Verifying Dynamic Trait Objects in Rust

Alexa VanHattum
avh@cs.cornell.edu
Cornell University, Amazon
Ithaca, USA

Daniel Schwartz-Narbonne
dsn@amazon.com
Amazon
New York City, USA

Nathan Chong
ncchong@amazon.com
Amazon
Boston, USA

Adrian Sampson
asampson@cs.cornell.edu
Cornell University
Ithaca, USA

ABSTRACT

Rust has risen in prominence as a systems programming language in large part due to its focus on reliability. The language's advanced type system and borrow checker eliminate certain classes of memory safety violations. But for critical pieces of code, teams need assurance beyond what the type checker alone can provide. Verification tools for Rust can check other properties, from memory faults in unsafe Rust code to user-defined correctness assertions. This paper particularly focuses on the challenges in reasoning about Rust's *dynamic trait objects*, a feature that provides dynamic dispatch for function abstractions. While the explicit `dyn` keyword that denotes dynamic dispatch is used in 37% of the 500 most-downloaded Rust libraries (crates), dynamic dispatch is implicitly linked into 70%. To our knowledge, our open-source Kani Rust Verifier is the first symbolic modeling checking tool for Rust that can verify correctness while supporting the breadth of dynamic trait objects, including dynamically dispatched closures. We show how our system uses semantic trait information from Rust's Mid-level Intermediate Representation (an advantage over targeting a language-agnostic level such as LLVM) to improve verification performance by 5%–15× for examples from open-source virtualization software. Finally, we share an open-source suite of verification test cases for dynamic trait objects.

CCS CONCEPTS

• **Software and its engineering** → *Formal software verification*; • **General and reference** → **Verification**; • **Theory of computation** → **Verification by model checking**.

KEYWORDS

Rust, verification, model checking, dynamic dispatch

ACM Reference Format:
Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. 2022. Verifying Dynamic Trait Objects in Rust. In *44nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3510457.3513031

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE-SEIP '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9226-6/22/05.
https://doi.org/10.1145/3510457.3513031

1 INTRODUCTION

Rust has made significant inroads as a popular *safe* systems programming language over the decade since its release. Stack Overflow has named Rust the "most loved language" every year since 2016.[1] One of the language's main selling points is its focus on reliability—the ownership type system is a success story of programming language memory safety research breaking into the mainstream. The borrow checker eliminates certain high-impact classes of bugs, including null pointer dereferences, use-after-frees, and most forms of leaked memory. A team writing safety- or security-critical code, though, may seek an even higher level of assurance than what the current type system alone provides.

While Rust's type system rules out most memory safety bugs in checked *safe* code, there remain many ways for execution to go wrong. The language provides an "unsafe" dialect that allows programmers to bypass restrictions to regain more expressivity for lower-level regions of code. Even in safe Rust regions, the type system does not rule out dynamic panics from out-of-bounds or indexing errors (for example, consider the well-type-checked snippet `let v = vec![1, 2]; v[3]`). Finally, engineers may want assurance of *functional correctness*—the ability to assert specific properties about the result of a program under all possible inputs.

We are building an open-source tool, the Kani Rust Verifier (Kani), for sound, bit-precise symbolic analysis of Rust programs—initially motivated by use cases at Amazon Web Services (AWS). In our previous work on symbolic correctness proofs of production C code, we found that (1) embedding specification into proof harnesses similar to unit tests, and (2) integration with existing developer workflows were key to broad impact on software engineering teams [5]. To this end, one of our primary goals with the Kani project is to support enough of the Rust language surface to seamlessly integrate into large, existing projects. In Section 4.2, we show how Kani performs on components of the open-source Firecracker virtual machine monitor,[2] which provides virtualization for two publicly-available serverless compute services at Amazon Web Services: Lambda and Fargate [1].

We have found an unexpected challenge in Rust language coverage to be correctly modeling dynamic dispatch through virtual

[1] https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/
[2] https://firecracker-microvm.github.io/

method tables. Rust does not have classes or class inheritance like other object-oriented languages; rather, *traits* are the primary mechanism for defining interfaces and abstracting over implementations. The official Rust blog states:[3]

> The trait system is the secret sauce that gives Rust the ergonomic, expressive feel of high-level languages while retaining low-level control over code execution and data representation.

By default, trait method calls are *monomorphized*—that is, the compiler statically resolves which concrete function to call at each function call site (see Section 2.1).

However, users can add a `dyn` keyword to gain the expressivity of dynamic dispatch to trade-off dynamic runtime for improved code size and compilation times (see Section 2.2). Further, Rust's closures, or anonymous functions, can also be dynamically dispatched through trait objects. As we show in Section 4.1, 37% of the 500 most popular Rust crates (packages) explicitly invoke dynamic dispatch in their source code, and 70% implicitly include code that uses it.

Despite Rust's minimal runtime, supporting these dynamic trait objects is challenging because (1) they require non-trivial dynamic dispatch semantics that are not explicitly specified in any Rust documentation, and (2) they require heavy use of function pointers, which can be challenging for static analysis and symbolic execution algorithms [16, 18]. While dynamic trait objects are easy to avoid in hand-crafted verification examples, their use in the Rust standard library and throughout realistic, real-world crates motivates providing full support within our Kani tool. We have also seen in practice that faithfully modeling dynamic trait semantics causes our verification times to become intractable due to the large number of function pointers. In Section 3.3, we show how Kani leverages semantic information about traits to restrict the number of possible targets for function pointers, moving a Firecracker proof from intractable to completing successfully in 16 minutes.

Verification for Rust is a growing field, but to the best of our knowledge, Kani is the only symbolic model checking tool that targets Rust's Mid-level Intermediate Representation (MIR) and can reason about dynamic trait objects and dynamic closures. Other verification tools that target MIR either do not provide soundness guarantees over symbolic inputs (MIRI [15], MIRAI [9]) or do not support all cases of dynamic traits (Prusti [2], CRUST [23], Crux-MIR [10]); other tools target LLVM-IR and thus do not leverage MIR-level type information (SMACK [3], SeaHorn [11], RVT-KLEE [21]).

Kani is implemented as a backend for the Rust compiler that uses a mature, industrial-strength model checking tool—the C Bounded Model Checker (CBMC) [6]—as a verification engine. Kani translates Rust's Mid-level Intermediate Representation (MIR) into *Goto-C*, CBMC's C-like intermediate representation. Specifications in Kani are written as Rust-source-level `assert!(...)` statements, with simple extensions to specify assumptions and nondeterministic symbolic input (Section 3.1). Kani can be invoked on individual Rust files or on crates with the `Cargo` Rust build tool. In addition to the user-added assertions, Kani checks for arithmetic overflow, out-of-bounds memory accesses, and invalid pointers. CBMC performs bounded unrolling of loops and recursion in the program, but Kani by default is run with assertions that guarantee that if code is verified, loops are sufficiently unrolled (via an assertion that any iterations beyond the unrolling bounds are unreachable).

[3]https://blog.rust-lang.org/2015/05/11/traits.html

In this paper, we identify dynamic trait objects as an essential language feature for Rust verification tools to tackle in order to enable use on large, real-world Rust projects. Our contributions are as follows:

(1) We describe the Kani Rust Verifier, an open-source bit-precise symbolic model checker for Rust programs. We show that covering dynamic trait objects semantics is necessary to reason about real-world Rust, and we identify nuanced interactions between dynamic dispatch and the Rust borrow checker that must be correctly modeled by tools that target Rust's Mid-level Intermediate Representation (MIR).
(2) We show how Kani uses MIR-level semantic information about traits to restrict possible targets for function pointers, which pose a well-known performance challenge for symbolic execution tools.
(3) We provide a case study on the open-source Firecracker repository that shows that function pointer restrictions unlock a previously intractable proof, with verification performance (under 20 minutes) suitable for use in continuous integration.
(4) We share an open-source suite of verification test cases for dynamic trait objects and compare the results of several related tools.

2 RUST TRAIT OVERVIEW

Traits are a core Rust language feature for specifying when types should share a common interface. By default, Rust uses a monomorphization process to concretize each possible method implementation with a specific type. But, programmers can instead opt-in to dynamic dispatch when they use a trait to trade-off runtime performance with improved code size and compilation times.

2.1 Traits and Monomorphization

To understand dynamic dispatch, we first describe Rust's default static dispatch techniques for trait objects. We start with a motivating example which defines an interface for objects that have an integer count method:

```rust
trait Countable { fn count(&self) -> usize; }
```

We can implement this trait for two data structures, the Rust standard library's `Vec` and our own custom `Bucket` struct:

```rust
impl Countable for Vec<i8> {
  fn count(&self) -> usize { self.len() }
}
impl Countable for Bucket {
  fn count(&self) -> usize { self.item_count }
}
```

Now, we can use the `Countable` type to refer to any object that implements the trait:

```rust
fn print_count<T: Countable>(obj: T) {
  print!("Count = {}", obj.count());
}
```

This implementation specifies that the function takes a generic type `T` that must implement the `Countable` trait. Lower-level languages

like assembly do not, of course, support generics. How, then, does the compiler resolve line 2 of `print_count` into an actual function call jump (that is, which `count` implementation should be called)?

By default, the Rust compiler uses *monomorphization:* it creates a specialized `print_count` function for each concrete type. This process happens at the MIR level, but the effect is roughly equivalent to this Rust source code:

```
fn print_count_vec_i8(obj: Vec<i8>) {
  print!("Count = {}", obj.count::<Vec<i8>>());
}
fn print_count_bucket(obj: Bucket) {
  print!("Count = {}", obj.count::<Bucket>());
}
```

Monomorphization means that every function that uses a generic type bound must be duplicated for every possible implementation.

2.1.1 Closures as dynamic trait objects.

Closures are anonymous functions that can capture (and if specified, mutate) values in the environment where they are defined. Each closure has its own unique concrete type (that is, even closures that share the same signature do not share a concrete type.) This creates a difficulty: what type should be used when a closure is passed into a higher-order function, such as `map`? Rust solves this using traits: all closures must implement at least one of three standard-library-defined traits: `FnOnce`, `FnMut`, or `Fn`, depending on whether they consume, mutably reference, or immutably reference the captured environment (see Section 3.2.4).

For example, we could define a function that takes in an item cost and a closure to calculate the price of that item with tax:

```
fn price<T: Fn(f32)->f32>(cost: f32, with_tax: T)
  -> f32 {  with_tax(cost) }
```

To call this function, we simply specify the closures we want as the second argument:

```
let tax_rate = 1.1;
price(5., |a| a * tax_rate); // Price is: 5.5
price(5., |a| a + 2.);       // Price is: 7
```

Rust will monomorphize the code at compile time to call the right implementation (we use `[closure@...]` to represent the closure environment, which stores the `tax_rate` in the first closure and is empty in the second):

```
fn see_price_closure@main:1(cost: f32) -> f32 {
  closure@main:1([closure@main:1], cost)
}
fn see_price_closure@main:2(cost: f32) -> f32 ...
```

2.1.2 The costs of traits.

With this monomorphization strategy, developers pay no run-time efficiency cost compared to code that manually specifies each implementation without using generics or abstraction. However, monomorphization can have undesirable effects: an increase in code size and compilation time, especially as the number of possible implementations grows.

Verification tools can often avoid reasoning about monomorphization by consuming Rust code *after* monomorphization completes, either by running MIR's default monomorphizer or by targeting a lower-level of code in compilation, such as LLVM IR. From the perspective of a verification tool, it is feasible to handle Rust code with statically dispatched trait objects by instead using only the monomorphized, concrete functions.

2.2 Dynamic Trait Objects

To trade-off runtime efficiency with improved code side and compilation time, developers can use *dynamic trait objects* to opt in to dynamic dispatch (and out of monomorphization). For example, using our same `Countable` trait, a developer could have this alternative implementation of `print_count`:

```
fn print_count(obj: &dyn Countable) {
  print!("Count = {}", obj.count());
}
```

To pass a trait object to this function, developers need to cast it as a dynamic trait object:

```
print_count(&Bucket::new(1) as &dyn Countable);
```

Here, the `dyn` keyword expresses that this object should have method calls dynamically dispatched. That is, the Rust compiler will use a different strategy to answer the question: "which implementation should `obj.count()` call?"

Rather than creating a new function signature per concrete type for `print_count`, the Rust compiler will use a single instance of `print_count` that takes a single type that can represent all objects that implement `Countable`. In Rust, this type is an instance of a *fat pointer*—a double-wide pointer type that represents both data and essential metadata. Fat pointers for dynamic trait objects consist of a `data` pointer to the object itself and a pointer to the *virtual method table* (vtable) [7] that maps trait-defined methods to their implementations.

2.2.1 Rust's implementation using vtables.

Rust Mid-level Intermediate Representation (MIR) uses abstract trait types, so it is up to each backend to implement vtables as they lower to their corresponding lower-level representation. Because vtables require jumps to a dynamically computed address, they can potentially be exploited in security attacks (e.g., in C++ [13]), and hence their precise implementation has security implications. Although Rust's informal specification does not specify the exact vtable layout, MIR provides utility functions for building vtables of a specific form. When we lack documented semantics for how Rust treats dynamic trait objects, we use the canonical LLVM backend as a reference. Our descriptions are based on Rust 1.55.0, the latest version of the compiler at the time of writing.

In the canonical LLVM backend for the Rust compiler, vtables have a specific layout that contains object metadata (the size and alignment of the data) as well as pointers for each method implementation. Every vtable includes a pointer to the concrete type's `drop` (destructor) method implementation. The remainder of the vtable contains pointers to the concrete implementation of all methods defined by that trait. A new vtable is defined at compile time for every cast statement between a unique pair of concrete object type and trait type, and stored in a new global variable. Dynamic trait objects that share the same concrete type can thus share the same vtable.

The vtable for our `Countable` example is (conceptually):

sizeof<Bucket>	8
align<Bucket>	8
&Bucket::drop	0x7ffe02d0ba88
&Bucket::count	0x7ffe02d0ba90

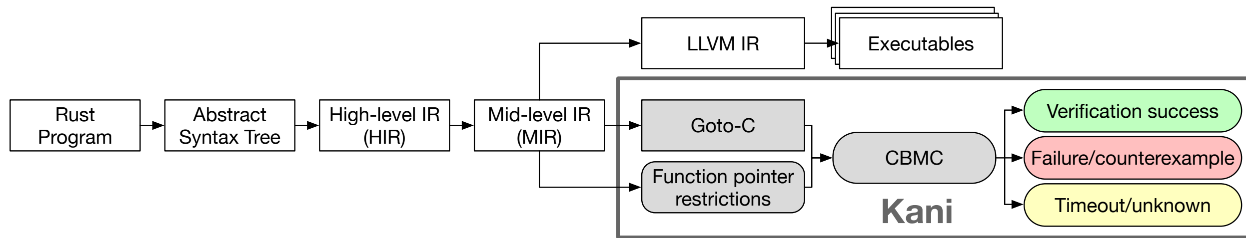

Figure 1: Architecture of Kani. Unfilled indicates the Rust compiler with the canonical LLVM backend; filled indicates Kani. *IR* is Intermediate Representation. Kani translates Rust's MIR to CBMC's C-like Goto-C language, uses MIR type information to emit function pointer restrictions, and outputs a successful verification, a failure with a counterexample trace, or a timeout.

The fat pointer for our `&Bucket as &dyn Countable` object would have one pointer to the `Bucket` and one pointer to the vtable above. Calls to methods that take `self` then can pass the `data` pointer as self. For example, `print_count` would be implemented as roughly:

```
1  fn print_count(obj: &dyn Countable) {
2    print!("Count = {}",
3           *(obj.vtable.count)(obj.vtable.data));
4  }
```

For dynamic closures, the `data` half of the fat pointer points to the closure's environment. The vtable consists of the same `size`, `align`, and `drop` metadata, then pointers to functions defined for `Fn`, `FnMut`, and/or `FnOnce`.

2.2.2 Summary. Dynamic trait objects allow developers to compile code with dynamic objects that carry metadata specifying which trait implementations of methods to call, rather than statically duplicating code through monomorphization. Dynamic trait objects are used throughout the Rust standard library, so even if programmers do not opt-in to dynamic dispatch within their own source code, they are likely to pull in Rust source that constructs and uses vtables (see Section 4.1). Rust's dynamic dispatch poses a challenge for verification both because *how* to implement them is not precisely specified by the Rust language definition, and because function pointers require pointer analyses that are a known challenge for symbolic reasoning [16, 18].

3 METHODOLOGY & IMPLEMENTATION

3.1 The Kani Rust Verifier

Architecturally, Kani is implemented as code generation backend to the `rustc` compiler (Figure 1).[4] Instead of translating to machine-code (e.g., via the LLVM compiler infrastructures for the standard backend or Cranelift for an experimental debug backend), Kani translates to `Goto-C`, the C-like intermediate representation for CBMC [6]. Kani then invokes CBMC on the generated `goto` program, which ultimately runs symbolic execution and discharges formulas to an off-the-shelf SAT or SMT solver (by default, MiniSAT [8]).

3.1.1 Properties checked and soundness. Kani by default checks for memory safety (pointer type safety, invalid pointer indexing in unsafe code, slice/vector out-of-bounds), arithmetic overflow, runtime panics, and violations of user-added assertions. Users can additionally specify `assert!` and `kani::assume` statements using Rust syntax. To reason about all possible inputs, users specify variables as non-deterministic symbolic inputs using a special generic `kani::any<T>()` function. We use *sound* to indicate that Kani never misses violations of the checked properties in the `rustc`-produced binary execution on some input. We have made the conscious choice when developing Kani to prioritize soundness over completeness, so Kani fails prior to verification if it encounters a Rust language feature it does not yet support. Kani currently focuses on sequential Rust and thus fails on any concurrency constructs. Kani also fails on some compiler intrinsics, including a subset of SIMD (vector single instruction, multiple data) operations.

While CBMC can act as a bounded model checker, Kani uses it for unbounded verification. By default, CBMC is bounded because it requires either a heuristic or a user-specified unrolling bound to unroll each loop and set of recursive function calls. When symbolic execution reaches the specified bound, CBMC defaults to inserting an `assume(false)`, which stops further exploration of the execution. However, CBMC provides an `unwinding-assertions` flag that asserts that any loop iteration beyond the specified bounds is unreachable. Kani enables this flag—this causes us to be potentially incomplete on programs where the bounds cannot be specified, but provides an assurance of soundness for all cases where Kani returns "SUCCESS".

3.1.2 Choice of input representation. The Rust compiler translates a Rust program between a series of increasingly low-level representations, as shown in Figure 1. One of the key architectural choices when designing a Rust verifier is what level of representation the verification tool should take as input. Each level has both advantages and disadvantages for verification. On the one hand, each step lower in the representation tends to use a smaller set of more uniform constructs. Defining a formal semantics is therefore easier at lower levels. Tools such as SMACK operate at the LLVM intermediate representation (LLVM-IR) level, which has the additional benefit of allowing a shared verification backend between different languages, such as C and C++.

On the other hand, lower-level representations lose information about the original structure of the program and hence about the original intent of the programmer. For example, the compiler may give implementation-defined semantics in a lower-level representation to an operation that is undefined behavior at a higher level. We have found that the Rust Mid-level Intermediate Representation (MIR) to be an effective interface for verification. MIR is a (fairly)

[4] https://rustc-dev-guide.rust-lang.org/backend/backend-agnostic.html

clean and compact representation that retains most of the semantic Rust type information. Kani invokes monomorphization before analysis takes place, so we do not explicitly need to reason about generic constructs. As we demonstrate in Section 4.2, MIR's rich type information is crucial to enabling high-performance verification of dynamic trait objects.

3.1.3 The need for bit-precision. `unsafe` Rust code can both read and modify objects as a collection of raw bytes, bypassing the borrow checker and type system. For example, Rust code can use `transmute` to reinterpret bytes of one type as bytes of another or use raw pointer indexing to directly view and modify the bytes of a type. These features are used for performance and portability benefits in production Rust code, including in the Rust standard library.

For example, the standard library's `OsStr` implementation notes:

```
1  /* FIXME: `OsStr::from_inner` current
2  implementation relies on `OsStr` being
3  layout-compatible with `Slice`... */
4  pub struct OsStr { inner: Slice, }
```

In order to verify such code, it is necessary that the bitwise layouts used by Kani match those used by the Rust compiler itself. While relying on implementation details like this is undefined behavior for source-level Rust code, the standard library is able to rely on stronger implementation-level guarantees from the Rust compiler. Kani's CBMC backend provides the bit-level reasoning necessary to handle such cases.

3.2 Dynamic Trait Objects in Kani

Goto-C (and C) do not have native support for method dispatch, so Kani must lower MIR to C in a manner that removes traits but maintains the same semantics. Our primary strategy is to follow the LLVM backend's vtable implementation, emitting Goto-C instead of LLVM IR.

3.2.1 vtable construction. Dynamic objects are created at cast sites, where a concrete type is cast to a `dyn` type explicitly or implicitly. Like the LLVM code generation backend, Kani keeps a cache of vtables that constructs a new vtable for every unique concrete object, trait type tuple. Vtables generated by Kani are Goto-C structs that map the metadata identifier to the corresponding data.

Naming vtables fields was less straightforward than we anticipated. In the LLVM code generation backend, vtables are global allocations *without* named fields (rather, each individual element is accessed through pointer arithmetic). To keep our generated Goto-C code more debuggable by Kani developers (and counterexample traces more readable for users), we opted to use a struct with named fields (because each field is the size of one pointer, the memory layout is the same). An earlier version of Kani mapped the method name to the method implementation function pointer. However, we found this failed to handle cases where an object implemented two traits with the same method name.

Unlike some other languages, Rust allows a type to implement two traits with identically-named methods (regardless of whether their signature is the same):

```
1  trait A { fn is_odd(&self) -> i32; }
2  trait B { fn is_odd(&self) -> bool; }
3  impl A for i32 { ... };
4  impl B for i32 { ... }
5  trait C : A + B {}
6  impl C for i32 {}
7  // The vtable for x has two 'is_odd' entries
8  let x: &dyn C = &3 as &dyn C;
```

To resolve this ambiguity, Kani now uses *the index of the item* in the vector returned from a Rust MIR API call—`vtable_entries`—to uniquely identify methods. We confirmed this strategy in informal public discussions with Rust compiler developers.[5]

Specifically, we create a new vtable when we see a cast from a sized pointer type to an `unsized` (non-slice) pointer type, where we have not already created a vtable for this concrete type, trait type pairing. At construction, we iterate over the Rust compiler's new (June 2021) `vtable_entries`[6] results. We construct `size` and `alignment` using the Rust compiler's API for layout and drop resolution.[7] For each method defined explicitly for that trait type, we add an entry indexed by position in the canonical `vtable_entries`.

3.2.2 Virtual calls through vtables. Dynamic dispatch occurs when a statement calls a method on a dynamic trait object.

At the MIR level, we construct a dynamic call through a vtable when we encounter a virtual call terminator. We obtain the object's self pointer and vtable pointer by accessing the respective components of the fat pointer. We use the index `idx` provided by the virtual call object to determine the vtable method—which corresponds with the index into the vector returned by the Rust compiler's `vtable_entries`.

3.2.3 Casts of dynamic objects. Rust does not currently support general dynamic trait *upcasting* (see Section 6): i.e., one cannot cast an object of type `&dyn Foo` to one of type `&dyn Bar` even if one is a subtype of the other (unlike, for example, Java subtyping). The underlying reason is that Rust prefers to stay as close to *zero cost abstractions*[8] as possible—giving users high level language features without sacrificing performance. Totally generic trait upcasting would require modifying or rebuilding vtables (or additional pointer indirection), imposing a runtime cost.

Kani initially encoded the assumption that dynamic trait objects could thus not be the *source* of cast statements. When we tested Kani on the standard library, Kani found violations of this assumption when handling types like `&dyn Error + Send`. Looking more into the Rust documentation for traits, we found:

> The `Send`, `Sync`, [...] and `RefUnwindSafe` traits are auto traits. *Auto traits have special properties.* [...]

Because auto traits like `Send` *have no associated methods*, the underlying vtable does not need to change when a cast involves only auto-traits. The Rust compiler therefore allows adding and removing auto traits in dynamic trait objects casts, breaking Kani's initial assumption. To reason about the Rust standard library as-is, verification tools must be able to handle this type of cast.

[5]https://rust-lang.zulipchat.com/#narrow/stream/144729-wg-traits/topic/.E2.9C.94.20object.20upcasting/near/246857652
[6]https://doc.rust-lang.org/nightly/nightly-rustc/rustc_trait_selection/traits/fn.vtable_entries.html
[7]https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/ty/layout/trait.LayoutOf.html,https://doc.rust-lang.org/beta/nightly-rustc/rustc_middle/ty/instance/struct.Instance.html#method.resolve_drop_in_place
[8]https://blog.rust-lang.org/2015/05/11/traits.html

3.2.4 Closure signatures. Our initial implementation of dynamic trait objects in Kani (which, like the current version, prioritized soundness over completeness) failed to verify due to a CBMC pointer error on the following input:

```
let f: Box<dyn FnOnce(i8)> = Box::new(|x| {
  assert!(x == 1);
});
f(1);
```

A nearly-identical version of this case with `Fn` replacing of `FnOnce` verified successfully. The root issue was a surprisingly subtle interaction between Rust's borrow checker and dynamic dispatch.

The Rust documentation includes the following:

> Use `FnOnce` as a bound when you want to accept a parameter of function-like type and only need to call it once. If you need to call the parameter repeatedly, use `FnMut` as a bound; if you also need it to not mutate state, use `Fn`.

FnOnce thus has provides a method signature that *moves* ownership of its self type by taking it by-value: `fn call_once(self, args: Args)-> Self::Output;` This allows the Rust borrow checker to give errors on attempted reuse such as:

> `'f' moved due to this call. This value implements 'FnOnce', which causes it to be moved when called`

In comparison, *Fn* has this method signature: `fn call(&self, args : Args)-> Self::Output;`. Both `Fn` and `FnOnce` are used for dynamic dispatch via vtable calls, using the `self` parameter as one argument. This is the root cause of our verification failure—the machinery we have described for closures and vtables requires that the vtable's `self` argument be a *pointer* to an object.

Rust uses a *vtable shim* to work around this mismatch:

```
/*`<T as Trait>::method` where `method` receives
unsizeable `self: Self`...The generated shim
will take `Self` via `*mut Self` - conceptually
this is `&owned Self` - and dereference the
argument to call the original function. */
VtableShim(DefId),
```

However, the full translation is not complete at the MIR level: before code generation, backends must be sure to correct the function call signature, for example from the Rust compiler:

```
if let InstanceDef::VtableShim(..) = self.def {
// Modify `fn(self, ...)` to `fn(self: *mut Self, ...)`
```

Backends can either disregard the MIR function signature and use a separate `fn_abi_of_instance`, or apply this same correction to the MIR function signature. Verification tools can reasonably make either choice—but using the MIR function signature alone in this case will lead to incorrect results.

3.3 Leveraging Trait Semantics for Function Pointer Restrictions

One of Kani's key advantages over more language-agnostic verification tools is that it can exploit Rust's semantics to improve verification completeness and performance. While other tools (i.e., SMACK [3], RVT-KLEE, RVT-SeaHorn [21]) that work at the level of LLVM IR must work with vtables as opaque allocations generated by the standard Rust backend, Kani can offer a more direct interpretation of dynamic dispatch that allows us to combat path explosion in the verification state space. At the MIR level, we have access to rich trait information; by the time Rust is lowered to LLVM, traits are gone and replaced by non-specific LLVM pointer types which are more difficult to reason about. Specifically, Kani uses information about dynamic trait object creation (at object cast sites) and use (at function call sites) to restrict the set of possible targets for vtable function pointer calls.

3.3.1 The verification challenge of dynamic dispatch. In general, indirect function calls pose a scalability challenge for program verification due to the pointer analysis involved [16, 18]. Before running symbolic execution, CBMC removes all function pointers by lowering them to conditional `if` blocks between possible target functions. By default, CBMC considers all functions in the code generation unit of the correct function signature to be possible targets (this is sound when CBMC is run with pointer checks, which verify that all pointers are to objects of the correct type.)

In a simple case, this permissive approach works well. Consider two functions with the same signature:

```
fn a(x: i32) -> bool { x == 2 }
fn b(x: i32) -> bool { x != 0 }
```

When a pointer to a function of this type is used, for example, `(*f)(2)` CBMC's algorithm conceptually emits the following:

```
if (f == &a)      a(2);
else if (f == &b) b(2);  // ...
```

CBMC can then use standard symbolic execution techniques for conditional control flow to soundly reason about this code.

This strategy becomes problematic when run on large code bases that pull in numerous dependent crates. *Every* dynamic trait object uses a function pointer every time a method is called, because each trait-defined method call is resolved through a vtable entry. The number of possible function pointer targets especially proliferates for calls to `drop`—the destructor function. Every object's `drop` function signature shares a shape: a method that takes a single `self` parameter and returns the unit type (analogous to a void return in C). When the `self` type is something from the standard library, such as `std::io:Error`, the number of possibilities skyrockets. In Section 4.2, we show how such a case can lead to hundreds of possible targets, rendering this approach to verification intractable.

3.3.2 Restricting call destinations using Rust semantics. We recognized that, with Kani's semantic understanding of traits at the MIR level, we have a much more precise notion of which implementations vtable function pointers could target. In particular, we can guarantee (short of the user using `unsafe` to transmute the vtable memory) that a call through a vtable will be one of the trait-defined methods for that trait type that we have encountered during code generation. To maintain soundness even under unsafe memory transmutes, we `assert!(false)` if the actual function pointer does not match one of our identified possibilities. This also allows us to soundly under-approximate possible targets by not explicitly accounting for casts between trait types, as described in Section 3.2.3.

We implement function pointer restrictions by tracking possible implementations (at object cast sites, when the vtable is built) and

uses of vtable pointers (at function call sites). For the first, we build a map that builds a set of possible implementations (in our case, symbol names for each Goto-C function) keyed by the tuple of trait type and method index for vtables of that trait type. For call sites, we build a list of `structs` with trait name, method index for vtables of that trait type, and information for identifying that call site in CBMC. Because dynamic dispatch calls can occur across crate boundaries, we emit a file with this information for each crate (using a stable unique hash for trait types). Finally, after code generation, we combine the per-crate data by iterating over the list of call sites and looking up the possible implementations we have found for that trait index tuple. When there are no possible implementations (possible in functions that are never invoked), we emit the empty set.

Kani combines the restrictions for each Rust dependency and the crate itself into an auxiliary JSON file to be consumed by CBMC as function pointer restrictions. We have seen CBMC times drop by an order of magnitude with this restriction strategy, as we describe in the next section.

4 EVALUATION

For our evaluation, we used an Amazon EC2 `m5d.4xlarge` instance with 16 cores and 64GB of memory, running Ubuntu 20.04.2.[9]

4.1 Prevalence of Dynamic Trait Objects

We conducted a simple empirical study to estimate the prevalence of dynamic trait objects in the 500 most downloaded crates on crates.io, the Rust package repository. We found that while only 185 of these 500 crates (37%) use the explicit `dyn` keyword within their source code, 349 (70%) include at least one vtable when compiled with rustc.

We downloaded the top 500 crates sorted by greatest number of downloads on October 2, 2021. To estimate the implicit use of dynamic trait objects, we invoked a debug build of the Rust compiler via `cargo build` and searched the debug output for the line `get_vtable`, which is logged at vtable use. This is likely an overestimate of the dynamic trait objects that are actually used in functionality a user might want to verify for these crates, but it does provide an indication of how often verification tools that integrate with Cargo will encounter linked dynamically-dispatched code.

4.2 Case Study: Firecracker

As a real-world case study, we consider how two different variants of Kani—one without vtable function pointer restrictions, and one with—perform on examples from the open-source Firecracker hypervisor. This case study highlights the challenges in moving from small, standalone verification examples to proofs that sit alongside large scale codebases.

Implemented in Rust, Firecracker provides the underlying virtualization technology for two publicly-available serverless compute services at Amazon Web Services: Lambda and Fargate [1]. A core characteristic of serverless computing is multitenancy, meaning that multiple customer workloads (e.g., functions or containers) may run on the same hardware. Consequently, Firecracker is crucial for ensuring the *isolation* of customer workloads.

[9] https://github.com/avanhatt/icse22ae-kani

```
1  pub trait BusDevice: AsAny + Send {
2    fn read(&mut self, offset: u64,
3            data: &mut [u8]);
4    fn write(&mut self, offset: u64, data: &[u8]);
5  }
```

Figure 2: The `BusDevice` trait used for explicit dynamic dispatch in Firecracker's serial device.

```
1  fn serial_harness() {
2    let mut serial = SerialDevice {
3      serial: Serial::new( ... ) };
4    // Model arbitrary input as symbolic
5    let bytes: [u8; 1] = kani::any();
6    let mut buf = [0x00; 1];
7    // Call functions-under-verification
8    <dyn BusDevice>::write(&mut serial, 0u64, &bytes);
9    <dyn BusDevice>::read(&mut serial, 0u64, &mut buf);
10   assert!(bytes[0] == buf[0]);
11 }
```

Figure 3: Our proof harness for simple `read`/`write` functionality. `kani::any()` is an Kani construct that returns a nondeterministic, symbolic value of the inferred type.

4.2.1 Firecracker Serial Device. Firecracker provides console emulation for a guest virtual machine by emulating a serial device (16550A UART). The guest virtual machine sends and receives bytes by writing and reading to device registers mapped into the guest memory. Since read and writing to a device through memory is a common interface, Firecracker defines a trait `BusDevice` which defines `write` and `read` methods (Figure 2). Multiple devices are wrapped in a `Bus` container which maps address ranges to a particular device and routes write and read requests as dynamic calls to the underlying device.

Verification task. We aim to demonstrate a simple proof harness using the serial device behavior in loopback mode, where bytes are read and written to the same port. Firecrack's serial device specifies that only a single byte can be read or written in a given call. Figure 3 shows a small proof harness that checks that for *any* single byte we can write through the dynamically-dispatched call, the same byte is read back. Kani checks this user-added assertion, as well as memory safety, arithmetic overflows and division by zero, and pointer safety.

Function pointer restriction. With our function pointer restrictions enabled, Kani identifies exactly the correct function pointer to call for both `read` and `write`. In Kani without function pointer restrictions, CBMC's default function pointer strategy finds 8 possible calls for each of `read` and `write`. For example, the call for `write` includes these two options, which are from an entirely different module of Firecracker but are included because they share the same function signature:

```
1  if(v.vtable->6 == Block_VirtioDevice_read_config)
2    goto __CPROVER_DUMP_L12;
3  if(v.vtable->6 == Block_VirtioDevice_write_config)
4    goto __CPROVER_DUMP_L12;
```

```
pub fn parse(
    avail_desc: &DescriptorChain,
    mem: &GuestMemoryMmap,
) -> result::Result<Request, Error> { ... }
```

Figure 4: parse, our function-under-verification.

For this simple illustrative case, Kani runs in 4 minutes and 4 seconds with the restrictions and 4 minutes and 13 seconds without, representing a modest 5% speedup (with a tradeoff in code size increasing from 1.14GB to 1.20GB due to the auxiliary restrictions files). In the next example, we show how implicit vtable calls can cause far worse performance differences.

4.2.2 Firecracker Block Device Parser. For our next function-under-verification, we consider the emulated block device available to guest Virtual Machines (VM)s for storage (i.e., reading and writing to disk). The device is visible to the guest VM through MMIO (memory-mapped IO), using the virtio API [22]. The guest VM allocates a set of `virtqueue` data structures in guest memory to support generic data transport between the guest and hypervisor. Each entry in a `virtqueue` is a descriptor: a pointer with metadata, such as length and read/write permissions, to a buffer in guest memory. Descriptors can be chained so that multiple buffers can be transported in a single transaction. For the block device, a read (respectively, write) transaction consists of three descriptors pointing to three buffers, containing (1) the request type and disk sector, (2) the data buffer to be filled/read, (3) a status byte returned by the device. The primary task of the emulated block device is to parse and execute guest transactions that it receives through this interface.

Verification task. Isolation between guest VMs requires that no input from a guest, no matter how malformed, can cause Firecracker to panic. Figure 5 gives a straightforward proof harness for the `parse` function of the block device (Figure 4). The parse function is responsible for taking the raw untrusted bytes of descriptors from guest memory and returning either a request object or an error. We use symbolic inputs (generated with `kani::any()` on line 3) to model input from the guest as well as to over-approximate data values read from the guest memory. Successfully verifying this proof harness using Kani shows that the block device has no runtime panics under any guest behavior.[10] Kani can be used to verify deeper functional properties, in addition to panic freedom—for the purpose of this case study, we note that even this simple harness is intractable without MIR-level type reasoning.

Although the code-under-verification never uses the `dyn` keyword to explicitly invoke a dynamic trait, the `parse` function returns the type `result::Result<Request, Error>`, where `Error` is a custom enum `devices::virtio::block::Error`. As shown in Figure 6, one enum value uses the standard library type `std::io::Error`, which is implemented using traits. When the returned object goes out of scope (when `block_proof_harness` returns), Rust automatically inserts a call to destruct the object with `std::ptr::drop_in_place`. This `drop_in_place` function uses a dynamic trait object of type `Drop`, which is routed through the object's vtable.

[10]Running this proof with the default set of Kani flags gives spurious pointer check errors (which we are investigating), so the results in this section are for Kani with pointer checks disabled.

```
fn block_proof_harness() {
  // Model arbitrary descriptor from guest as symbolic
  let desc : DescriptorChain = kani::any();
  // ..., call function-under-verification
  match parse(&desc, /*...*/) {
    Ok(req) => {},
    Err(_) => {},
  }
}
```

Figure 5: Our proof harness for parse.

```
pub enum devices::virtio::block::Error {
  // Guest gave us a descriptor that was too short
  DescriptorLengthTooSmall,
  // Getting a block's metadata fails for any reason
  GetFileMetadata(std::io::Error),
  // ...
}
```

Figure 6: The error type used in the Result returned by parse.

Impact on verification. Even in this simple case, the hidden use of dynamic trait objects poses a huge challenge for verification with CBMC—`std::io::Error` is so commonly used within Firecracker and its dependencies that CBMC identifies 314 possible function targets for this virtual call to `drop`. Each of these functions must then be unwound for symbolic execution. CBMC's symbolic execution engine was unable to complete this unwinding within a four hour timeout (and hence never even reached the stage of discharging the actual proof obligations to a satisfiability solver).

Our trait-based function pointer restrictions allow our proof harness for parse to terminate successfully in 16 minutes—at least a 15× improvement in verification performance. Code size again increases slightly, from 0.96GB for the proof harness without restrictions to 1.02GB for the successful proof. For the problematic call to `drop` on `std::io::Error`, Kani correctly identified that the `GetFileMetadata(std::io::Error)` type is never used in this harness or function-under-verification. That is, Kani emits 0 possible functions that could actually be the target of the precise `Error` type in this context, rather than the extremely permissive 314 possible options. Since Kani soundly replaces the call to `drop` with `assert(false)`, verification of the test case also serves as verification of the function-pointer restriction set. As an additional sanity check, we modified the function-under-verification to non-deterministically return a `std::io::Error` in some cases, which caused Kani to fail with spurious, false positive verification errors. Our manual inspection of these failures indicates that they do not affect soundness, but we are investigating them as a top priority.

4.3 Dynamic Dispatch Test Suite

In developing Kani, we have produced a suite of over 40 verification test cases for dynamic trait objects. This test suite has been open source throughout its development.[11] We encourage other developers of Rust verification tools to use and modify these test cases as they add more support for dynamic trait objects. Our full

[11]https://github.com/model-checking/kani

Tool		**Kani**	**Crux-MIR**	**RVT-SH**	**RVT-KLEE**	**SMACK**
Focus		Soundness	Soundness	Soundness	Bug-finding	Soundness
Test name	**Code snippet**					
Simple trait, pointer	&3 as &dyn T	✓	✓	✓	✓	✓
Simple trait, boxed	Box::new(o) as Box<dyn T>	✓	×	×	✓	×
Auto trait, pointer	&3 as &dyn Send	✓	×	×	✓	✓
Fn closure, pointer	$||{} as &dyn Fn	✓	×	✓	✓	✓
FnOnce closure, boxed	Box::new(||{}) as Box<FnOnce>	✓	×	×	✓	×
Generic trait, pointer	trait T: S<i8> + S<u8>	✓	×	✓	✓	✓
Explicit drop, boxed	impl Drop for T...Box<dyn T>	✓	×	✓	✓	×
Explicit drop, pointer	impl Drop for T...&dyn T	✓	✓	✓	✓	✓

Table 1: Dynamic trait object test cases, per tool (✓ is supported, × is unsupported). Kani is this work. Crux-MIR is an MIR-based "static simulator." RVT-SH (RVT-SeaHorn), RVT-KLEE, and SMACK all start with LLVM IR; respectively, they are a model checker, a bug finder, and a Boogie-based verifier. Prusti and CRUST did not support any form of dynamic trait object.

suite includes versions of the functions-under-verification that are expected to succeed and versions that are expected to fail.

Table 1 shows our understanding of other tools' support for a subset of test cases. We used the following versions:

- SMACK: `version 2.8.0`.
- Crux-MIR: commit hash `3451423`.
- Rust Verification Tools: commit hash `b179e90`.
- Prusti: `rustc 1.56.0-nightly (3d0774d0d 2021-08-18)`.
- Crust: no longer actively developed, the paper specified that dynamic traits were unsupported [23].

5 TRUSTED COMPUTE BASE & LIMITATIONS

Kani is designed as a sound verifier with respect to the properties checked, but because neither MIR nor Goto-C currently have formal semantics, the full Kani toolchain itself is not formally verified. Kani's trusted compute base includes our translation from MIR to Goto-C, CBMC itself, and the backend SAT or SMT solver.

Compared to some other Rust formal methods tools, Kani's use of Rust syntax for assertions and assumptions limits us to a smaller space of expressible properties. Supporting richer specifications—including support for first-class loop invariants, explicit existential quantifiers, and modular verification—is future work.

6 DISCUSSION & FUTURE WORK

In this paper, we have outlined how a language feature that is thought to be well-understood—dynamic dispatch—can pose unanticipated verification challenges. Prior efforts to formalize Rust semantics have (reasonably) focused on other unique language features, primarily the borrow checker. For example, the RustBelt[14] project's λ_{Rust} "omits some orthogonal features of Rust such as traits (which are akin to Haskell type classes)". The Oxide: Essence of Rust[24] paper similarly references Haskell type classes and does not see traits as an "essential part of Rust". We have a slightly different goal than this prior work: because we want to embed verification of Rust in real world codebases, we *needed* to wrangle with the semantics of trait objects, and we found that doing so was far from trivial. From this, we can argue a point broader than just Rust—verification tool designers should be prepared to model the complex and subtle ways *all* language features interact. This is especially true when languages provide a standard library that is not formally specified but uses the desired language feature.

Trait upcasting coercion. Rust has an in-progress proposal to add a *trait upcasting coercion* feature for dynamic trait objects.[12] This language feature would allow developers to cast between dynamic trait object types as long as the source type is a subtrait of the destination type. Implementing such coercions requires a more complicated vtable strategy, since they require the underlying vtable to change. Kani could be extended to support these trait coercions once they are enabled by default in Rust. To do so, we would need to extend our vtable generation and method lookup to model the Rust compiler semantics, which will likely be vtables with a nested structure that can require multiple pointer indirections. This would also be additional motivation to extend our existing strategy for restricting function pointers to include directed type cast information.

7 RELATED WORK

MIR-based verification. Other tools target Rust's Mid-level Intermediate Representation; but to our knowledge no other tool provides sound verification of symbolic inputs and supports the breadth of dynamic trait objects.

CRUST [23] is a similar bounded model checker for Rust that also uses the CBMC tool as a verification backend. However, CRUST explicitly does not support dynamic trait objects or dynamic dispatch and is no longer being actively developed.

Prusti [2] is a Rust compiler plugin built on the Viper verification infrastructure that can verify user-added specifications, as well as the absence of panics. Like Kani, Prusti leverages MIR type information to improve verification results. Prusti has a more expressive language for proof annotations than Kani, including supporting loop invariants that allow verification of programs Kani cannot currently verify. However, Prusti has limited support for unsafe code and does not support dynamic trait objects (our tests fail with

[12] https://rust-lang.github.io/dyn-upcasting-coercion-initiative/CHARTER.html

compiler errors). A recent extension to Prusti adds additional support for closures; however, to our knowledge, this extension does not handle dynamic closures [25].

Crux-MIR is a symbolic execution that similarly targets Rust's MIR [10] using Galois' Crucible verification infrastructure. Crux-MIR can verify simple cases of dynamic dispatch through `&dyn` pointer references. However, the tool fails with unimplemented for boxed dynamic objects (e.g., `Box<dyn T>`) and dynamic closure objects (e.g., `&dyn Fn()-> i32`).

Facebook's experimental MIRAI is an abstract interpreter for MIR [9]. MIRAI explicitly prioritizes a low false-positive rate for bugs rather than a low false-negative rate, and thus does not claim to provide sound verification.

LLVM-IR based Rust verification. Several LLVM-based tools have been extended to better support Rust code. As we showed in Section 4.2, the generality of supporting Rust at the LLVM IR level comes with the downside of being unable to apply Rust-type-level semantic understanding. However, LLVM-backed solutions tend to be less dependent on supporting changes to Rust, which currently evolves more quickly than LLVM.

The SMACK toolchain has been used to verify Rust by using the existing `rustc` backend to produce LLVM IR [3]. SMACK's toolchain was initially designed to primarily support Clang as a frontend and thus required changes (primarily to alias analysis) to support Rust programs. Further, SMACK's handling of the `Box` datatype requires that the box type be `Sized`, which seems to render the tool unable to reason about boxed dynamic closures.

Google Research's Rust Verification Tools (RVT) Project [21] aimed to build on a range of existing verification tools, from property testing to symbolic execution. Their tool supports multiple symbolic execution engines, each based on LLVM IR. RVT includes a KLEE [4] backend that can cover our full test suite of cases. However, KLEE is designed with a focus toward bug finding rather than unbounded, sound verification. RVT's SeaHorn backend uses the SeaHorn Verification Framework [11] and provides sound verification, but fails on some boxed closure test cases.

Analyzing virtual calls. Indirect function calls pose well-known problems for program analysis in general because identifying the code being invoked entails pointer analysis [16, 18]. Symbolic execution tools, for example, sometimes resort to requiring user annotations to handle indirect calls [20, Section 3.4] [17]. In languages with built-in support for virtual calls, such as object-oriented languages, optimizing compilers typically attempt *devirtualization*, opportunistically replacing indirect calls with direct calls when pointer information is sufficient, to make programs more analyzable [12, 19]. The function-pointer restriction technique in this work (Section 3.3) resembles a form of devirtualization that relies solely on type information from Rust's trait system, with the goal of improving model-checking efficiency and precision.

8 CONCLUSION

For verification of Rust to be deployed in large-scale projects, tools need to reason about the dynamic trait objects that are pervasive throughout the Rust standard library. In this paper, we demonstrated how our model-checking tool, Kani, successfully translates Rust's dynamic trait semantics. We show that by targeting Rust at the Mid-level Intermediate Representation level rather than LLVM-IR, we can leverage trait-based type information to improve verification time up to 15×. Our Firecracker case study highlights how this semantic understanding of traits unlocked previously intractable verification results. We encourage the other verification projects to use and build on our open-source suite of tests for dynamic dispatch, and we look forward to working with the Rust community to build an ecosystem where developers can verify functional correctness of security- and safety-critical Rust programs.

9 ACKNOWLEDGEMENTS

We thank Ted Kaminski, Zyad Hassan, Kareem Khazem, Adrian Palacios, Niko Matsakis, Michael Tautschnig, Rachit Nigam, and the anonymous reviewers for their help and feedback. The first author was partially supported by an NSF GRFP under Grant No. DGE-1650441. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] Alexandru Agache et al. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*.
[2] Vytautas Astrauskas et al. 2019. Leveraging Rust Types for Modular Specification and Verification. In *OOPSLA*.
[3] Marek Baranowski, Shaobo He, and Zvonimir Rakamaric. 2018. Verifying Rust Programs with SMACK. In *ATVA*.
[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*.
[5] Nathan Chong et al. 2020. Code-Level Model Checking in the Software Development Workflow. In *ICSE-SEIP*.
[6] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *TACAS*.
[7] Karel Driesen and Urs Hölzle. 1996. The Direct Cost of Virtual Function Calls in C++. In *OOPSLA*.
[8] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *SAT*.
[9] Facebook Experimental. 2021. MIRAI. https://github.com/facebookexperimental/MIRAI.
[10] Galois, Inc. 2020. Crux: Introducing our new open-source tool for software verification. https://galois.com/blog/2020/10/crux-introducing-our-new-open-source-tool-for-software-verification/.
[11] Arie Gurfinkel et al. 2015. The SeaHorn Verification Framework. In *CAV*.
[12] Kazuaki Ishizaki et al. 2000. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *OOPSLA*.
[13] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *NDSS*.
[14] Ralf Jung et al. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. In *POPL*.
[15] Ralf Jung et al. 2019. Stacked Borrows: An Aliasing Model for Rust. In *POPL*.
[16] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *CCS*.
[17] Petar Maksimović et al. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *CAV*.
[18] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2004. Precise Call Graphs for C Programs with Function Pointers. In *ASE*.
[19] Piotr Padlewski. 2017. Devirtualization in LLVM. In *SPLASH Companion*.
[20] David A. Ramos. 2015. *Under-constrained symbolic execution: correctness checking for real code*. Ph.D. Dissertation. Stanford University. https://searchworks.stanford.edu/view/11061347.
[21] Alastair Reid et al. 2020. Towards making formal methods normal: meeting developers where they are. In *arXiv*.
[22] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*.
[23] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. CRUST: A Bounded Verifier for Rust. In *ASE*.
[24] Aaron Weiss et al. 2019. Oxide: The Essence of Rust. https://arxiv.org/abs/1903.00982.
[25] Fabian Wolff, Aurel Bílý, et al. 2021. Modular Specification and Verification of Closures in Rust. In *OOPSLA*.