






# Automatically Enforcing Rust Trait Properties

Twain Byrnes<sup>(✉)</sup>, Yoshiki Takashima, and Limin Jia

Carnegie Mellon University, Pittsburgh, PA, USA  
{binarynewts,ytakashima,liminjia}@cmu.edu

**Abstract.** As Rust’s popularity increases, the need for ensuring correctness properties of software written in Rust also increases. In recent years, much work has been done to develop tools to analyze Rust programs, including Property-Based Testing (PBT), model checking, and verification tools. However, developers still need to specify the properties that need to be analyzed and write test harnesses to perform the analysis. We observe that one kind of correctness properties that has been overlooked is correctness invariants of Rust trait implementations; for instance, implementations of the equality trait need to be reflexive, symmetric, and transitive. In this paper, we develop a fully automated tool that allows developers to analyze their implementations of a set of built-in Rust traits. We encoded the test harnesses for the correctness properties of these traits and use Kani to verify them. We evaluated our tool over six open-source Rust libraries and identified three issues in *PROST!*, a protocol buffer library with nearly 40 million downloads.

**Keywords:** Rust · Traits · Software model checking

## 1 Introduction

The Rust programming language [17] has been steadily increasing in popularity for projects requiring a high level of precision and performance. Rust is used in the Linux Kernel [10, 25], Dropbox’s file-syncing engine [13], Discord [12], several Amazon Web Services (AWS) projects including S3, EC2, and Lambda [19], and many other high-profile projects. Rust has gained such wide adoption, in large part, due to its memory safety guarantees. However, Rust developers must still perform additional testing or verification to ensure functional correctness properties, specific to their code.

To help programmers analyze Rust programs, several tools are being developed [8, 11, 14, 16, 18]. For instance, *proptest* [23], a popular Property-Based Testing (PBT) [20] tool for Rust, is used in over 1500 crates according to *crates.io* [2]. There are several formal verification tools for Rust [1], including concurrency checkers and dynamic symbolic executors [3, 5, 6].

However, one road block preventing more Rust developers from using these tools is that they need to write test or verification harnesses for each of the

properties that they want to analyze. This is particularly problematic for Rust developers without a formal methods background, or for those who do not wish to spend time creating these tests. Therefore, exploring avenues for lowering developers' burden and automating the specification and analysis of correctness properties of Rust programs can be beneficial towards making Rust software more correct and secure. For instance, prior work has explored automatically converting existing test harnesses that developers wrote for `proptest` to harnesses that Kani, a model checker for Rust can use [22]. As a result, developers can verify their code using Kani without additional manual effort.

In this paper, we identify further areas where Rust developers can benefit from automated code verification. We observe that one kind of correctness properties that has been overlooked is correctness invariants of Rust trait implementations; for instance, a function implementing the equality trait for objects of a particular type must be transitive, symmetric, and reflexive. Rust traits, similar to interfaces in other languages, support shared functionality across types. Like equality, some Rust traits have invariants that are not checked by Rust's type-checker, and are thus left to the programmers to implement correctly.

We develop `TraitInv` to verify Rust trait invariants, as an addition to the Kani VS Code Extension [4]. `TraitInv`, like PBT, is based on the idea that certain properties of the output of specific methods must exist, regardless of the input. We identify invariants of commonly-used traits and create modular harnesses that can be inserted into a user's code with little intervention. Once inserted, these harnesses are verified by Kani. The harnesses themselves can be used by `proptest` by replacing symbolic value generators with random ones.

We evaluate `TraitInv` on 42 trait implementations from six libraries from `crates.io` [2]. The evaluation results show that `TraitInv` can create Kani testing harnesses on a wide variety of libraries over many traits. To answer the question of whether developers need their trait implementations verified for correctness invariants, we then use Kani to verify all the created harnesses. We discovered three issues in one trait implementation in `PROST!`, a popular and heavily tested Rust library with 40 million downloads. The tool is open source at <https://github.com/binarynewts/kani-vscode-extension>.

The rest of this paper is organized as follows. In Sect. 2, we review Rust traits and Kani and discuss related work. Then we describe specific traits and their invariants that `TraitInv` supports in Sect. 3. Next, we present the implementation and evaluation results in Sect. 4. Limitations of `TraitInv` and future work are discussed in Sect. 5.

## 2 Background and Related Work

In this section, we first discuss Rust traits, then provide the background of Kani, the Rust model checker that our tool is built on; we then discuss related tools.

## 2.1 Rust Traits

Rust traits define shared behavior abstractly, and are used like interfaces in other languages. Traits allow for the same methods to be called on many different types with a shared expected behavior.

For example, the `PartialOrd` trait allows for a partial order to be instantiated for a custom type, and the `Ord` trait can be used to implement a total order. `PartialOrd` requires the following partial comparison method:

```
partial_cmp(&self, other: &Rhs) -> Option<Ordering>.
```

This method takes two objects and returns a `Some` constructor of the proper `Ordering` type (`>`, `<`, or `=`) if they can be compared and a `None` constructor otherwise. The total order trait, `Ord`, requires the following method, which takes two objects, compares them, and returns an object of the `Ordering` type:

```
cmp(&self, other: &Self) -> Ordering.
```

Some basic Rust trait implementations can be derived automatically by the compiler. However, traits that require more complex behavior need to be user-defined. Listed below is an example user-defined implementation of a `PartialOrd` and `Ord`, taken from `crypto-bigint/src/limb/cmp.rs`, lines 93–116.

---

```

1  impl Ord for Limb {
2      fn cmp(&self, other: &Self) -> Ordering {
3          let mut n = 0i8;
4          n -= self.ct_lt(other).unwrap_u8() as i8;
5          n += self.ct_gt(other).unwrap_u8() as i8;
6
7          match n {
8              -1 => Ordering::Less,
9              1  => Ordering::Greater,
10             _  => {
11                 debug_assert_eq!(n, 0);
12                 debug_assert!(bool::from(self.ct_eq(other)));
13                 Ordering::Equal
14             }
15         }
16     }
17 }
18
19 impl PartialOrd for Limb {
20     #[inline]
21     fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
22         Some(self.cmp(other))
23     }
24 }

```

---

Rust traits have properties that the programmer must enforce on their implementation. For example, `partial_cmp` needs to be transitive.

## 2.2 Kani

The Kani Rust Verifier [24] uses bit-precise model checking and is built on top of CBMC [15]. Kani has been used to verify components of `s2n-quick` [21], an AWS-developed cryptographic library, and several popular Rust crates [22].

Kani harnesses analyze properties that should be maintained within code. For example, we may want to check that a function that takes in a `u64` and is supposed to return an even integer actually meets this return specification. To do this, we could write the following Kani harness, where the function `return_even` is called and the `assert` statement checks that the result is even.

---

```

1  #[kani::proof]
2  fn from_test() {
3      let num: u64 = kani::any();
4      assert!(return_even(num))
5  }
```

---

Kani is able to provide formal guarantees by symbolically executing the harness, allowing it to analyze properties checked by the harness for every possible input. In order for Kani to be able to generate symbolic inputs of a type, the type must implement the `kani::Arbitrary` trait. The Kani developers have implemented this trait for most primitive types and some types in the standard library. If the objects are enumeration or structure types whose fields implement `kani::Arbitrary`, it can be derived by placing `#[cfg_attr(kani, derive(kani::Arbitrary))]` above the type declaration. If not, the Kani user will need to implement `kani::Arbitrary`. This trait contains only the method `any() -> Self`, which takes no argument and returns a value of the type that `kani::Arbitrary` implements (`Self`). There are limitations on types that Kani can efficiently generate symbolic inputs for, which we detail in Sect. 5.

## 2.3 Related Work

`propproof` [22] is a tool that converts existing PBT harnesses into Kani harnesses so that they can be formally verified. `TraitInv` implements Kani harnesses and can be combined with `propproof` to verify trait invariants and application-specific invariants of Rust programs.

Erdirin developed a tool for verifying user-defined correctness properties of Rust programs, including trait invariants like ours [9]. Unlike `TraitInv`, which automatically generates test harnesses based on trait names, their tool requires user-provided annotations to generate test harnesses. It uses Prusti [7], a Rust verifier based on the Viper verification infrastructure.

### 3 Verifying Rust Trait Invariants

We explain how programmers can use `TraitInv` to analyze their trait implementations and explain its capabilities.

#### 3.1 Workflow for Verifying Trait Invariants

After launching the Kani VS Code Extension with `TraitInv`, Rust programmers can easily add Kani testing harnesses. To use this tool, one must navigate to the code file, highlight the line with their trait implementation declaration, bring up the Command Palette, and select “Add Trait Test”. Upon selecting this option, the tool generates the applicable Kani testing harness and inserts it directly above the implementation line.

#### 3.2 Harnesses for Trait Invariants

We implemented harnesses for invariants of frequently used traits within the Rust standard library. Next, we describe all the supported traits and their invariants. **From/Into**: Rust allows for conversion of values of a source type to values of a (different) target type using the `From` and `Into` traits. In the following code, we convert from a `u8` array of length 12 to a custom type, `ObjectId`, a struct with an `id` field of type length 12 `u8` array. We do this by simply returning an `ObjectId` instance, populating its `id` parameter with the array passed in.

---

```

1  struct ObjectId {
2      id: [u8; 12],
3  }
4
5  impl From<[u8; 12]> for ObjectId {
6      fn from(bytes: [u8; 12]) -> ObjectId {
7          Self { id: bytes }
8      }
9  }

```

---

The only requirement for such a conversion is that it does not panic (e.g. unwrapping `None` or dividing by 0). Therefore, the harness consists of code that first instantiates a symbolic input of the source type to be converted, and then calls the `from` method to create an instance of the target type from the value. Kani will report any panic behavior. Below, we provide an example of a Kani test harness for the above implementation of `From`.

---

```

1  #[kani::proof]
2  fn from_test() {
3      let t: [u8; 12] = kani::any();
4      let _ = ObjectId::from(t);
5  }

```

---

**PartialEq/Eq:** Rust types that implement the `PartialEq` trait allow a partial equality relation between objects of those types. `PartialEq` can be implemented for one type or across two types (e.g., comparing `i32` and `u32`).

To implement this trait, the user must implement the following method:

```
eq(&self, other: &Rhs) -> bool
```

This method takes two objects and returns a boolean stating whether or not these two objects are equal. This method is allowed to throw errors, and states that the two objects are not comparable in this instance. This distinction allows for instances like floating point numbers, where we do not want anything to equal NaN, including NaN.

`PartialEq` must obey transitivity, symmetry, and be consistent between not equal and the negation of equality. If this trait is implemented between two types, all `PartialEq` properties must hold across any combination of types.

The equality trait, `Eq`, must obey the rules of `PartialEq`, but additionally requires reflexivity and totality. It can only compare instances of the same type.

Below, we include an example of a `TraitInv`-generated Kani harness for properties of `PartialEq` implemented on the type `MyType`, a placeholder type for illustrative purposes. Comments point out what each assertion checks for.

---

```

1  #[kani::proof]
2  fn partialeq_test() {
3      let a: MyType = kani::any();
4      let b: MyType = kani::any();
5      let c: MyType = kani::any();
6      if a == b { assert!(b == a); } // symmetry
7      if (a == b) && (b == c) { assert!(a == c); } //
   ↳ transitivity
8      if a != b { assert!(!(a == b)); } // ne eq consistency
9      if !(a == b) { assert!(a != b); } // eq ne consistency
10 }

```

---

Note that we do not need to add a test for symmetry by swapping `a` and `b`, since they are of the same type, and this case will be tested regardless.

**PartialOrd/Ord:** As shown in Sect. 2, Rust has a partial order trait called `PartialOrd`, which requires a method

```
partial_cmp(&self, other: &Rhs) -> Option<Ordering>.
```

This method takes two objects and, if they can be compared, returns the `Some` constructor of the proper `Ordering` type, or else returns the `None` constructor. This method must be consistent with the following infix comparators: `<`, `>`, `==`. By consistent, we mean that two objects can be compared with one of the operators if and only if invoking the `partial_cmp` method on those two objects yields the `Some` constructor of the same `Ordering` type. Additionally, `>=` and `<=` must expand in the natural way. All of these requirements, other than consistency with `==`, are ensured by the default implementation of the four methods describing less than (or equal to) and greater than (or equal to). However, the programmer may choose to override these default implementations, in which case conformance is not guaranteed and must be checked.

`PartialOrd` requires transitivity between `<`, `>`, and `==`, as well as consistency between not equal and the negation of equality. Since `PartialOrd` is required to be consistent with `PartialEq`, this last consistency is ensured. Finally, `PartialOrd` requires duality: that `a < b` if and only if `b > a`.

The total order trait, `Ord`, requires a method

```
cmp(&self, other: &Self) -> Ordering,
```

which must be consistent with `PartialOrd`'s `partial_cmp`. This consistency can be broken when some traits are derived and others are implemented manually, or if they are implemented manually in a way that is not future proof.

`AsMut/AsRef`: These traits provide an interface to define types that represent static or mutable references of another type. For example, `Vec<T>` has an `AsRef` implementation for `[T]` because static vector references are also slices. Both `AsRef` and `AsMut` conversions have the same requirement as the `From` trait, in that they must not crash during conversion.

## 4 Implementation and Evaluation

We implemented harnesses for the traits described in Sect. 3 as an addition to the Kani VS Code Extension, automating the generation and insertion of these harnesses. We evaluated `TraitInv` on Rust libraries to answer the following questions: can `TraitInv` enable efficient verification of trait invariants across different traits, types, and libraries; and can `TraitInv` find bugs in trait implementations in libraries.

### 4.1 Implementation

`TraitInv` is implemented as a fork of the Kani VS Code Extension. Including this tool, the extension contains around 195K lines of typescript code, with around 500 lines of typescript code added to original, containing the harness encodings. Other than containing the encodings, this code reads the highlighted line, strips it down to figure out what trait and types it must create a harness for, creates the harness, and inserts it into the user's code immediately above

the highlighted line. The tool is launched along with the rest of the Kani VS Code extension, enabling potential users to use the tool without installing any tooling from outside the VS Code GUI.

## 4.2 Evaluation

**Experiment Setup** We ran our benchmarks on an Intel(R) Core(TM) i7-7700HQ with 4 cores and 16GB of RAM running Ubuntu 20.04.5 LTS. We used Kani Rust Verifier version 0.33.0 on CBMC version 5.88.1 with CaDiCaL.

**Dataset.** We pulled highly-downloaded Rust crates from the Rust community’s crate registry, `crates.io` [2]. We looked for crates that contained some of the implementations of the above traits and had at least several million downloads. We created 42 harnesses across six libraries, and ran these harnesses with Kani. The traits and libraries are summarized in Table 1. The Types column details the types that the trait was implemented on or between, and may be slightly edited from the original implementation line for ease of reading and comprehension.

For traits implemented on polymorphic types, we can only analyze concrete instantiations of them. This is because Kani is bit-precise and needs concrete types to know the memory layout of the values. We list the concrete instantiations of the polymorphic types below.

*Test 1.* Since `chrono::DateTime` is parametric, we were only able to test it on one possible timezone. We chose `chrono::Utc`, since the other options were `chrono::Local`, which would lead to inconsistent testing based on where the testing takes place geographically, and `chrono::FixedOffset`, which requires an input to offset from UTC time, and is thus similar to simply using `chrono::Utc`. These other options could potentially lead to bugs that choosing `chrono::Utc` might not, which we did not test.

*Tests 11, 12.* Both of these tests create symbolic values of `Checked<T>`, which is parametric, so we were only able to test it on one possible type. We chose `U64`, a custom type from the `crypto-bigint` library similar to the standard `u64`, since that was the only type that we found used with `Checked` in the entire library.

*Tests 29, 34.* These tests were for the `StackVec<usize>` type, so we were forced to choose a `usize` to implement these tests for. Without choosing, Rust would not know what size to allot for this, and would lead to an error.

*Tests 39–42.* `Uint<usize>` is generated in these tests, and takes in a `usize`. We had to fix a `usize` to implement `kani::Arbitrary` for and test on, otherwise the items would not have a fixed size and running Kani would lead to an error.



**Table 1.** Harnesses used to evaluate effectiveness of synthesis. **\*\*chrono::DateTime<T: chrono::TimeZone> for Bson.**

	Trait	Library	Types	Time(s)	Mem(GB)
1	From	bson-rust	*	11.55	0.120
2	From	bson-rust	u8 for BinarySubtype	2.80	0.046
3	From	bson-rust	BinarySubtype for u8	2.63	0.045
4	From	bson-rust	f32 for Bson	7.89	0.038
5	From	bson-rust	f64 for Bson	5.52	0.048
6	From	bson-rust	bool for Bson	4.54	0.048
7	From	bson-rust	i32 for Bson	3.87	0.048
8	From	bson-rust	i64 for Bson	4.76	0.048
9	From	bson-rust	Decimal128 for Bson	4.70	0.048
10	From	bson-rust	[u8; 12] for ObjectId	2.42	0.045
11	From	crypto-bigint	Checked<T> for CtOption<T>	0.74	0.029
12	From	crypto-bigint	Checked<T> for Option<T>	0.74	0.030
13	From	crypto-bigint	u8 for Limb	0.61	0.029
14	From	crypto-bigint	u16 for Limb	0.61	0.029
15	From	crypto-bigint	u32 for Limb	0.60	0.029
16	From	crypto-bigint	u64 for Limb	0.62	0.030
17	From	crypto-bigint	Limb for Word	0.70	0.029
18	From	crypto-bigint	Limb for WideWord	0.61	0.029
19	From	proptest	(usize, usize) for SizeRange	0.48	0.026
20	From	proptest	usize for SizeRange	0.46	0.026
21	From	proptest	f64 for Probability	0.50	0.026
22	From	proptest	Probability for f64	0.49	0.026
23	From	prost	Timestamp for DateTime	434.63	0.452
24	From	prost	DateTime for Timestamp	3.18	0.073
25	From	prost	EncodeError for std::io::Error	4.21	0.282
26	PartialEq	crypto-bigint	Limb	1.85	0.043
27	PartialEq	rust-lexical	f16	5.09	0.093
28	PartialEq	rust-lexical	bf16	0.67	0.022
29	PartialEq	rust-lexical	StackVec<usize>	5.13	0.112
30	PartialEq	sharded-slab	State	0.42	0.118
31	PartialEq	sharded-slab	DontDropMe	1.73	0.135
32	Eq	crypto-bigint	Limb	1.92	0.043
33	Eq	prost	Timestamp	1.58	0.207
34	Eq	rust-lexical	StackVec<usize>	8.14	0.158
35	PartialOrd	crypto-bigint	Limb	37.90	0.334
36	PartialOrd	rust-lexical	f16	56.84	0.446
37	PartialOrd	rust-lexical	bf16	5.69	0.064
38	Ord	crypto-bigint	Limb	28.75	0.226
39	AsRef	crypto-bigint	[Word; usize] for Uint<usize>	1.30	0.057
40	AsRef	crypto-bigint	[Limb] for Uint<usize>	1.23	0.056
41	AsMut	crypto-bigint	[Word; usize] for Uint<usize>	1.34	0.057
42	AsMut	crypto-bigint	[Limb] for Uint<usize>	1.32	0.057

### 4.3 Results

We ran our 42 Kani test harnesses and collected the results into Table 1.

**Performance of TraitInv.** The last two columns of Table 1 document the time spent verifying the given harness in seconds, and maximum memory usage measured in gigabytes, respectively. Most of our tests were able to complete in under ten seconds, many of them under one second. Though we had an outlier at 434.64s, none of the other tests took over a minute to complete. Each test uses under 0.5 GB of memory, with only a little over a quarter of the tests using over 0.1GB of memory. Thus, our tool has been shown to be efficient.

There is a notable disparity in time taken and memory used between tests 23 and 24. Though these `From` implementations are made to convert between the same two types, they appear to take vastly different amounts of time to verify. Conversion from `Timestamp` to `DateTime` takes over 400s, by far the longest amount of time on the table, while its counterpart, converting in the other direction, takes only three seconds. This is due to the fact that the former takes over 400s to verify as correct, while its counterpart takes only three seconds to terminate due to finding bugs. The same factor is responsible for the difference in maximum memory usage between benchmarks 23 and 24.

Verification for tests 29 and 34 took a 5.13 and 8.14s respectively, considerably longer than those taking under a second. This is because these types involved loops: Kani will indefinitely unroll loops unless a maximum loop bound is obvious. Finding a maximum bound is very rare, and it is much more common for Kani to get stuck attempting to unwind a loop. By telling Kani how far to unwind, it is able to set an upper bound and try to verify from there. The lower the bound, the more efficient Kani is, but when set too low Kani may not be able to formally verify every case and will mark a failed check. The necessary unwinding on these tests explain the relatively lengthy test time.

**Bugs Identified.** Using `TraitInv`, we identified three issues with benchmark 24, an implementation of the `From` trait that converts objects of `PROST!`'s `DateTime` type to objects of its `Timestamp` type. There is already a PBT harness for this trait implementation, yet the errors had not been found. We followed a responsible disclosure process in reporting the bug described in this paper, reporting the bug on the `PROST!` GitHub issues page.

The `from` method that converts from `DateTime` to `Timestamp` was implemented by calling several different functions, including ones which convert the number of years into seconds and the number of months into seconds. All of the errors were found in the function converting the year into a number of seconds. The problematic code from `prost/prost-types/src/datettime.rs` is shown below. We leave out sections of the function with no problems, since the full function is around 80 lines. Lines 2–3 set up some variables for the rest of the function. Line 6 contains an early return for years 1900–2038, since these are assumed to be seen more often and should thus be able to be computed more quickly. Lines 10–11 set up more variables to be used in the code excluded on

line 13. Finally, lines 15–19 do some ending computations and return the tuple of the number of seconds and a boolean for whether or not it is a leap year.

---

```

1  pub(crate) fn year_to_seconds(year: i64) -> (i128, bool) {
2      let is_leap;
3      let year = year - 1900;
4
5      // Fast path for years 1900 - 2038.
6      if year as u64 <= 138 { ... }
7
8      ...
9
10     let mut cycles: i64 = (year - 100) / 400;
11     let mut rem: i64 = (year - 100)
12
13     ...
14
15     (
16         i128::from((year - 100) * 31_536_000) +
17     ↪ i128::from(leaps * 86400 + 946_684_800 + 86400),
18         is_leap,
19     )
20 }

```

---

Next, we discuss the three bugs found.

**Subtraction Overflow on Line 3.** At line 3 in the above function, we see a subtraction of 1900 from the year. There is no check to ensure that the year is within a valid range, but the conversion must work for any year. For example, when this function is called with the year `i64::MIN + 1800`, we will overflow.

**Subtraction Overflow on Line 10.** There is a similar error on line 10, where no check is done to ensure that 100 can actually be subtracted from the current value of the year. For example, we would overflow if we input the year `i64::MIN + 1950`, since we first reset this to 50 on line 3, then attempt to subtract 100 from it. Though line 11 also contains subtraction by 100 without a check, Kani does not point this out, as it is the exact same issue. If the issue on line 10 were fixed without also addressing the subtraction on line 11, Kani would point to line 11 as an issue.

**Multiplication Overflow on Line 16.** The return value computation has a multiplication overflow, which would yield an incorrect number of seconds. Setting the year to a large negative number causes the multiplication `(year - 100) * 31_536_000` on line 16 to overflow. Kani returned `year=-6192467633871255600` as a concrete example that causes such an overflow.

## 5 Limitations and Future Work

We have seen that we can implement some traits across two types. We may wish to implement some traits across three or more types. We can do this by implementing a trait across two types in one implementation, and two types in a separate implementation with overlap. There are some traits which have special invariants for this case. Both `PartialEq` and `PartialOrd` may be defined across an arbitrary number of types, in which case transitivity must hold across types. This invariant is not currently checked by `TraitInv`, as it looks at one implementation line at a time. There were no such instances of a trait implemented across more than two types in the test set, so this is likely a rare occurrence.

Since `TraitInv` is built on top of Kani, it inherits all of Kani’s limitations. Kani can derive some implementations of its `Arbitrary` trait, though it cannot for many types, often leading to manual implementations. For instance, programmers cannot implement `kani::Arbitrary` for types with unbounded size, like strings and vectors. Kani can create inputs for the above types of any size less than or equal to some bound with large verification time, and is much more efficient when told to create objects of a specific size. Kani also struggles to generate `Arbitrary` implementations for polymorphic types like `struct InnerArray<T, const N: usize>([T; N]);`, which requires both a type and size input in order to implement `kani::Arbitrary`; we cannot test on all potential parameterizations. The standard workaround for a parametric type is to pick a specific parameterization and implement `kani::Arbitrary` for that type.

Improving generation in these cases would be extremely beneficial, increasing modularity and decreasing the amount of time spent by users on formal verification. Additionally, there are several performance issues with Kani, stopping it from working on larger crates. We used our tool to create tests for the AWS SDK for Rust, but Kani fails to verify these tests, despite their similarity to tests that Kani has no trouble with in smaller libraries.

## 6 Conclusion

We introduced a fully automated harness synthesis tool as an extension to the Kani VS Code Extension, allowing Rust programmers with no formal verification expertise to ensure correctness of their Rust trait implementations. We enable Rust programmers to generate Kani testing harnesses for their trait implementations at the click of a button, which can be checked by Kani and provide formal verification that their trait implementations follow the necessary invariants. Our evaluation shows that such harnesses can be used across a wide variety of Rust libraries, and can identify issues in heavily tested real-world code.

**Acknowledgements.** We would like to thank the anonymous reviewers for their feedback on our paper. This work was partially funded by the National Science Foundation via grants CNS2114148 and an Amazon Research Award, Fall 2022 CFP. Any opinions,

findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of Amazon.

## References

1. Rust verification tools (2021). <https://rust-formal-methods.github.io/tools.html>
2. crates.io: Rust Package Registry (2023). <https://crates.io/>
3. haybale (2023). <https://github.com/PLSysSec/haybale>
4. Introducing the kani vs code extension (2023). <https://model-checking.github.io/kani-verifier-blog/2023/06/30/introducing-the-kani-vscode-extension.html>
5. Loom (2023). <https://github.com/tokio-rs/loom>
6. Shuttle (2023). <https://www.shuttle.rs/>
7. Astrauskas, V., et al.: The Prusti project: formal verification for rust (invited). In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NFM 2022. LNCS, vol. 13260, pp. 88–108. Springer, Cham (2022). [https://doi.org/10.1007/978-3-031-06773-0\\_5](https://doi.org/10.1007/978-3-031-06773-0_5)
8. Denis, X., Jourdan, J.H., Marché, C.: Creusot: a foundry for the deductive verification of rust programs. In: Riesco, A., Zhang, M. (eds.) ICFEM 2022. LNCS, vol. 13478, pp. 90–105. Springer, Cham (2022). [https://doi.org/10.1007/978-3-031-17244-1\\_6](https://doi.org/10.1007/978-3-031-17244-1_6), <https://hal.inria.fr/hal-03737878>
9. Erdin, M.: Verification of Rust Generics, Typestates, and Traits. Master’s thesis, ETH Zürich (2019)
10. Filho, W.A.: Rust in the Linux kernel, April 2021. <https://security.googleblog.com/2021/04/rust-in-linux-kernel.html>
11. Ho, S., Protzenko, J.: Aeneas: rust verification by functional translation. Proc. ACM Program. Lang. 6(ICFP), 116:711–116:741 (2022). <https://doi.org/10.1145/3547647>
12. Howarth, J.: Why discord is switching from go to rust (2020). <https://discord.com/blog/why-discord-is-switching-from-go-to-rust>
13. Jayakar, S.: Rewriting the heart of our sync engine (2020). <https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine>
14. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: securing the foundations of the Rust programming language. Proc. ACM Program. Lang. 2(POPL), 66:1–66:34 (2017). <https://doi.org/10.1145/3158154>
15. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-642-54862-8\\_26](https://doi.org/10.1007/978-3-642-54862-8_26)
16. Lehmann, N., Geller, A., Vazou, N., Jhala, R.: Flux: Liquid Types for Rust, November 2022. <https://doi.org/10.48550/arXiv.2207.04034>, <http://arxiv.org/abs/2207.04034>
17. Matsakis, N.D., Klock, F.S.: The rust language. In: Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology. HILT ’14, pp. 103–104. Association for Computing Machinery, New York, NY, USA, October 2014. <https://doi.org/10.1145/2663171.2663188>
18. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based Verification for Rust Programs. ACM Trans. Program. Lang. Syst. 43, 15:1–15:54 (2021). <https://doi.org/10.1145/3462205>
19. Miller, S., Lerche, C.: Sustainability with Rust | AWS Open Source Blog, February 2022. <https://aws.amazon.com/blogs/opensource/sustainability-with-rust/>, section: Developer Tools

20. Paraskevopoulou, Z., Hrițcu, C., Dénès, M., Lampropoulos, L., Pierce, B.C.: Foundational property-based testing. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 325–343. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-22102-1\\_22](https://doi.org/10.1007/978-3-319-22102-1_22)
21. Schwartz-Narbonne, D.: Use Kani action in CI by danielsn · Pull Request #1556 · aws/s2n-quic, October 2022. <https://github.com/aws/s2n-quic/pull/1556>
22. Takashima, Y.: Proptest: free model-checking harnesses from PBT. In: ESEC/FSE (2023)
23. The proptest developers: Proptest, May 2023. <https://github.com/proptest-rs/proptest>
24. VanHattum, A., Schwartz-Narbonne, D., Chong, N., Sampson, A.: Verifying dynamic trait objects in rust. In: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. ICSE-SEIP '22, pp. 321–330. Association for Computing Machinery (2022). <https://doi.org/10.1145/3510457.3513031>
25. Vaughan-Nichols, S.J.: Linux kernel 6.1: rusty release could be a game-changer (2023). [https://www.theregister.com/2022/12/09/linux\\_kernel\\_61\\_column/](https://www.theregister.com/2022/12/09/linux_kernel_61_column/)