they can take arbitrary values at runtime. Hence, a solution supporting this use case must skip the uninitialized top-level parameters. With additional engineering effort, it is possible to extend the MCG construction algorithm to support such entry points by filtering out dynamic dispatch associated with the top-level sources. However, the MCG constructed from such entry points will be inherently incomplete, since it is impossible to know which values top-level generics or dynamic dispatch objects will take at runtime.

The current implementation of PEAR omits assert, unwind, or inline assembly nodes in its MCG construction. Neither of the PEAR-based analyses in our evaluation makes use of these nodes, but, they can be included into the MCG with more engineering effort.

5 Application Case Studies

We used PEAR to express the static analyses from two real-world systems: Sesame [DAA+24] and Sniffer [TAS+25], which we describe below.

5.1 Sesame

Sesame is a framework for end-to-end privacy policy enforcement in web applications. Sesame wraps data in policy containers that couple data with policies that apply to it. To operate on the data, developers using Sesame must access it through vetted *privacy regions*. Sesame prevents privacy regions from leaking protected data by applying static analysis and sandboxing to the regions.

Sesame's static analyzer, Scrutinizer, collects all available function bodies and resolves dynamic dispatch along the way. To resolve dynamic dispatch, it uses small-step abstract interpretation, implemented via the dataflow analysis framework of the Rust compiler. Scrutinizer then organizes the function bodies it discovered into a call graph, which represents a union of the program's call graphs from all possible executions. Afterward, Scrutinizer runs its leakage-freedom analyses on the call graph.

5.1.1 Requirements for Scrutinizer

To uphold the guarantees of Sesame, Scrutinizer should label any leaking privacy regions as such. In other words, the false negative rate of Scrutinizer must be zero. On the other end, Sesame provides a fallback mechanism for enforcing non-leakage of privacy regions that Scrutinizer fails to verify as non-leaking. Hence, Scrutinizer can conservatively label some non-leaking regions as leaking.

5.1.2 PEAR-enabled Scrutinizer

We notice two facts about Scrutinizer. First, the bulk of its source code implements the call graph construction, while its analysis part is small and self-contained. Second, most of the limitations of Scrutinizer stem from the imperfections of its call graph construction. These facts make Scrutinizer a static analysis that could benefit from having access to MCGs.

Local analysis for PEAR-enabled Scrutinizer retrieves borrow checker facts needed to run its variable flow analysis and the function bodies to which the borrow checker facts apply. Global analysis for PEAR-enabled Scrutinizer runs the original leakage analysis using the data from the local analysis to approximate variable flows.

5.2 Sniffer

Sniffer is a targeted auditing tool using static analysis to alleviate auditor burden. One of Sniffer's use cases is auditing extensions for systems that depend on properties of safe Rust

not covered by Rust's UB-freedom guarantees. Examples of these properties include encapsulation of private `struct` members, absence of interior mutability, or panic-freedom.

Sniffer runs its analysis over an input crate and its dependencies. Similarly to Scrutinizer, Sniffer builds the call graph of all transitively reachable functions to ensure the exhaustiveness of its analysis. Users of Sniffer implement a check for the target property of interest as a visitor over this call graph. Sniffer recursively runs the visitor on the call graph, taking note of the paths through which each node is reachable. Sniffer reports the regions that fail the check and the paths through which the region is reachable. The auditor then uses the call chain information to decide if the region truly violates the property.

5.2.1 Requirements for Sniffer

For the auditing process to be correct, Sniffer should label any region violating the desired property as such. Similarly to Scrutinizer, the false negative rate of Sniffer should be zero. Since there is a person in the loop reviewing the regions, Sniffer can surface some benign regions as potentially violating the property.

5.2.2 PEAR-enabled Sniffer

Like Scrutinizer, the bulk of the original Sniffer code implemented call graph construction and suffered from its limitations. In addition, a big part of the original Sniffer codebase consisted of its own version of local analysis and the serialization infrastructure. Sniffer had to manually recompile the crates of interest to obtain the information the compiler otherwise discarded. Hence, Sniffer could benefit from having access to MCGs, PEAR local analysis architecture, and the serialization infrastructure.

Sniffer uses PEAR to implement analyses that find unsafe blocks, code that breaks encapsulation of private `struct` members, uses of interior mutability, and panics. The Sniffer authors used these analyses to audit multiple real-world systems.

Local analysis for PEAR-enabled Sniffer retrieves HIR-level information for each function body, since the compiler discards it after each crate compilation. Global analysis for PEAR-enabled Sniffer traverses the MCG to find function bodies containing patterns that could break system-level guarantees.

6 Evaluation

We ask the following questions to evaluate PEAR:

1. Does PEAR approximate dynamic dispatch correctly? (§6.1)
2. How precise is the MCG construction algorithm of PEAR, and what impact does it have on the accuracy of static analysis systems? (§6.2)
3. How much developer effort is required to use PEAR? (§6.3)

6.1 Dynamic Dispatch Resolution Correctness

Having PEAR resolve dynamic dispatch correctly is a prerequisite for the soundness of any analysis built on top of PEAR. To test the correctness of PEAR dynamic dispatch resolution, we use an open-source benchmark and supplement it with additional tests.

VanHattum et al. [VSC+22] contributed an extensive test suite for verifying dynamic trait objects in Rust and used it to check the dynamic dispatch resolution correctness of the Kani Rust Verifier tool. This test suite is available as a part of the source code repository for Kani [Kan21] and has grown over time. To use this test suite, we adapt the test cases to PEAR by filtering out any Kani-specific checks and encoding the test cases as assertions that all functions behind dynamic dispatch are present in the MCG for the program. A good result for PEAR would show PEAR MCG construction being sound, i.e., PEAR correctly identifies all functions executed at runtime.

PEAR correctly resolves dynamic dispatch for all available test cases. In addition to the test cases from VanHattum et al., we provide more complicated test cases, which test trait object chains, function pointer casts, and global statics. Figure 8 shows the details of the test cases.

6.2 Accuracy of MCG Construction

Since the MCG overapproximates the central call graph of the program, it can contain nodes that correspond to the functions that will never be executed by the program. To serve as a useful foundation for other static analysis systems, the overapproximation introduced by the MCG construction algorithm should not detract from the accuracy of the client static analyses.

To evaluate the impact of PEAR on the accuracy of existing static analysis systems, we run experiments from Sesame [DAA+24] and Sniffer [TAS+25] on the versions of these systems that we reimplemented on top of PEAR. A good result for PEAR would show that PEAR-enabled Sesame and Sniffer satisfy the requirements from §5.1.1 and §5.2.1. Specifically, PEAR-enabled Sesame and Sniffer should show equal or lower false positive

Test Case	Code Snippet	Author
Boxed trait	`Box::new(...) as Box<dyn T>`	VanHattum et al. [VSC+22]
Boxed auto trait	`Box::new(...) as Box<dyn Send + Sync>`	VanHattum et al. [VSC+22]
Closure reference	`Box::new(|| {...}) as &dyn Fn`	VanHattum et al. [VSC+22]
Function pointer reference	`&func as &dyn Fn`	VanHattum et al. [VSC+22]
Boxed closure	`Box::new(|| {...}) as Box<dyn FnOnce>`	VanHattum et al. [VSC+22]
Boxed function pointer	`Box::new(&func) as Box<dyn FnOnce>`	VanHattum et al. [VSC+22]
Nested closures	`Box::new(Box::new(|| {})) as Box<Box<dyn FnOnce>>`	VanHattum et al. [VSC+22]
Closure as function pointer	`|| {...} as fn() -> ()`	VanHattum et al. [VSC+22]
Trait upcasting	`trait T: U; <dyn T as U>::method(...)`	VanHattum et al. [VSC+22]
Unsized reference casts	`&(dyn T + Sync) as &(dyn T)`	VanHattum et al. [VSC+22]
Unsized `Box` casts	`Box<dyn T + Sync> as Box<dyn T>`	VanHattum et al. [VSC+22]
Unsized `Rc` casts	`Rc<dyn T + Sync> as Rc<dyn T>`	VanHattum et al. [VSC+22]
Explicit `Box` drop	`impl Drop for T; Box<dyn T>`	VanHattum et al. [VSC+22]
Explicit reference drop	`impl Drop for T; &dyn T`	VanHattum et al. [VSC+22]
Trait chains	`trait A : B + C, trait B : D, trait C: D`	Newly developed
Function pointers	`func as fn() -> ()`	Newly developed
Stored function pointer	`struct Wrapper(fn() -> ())`	Newly developed
Static function	`static FUNC: fn() -> ()`	Newly developed
Static closure	`static FN = || )`	Newly developed

Figure 8: PEAR passes a comprehensive test suite of dynamic dispatch test cases.

		Ground Truth		
		Non-leaking	Leaking	Unresolved
Original	Non-leaking	25	0	0
	Leaking	8	46	0
	Skipped	0	0	34
PEAR enabled	Non-leaking	25	0	0
	Leaking	8	46	0
	Skipped	0	0	34

(a) Running static analysis from Sesame on methods from `Vec`.

		Ground Truth		
		Non-leaking	Leaking	Unresolved
Original	Non-leaking	14	0	0
	Leaking	10	16	0
	Skipped	0	0	4
PEAR enabled	Non-leaking	14	0	0
	Leaking	10	16	0
	Skipped	0	0	4

(b) Running Scrutinizer on methods from `HashMap`.

Figure 9: Analyzing non-leakage of methods from the Rust standard library containers. PEAR-based Scrutinizer classifies the same number of non-leaking methods as the original Scrutinizer while remaining sound (rejecting all leaking methods).

rates compared to the original versions of these systems. At the same time, PEAR-enabled Sesame and Sniffer should show zero false negative rates, in line with the original versions.

6.2.1 Sesame

Sesame's static analysis, Scrutinizer, includes a comprehensive internal test suite, which PEAR-enabled Scrutinizer passes.

Furthermore, Scrutinizer includes an internal accuracy benchmark. The benchmark analyzes methods from two containers from the Rust standard library, `Vec`, and `HashMap`, for leakage-freedom. The Rust standard library includes a lot of optimizations and multi-level dynamic dispatch, which makes this benchmark a challenging task for static analysis.

Figure 9 shows the results of running PEAR-enabled Scrutinizer on this benchmark together with the results from the original Scrutinizer for reference. PEAR-enabled Scrutinizer correctly identifies all leaking methods (the intersection of "Ground Truth / Leaking"

Application	Leakage-Free	Accepted (Original)	Accepted (PEAR)
YouChat	3	3	3
Voltron	3	3	3
Portfolio	55	43	43
WebSubmit	19	17	17

Figure 10: The accuracy of PEAR-enabled Scrutinizer is comparable to the accuracy of the original one. PEAR-based Scrutinizer accepts all leakage-free regions that the original Scrutinizer accepts.

	Cargo Geiger	Cargo Scan	Sniffer (MCG only)	Sniffer (MCG + analysis)
Identified Regions	33,735	43,761	9,982	12
Problematic Regions	1	1	1	1

Figure 11: PEAR-based Sniffer is more accurate than the alternatives. MCG construction alone improves the accuracy by 3.4× compared to Cargo Geiger and by 4.4×, compared to Cargo Scan.

and "PEAR-enabled / Non-leaking" in each of the tables of Figure 9 is zero). This empirically confirms the soundness of PEAR-enabled Scrutinizer, in line with the zero false negative rate requirement.

PEAR-enabled Scrutinizer also correctly identifies the same number of non-leaking methods as the original Scrutinizer (see the "Ground Truth / Non-leaking" column in each of the tables of Figure 9). Despite performing a more coarse-grained dynamic dispatch resolution compared to the original Scrutinizer, PEAR-based Scrutinizer achieves an equal false-positive rate. In the context of Sesame, a higher false-positive rate would mean having to rely less on sandboxing, which could negatively affect performance. Similarly to the original Scrutinizer, PEAR-enabled Scrutinizer does not run on methods with non-instantiated generics, which we report in the "Unresolved" column of Figure 9.

As a part of the Sesame evaluation, its authors run Scrutinizer on privacy regions from four applications, both newly developed for Sesame and ported to Sesame. Figure 10 shows the results of running PEAR-enabled Scrutinizer on the applications. PEAR-enabled Scrutinizer successfully verifies all of the regions from the applications, which shows that the false positive rate of PEAR-enabled Scrutinizer is comparable to the original one.

In total, the experiment shows that porting Scrutinizer to PEAR enabled comparable accuracy while maintaining soundness.

6.2.2 Sniffer

We focus on two experiments from Sniffer: the first one comparing the number of false positives produced by Sniffer to alternative approaches, and the second one comparing the original Sniffer to PEAR-enabled Sniffer.

The first experiment finds occurrences of encapsulation-breaking code that would endanger Sesame's guarantees in an endpoint from Sesame. To provide a baseline for developer burden with existing tools, the experiment also applied Cargo Geiger [Car15] and Cargo Scan [ZTJ+24] to this endpoint.

Figure 11 shows the results. Sniffer identifies 12 calls with potential violations, while Cargo Geiger reported 33,735 potentially "used" `unsafe` items, and Cargo Scan reported 43,761 items to audit. Even without specifically targeting encapsulation-breaking, bare-bones Sniffer reporting all unsafe regions present in the MCG decreases the auditing burden by 3.4× compared to Cargo Geiger and by 4.4×, compared to Cargo Scan. PEAR-enabled Sniffer that runs its encapsulation-breaking analysis produces an auditor burden on the order of tens of regions, while the alternatives produce an auditing burden multiple magnitudes higher. This serves as an indirect testament to the usefulness of PEAR with respect to increasing the accuracy of Rust static analyses.

The second experiment runs Sniffer's unsafety-freedom analysis on a crate called `dryoc`, which claims to be "mostly free of unsafe code". Both the original Sniffer and PEAR-enabled Sniffer flag reachable unsafe regions of code. These regions serve as entry points for the library developers to audit the crate and ensure that unsafe regions do not interfere with the correctness and security goals of the crate.

For each of the analyzed functions, the original Sniffer reports tens of unresolved functions due to dynamic dispatch, while PEAR-enabled Sniffer resolves all dynamic dispatch. Even though PEAR-based Sniffer resolves dynamic dispatch fully, it yields a comparable auditor burden, with 13.85 unsafe functions per top-level public API function on average, compared with 10.02 on average for the original Sniffer.

6.3 Developer Effort

We qualitatively evaluate the developer effort required to port Sesame's static analysis, Scrutinizer, (§5.1) and Sniffer (§5.2) to PEAR.

6.3.1 Sesame

Leakage-freedom analysis in the original Scrutinizer was already decoupled from its collection algorithm. Hence, porting Scrutinizer to PEAR amounted to swapping the collection algorithm for PEAR-provided MCG generation and splitting the leakage-freedom analysis into local and global analyses to make use of PEAR APIs. We estimate that porting Scru-

tinizer to PEAR required approximately 40 person-hours for a person familiar with Rust static analysis, which we expect most of the PEAR clients are.

PEAR-based Scrutinizer comprises 1k sLoC of Rust compared to the 4.2k sLoC of the original Scrutinizer. Most of the reduction comes from having access to the MCG, which makes the collection unnecessary. Furthermore, strict delineation between local and global analyses provides a framework for the analysis, making it simpler. We discuss the problems we encountered during the porting process in the next section (§7).

6.3.2 Sniffer

Sniffer initially used the collection algorithm from Scrutinizer, which was brittle and specific to the needs of Sesame. Since Scrutinizer was tailored for small and self-contained regions of code, running it on the whole crates resulted in errors because Scrutinizer encountered code constructs its collection algorithm does not support.

Shifting to PEAR brought multiple benefits. PEAR-enabled Sniffer is able to analyze a wider range of programs, since, to the best of our knowledge, PEAR supports MCG generation for all language features available in its target Rust version. Furthermore, it became easier for Sniffer authors to add new analyses to Sniffer, since PEAR handles most of the compiler configuration and allows plugging local and global analyses while keeping the rest of the infrastructure intact. Finally, since PEAR marks approximated dynamic dispatch as such, Sniffer could provide additional information to its users to help them figure out if something is a potential false positive. Upon encountering a node marked as generated due to dynamic dispatch approximation, Sniffer surfaces this information to the user.

PEAR-enabled Sniffer comprises 1.4k sLoC of Rust among its four analyses: unsafety-freedom, encapsulation-breaking, presence of interior mutability, and panic-freedom. This suggests an approximate 350 sLoC per analysis, which we consider modest. In comparison, the original Sniffer comprises 4.2k sLoC of Rust, including the collection infrastructure from Scrutinizer. Porting Sniffer to PEAR took 15 person-hours for an experienced compiler plugin developer. Most of the time was spent on figuring out how the existing analysis mapped to global and local analyses of PEAR.

7 Discussion

We discuss the limitations of PEAR we encountered while using it with Sesame and Sniffer.

7.1 Compiler Versions

The Rust compiler lacks API stability guarantees across different compiler versions. As a result, each compiler plugin must be pinned to a specific compiler version, which plugin users must use to run it. This limitation is not specific to PEAR, but characteristic of the overall Rust compiler plugin ecosystem.

PEAR's nature as a compiler plugin limits the usability of PEAR in multiple ways. First, PEAR clients must ensure that the crates they are using PEAR with compile with the compiler plugin version PEAR chose. From our experience, most crates support Rust compiler versions dating one to two years back. Second, using PEAR with sibling tools to develop richer static analyses requires ensuring that the compiler version used by a sibling tool matches the compiler version used by PEAR. In practice, this means that the client of PEAR will need to vendor modified versions of sibling tools to ensure version compatibility, which is what we had to do with Flowistry [CPA+22] for supporting analysis in Sesame. Third, migrating between compiler versions presents its own set of challenges. For instance, changes to the compiler do not take into account the external users and might break the invariants systems like PEAR depend on. Without a strong test suite, detecting semantic changes that do not change the externally visible APIs becomes hard, as the Rust compiler is under active development.

Stable MIR [Sta] addresses some of these problems by providing alternative MIR APIs stable across compiler versions. However, opting into Stable MIR requires significant engineering effort, since some of the APIs are different from the original compiler APIs. Furthermore, Stable MIR is a strict subset of the optimized MIR in terms of available information. Hence, some of the information used by static analysis tools, like borrow checker information for Scrutinizer or unsafety information for Sniffer, is no longer available for them in Stable MIR. Such static analyses would be forced to obtain the information unavailable in Stable MIR via original compiler APIs, which brings back a part of the original problem.

7.2 Differences in MIR Optimization Levels

PEAR constructs MCGs using an optimized version of MIR for efficiency reasons. However, some static analysis tools for Rust operate over different MIR versions.

For example, when porting Sesame to PEAR, we encountered a problem where Flowistry performed its analysis over MIR obtained after borrow checking. It then computed variable-granularity data and control flow in terms of objects in that MIR version, which

did not directly map to the objects in the optimized MIR. Solving this required a workaround of saving the version of MIR on which Flowistry operates for each of the relevant functions. We then converted constructs from that MIR version to the constructs from optimized MIR, which PEAR uses. With extra engineering effort, however, it is possible to develop a version of PEAR, which performs its analysis over MIR at different optimization levels.

7.3 MCG Anti-Patterns

We observed one pattern in Rust that results in particularly low precision for MCGs. The reachability analysis of PEAR disambiguates function-pointer-like constructs exclusively by their signatures. While sound, creating many function pointers with the same signature results in PEAR assuming that they are interchangeable. Specifically, PEAR assumes that whenever a function behind one of the function pointers is called, any of the function pointers with a given signature can be called. This pattern makes PEAR include many nodes in MCGs that represent calls that would never occur in reality.

In general, this situation is quite rare, but a possible way to construct this pattern is by having a widely used trait whose implementation creates a function pointer under the hood for each of the implementations. The formatting infrastructure from the Rust standard library exhibits exactly this pattern. The default implementation of the `Display` trait converts any formatter for the implementer type into a function pointer, with the same function signature for each one of them.

Cases like this could be disambiguated by keeping track of the calling contexts in which such calls occur to disambiguate them further.

8 Conclusion

PEAR is a framework for writing static analyses that fills in the gap in the lack of tooling for interprocedural analyses in Rust. PEAR's core abstraction is a new Rust program representation, the monomorphized call graph (MCG). Our evaluation shows that PEAR constructs MCGs correctly by soundly resolving dynamic dispatch, has good specificity to support the needs of real-world static analysis systems, and requires modest developer effort. While some future work is needed to improve the usability of the system with other projects from the Rust static analysis ecosystem, the current prototype brings us one step closer to practical, widely-available interprocedural analyses of Rust programs.

References

[ABF+22] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. "The Prusti Project: Formal Verification for Rust". In: *NASA Formal Methods*. Edited by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Springer International Publishing, 2022, pages 88–108 (cited on pages 6, 7, 11).

[AZS+23] Justus Adam, Livia Zhu, Malte Schwarzkopf, Will Crichton, and Carolyn Zech. *Paralegal*. 2023. URL: `https://github.com/brownsys/paralegal` (cited on pages 6, 8).

[BKA+18] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. "Putting the "Micro" Back in Microservice". In: *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC)*. Boston, MA, July 2018, pages 645–650 (cited on page 6).

[BKA+21] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. "Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pages 84–99 (cited on pages 6, 10).

[BS96] David F. Bacon and Peter F. Sweeney. "Fast static analysis of C++ virtual function calls". In: *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '96. San Jose, California, USA: Association for Computing Machinery, 1996, pages 324–341 (cited on pages 8, 11).

[CAG+23] Will Crichton, Justus Adam, Gavin Gray, and Michael Coblenz. *rustc_plugin*. 2023. URL: `https://github.com/cognitive-engineering-lab/rustc_plugin` (cited on page 10).

[Car15] Cargo Geiger Developer Team. *Cargo Geiger*. 2015. URL: `https://github.com/geiger-rs/cargo-geiger` (cited on pages 6, 29).

[CPA+22] Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. "Modular Information Flow through Ownership". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. San Diego, California, USA, 2022, pages 1–14 (cited on pages 6, 31).

[DAA+24] Kinan Dak Albab, Artem Agvanian, Allen Aby, Corinn Tiffany, Alexander Portland, Sarah Ridley, and Malte Schwarzkopf. "Sesame: Practical End-to-End Privacy Compliance with Policy Containers and Privacy Regions". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. SOSP '24. Austin, TX, USA: Association for Computing Machinery, 2024, pages 709–725 (cited on pages 6–8, 10, 12, 23, 25).

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. "Optimization of object-oriented programs using static class hierarchy analysis". In: *Proceedings of the ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995 9*. Springer. 1995, pages 77–101 (cited on page 11).

[GC23] Gavin Gray and Will Crichton. *Debugging Trait Errors as Logic Programs*. 2023. arXiv: `2309.05137 [cs.PL]` (cited on page 6).

[Kan21] Kani Developer Team. *Kani Rust Verifier*. 2021. URL: `https://github.com/model-checking/kani` (cited on page 25).

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '04. Palo Alto, California, 2004, page 75 (cited on page 10).

[LHC+23] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. "Verus: Verifying rust programs using linear ghost types". In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA1 (2023), pages 286–315 (cited on page 6).

[LHG+24] Wei Li, Dongjie He, Yujiang Gui, Wenguang Chen, and Jingling Xue. "A Context-Sensitive Pointer Analysis Framework for Rust and Its Application to Call Graph Construction". In: *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. CC 2024. Edinburgh, United Kingdom, 2024, pages 60–72 (cited on page 11).

[LTB+24a] Ada Lamba, Max Taylor, Vincent Beardsley, Jacob Bambeck, Michael D. Bond, and Zhiqiang Lin. "Cocoon: Static Information Flow Control in Rust". In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1 (Apr. 2024) (cited on page 6).

[LTB+24b] Hayley LeBlanc, Nathan Taylor, James Bornholt, and Vijay Chidambaram. "SquirrelFS: using the Rust compiler to check file-system crash consistency". In: *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA, July 2024, pages 387–404 (cited on page 6).

[LWS+21] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. "MirChecker: Detecting Bugs in Rust Programs via Static Analysis". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pages 2183–2196 (cited on pages 6, 10).

[Mig07] Matthew Might. "Environment Analysis of Higher-Order Languages". PhD thesis. Georgia Institute of Technology, June 2007 (cited on page 12).

[MQL+24] Haoran Ma, Yifan Qiao, Shi Liu, Shan Yu, Yuanjiang Ni, Qingda Lu, Jiesheng Wu, Yiying Zhang, Miryung Kim, and Harry Xu. "DRust: Language-Guided

Distributed Shared Memory with Fine Granularity, Full Transparency, and Ultra Efficiency". In: *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA, July 2024, pages 97–115 (cited on page 6).

[MWD+23] Molly MacLaren, Ruochen Wang, William Duan, Serena Chen, Michael Coblenz, and Jewelle Elizabeth Tatad. *Situationally Adaptive Language Tutor (SALT)*. 2023. URL: `https://github.com/mojeanmac/vscode-salt` (cited on page 6).

[NG21] Emil Jørgensen Njor and Hilmar Gústafsson. "Static Taint Analysis in Rust". Master's thesis. Aalborg University, 2021 (cited on page 11).

[NHD+20] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. "RedLeaf: Isolation and Communication in a Safe Operating System". In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. Nov. 2020, pages 21–39 (cited on page 6).

[PHJ+16] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. "NetBricks: Taking the V out of NFV". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA, Nov. 2016, pages 203–216 (cited on page 6).

[Rusa] Rust Compiler Developers. *MIR dataflow - Rust Compiler Development Guide*. URL: `https://rustc-dev-guide.rust-lang.org/mir/dataflow.html` (cited on page 6).

[Rusb] Rust Compiler Developers. *MIR visitor and traversal - Rust Compiler Development Guide*. URL: `https://rustc-dev-guide.rust-lang.org/mir/visitor.html` (cited on page 6).

[Rusc] Rust Compiler Developers. *rustc_driver and rustc_interface - Rust Compiler Development Guide*. URL: `https://rustc-dev-guide.rust-lang.org/rustc-driver/intro.html` (cited on pages 6, 10).

[Rusd] Rust Compiler Developers. *rustc_monomorphize::collector - Rust*. URL: `https://doc.rust-lang.org/1.77.0/nightly-rustc/rustc_monomorphize/collector/` (cited on page 17).

[Ruse] Rust Compiler Developers. *The borrow checker - Rust Compiler Development Guide*. URL: `https://rustc-dev-guide.rust-lang.org/borrow_check.html` (cited on page 6).

[Rusf] Rust Compiler Developers. *Unsafety Checking - Rust Compiler Development Guide*. URL: `https://rustc-dev-guide.rust-lang.org/unsafety-checking.html` (cited on page 6).

[Shi91] Olin Grigsby Shivers. "Control-flow analysis of higher-order languages or taming lambda". PhD thesis. Carnegie Mellon University, 1991 (cited on page 11).

[Sri92] Amitabh Srivastava. “Unreachable procedures in object-oriented programming”. In: *ACM Lett. Program. Lang. Syst.* 1.4 (Dec. 1992), pages 355–364 (cited on page 11).

[Sta] Stable MIR Librarification Project Group. *Stable MIR Librarification Project*. URL: `https://rust-lang.github.io/project-stable-mir` (cited on pages 10, 31).

[TAS+25] Corinn Tiffany, Artem Agvanian, Ziyun Song, Kinan Dak Albab, Will Crichton, Philip Levis, Akshay Narayan, Deepti Raghavan, and Malte Schwarzkopf. “Rust Isn’t a Silver Bullet for Systems Research (Yet)”. (Preprint). 2025 (cited on pages 8, 23, 25).

[TP00] Frank Tip and Jens Palsberg. “Scalable propagation-based call graph construction algorithms”. In: *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’00. Minneapolis, Minnesota, USA: Association for Computing Machinery, 2000, pages 281–293 (cited on page 11).

[VSC+22] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. “Verifying Dynamic Trait Objects in Rust”. In: *Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2022, pages 321–330 (cited on pages 6, 7, 11, 25, 26).

[ZTJ+24] Lydia Zoghbi, David Thien, Ranjit Jhala, Deian Stefan, and Caleb Stanford. “Auditing Rust Crates Effectively”. (Preprint). 2024 (cited on pages 6, 29).