Crabtree: Rust API Test Synthesis Guided by Coverage and Type

YOSHIKI TAKASHIMA, Carnegie Mellon University, USA
CHANHEE CHO, Carnegie Mellon University, USA
RUBEN MARTINS, Carnegie Mellon University, USA
LIMIN JIA, Carnegie Mellon University, USA
CORINA S. PĂSĂREANU, Carnegie Mellon University, USA

Rust type system constrains pointer operations, preventing bugs such as use-after-free. However, these constraints may be too strict for programming tasks such as implementing cyclic data structures. For such tasks, programmers can temporarily suspend checks using the `unsafe` keyword. Rust libraries wrap `unsafe` code blocks and expose higher-level APIs. They need to be extensively tested to uncover memory-safety bugs that can only be triggered by unexpected API call sequences or inputs. While prior works have attempted to automatically test Rust library APIs, they fail to test APIs with common Rust features, such as polymorphism, traits, and higher-order functions, or they have scalability issues and can only generate tests for a small number of combined APIs.

We propose Crabtree, a testing tool for Rust library APIs that can automatically synthesize test cases with native support for Rust traits and higher-order functions. Our tool improves upon the test synthesis algorithms of prior works by combining synthesis and fuzzing through a coverage- and type-guided search algorithm that intelligently grows test programs and input corpus towards testing more code. To the best of our knowledge, our tool is the first to generate well-typed tests for libraries that make use of higher-order trait functions. Evaluation of Crabtree on 30 libraries found four previously unreported memory-safety bugs, all of which were accepted by the respective authors.

CCS Concepts: • **software and its engineering → software maintenance tools**; *software reliability*.

Additional Key Words and Phrases: Rust, API testing, program synthesis, fuzzing

ACM Reference Format:
Yoshiki Takashima, Chanhee Cho, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. 2024. Crabtree: Rust API Test Synthesis Guided by Coverage and Type. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 293 (October 2024), 30 pages. https://doi.org/10.1145/3689733

1 Introduction

The Rust language [26] is becoming popular in systems development. Rust is now used in critical systems such as the Linux kernel [9, 36], Google's Project Fuchsia operating system [15], and Firefox's core graphics engine [28]. One main advantage of Rust, which is driving its increased adoption, is its focus on memory safety, achieved through ownership and borrowing, two key concepts enforced by Rust type system. Rust type system carefully tracks references and ensures that there can be only one active mutable reference to a given piece of memory at a time and

Authors' Contact Information: Yoshiki Takashima, Carnegie Mellon University, Pittsburgh, USA, ytakashi@andrew.cmu.edu; Chanhee Cho, Carnegie Mellon University, Pittsburgh, USA, chanheec@andrew.cmu.edu; Ruben Martins, Carnegie Mellon University, Pittsburgh, USA, rubenm@andrew.cmu.edu; Limin Jia, Carnegie Mellon University, Pittsburgh, USA, liminjia@andrew.cmu.edu; Corina S. Păsăreanu, Carnegie Mellon University, Pittsburgh, USA, pcorina@andrew.cmu.edu.

ACM 2475-1421/2024/10-ART293
https://doi.org/10.1145/3689733

that only one thread can mutate a given piece of memory. By enforcing these constraints, Rust eliminates classes of shared memory bugs such as use-after-free and data races.

Purely safe Rust is not ergonomic for some systems programming tasks (e.g., implementing cyclic data structures like doubly linked lists). To accommodate these tasks, Rust provides the `unsafe` keyword to allow programmers to perform unchecked pointer manipulation. These `unsafe` code blocks, commonly written by experts, are often wrapped in safe library APIs that expose a clean interface to others. This design pattern of mostly safe code wrapping around a small core of expert-written `unsafe` code is the key to Rust's approach to balancing performance and safety.

However, using `unsafe` code improperly risks resurrecting memory bugs vanquished by the type system. Recent surveys [32, 48] show the danger of carelessly using `unsafe` Rust, which can lead to potential vulnerabilities. Prior work on Rust library API testing [21, 41] uncovered numerous bugs, including exploitable vulnerabilities [1]. To mitigate this risk and help developers identify these bugs in their libraries, researchers have proposed automatic testing techniques for Rust library APIs. RULF [21] is a fuzz driver synthesizer that uses graph search to combine APIs and build drivers. While RULF has been highly successful at fuzzing parsers and APIs with concrete, non-polymorphic input types, it cannot handle polymorphic APIs. Polymorphic APIs are prevalent in Rust programs [45]; consequently, a large fraction of the Rust libraries cannot benefit from RULF. Another approach, SyRust [41], uses constraint solving and synthesis techniques to generate well-typed API call sequences over both concrete and polymorphic Rust APIs. However, SyRust suffers from scalability limitations that prevent it from testing beyond 20 or so APIs with a larger number of APIs leading to combinatorial explosion of mostly useless tests and frequent timeouts in the constraint solver. In addition, SyRust only runs the synthesized tests with fixed inputs without trying diverse sets of inputs, further limiting testing completeness. To improve the quality and coverage of the automatically generated test sequences, we combine ideas from fuzzing and synthesis to develop Crabtree. Crabtree builds upon SyRust to generate API call sequences and overcomes SyRust's scalability limitations by using both the coverage and type information of previously generated API call sequences, choosing such sequences for prefixes that generate longer sequences in future iterations of synthesis. Crabtree gives priority to API call sequences that generate a return value with a previously unseen (new) type in addition to those that produce new code coverage.

To provide better support for polymorphic APIs, Crabtree's synthesis algorithm encodes trait constraints such that the instantiation of polymorphic types during synthesis is more targeted. This greatly reduces the number of ill-typed tests due to trait constraints and enables Crabtree to achieve better scalability than SyRust. An added benefit of encoding trait constraints is that Crabtree can generate calls to APIs that take functions as arguments. This is because Rust's function closures and function pointers, which can be passed as arguments to higher-order functions and be returned as arguments, all implement a variant of the `Fn` trait. As a result, Crabtree is able to generate tests that supply higher-order functions with arguments of the correct type. To handle cases where no existing APIs in the target library can be used as arguments to higher-order APIs, Crabtree synthesizes a closure of the correct type.

Crabtree also incorporates a fuzzer to generate diverse inputs to API call sequences. Crabtree uses a notion of *fuzzable* types to guide input generation. Different values for an input of a "fuzzable" type are likely to explore different code execution paths of the API, while all values of a "non-fuzzable" type are likely to lead to the same behavior of the API (e.g., input with a polymorphic type). Therefore, the fuzzer only mutates inputs of fuzzable types. However, trait constraint on polymorphic types provides us with an opportunity to fuzz APIs with polymorphic type signatures, which are otherwise considered nonfuzzable. More concretely, Crabtree's fuzzing process considers fuzzable, polymorphic inputs constrained by traits that require the input to be in the format of

Table 1. Comparison of Crabtree, SyRust, and RULF.

	Crabtree	SyRust	RULF
Supports for traits and higher-order functions	✓	×	×
Fuzzing inputs	✓	×	✓
Methods for generating well-typed Rust programs	SAT encoding of typing constraints		graph-based search
Size of libraries tested	∼100 APIs	∼20 APIs	∼100 APIs

a binary buffer (e.g., the `Read` trait). To further improve the efficiency of fuzzing, we propose a hierarchical organization of fuzzer-generated input corpus that avoids duplicate work across repeated runs of the fuzzer on similar API sequences. Table 1 summarizes the features supported by Crabtree, SyRust, and RULF.

Our evaluation shows that Crabtree's capabilities enable better Rust API testing as compared to SyRust, in both scale and coverage. While it is not always the case that Crabtree achieves better coverage than RULF on all benchmarks, Crabtree's novel features enable it to handle libraries that are beyond RULF's reach: Crabtree achieves higher test coverage in difficult-to-test APIs such as iterators and trait functions. Crabtree finds four previously unreported bugs confirmed by the respective library authors.

We summarize our contributions as follows.

- We re-architect SyRust's synthesis engine to be guided by test coverage and type discovered to generate more effective tests.
- We extend the polymorphism support of SyRust to enable testing with traits, a major feature of Rust, and higher-order functions.
- We connect the synthesis engine with a fuzzer optimized to generate only inputs for "fuzzable" types and store input corpus in a hierarchical tree structure.
- We implement Crabtree, a testing tool for Rust libraries, and evaluate Crabtree on Rust libraries to show it can find bugs involving traits and requiring input mutations while scaling to libraries with a large number of APIs.

2 Background and Motivation

We briefly review Rust language features and existing work on automatically testing Rust library APIs. We then highlight challenges in this domain to motivate Crabtree.

2.1 Background

Rust Types and Traits. We explain key Rust features relevant to this work via a running example shown in Fig. 1. Lines 1-3 show the harness for fuzzing, which calls the test function and we will explain in detail Section 3; lines 5-10 show a test function that makes a sequence of API calls to test the `integer-encoding` library; lines 12-19 show snippets from the `integer-encoding` library.

An important feature of Rust type system is polymorphism, which allows types to be parameterized using other types. For example, the polymorphic vector type `Vec<T>` is parameterized over its element type `T`. The type variable `T` can be instantiated with concrete types. For instance, on line 5, the argument to the `test` function is `Vec<u16>`.

```
 1  fn fuzz(fuzzer_bits : &[u16]) {
 2    if fuzzer_bits.len() > 0 { test(fuzzer_bits[0], fromuzzer_bits[1..].clone()); }
 3  }
 4
 5  fn test(integer: u16, rest: Vec<u16>) {
 6    let mut vecint : Vec<u8> = rest.iter().map(|elem : &u16| elem.to_be_bytes()[0]);
 7    let buf &mut [u8] = vecint.as_mut();
 8    integer.encode_fixed(buf)
 9    let integer64: u64 = <u64 as fixed::FixedInt>::decode_fixed(buf);
10  }
11
12  impl FixedInt for u8 { // Library-side code
13    const REQUIRED_SPACE: 8;
14    type ENCODED: [u8]; // for explanation only
15    fn encode_fixed(self, dst: &mut [u8]) {...}
16    fn decode_fixed(src: &[u8]) -> Self
17    //...
18  }
19  impl FixedInt for u64 {...}
```

Fig. 1. Test function exposing a bug from `integer-encoding` crate.

Rust allows programmers to define shared behavior among types, through *traits*. Traits are implemented on types and define a shared interface to interact with all the types that implement this trait. For example, the `FixedInt` trait defines an interface to encode and decode values of a type from fixed-bitwidth bytes. Code on lines 12-18 defines this trait for the type `u8`. It can also be defined for other types (e.g., `u64` on line 19). Traits can also have associated constants (`REQUIRED_SPACE` on line 13) and types (`ENCODED` on line 14) [1]; `encode_fixed` and `decode_fixed` (lines 15-16) are the interfaces that turn an integer to its `u8` encoding, store it in a mutable buffer, and decode a buffer of `u8` encoding to an integer, respectively. The test case (`fn test`) uses these trait APIs; it first makes a mutable buffer (line 7), then writes the encoded `u8` into the buffer (line 8), and finally decodes `u64` from the same buffer (line 9). Programmers can restrict the concrete types used to instantiate a type variable by specifying a trait constraint such as `T : FixedInt<REQUIRED_SPACE=1>`. The compiler will reject any use of the polymorphic type where the trait constraint is not met. Rust also allows programmers to define their own data types and traits, providing additional flexibility.

Rust's first-class functions such as function pointers and closures all implement a variant of the `Fn` trait. As a result, the type specifications of Rust's higher-order functions also use some variant of the `Fn` traits. Other than `Fn`, the other variants are `FnOnce`, for ownership-moving functions that are called at most once, and `FnMut`, which have exclusive write access to variables it captured. For example, `map` is a higher-order function of type signature `_ : Iter<A>` $\rightarrow$ `(F : Fn(A)` $\rightarrow$ `B)` $\rightarrow$ `_:Iter<B>` that takes an iterator over type `A`, and any function of type `F : Fn(A)` $\rightarrow$ `B`, and produces an iterator over type B `_:Iter<B>`. `Fn(A)` $\rightarrow$ `B` is a trait automatically implemented by function closures like `|elem : &u16| {...}` whose type signature is `A` $\rightarrow$ `B`. _ is Rust syntax for anonymous type variables; different occurrences of it do not necessarily represent the same type. Closures may also capture ownership of variables if they are annotated with the `move` keyword. These implement a more specific trait `FnOnce(A)` $\rightarrow$ `B` or `FnMut(A)` $\rightarrow$ `B` depending on whether they capture mutable references or ownership. These closures present a challenge to test synthesizers that can only generate straight-line API call sequences as it does not generate the bodies of function closures, which live in a different scope from the top-level API call sequences.

[1]The type `ENCODED` line 14 is added by us to demonstrate associated types in traits.

Automatic Testing of Rust Libraries. A common way to automatically test programs is coverage-guided fuzzing [16, 25, 37, 50]. The technique automatically discovers interesting inputs to a program by mutating the inputs until the execution reaches a previously unseen location in the code (coverage increase). Coverage-increasing inputs are kept and further mutated, guiding the search for more inputs. This list of inputs is called the *input corpus*.

To test Rust library APIs using fuzzing, fuzzing drivers (harnesses) in the form of sequences of API calls are also needed. Building such fuzz drivers is challenging because of the complex Rust typing constraints that need to be satisfied. RULF [21] is a fuzz driver synthesizer that uses graph search to combine different API calls and build drivers. RULF cannot handle polymorphic APIs.

In contrast, the constraint-based synthesis tool SyRust [41] can generate well-typed tests for Rust library APIs and supports polymorphic types. The tests that SyRust [41] synthesizes model straight-line sequences of API calls. The syntax is shown below, where τ is any valid type in Rust.

$$\begin{array}{llll} \textit{Program} & P & := & L \mid L; P \\ \textit{Line} & L & := & f(V) \mid \texttt{let}\ v : \tau = f(V) \mid \texttt{let}\ v : \&\tau = \&\, v \mid \texttt{let}\ v : \&\texttt{mut}\ \tau = \&\texttt{mut}\ v \\ \textit{Vars} & V & := & v_1, \ldots, v_k \end{array}$$

A program P is a sequence of lines $L_1; \ldots; L_n$. Arbitrary combinations of lines are not guaranteed to be well-typed. For example, the syntax allows a line that uses variables before they are declared. SyRust encodes the problem of synthesizing well-typed API combinations into Boolean Satisfability [7] and leverages off-the-shelf logical solvers [6] for synthesis. The logical encoding builds a system of constraints that arise out of matching input types with variables in the context. Given a possibly polymorphic API with signature $f : (\tau_1, \ldots, \tau_n) \rightarrow \tau$, and inputs to f of types $v_1 : \sigma_1, \ldots, v_n : \sigma_n$, SyRust defines two sets of validity constraints at the level of a single line.

- Individual types have compatible shape: $\texttt{unify}(\tau_i, \sigma_i) \mapsto \Gamma_i$. If successful, `unify` also generates a type variable typing context Γ_i that unifies τ_i and σ_i.
- Consistent type variable assignments for $\Gamma_1, \ldots, \Gamma_n$. Type variables in SyRust are bound at the function level, so if the same type variable occurs in different inputs, all inputs have to assign the type variable to consistent types. SyRust encodes this constraint with boolean logic variables, each representing the fact that a Rust variable was used as input to a particular API. For example, consider calling the API `push : (&mut Vec<T>, T)` $\rightarrow$ `()` from a type context `{v0: Vec<u16>, v1: u16, v2: u8}`. SyRust will create 3 variables, one for each of the following valid unification `{(v0, &mut Vec<T>)`$\mapsto$ `{T = u16}, (v1, T)`$\mapsto$ `{T = u16},(v2, T)`$\mapsto$ `{T = u8}}`. Observe that `T` maps to incompatible types of `u8` and `u16` if all of the three are considered valid unification. SyRust uses additional logical constraints to ensure type variable compatibility. In this case, if the variable representing the unification of `(v0, &mut Vec<T>)` is true, then the variable representing `(v2, T)` must be false.
- Finally, some APIs require additional constraints due to Rust's ownership type system. Consider `append : (&mut Vec<T>, &mut Vec<T>)` $\rightarrow$ `()`. To prevent racing mutations on the same vector at the same time, Rust does not allow the same variable to be mutably borrowed more than once, making API invocations like `append(&mut v0, &mut v0)` ill typed. To prevent these patterns, SyRust enforces an at-most-one constraint over such uses. Other ownership restrictions are handled similarly.

These constraints are extended across lines to ensure $L_1; \ldots; L_n$ are well-typed in Rust. Since SyRust models borrowing as APIs, restrictions on borrowing and ownership are also encoded at this level. The solutions to the constraints yield well-typed programs.

While SyRust does not explicitly model traits, it leverages feedback from `rustc` to reduce the number of ill-typed programs synthesized. For example, SyRust might generate an API call like `encode_fixed(String::new(),&[])`, ignoring the fact that `String` type does not implement the trait

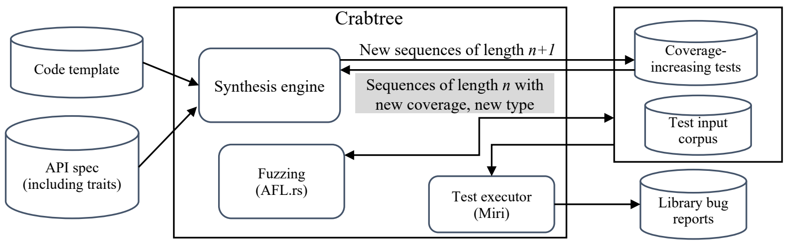

Crabtree
Code template
API spec (including traits)
Synthesis engine
New sequences of length n+1
Sequences of length n with new coverage, new type
Coverage-increasing tests
Test input corpus
Fuzzing (AFL.rs)
Test executor (Miri)
Library bug reports

Fig. 2. Overview of Crabtree

`FixedInt`. When `rustc` rejects this API call with a type error, SyRust attaches a metadata to the first input type of `encode_fixed` to prevent a variable of type `String` from being used for that position. In the next round of the synthesis, SyRust will generate a call using a variable of a different type. Given enough time, SyRust may eventually stumble upon a variable with required trait constraints. However, this iterative refinement loop is very inefficient.

2.2 Motivation

Revisiting our example in Fig. 1, the sequence of API calls in `test` triggers a memory safety bug in a prior version of `integer-encoding-rs`, a popular encoding library with 6M downloads. The bug is triggered on line 9 when attempting to decode a buffer that does not have enough bits. This sequence triggers an out-of-bounds access on `::decode_fixed` as it tries to read beyond the limit of the empty array to parse an integer. Note that this bug is triggered only when (1) the decoded bit size is bigger than the encoded bit size, and (2) the original buffer is too short for `u64`.

To automatically construct the test that can successfully trigger the bug, we first need to be able to deal with polymorphic types and traits. These features have been challenging for prior tools like SyRust and RULF with both tools failing to support traits directly. Furthermore, the bug can be triggered only by an array of length 1. Without a diverse set of inputs, which could be generated via fuzzing, the bug is likely to be missed. In this work, we aim to combine the strengths of constraint-based synthesis with coverage-guided fuzzing to enable the automatic testing of Rust library APIs, including those that are polymorphic and use traits, which could not be handled by previous techniques and tools.

However, naively synthesizing long API sequences and generating inputs for synthesized test sequences can be prohibitively expensive, as the number of API combinations can grow exponentially and each test takes a few seconds to compile and run. Fuzzers are intended to test a few well-written harnesses, not a million harnesses that are closely related. In this work we address the challenge of effectively combining test sequence synthesis with input fuzzing for scalable Rust API testing. To this end, we develop methods that guide the synthesis and fuzzing using typing and coverage information and reduce unnecessary or duplicate work by sharing the input corpus between similar API call sequences.

3 Crabtree Overview

We give a high-level overview of Crabtree (as illustrated in Fig. 2) and discuss how we achieve the synergy between synthesis and fuzzing to improve the effectiveness of both. Crabtree takes a template and API data as the inputs and produces bug reports, along with synthesized test cases, annotated with their coverage, and the test input corpus generated by the fuzzer. Crabtree has three main components: a synthesis engine, which is based on SyRust but improves it by guiding the test generation with both coverage and type information, a fuzzer, and a test executor, Miri [22], used to report crashes and memory errors (out-of-bounds behavior) when running tests along with their inputs. We explain each component below.

Template and API data. The template consists of two functions and harnessing code to interface with `cargo test`, the Rust testing toolchain, as shown in Fig. 3. The first function `test` is where the synthesized API call sequences will be inserted. This function takes in inputs that are used by the synthesized code. The second function `fuzz` is the fuzzing harness, which provides the `test` function with inputs. The input to the `fuzz` function consists of the fuzzer-generated bytes, which are converted into arguments of the correct type in the body of `fuzz`. For example, the `fuzz` function in Fig. 1 converts the bytes into two arguments of the type `u8` and `Vec<u16>` respectively (Fig. 1, line 2), before giving them to the `test` function. The template is provided with the Crabtree distribution and can be modified by the analyst to target specific libraries.

```
fn test(x: Input)  {
  // synthesized test goes here
}
fn fuzz(fuzzer_bits: &[u8]) {
  // setup
  test(...);
}
// cargo toolchain boilerplate
```
Fig. 3. High-level structure of Crabtree's template.

The API data includes type specifications of the APIs under test and is automatically generated using the `cargo doc` tool provided by the Rust developers. The second part of the API data documents which traits are implemented for which types; for example, `FixedInt` is implemented for `u8` with associated constant `REQUIRED_SPACE=1` is included there.

API Test Synthesis Guided by Coverage and Type Information. The synthesis engine is responsible for generating API call sequences. Our synthesizer extends SyRust's synthesis technique with the handling of traits and further implements novel techniques that prioritizes generating API call sequences based on new information discovered with testing; specifically, the engine extends previously generated sequences that trigger *new coverage* or use previously *unseen types* (illustrated as the grey box). The details of this synthesis engine will be discussed in Section 4.

Fuzzing and Input Corpus Management. Crabtree's fuzzer component generates the input corpus to the synthesized API call sequence. One of our key observations is that mutating inputs is only effective for certain data types when polymorphic types are involved, e.g., `[u8]` in our example; we call them *fuzzable types*. Fuzzing for other types, such as `[T]`, where `T` is totally unconstrained, is unlikely to improve coverage because the function using this type is agnostic to the exact value of `T`. One exception is `Read`: if we know `T` implements `Read`, then we classify it as fuzzable because `Read` is a trait for abstracting bitstream-like data structure, which a fuzzer can directly generate inputs for by sampling from the space of bitstrings and thus are fuzzable.

Crabtree will identify arguments of the APIs that are of fuzzable types and only generate an input corpus for them. The fuzzer-generated corpus is stored to allow it to be re-used as seeds to the next round of fuzzing or used as inputs by the test executor to identify bugs. Once fuzzing is done, the achieved coverage is measured and recorded, to be provided as feedback to the synthesis engine. Details of input corpus management will be given in Section 4.3.

Test Executor. The test programs generated by the synthesis engine are passed to the test executor, along with their inputs generated by the fuzzer. The test executor runs these programs using Miri [22], an external tool that serves as our bug definition. Any bugs detected will be recorded in a database for analysts to review.

Algorithm 1: High-Level Algorithm of Crabtree.

```
Input  : API type signatures $\mathcal{A}$, code template $\mathcal{T}$, initial seed input corpus $\mathcal{S}$
Output : priority queue of tests $Q$, set of crashes $\mathcal{B}$, set of tests $\mathcal{F}$, set of covered locations $C$
1  Procedure Crabtree($\mathcal{A}, \mathcal{T}, \mathcal{S}$):
2      $(Q, \mathcal{B}, \mathcal{F}, C) \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset)$
3      $Q \leftarrow$ enqueueWithPriority($Q$, ($\mathcal{T}$,$\mathcal{S}$), $(0, \text{utilityLookAhead}(\text{inputs}(\mathcal{T}), \mathcal{A}))$)
4      while $Q \neq \emptyset$ do
5          $(t, i) \leftarrow$ popBest($Q$)
6          for ($t_{new} \in$ growWithSynthesis($t$)) do
7              $(c_+, i_{new}) \leftarrow$ fuzz($t_{new}$, $i$, $C$)
8              $\mathcal{B} \leftarrow \mathcal{B} \cup$ runWithMiri($i_{new}$, $t_{new}$)
9              $u_{new} \leftarrow \text{utilityLookAhead}(\text{inputs}(t_{new}), \mathcal{A} \backslash \text{coveredAPIs}(C))$
10             $Q \leftarrow$ enqueueWithPriority($Q$, $(t_{new}, i_{new})$, $(|c_+|, u_{new})$ )
11             $(\mathcal{F}, C) \leftarrow (\mathcal{F} \cup \{t_{new}\}, C \cup c_+)$
12         end
13     end
14     return $(Q, \mathcal{B}, \mathcal{F}, C)$
```

4 Crabtree Algorithms

In this section, we describe the algorithms implemented by Crabtree and explain how the synthesis engine and the fuzzer interact. We also describe our approach to leveraging coverage and type information to guide the testing.

4.1 Top-Level Algorithm

We summarize the top-level algorithm of Crabtree in Alg. 1. Crabtree takes as inputs: API data $\mathcal{A}$, code template $\mathcal{T}$, and initial inputs $\mathcal{S}$, which are used for seeding the fuzzer. Crabtree returns a set of test cases $\mathcal{F}$, coverage data C, and a set of bug-inducing tests $\mathcal{B} \subseteq \mathcal{F}$ (if any bugs were found). The algorithm also keeps an internal state Q, which is a priority queue of previously tested API sequences, together with their respective inputs generated by the fuzzer. The weight of each element $(\mathcal{T}, \mathcal{S})$ in the queue is a pair of the coverage of executing $\mathcal{T}$ with inputs $\mathcal{S}$ and a integer roughly indicating how far is $\mathcal{T}$ from reaching all APIs (more details later). We also use the following intermediary variables: t_{new} is a test program synthesized by extending a previously generated program t. i_{new} and c_+ are the set of inputs generated and the increase in coverage by fuzzing t_{new} with seed i respectively.

The algorithm first initializes the data structures to empty (line 2). Then, the pair of the template $\mathcal{T}$ and the initial fuzzer inputs $\mathcal{S}$ is placed in the priority queue to seed the generation process (line 3-4). $\mathcal{T}$ contains a Rust testing harness that runs the synthesized test and an empty test case. The first part of the weight for this pair is 0, indicating that an empty program produces zero coverage. The second part of the weight is computed using the utility function $\text{utilityLookAhead}(\text{inputs}(\mathcal{T}), \mathcal{A})$. We describe utilityLookAhead in detail in Section 4.2. At a high level, utilityLookAhead computes the number of library APIs that cannot be called from the context following this test case because they require arguments of types not easily constructed from existing variables. The first argument to the utility function is the set of types for which $\mathcal{T}$ has constructed variables for and we collect them using the function inputs($\mathcal{T}$). For example, in Fig. 1, inputs($\mathcal{T}$) = `{integer: u16, rest:`

`Vec<u16>`}. The second argument is the set of APIs that the test does not call. Here it is the set of all APIs $\mathcal{A}$ in the library under test, as we have not generated anything yet.

The outermost while loop (lines 5-17) iterates over previously generated tests and inputs stored in Q to generate more tests and inputs, until Q is empty. The tests and inputs with the highest priority (i.e., they improve upon coverage or generate new types) are used first (line 6). From Q, we pick the highest priority test case t that roughly translates to t; i has the highest new coverage and produces a context that enables the highest number of APIs to be called, compared to other pairs in the queue. Leveraging synthesis, we grow this test case to produce a set of new tests. Each new test t_{new} adds one more line to t while still being well-typed with respect to Rust compiler type checking (line 7). The inner `for` loop iterates over the set of new tests. For each t_{new}, reports coverage and produces a new input corpus set i_{new} (line 8). To detect bugs, we run Miri on the test cases with the fuzzer-generated inputs, including crash-inducing inputs, and record any bugs found (line 9). We then compute the utility score u_{new} for the current test t_{new} and the set of APIs that have not been reached by the current set of tests. Here $\mathsf{coveredAPIs}(C)$ returns the set of APIs that have been called given the current coverage so far. The test with a weight based on coverage and potential to reach new APIs is enqueued (lines 11-13). We also update the C and $\mathcal{F}$ accordingly (lines 14-15)

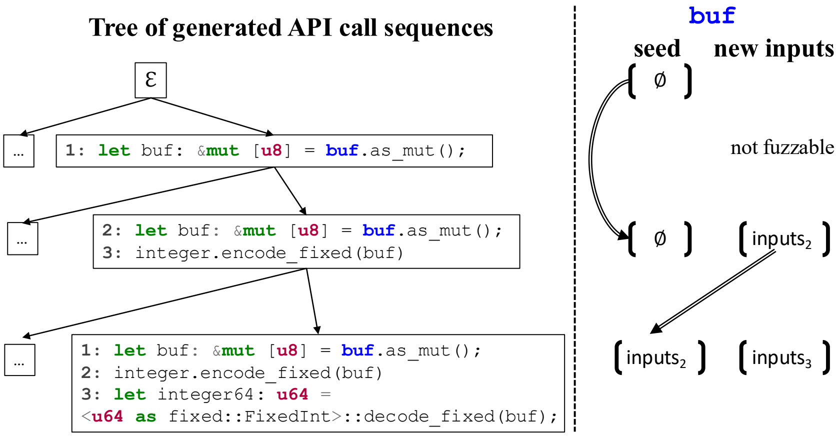
4.2 Synthesis Algorithm

Our synthesis algorithm (`growWithSynthesis`) builds on SyRust's synthesis algorithm, which in turn builds on prior work [13, 20, 41] that addresses synthesizing well-typed programs. We omit the details on the constraints that produce well-typed Rust programs here and instead focus on Crabtree's addition to the synthesis algorithm—the algorithm that guides the synthesis engine towards more useful test cases, i.e., synthesized tests that cover more APIs and code under test and produce more previously-unseen types.

Systematically generating new well-typed programs of the form $L_1; \ldots; L_n$, for each n and testing them can be prohibitively expensive. It takes our test executor Miri around 5 seconds to run one test case. Furthermore, fuzzing the inputs for each test parameter using coverage-guided fuzzers [50] needs to apply many mutations to each input to make progress.

To speed up test generation, we propose to synthesize *extensions* of selected previously generated programs, instead of generating each new test sequence from scratch. To this end, we represent the sequence of generated API calls in a hierarchical structure (a tree), as shown on the left of Fig. 4. The root of the tree is an empty test case, with each layer consisting of programs of a given length. A node's parent is the program without the last line in the node's program, and its children are programs extended by exactly one API call. The synthesis algorithm will extend the tree, by adding new lines to existing programs. The synthesis is *guided* by coverage and type information, to efficiently search for diverse sequences that increase testing coverage and uncover new types.

Using Coverage to Guide Synthesis. Recall the example in Fig. 4. Suppose a program consists of lines 1 and 2 and running that program, Crabtree achieves coverage gains in the library code. Crabtree notices the coverage gain and prioritizes using lines 1 and 2 as a prefix for generating longer programs, over prefixes that do not have coverage gains. Generally, for each program $P = L_1; \ldots; L_n$ representing a test case that increased coverage, we generate new programs P' with the constraint $P' = P; L_{n+1}$. Here coverage can be line or branch coverage. This heuristic is not guaranteed to obtain the maximum coverage (achievable for a sequence length), as it can get stuck in local maxima where what were initially good API calls (in terms of achieving high coverage) may force the tool down an unproductive path, preventing longer API call sequences to form. Next, we explain how to use types to alleviate this problem.



```
1: let buf: &mut [u8] = buf.as_mut();
```

```
2: let buf: &mut [u8] = buf.as_mut();
3: integer.encode_fixed(buf)
```

```
1: let buf: &mut [u8] = buf.as_mut();
2: integer.encode_fixed(buf)
3: let integer64: u64 =
<u64 as fixed::FixedInt>::decode_fixed(buf);
```

Fig. 4. A tree representation of test cases and hierarchical input storage.

Using Type Information to Guide Synthesis. Coverage guidance may penalize using APIs that do not increase coverage, but are necessary to enable subsequent coverage-increasing API calls to be generated. For instance, APIs such as `as_ref` and language primitives such as `&mut` do not produce coverage gains, but are required to produce variables of the right type needed to call many other APIs. In our example from Fig. 4, line 1 produces a previously unknown type `&mut [u8]` that allows more APIs to be called. Similar effects can also be observed with other APIs, not just using the above mentioned language primitives. To overcome this limitation, we should also favor programs that produce a previously unseen type, and guide the synthesis engine towards using this type. In our example, the test case consisting of just line 1 should also be selected for extension.

We define a utility function, which takes as inputs, a set of types, for which the current test can generate variables for, and a set of APIs that Crabtree has not generated any call to, and returns a measure indicating how many APIs have input types for which Crabtree cannot generate variables and therefore Crabtree cannot generate programs to call those APIs. This number decreases when a new type is found, and therefore a lower number corresponds to higher priority in the queue. A first draft of the utility function is as follows:

$$\mathsf{utilityLookAhead}(T, A) = |\{a \in A \mid \mathsf{inputsTypes}(a) \not\subseteq T\}|$$

Here $\mathsf{inputsTypes}(a)$ collect the set of a's argument types.

While the number of previously uncovered APIs that can be called after this test is a reasonable heuristic, it may not be optimal. For example, the utility of a newly found type would be low if we only look at that type and not references to it because most Rust APIs take references for ergonomic reasons (no ownership movement). Therefore, a more refined heuristic is to compute a set of types reachable from the original set by using borrowing alone. We refer to this as the *borrow-aware closure* over mutable and static references of the types available. We are careful to avoid semantic-violating cases like making two mutable references from the same non-copy type. The utility function Crabtree uses is as follows:

$$\mathsf{utilityLookAhead}(T, A) = \min_{b \in \mathsf{closure}(T)} (|b \setminus T| + |\{a \in A \mid \mathsf{inputsTypes}(a) \not\subseteq b\}|)$$

Here closure takes T, a multiset of types and produces a set of Rust semantic-aware closure over `&` and `& mut` of T. The closure is finite because the Rust type system converts `&&` to `&` and similar conversions hold for mutable references; mixed combinations are not permitted. The closure of the type context with a single non-reference type `String` is the set of type contexts of size 3. The first element is {`String` }, the second element is {`String`, `&mut String` }, and the third is {`String`, `& String` }. The context {`&mut String`, `&mut String` } is not permitted because that violates the Rust typing rule that there is at most one mutable reference. $|b \setminus T|$ is the number of borrowing operations required to get to this closure element b because each *new* reference type in b is made by a borrowing operation. $\{a \in A | \mathsf{inputTypes}(a) \not\subseteq b\}$ is the set of APIs not reachable from this new closure element.

While in theory this lookahead utility function can use APIs with more complex semantics than borrowing to compute reachable types, this will increase the time required to compute the heuristics because the number of APIs used in enumeration will grow. We posit that borrowing is a reasonable subset since reference types occur very frequently in APIs for ergonomic reasons.

4.3 Input Corpus Generation and Management

We now describe how we fuzz Rust APIs by mutating inputs to the API sequences we generate. Since the input types of the test program are given in the template, we fuzz the API sequence by fuzzing the template. We focus on two key challenges encountered when fuzzing Rust APIs. First, since our approach handles polymorphic APIs, we must figure out which APIs are worth fuzzing even when they are polymorphic and their underlying behavior is not known. In particular, the behavior of some polymorphic functions might be completely independent of the value of input. For example, the behavior of pushing an element to a vector is the same for all values of the element to be pushed (that's also why the push function is polymorphic). Second, since many API call sequences are similar to each other, we must determine how to effectively share the input corpus between API sequences. To address the first challenge, we leverage the Rust type and trait system to guide our decision of when to fuzz. Our insight is that, in Rust, even fully polymorphic inputs might be worth fuzzing if the input polymorphic type implements traits that indicate this type behaves like a binary buffer, so a fuzzer can directly generate input for it without knowing its concrete type. For the second challenge, we employ a hierarchical input corpus management that allows reuse of the input corpus across iterations for test cases that are deemed similar, i.e., share the same prefix.

Fuzzable Types and Traits. We use an example to illustrate how to use Rust's trait system to identify which APIs might be worth fuzzing. Consider line 1 and 2 from our example in Fig. 4. Lines 1 and 2 have API calls with the following types respectively:

- `as_mut: (S: std::convert::AsMut⟨T⟩) →(&mut T)`
- `encode_fixed: (T: fixed::FixedInt⟨...⟩, dst: &mut [u8])→()`
  where `fixed::FixedInt⟨...⟩` is a sub-trait of `std::io::Read`

A lightweight static way of distinguishing between the two programs above is to leverage the Rust trait system. In the first program, the input is passed to an API that takes an input implementing the trait `std::convert::AsMut⟨T⟩`. Given this trait annotation, the implementation of `as_mut` cannot treat the type in a more specific way than generating a mutable reference. `as_mut`'s behavior is not affected by changing its input, as it simply converts a reference to a mutable reference for type checking purposes. As such, there is no point in fuzzing `as_mut` function. So a program consisting of only line 1 will not be fuzzed and existing seeds will be run.

The second API takes in a value of a type that implements `fixed::FixedInt⟨...⟩`. Since this trait is a sub-trait of `std::io::Read`, we know the implementation can treat this input as a binary-buffer and fuzzing those bytes can induce different program behavior, even though we do not know the underlying structure of the type. We characterize such types as "fuzzable". Thus, a program consisting of lines 1 and 2 should be marked for fuzzing on variable `rest: Vec<u16>`. This fuzzable characterization applies to non-polymorphic APIs as well; types for which fuzzer knows how to generate inputs for are fuzzable.

Our algorithm relies on an analyst-provided *allow-list* of fuzzable types and traits. For example, the `Read` trait is fuzzable, as explained above. While we construct this list manually by reviewing the standard library, it can be generated automatically or added by the analyst if it is amenable to direct mutation by the fuzzer, such as providing an iterator over `u8`.

Hierarchical Input Corpus Management. Even with restrictions to fuzzable types and traits, the fuzzer is still left with a huge number of programs to fuzz. In particular, as the API call sequence grows longer by the synthesis algorithm, additional fuzzable API calls may appear. Recall the 3-line example from Fig. 4. Our algorithm constructs this program incrementally from prefixes, so it sees the program consisting of line 1 and a program consisting of lines 1 and 2 before it sees the final 3-line program. Since fuzzing all the prefixes does not scale, Crabtree re-uses the input corpus for the 2-line program to run the 3-line program. The details of this process are given below. Consider the 2-line (lines 1-2) and 3-line program (lines 1-3) from our example. Fuzzing is applied to the 2-line program, generating a corpus of inputs that tests this 2-line program. Given that the 3-line program shares the first 2 lines with the 2-line program, it would be a waste to restart over from the seeds. Instead, we use the fuzzer-generated corpus from the 2-line program as the seeds to the fuzzing of the 3-line program. This is illustrated on the right side of Fig. 4. Suppose fuzzing the 2-line program generates the corpus `HELLO`, `WORD`. The first input `HELLO` executes successfully, and the second input `WORD` crashes the program.

When fuzzing a 3-line program that extends the 2-line one, we need not start with a naive seed. Instead, we hot start the fuzzing by carrying over the input corpus of the 2-line prefix. We filter out crashing inputs that do not reach the new line, giving us the seed corpus of `HELLO`. This allows us to prune the corpus as we move down the tree, and the number of leaves becomes numerous. Pruning is especially effective for parsers of structured data like JSON or HTML because most random inputs will fail the initial parse API call. These parser-like functions are worth fuzzing, but fuzzing an extended API calls with parser calls in the first few lines will waste effort as most inputs will not pass the parser line or cause the program after the parser line to be executed. If at any point, the fuzzer fails to generate new inputs that reach beyond that line without crashing, then we stop fuzzing and re-play the current corpus set.

5 Traits in Synthesis

Given the prevalence of traits in data structures and other libraries, Crabtree needs to model traits precisely for effective testing. Prior works like SyRust [41] ignored traits and treated type variables as unconstrained, leading SyRust to generate a large number of programs with trait errors.

We propose to explicitly model the trait semantics of Rust to avoid this expensive error-guided loop and synthesize tests containing calls to trait functions implemented in the library under test. Our model captures the semantics of Rust traits with concrete associated traits and constants, with generics allowed as parameters to traits (`Trait<A>`) and as implementation of traits on generic types (`impl Trait<A> for B`). To fully incorporate traits into our synthesis algorithm, we need to solve two problems: first, how to make the synthesis algorithm aware of the set of types that implement traits required by the inputs of an API such that the API can be called with well-typed arguments; and second, how to propagate concretized types, including associated types within traits to the return type of an API call, such that the returned results can be effectively used later as arguments to other APIs. Next, we describe how we solve the above two problems.

Type and Trait Matching. Rust's trait system is multi-layered and hierarchical: traits can be parameterized by type variables and traits can have sub-traits and sup-traits. Briefly, Rust permits implementations of one trait on another, making the latter a sub-trait. For example, the `FixedInt` trait is a sub-trait of `Read` and any type implementing `FixedInt` automatically implements `Read`. As part of its input, Crabtree receives a database recording which types implement which traits along with the API type specifications. We generate the trait implementation database by extracting trait implementation information from the Rust compiler. This database includes traits that appear in the library under test and any libraries chosen by the analyst, such as the Rust standard library. The type/trait pairs in the database are organized hierarchically, based on subtype and subtrait relations. Concretely, the database records the lattice hierarchy of traits implementing one on the other and traits implementing on types. For example, a database representing the code from Fig. 1 would contain the following data `[(FixedInt<>, u8), (Read, _: FixedInt)]`, meaning `u8` implements `FixedInt` and any type implementing `FixedInt` implements the `Read` trait. We can query this database and ask whether `u8` implements trait `Read`; we would first match the first tuple to derive `u8` implements `FixedInt`, and then using this information, we can match the second tuple to conclude that this is true: `u8` implements `Read`. Now using this database, the synthesis algorithm first checks that the trait is implemented on a type, before a variable of that type is used as input to a function that requires a trait to be implemented on the input type.

Consider the `encode_fixed` function which has a type signature of `(T : FixedInt, &mut [u8])` → `()`. It writes an integer into a `u8` buffer. Next, we show how Crabtree decides whether a variable of the type `u8` can be used as the first argument to call `encode_fixed` on line 8 of Fig. 1. The argument type is constrained with a trait `FixedInt`. We need to check that `FixedInt` is implemented for `u8`. We query the database of trait implementations, and the result shows that the check passes.

While we find the exact match in this case, our matching algorithm can handle the hierarchy of types and traits and permits a variable whose type implements a subtrait (e.g., `FixedInt`) of the trait (e.g., `Read`) required by the function type to be used to call the function.

The Rust type system permits negative trait implementations as well. These trait constraints state that a given trait *cannot* be implemented for a given type. An example use is to negate the `Sync` trait indicating a type cannot be sent across threads, preventing it from being passed to functions that assume a thread-safe object. The database of trait implementations also contains negative trait information and is checked by our matching algorithm.

Encoding Trait Semantics to SAT. We encode the matching algorithm sketched above as constraints in SAT. So far, we have only discussed how to match one input variable to one input type. However, to generate a well-typed API call, we need to consider all the variables and argument types in the API call. In particular, if the same variables appear as two different inputs to the same function (e.g., `f(x,x)`), their types must be unified to compatible inputs. SyRust already considers these types of compatibility constraints without traits, which Crabtree builds on to include additional constraints resulted from trait constraints. First, our trait system models the additional typing constraints required by trait matching. This is done by adding constraints imposed by traits to the output of unify. Once this is done, the consistency rules of SyRust will automatically enforce consistent assignment to type variables regardless of whether they occur in traits or types.

Consider the function `write_to : (T:FixedInt, [u8;T::REQUIRED_SPACE])→ ()` that writes the first input to a finite array. This requires that the second input's size match the associated constant of the first input's trait implementation. We add these constraints to the output of unify, and prevent cases like matching `u8`, which has `REQUIRED_SPACE=1`, with `[u8;3]`. Like with type variables, we add compatibility constraints with an implication: if Crabtree chooses the former variable as input, then the it cannot choose the latter in the same solution as input.

Output Type Resolution. While checking function inputs to ensure trait constraints are met is sufficient for generating a well-typed API call, it is insufficient for generating longer API sequences that uses the result of a API call if the API's return type is polymorphic. SyRust's synthesis algorithm performs error-guided refinement of API specifications that creates concretized forks of polymorphic APIs so that the output of polymorphic functions can be used later. For example, `encode_fixed` will turn into `encode_fixed_1` : `(u8, &[u8])` → `()` and `encode_fixed_2` : `(T, &[u8])` → `()` with metadata that prevents `T` from unifying with `u8`. However, without considering trait constraints, there will be too many possible types to be used to create a fork. As a result, this method would generate too many forks, making it unscalable.

We extend this algorithm to trait-polymorphic functions so the forks are meaningful; i.e., they are more likely to satisfy the trait constraints. Of particular interest are functions whose output types are parameterized by trait-associated types (e.g., `FixedInt::ENCODED` from Fig. 1) and constants (`REQUIRED_SPACE`). The associated types and constants need to become concrete as early as possible so they can be used to call more APIs. Consider the following function:

$$\texttt{to\_array} : (\texttt{T} : \texttt{fixed} :: \texttt{FixedInt}) \longrightarrow [\texttt{u8}; \texttt{T} :: \texttt{REQUIRED\_SPACE}]$$

It returns a fixed-size array of the exact-required size containing the encoding of the input. Observe that the value of `T::REQUIRED_SPACE` depends on what T is. If T = u8, our concrete fork will return [u8; 1]. While it is possible to fork and set the input type to `u8`, this refinement is too restrictive. We concretize inputs only when it is required to determine the output. Instead, the forked API's input type will be `T`: `fixed::FixedInt` ⟨`REQUIRED_SPACE=1` ⟩, not `u8`. This allows us to represent all types that implement `fixed::FixedInt` with associated constant `REQUIRED_SPACE=1` rather than having one fork for every type.

Validity of the Rules. To the best of our knowledge, the trait matching rules that we implemented, as sketched out in this sections, are identical to those used by the Rust type checker. And thus, for well-typed Rust API invocation in our syntax, the rules would allow the program to be synthesized. On the other direction, all the synthesized programs will not violate trait requirements by the Rust type checker. This excludes unsupported features, whose full list can be found in Section 8.

6 Synthesizing Calls to Higher-Order Functions

We present a synthesis strategy for calling higher-order functions and synthesizing function closures as inputs to them, leveraging the trait model presented in the previous section. There are three challenges to synthesizing code calling higher-order functions.

Boilerplate API Chains. A call to a higher-order function often involves a boilerplate consisting of several API calls. Taking iterator APIs as an example: first, an iterator `Iter<A>` is created, then the higher-order function is applied to the iterator, and finally, the `collect` method is called on the iterator and create the final result from the results of evaluating the iterator. If Crabtree tried to synthesize this sequence, it would waste time searching through API combinations that do not form the above mentioned call chain.

Synthesizing Programs Beyond API call sequences. Function closures break the assumption that the synthesized program is a sequence of API calls, which SyRust and RULF rely on. Crabtree needs to relax the syntax of the programs to be synthesized to make it possible to generate both the closure body needed by higher-order functions and the API call sequences.

Typing Constraints. For some functions like `map` and `filter`, the closure they take as inputs needs to return a particular type. We need to make the synthesis algorithm aware of the internal types of iterators and outputs of maps over data structures. Next, we explain how `map` works, as it produces outputs of a type different from its inputs. `filter_map` and `from_fn` can be similarly encoded.

6.1 API and Syntax for Higher-Order Functions

To reduce the complexity of generating the call-chain boilerplate described above, we model `map` and related API call-chain boilerplate as a single API. `map` has the type signature `Iter<A>` $\rightarrow$ `(F : Fn(A)` $\rightarrow$ `B)` $\rightarrow$ `Iter<B>`. Since directly synthesizing the boilerplate is challenging (the space of 3-API sequences is large), we model this 3-API sequence as one polymorphic API (`map_collect`) shown in Fig 5 to avoid searching through the space.

The API `map_collect` takes iterable data structures of type `C` and a function of type `F` as input and returns a value of type `D`. To ensure that an iterator over type `A` can be generated from `C`, we constrain it with a trait annotation `IntoIterator<Item = A>` (line 4). Similarly, to ensure that the output type `D` can be created from an iterator over `B`, `D` is annotated with trait `FromIterator<B>` (line 5). Leveraging the Rust traits `IntoIterator` and `FromIterator` allows Crabtree to reuse the trait models and generate well-typed higher-order API calls by using this single API encompassing the boilerplate code directly.

```
1  fn <
2    A,B,
3    F: Fn(A) -> B,
4    C: IntoIterator<Item = A>,
5    D: FromIterator<B>,
6    >
7    map_collect(c: C, f: F) -> D {
8      c.into_iter().map(f).collect()
9   }
```

Fig. 5. `map_collect` API modeling the boilerplate at once.

The syntax of the programs synthesized by Crabtree is modified to allow synthesis of function closures. The `move` keyword indicates that this closure will capture variables instead of just borrowing them.

$$\textit{Line} \quad L \quad := \quad \ldots \mid f(V, |v_1|\{P\}) \mid f(V, \texttt{move}\ |v_1|\{P\})$$

6.2 Synthesizing the Closure Body

Given the syntax around higher-order functions, we provide a modification to the synthesis algorithm that allows it to generate bodies for function closures. Our algorithm addresses the challenge of incrementally synthesizing a program that can grow at many points with additional typing constraints inside the closure. Our insight is that the closure body can itself be modeled as a API call sequence, nested within another sequence, as long as we are careful about ownership and reference lifetimes. In particular, we can annotate each variable with a scope number so variables used within a closure can be removed from the context when the closure ends. To encode the typing constraint that the closure has to satisfy, we keep track of the required output (return) type of the closure while synthesizing the body and avoid growing the program outside the closure until this output type is reached.

Recall the higher-order function call on line 6 of Fig 1. This function call with intermediate type contexts seen by Crabtree is given in Fig 6. At each line, we show a pair of the type context modeled by Crabtree and the goal type. For example, in line 3, the context is `{integer_0 : u16, rest_0 : Vec<u16>}` and the goal type is `()` at the start. The goal type is the type that this scope (lines 5 to 8) is required to produce, `u8` in our case. Each of the variables ends with a number indicating the scope that the variable belongs to. Variables created in the main test scope will end with `_0` while those inside the closure end with `_1`. Crabtree synthesizes this program top-down. The first line is synthesized normally and the goal type is set to the unit type `()` since we do not have any typing constraints for the test function. In the second program line (line 4), the synthesizer calls `map_collect`, our syntactic sugar for map and collect functions. Now the following line will be inside the function closure. Inside the closure, `rest` is removed from the context because Rust typing rules prevent us from moving ownership while `elem` is available because that is the

```
 1  fn test(integer: u16, rest: Vec<u16>) {
 2  // context =\{...\}, goal-type = t
 3  // ({integer_0 : u16, rest_0 : Vec<u16>}, ())
 4    let mut vecint Vec<u8> = rest.iter().map(|elem : &u16| {
 5  // ({integer_0 : u16, rest_ref_0 : &Vec<u16>, elem_1 : &u16}, u8)
 6            let bytes : [u8;2] = elem.to_be_bytes();
 7  // ({integer_0 : u16, rest_ref_0 : &Vec<u16>, elem_1 : &u16, bytes_1: [u8;2]}, u8)
 8            bytes[0]
 9        }).collect();
10  // ({integer_0 : u16, rest_0 : Vec<u16>, vecint_0: Vec<u8>}, ())
11  }
```

Fig. 6. Test function exposing a bug from `integer-encoding`. The body of the closure is expanded for clarity. Each comment (blue lines starting with "//") is the context and goal type when synthesizing that line. Each variable is annotated with a scope number $\_i$. Goal type is set to the unit type () when outside the closure.

iterator element that the closure is running on. `elem` is marked with scope number 1 since it exists only inside the closure.

At this point, Crabtree only grows the program inside the closure and avoids growing more lines after the call to `map_collect`. This prunes the search space, and is sound because the program execution will never reach beyond the call to `map` unless a well-typed, non-panic expression is the body of the closure. The goal type inside the closure changes to `u8`, the required output type inferred from the type signature of the higher-order function (derived from type instantiations of `map_collect`). Variables created inside the closure like `bytes_1` are also marked with `_1` to be deleted when the closure body's scope ends. Synthesis continues until the goal type `u8` is in the context. Then Crabtree chooses whether to end the closure or continue inside.

6.3 Computing the Post-Call Context

Once Crabtree decides to end the closure, it needs to compute the context after the higher-order function call. The resulting context is the last context of Fig. 6 (line 10). The return type of the higher-order function `Vec<u8>` is derived from the type signature while variables created inside the closure such as `elem_1` and `bytes_1` are removed. Finally, the goal type reverts back to the unit type.

For other closure variants, Crabtree removes more variables after the closure scope closes to obey ownership typing rules of Rust closures. `move |...|` closures allow Rust programmers to move ownership of variables into the closure. Therefore, after an ownership-moving operation is done in the body of the closure, the variable's scope number needs to be that of closure's body's scope. The same procedure will remove this moved variable when the closure scope ends.

7 Implementation and Evaluation

We implemented Crabtree and evaluated its performance to answer the following research questions.

- **RQ1**: How effective is Crabtree at testing and finding bugs in realistic benchmarks?
- **RQ2**: How does Crabtree perform against recent prior works?
- **RQ3**: How do each of Crabtree's features contribute to the effectiveness of testing?

7.1 Implementation

Crabtree's synthesis engine is implemented with 23k lines of Rust. It uses Z3 [10] as the SAT solver during test synthesis. The fuzzing module is 200 lines of Rust on top of AFL.RS, a Rust port for the AFL fuzzer [50]. The test executor is based on Miri [43], the same as SyRust [41], but modified to measure coverage using `grcov` [29].

7.2 Experiment Setup

Benchmarks. Our benchmarks consist of 30 libraries selected from prior work and recent versions of popular libraries. We randomly picked 10 libraries from SyRust's benchmark set, 10 from RULF's benchmarks, and sampled 10 recently updated libraries with at least 100 stars or 100k downloads from `crates.io` [3] for a total of 30 libraries. Benchmarks from prior work were sampled due to computational constraints of running each benchmark.

We could not run SyRust and RULF on benchmarks other than their own because both tools rely on a modified Rust compiler that lack features needed to run other benchmarks. Crabtree is run on all 30 libraries. The benchmark configurations are given in Table 2. The first column displays the names of the tools. The next three columns indicate which sets of libraries were tested by which tool: ✓ means all libraries in that dataset were tested by that tool and "×" means that none of the libraries in that dataset were tested by that tool. The next column shows the time the tool spent on testing each library in hours. The following column indicates the number of APIs in each library selected for testing.

Table 2. Tool-Benchmark configurations for the evaluation.

	`crates.io` Set	SyRust Set	RULF Set	Hours per Library	# APIs tested per Library
Crabtree	✓	✓	✓	24	Full
SyRust	×	✓	×	10	15
RULF	×	×	✓	24	Full

Tool Inputs. We discuss how we selected the inputs for each tool for our evaluation.

- **SyRust**. To run SyRust on its own benchmarks, we use the configurations that ship with the SyRust artifact. These configuration files contain 15 author-picked APIs and an author-written template. These templates initialize variables that have types that the library APIs take as inputs, but do not provide as output. Without these variables, SyRust cannot synthesize well-typed API call sequences. While we considered passing a larger set of APIs to SyRust, we ended up using the original 15 due to the difficulty of running their API collection tool and a large number of SAT timeouts encountered when running SyRust on the full API set.
- **RULF**. RULF only takes Rust code as input. To run RULF, we modified its docker container to pin the Rust registry to November, 7 2021. RULF collects a set of inputs as seeds for each API call sequence it generates. Similar to SyRust, RULF also needs to provide the API call sequences with input variables of the correct type. RULF only generates a very restricted set of simple input types (i.e., `[u8], & str`). It implements a string literal collecting script [2], which scrapes the library under test for strings to seed the fuzzer with.
- **Crabtree**. Crabtree takes API type specification and trait data as input. We pass the entire API set of a library under test. For each library, we provide a template (Fig. 3) to be filled by Crabtree, similar to SyRust. When running Crabtree on SyRust benchmarks, we use the same template as SyRust. For RULF's benchmarks, Crabtree tests each library with a template generated by manually selecting the input types also used by RULF. We start Crabtree with the same seeds as RULF, by using the string literal collecting script provided by RULF. This provides a fair comparison against RULF. Finally, for the libraries picked from `crates.io`, we manually wrote the template by inspecting the APIs, collecting primitive types used by the APIs as inputs types, and providing variables of these collected types and arrays of these types as the starting point of synthesis.

The effort of constructing these templates is similar to that of writing a fuzz harness but without the need to explicitly call APIs under test. While we do not share templates between multiple libraries, sharing would be reasonable among libraries that have APIs with similar input types like an HTML parser and a JSON parser.

Environment Setup. We ran our experiments on a cluster consisting of 100 cores with 56 cores allocated to running Miri, 32 to fuzzing, and the remaining 12 to a database used to share state across machines. Crabtree is run for up to 250k tests. This number is based on a preliminary run where we observe coverage increases for libraries up to 220k tests. To answer RQ2, we use SyRust and RULF with benchmarks from the respective papers as the baseline. We run both tools as described in their respective papers: RULF for 24 hours and SyRust for 10 hours, both in wall-clock time. We choose the author-specified time limits that saturate the tool's performance instead of normalizing between tools because SyRust and RULF already differ in their compute requirements with SyRust supporting multi-machine setups and RULF scheduling only on a single machine. While most of the results can be replicated using a 10-hour run, Crabtree still generates tests that increase coverage up to the 15^{th} hour in some libraries like `http` where input structures are complex. The environment is set up to collect line and branch coverage information.

Tool setup Crabtree dynamically decides when to fuzz a synthesized test case. How long this fuzzing is done is statically set by the analyst. Our evaluation is run with a 5-minute constant fuzzer timeout. While this is too low for many benchmarks, a higher fuzzing time was not computationally feasible given the number of synthesized test cases. We will discuss the impact of our fuzzer time setup later in this section. To replicate RULF's experiment, its fuzzing is run for 24 hours. RULF's test generation is done before fuzzing, which takes only seconds.

Evaluation Metric. The primary metric for evaluating the tools is coverage. We use 3 different coverage metrics: API, line, and branch. API coverage is the set of APIs called. Line and branch coverage correspond to fragments of the source code under test. API coverage is computed from the public functions directly called by the respective tools while line and branch coverage is computed via `grcov`. When comparing SyRust and Crabtree, we also use the metric of coverage over time.

We do not directly compare real bugs found by these tools because they report different types of bugs and it is hard to compare them directly. For instance, SyRust and Crabtree use Miri [43] to report bugs and ignore unwrap failures due to a high false positive rate, while RULF also considers unwrap failures as bugs as well. To compare bugs found, we use the results of mutation testing where we use `mutants.rs` [42] to synthetically inject bugs into the library under test and measure the number of synthetic bugs each tool caught. Bugs generated by `mutants.rs` usually replace values with default values of the same type and may not cause incorrect memory behavior (out-of-bound behavior) such as double freeing in unsafe code or cause crashes that all three tools target. These bugs are best found by adding custom assertions on the outputs of APIs under test, which none of the above tools do out of the box. Therefore, one would expect only a fraction of the bugs to be found by these tools in the evaluation. Nonetheless, this experiment provides comparison between these tools regarding their bug finding capabilities when the bugs are not known a priori.

7.3 RQ1: Effectiveness in Testing

We evaluate Crabtree's effectiveness in testing libraries using the benchmark defined in the previous section. We run Crabtree for 250K test cases on the chosen libraries. The results of running Crabtree on the selected libraries are shown in Table 3. Benchmarks 1-10 are taken from SyRust; benchmarks 11-20 are from RULF, and the final set is benchmarks 21-30 are sampled from `crates.io`. We provide a detailed comparison with SyRust and RULF in the next section. Looking at Crabtree

Table 3. Crabtree's API/line/branch/mutant coverages on Rust libraries. Crabtree, SyRust, and RULF are abbreviated as CT, SR, and RU respectively. "-" denotes that tool was not tested on this benchmark. When one tool is strictly better than others at a particular benchmark and metric combination, it is marked in red.

		API				Line				Branch				Mutant			
		Total	CT	SR	RU	Total	CT	SR	RU	Total	CT	SR	RU	Total	CT	SR	RU
1	bitvec	235	112	14	-	1420	680	389	-	710	322	156	-	588	356	229	-
2	crossbeam	44	22	9	-	3450	203	162	-	5946	262	188	-	0	0	0	-
3	csv_core	202	11	9	-	1329	254	175	-	1957	112	70	-	378	27	25	-
4	encoding_rs	22	9	8	-	8244	686	233	-	7280	691	318	-	4205	814	305	-
5	url_encoding	5	2	2	-	100	23	23	-	97	23	23	-	35	32	32	-
6	bstr	72	11	4	-	3009	85	24	-	2050	100	0	-	298	60	9	-
7	bytes	99	31	14	-	1277	465	245	-	594	140	82	-	288	174	105	-
8	dashmap	149	15	4	-	693	98	83	-	482	134	134	-	210	51	48	-
9	num_rational	65	15	13	-	1430	160	115	-	3822	1462	516	-	616	187	124	-
10	smallvec	96	37	4	-	1123	287	187	-	1646	1123	995	-	359	139	80	-
11	url	76	39	-	67	2128	807	-	796	1878	265	-	265	223	134	-	134
12	http	211	113	-	26	3082	578	-	462	6616	273	-	4114	627	118	-	63
13	clap	194	32	-	66	5655	962	-	853	6271	434	-	342	733	209	-	198
14	serde_json	125	34	-	32	3027	304	-	217	2534	273	-	91	1009	103	-	48
15	bat	146	19	-	25	2134	537	-	479	1450	224	-	193	209	100	-	93
16	flate2	214	20	-	21	16603	114	-	105	8912	12	-	12	378	29	-	27
17	tui	259	70	-	70	2927	314	-	165	1614	143	-	95	340	63	-	33
18	xi_core_lib	244	41	-	58	7914	419	-	321	19235	167	-	118	707	84	-	64
19	regex	156	51	-	96	4293	1884	-	2920	2504	674	-	1416	587	293	-	289
20	regex_syntax	210	18	-	102	9510	276	-	1981	10178	109	-	1663	606	61	-	344
21	fixedbitset	58	15	-	-	962	90	-	-	845	29	-	-	64	20	-	-
22	integer_enc...	11	6	-	-	386	37	-	-	369	30	-	-	24	3	-	-
23	leapfrog	77	3	-	-	708	104	-	-	421	86	-	-	180	16	-	-
24	oxidebpf	35	5	-	-	2030	13	-	-	1956	0	-	-	171	3	-	-
25	roaring_rs	110	33	-	-	3251	304	-	-	2312	103	-	-	475	94	-	-
26	slab	47	15	-	-	756	66	-	-	928	60	-	-	67	17	-	-
27	sparsey	114	30	-	-	1597	590	-	-	911	219	-	-	388	125	-	-
28	triomphe	107	16	-	-	362	77	-	-	263	36	-	-	128	19	-	-
29	ubyte	45	17	-	-	288	28	-	-	474	26	-	-	25	3	-	-
30	sharded_slab	40	9	-	-	615	441	-	-	479	366	-	-	268	90	-	-

alone, in the best case, it is able to call almost half of the APIs in `bitvec` and cover 48% of the lines. However, there are some cases where Crabtree only covered 5% of the APIs and 0.6% of the lines. After manually examining selected libraries, the causes of the coverage variation that we identified is the lack of variables in the template that Crabtree can use to call APIs. The ratio of bugs caught over injected is similar to the ratio of covered code over the entire code base. The exception to this is `regex` where the ratio of bugs caught is much higher than that of covered code. This is because the mutations concentrated on trait functions that are covered by tests generated by Crabtree. Overall, only a small fraction of the bugs injected are caught by any of the tools, due to lack of application specific assertions checking output values of the APIs. The impact is most obvious for string- and byte-related libraries like `csv-core`, `encoding_rs`, `serde_json` where the default value of empty byte or string rarely crashes the test cases or leads to unsafe memory behavior on subsequent API calls. For `crossbeam`, `mutants.rs` reported zero mutations possible using their list of possible code mutations, because supported data types by `mutants.rs` are absent from `crossbeam`.

Crabtree identified 4 novel bugs in open-source libraries from the `crates.io` benchmarks. We describe each bug and provide the test that triggered the bug below.

- leapfrog, a nested hashmap library had an out-of-bounds read when inserting the pair consisting of the same elements. The issue is caused by the library incorrectly computing the buffer size for a bucket object in the HashMap. The HashMap data structure is polymorphic and with_capacity has a trait constraint: Hashable requirement for keys types of the hashmap. While SyRust could eventually generate this, it can only do so after trying at least 2^n ill-typed test cases where n is the number of available input types. This is because SyRust blindly uses input types to instantiate polymorphic type constructors like with_capacity without understanding traits.

```rust
fn test(i_0: usize, i_1: u64) {
  let mut x: HashMap<usize,usize> = HashMap::with_capacity(i_0);
  let y: &mut HashMap<usize,usize> = &mut x;
  y.insert(i_1,i_1);
}
```

- sparsey had an issue where a size-increasing operation during insertion performs a Rust memory swapping operation copy_nonoverlapping on overlapping data, which violates the safety invariant. The final function create requires that the input i_1 be of a type that implements the ComponentSet trait. This bug took the longest to find, at 45 minutes.

```rust
/// A,B,C are empty structs ()
fn test( i_0: sparsey::Layout, i_1 : (A,B,C)) {
  let z = &i_0;
  let mut x: World = World::with_layout(z);
  let y: &mut World = &mut x;
  let _: Entity = y.create(i_1);
}
```

- integer-encoding had a pointer-size mismatch bug when decoding into smaller than expected buffers. The code to induce this is shown in Fig. 1. This bug requires both traits and fuzzing to trigger; it needs the API to be called by an array of a particular length through trait API calls.
- oxidebpf, an eBPF-based packet-processing library had unaligned memory access during a conversion between a bit slice and a packet map definition. The conversion try_from is a trait API, and also requires fuzzing to generate a diverse set of input values from the array i_0 to include the one that triggers the bug.

```rust
fn test(i_0: &[u8]) {
  let x: MapDefinition = MapDefinition::try_from(i_0);
}
```

7.4 RQ2: Comparisons against SyRust and RULF

We compare Crabtree against SyRust and RULF. In addition to coverage numbers in Table 3, Table 4 reports the number of test cases generated and time spent finding bugs. Fig. 7 and Fig. 8 provide coverage-over-time comparisons against SyRust, RULF, and ablation variants of the tool with features turned off.

Compared to SyRust. Crabtree outperforms SyRust in API, line, and branch coverage for 9 out of the 10 benchmarks, achieving coverage gains ranging from 25% to 300% over SyRust, and has the identical performance as SyRust in the smallest library url_encoding. This is because the library is small enough to fit in the SyRust's set of the 15 APIs and SyRust's template already came with good inputs. Giving SyRust the full API set caused it to be stuck at generating 1 to 2 lines programs, which yielded zero or very low coverage because it spent most of the time spuriously calling the same functions or borrowing the same types. A preliminary evaluation with bitvec and

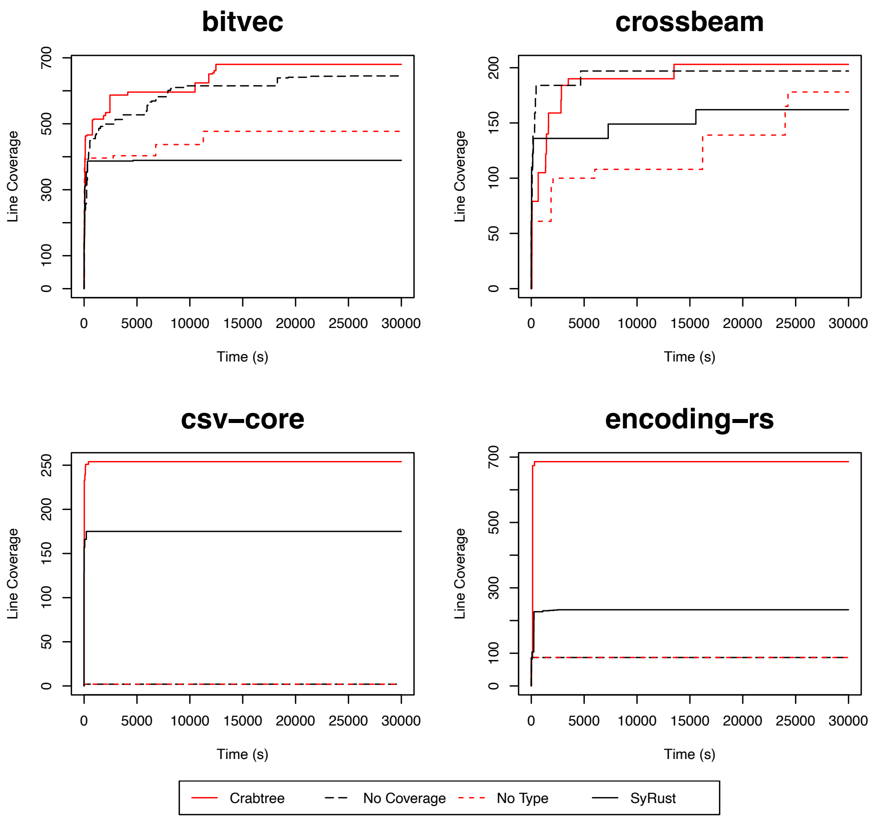


Fig. 7. Comparison of Crabtree and SyRust's coverage gain over time.

crossbeam suggests giving Crabtree the SyRust's set of 15 APIs yields the same coverage as SyRust with roughly the same time.

In bug-finding (the first 4 rows of Table 4), Crabtree catches all SyRust benchmark bugs with fewer tests. Neither tool finds bugs beyond the four initially reported by SyRust. Crabtree finds three out of the four bugs faster than SyRust. Crabtree took more time to find the bug crossbeam-2 because Crabtree generated several computationally heavy API sequences for Miri to run.

We plot the line coverage of bitvec, crossbeam, csv-core, and encoding-rs in Fig. 7. The plots show line coverage over time in seconds. The x-axis is the time spent testing in seconds and the y-axis is the line coverage. The figure is only plotted up to 10 hours, which is how long SyRust ran. Coverage gains after 10 hours are omitted. These coverage gains are minor and do not change the order of the tools and ablation variants. For bitvec, Crabtree outperforms SyRust: more code is tested with less time. Conversely, in crossbeam, Crabtree initially trails SyRust when it deals with combinatorial explosion from the bigger API set. The bigger API set works in Crabtree's favor in the later phase, allowing it to beat SyRust with a large margin by calling APIs outside of the 15 used by SyRust. Compared to the first two data structure libraries, Crabtree benefits from the larger API count early on when testing libraries with shallower API dependencies like csv-core and encoding-rs and the coverage converges very quickly.

Given the fuzzing phase, the cost of running each synthesized test with Crabtree is significantly higher than running it with SyRust where it runs once with concrete inputs. While it is difficult to give direct computational cost for two tools that allow arbitrary parallel deployment, we find that

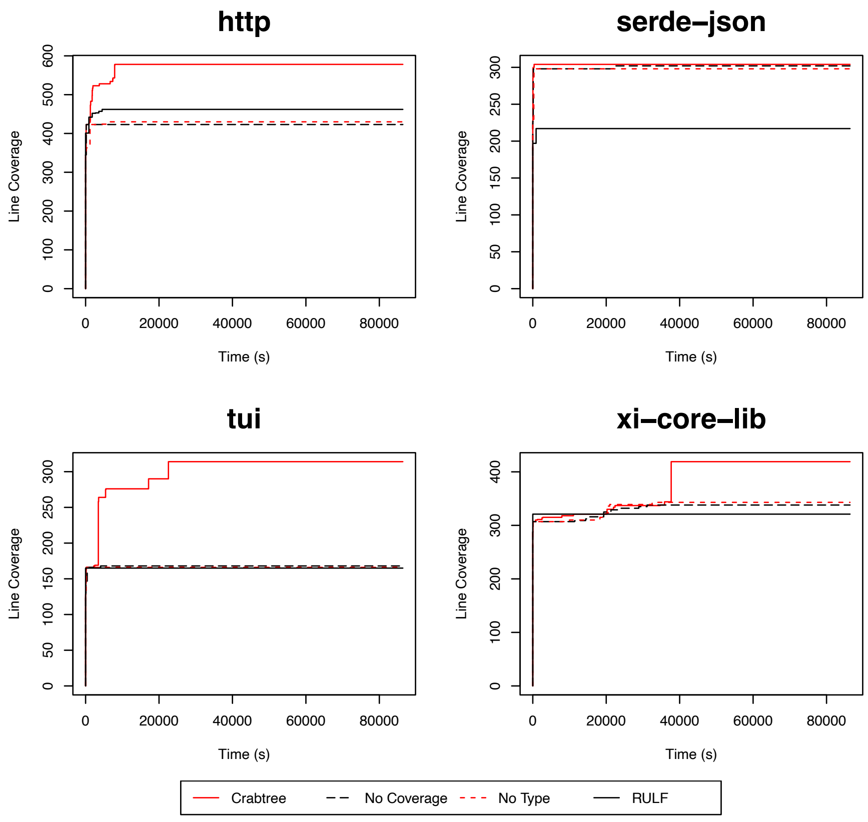


Fig. 8. Comparison of Crabtree and RULF of coverage gain over time.

Crabtree requires more computational resources than SyRust. While the fuzzer initially spent time mutating inputs, in the later phases, the fuzzer spends more time replaying the input corpus than mutating and generating new inputs.

Compared to RULF. For 8 out of 10 benchmarks, Crabtree performs roughly the same or better than RULF in line coverage and mutation catching. By supporting traits and polymorphism, Crabtree can handle a wider-range of APIs than RULF, allowing it to hit APIs with complex input and output types. However, Crabtree's coverage is significantly worse than that of RULF for `regex` and `regex-syntax`. Crabtree suffers from the previously noted issue of not been given adequate input types to call many APIs. There are types for which Crabtree cannot generate via API call sequences, no matter what APIs are called. As a consequence, some APIs will never be called because no variables of the input types can be found. For example, in `regex`, RULF generated 2 harnesses (template) with the input signature of `(i16, &str)` and `(&[u8], usize)` respectively. When we picked the former to use as our template, we miss APIs that require `usize` or `&[u8]` to trigger. As a consequence, Crabtree fails to catch `regex` bugs found by RULF.

API coverage is lower in Crabtree than RULF because RULF attempts to call every single function once while Crabtree selects APIs based on coverage, deprioritizing small functions with simple output types like getters that return primitive values. 5 benchmarks where Crabtree had lower API coverage than RULF but higher line coverage contained large numbers of these functions.

Table 4. The library number is the same as those in Table 3. When there are multiple bugs in one library, we add an integer suffix to the library name. "(s)" indicates time in seconds and "(Nb.)" indicates number of tests required to reach this bug.

	Bug	CT(Nb.)	CT(s)	SR(Nb.)	SR(s)	RULF(Nb.)	RULF(s)
1	bitvec	566	109	4527	200	-	-
2	crossbeam-1	21	10	20	4	-	-
2	crossbeam-2	31759	3480	98548	2850	-	-
3	encoding_rs	2281	239	7452	299	-	-
11	clap-1	32	47	-	-	24	<1
11	clap-2	40	53	-	-	30	873
13	url-1	2615	12851	-	-	35	2305
13	url-2	2717	12922	-	-	60	1585
17	tui-1	200111	12851	-	-	24	77
17	tui-2	200171	12874	-	-	25	41
17	tui-3	200178	12875	-	-	26	56
17	tui-4	200220	12912	-	-	27	21
18	xi-core-lib-1	65899	61853	-	-	16	11
18	xi-core-lib-2	65954	62112	-	-	21	72
18	xi-core-lib-3	65966	62231	-	-	23	25
18	xi-core-lib-4	65961	62215	-	-	29	67
19	regex	Not Found	N/A	-	-	19	20
22	integer-encoding	3	<1	-	-	-	-
23	leapfrog	852	59	-	-	-	-
24	oxidebpf	797	197	-	-	-	-
27	sparesey	2	<1	-	-	-	-

As for the lower branch coverage, we suspect that it is due to the low fuzzing time, which was 5 minutes. This can be seen in testing of `http` where most of the HTTP packet parser remained unreachable with only 5 minutes spent on fuzzing. Given the number of tests synthesized, it is not computationally possible to increase this timeout across the board. This could be improved with dynamic allocation of fuzzing resources, which we leave to future work.

We plot the line coverage over time graph for RULF benchmarks in Fig. 8 `http`, `serde-json`, `tui`, and `xi-core-lib`. For all of the four libraries, Crabtree achieves higher coverage than RULF. For `http` and `tui`, Crabtree without features performs worse because it contains complex APIs that require user-defined types. For `serde-json` and `xi-core-lib`, little difference exists between Crabtree and the variants without coverage or type because the structure and inputs to the APIs are less complex.

As seen in Table 4, while Crabtree replicates bugs found by RULF with the exception of `regex`, it is slower and generated more tests than RULF. This is because our tool prioritized other coverage-increasing API sequences than the one that triggers the bug. As seen in Fig. 8, this yields higher coverage in `tui` and `xi-core-lib`, but it takes longer for Crabtree to reach the bug. For `regex`, we did not replicate any of the RULF bugs due to low coverage on a parser for regex expressions that could be better tested if the tool spent more time fuzzing.

7.5 RQ3: Crabtree Features in Testing

We evaluate Crabtree's features and how they contribute to the effectiveness of testing Rust libraries. For search heuristics, we provide an ablation study using 8 benchmarks. `bitvec`, `crossbeam`, `csv-core`, and `encoding-rs` are selected from SyRust's benchmark set at random. `http`, `serde-json`, xi-core-lib, and tui are from RULF's benchmark set at random. Benchmarks from SyRust and RULF are shown in Fig. 7 and Fig. 8 respectively. For trait support and fuzzing, we provide a qualitative analysis as these are fundamental to the Crabtree and cannot be turned off individually without significantly impacting other components.

Type-Guided Synthesis. Across the benchmarks, we see that turning off the type heuristic resulted in slower coverage gains compared to having the heuristic on. The loss from having this heuristic off varied across libraries, with `csv-core` being hit the hardest when it failed to synthesize the mutable reference to the `CSVParser` type and thus failing to achieve any coverage beyond the constructor. Similarly, without type-guidance, the `Encoder` type is not synthesized for `encoding_rs`. For the RULF dataset, `http`, `tui`, and `xi-core-lib` all failed to reach key types without the type guidance. In `http`, `Request` type was not reached. In `tui` and `xi-core-lib`, both of which are stateful APIs, the type holding the primary state was not reached.

For some libraries like `crossbeam`, where there is no single API that returns a type that is needed as inputs by many other APIs, the type heuristic made Crabtree more efficient but most of the coverage could be hit without it. For `serde-json`, the effect of type guidance is very small because most of the complex APIs like iterators could be exercised by calling simple parser APIs with primitive string inputs, which in turn calls complex APIs. Therefore, testing `serde-json` needed neither deep API chains nor complex types that the type heuristic aims to build.

Coverage. Coverage guidance helps in some cases. With coverage guidance turned off, the coverage for `csv-core` and `encoding_rs` dropped significantly, to even below that of SyRust. For RULF benchmarks, `http`, `tui`, and `xi-core-lib`, Crabtree without coverage performed much worse than Crabtree because key types for these libraries permit "empty" values. Type guidance alone cannot tell whether an API sequence ran with a populated value or "empty" value of the same type. On the other hand, for `bitvec`, `crossbeam`, and `serde-json`, Crabtree's no-coverage variant trails Crabtree by only a small margin. In these libraries, many functions had small bodies and a large number of programmer-defined types already adequately guided the search, making coverage less important.

We see a significant difference between no type and no coverage guidance variants for `bitvec` and `crossbeam`. For these libraries, type guidance is more critical than coverage guidance because these libraries contain APIs with complex input types and reaching those types enable significantly higher coverage. For all other libraries, these two variants trail Crabtree at the same coverage level. Turning off either heuristic caused the tool to miss the same key types required by APIs to effectively test the library, and thus both had the same coverage. These libraries had a hybrid of both issues discussed previously: complex types required to call the APIs, and "empty" values for these types that required coverage guidance to avoid.

Trait Support. In comparison with SyRust, Crabtree eliminates trait errors by construction. In SyRust's benchmark set, roughly half of the ill-typed test cases were due to trait errors with 72303 / 143744 `bitvec` errors and 1248 / 1367 `crossbeam` errors being trait mismatches. Crabtree produces no such errors by construction. We note that `csv-core` and `encoding-rs` did not have extensive trait functions so both tools produced zero trait errors. The same goes for RULF, whose benchmark set excludes polymorphic, and thus trait-related, APIs. Eliminating trait errors also eliminated ill-typed tests around higher-order functions.

Analyzing the 30 libraries in our benchmark, roughly 30% of APIs were defined as trait functions. All the four bugs that we found involve traits. The first two bugs are in libraries that have polymorphic types with trait constraints. The last two bugs we encountered were inside functions that implement traits: (`integer-encoding`'s `FixedInt` and a `From` conversion in `oxide-bpf`). In addition to bug-finding, trait support has a positive impact on coverage as well. For benchmarks tui and xi_core_lib, the support for trait was critical for Crabtree's coverage improvements over RULF. This allowed Crabtree to call a more diverse set of APIs because it can generate default values for types required to call a function. Thus, the trait support in Crabtree is key to achieving higher coverage and finding bugs. For higher-order APIs, turning on our model increased the coverage of iterator APIs from 0% to 40%.

Higher-order functions that either take iterators as argument or return an iterator are about 5% of the APIs (178 out of 3511) in our benchmark. Among all traits implemented in the benchmark (738 trait implementations), 25% of them are iterator traits. Testing these implementations is important because iterator traits are frequently used by programmers looking for familiar trait interfaces to exploit library-specific features. Taking into consideration the types available through iterators, the set of all API calls enabled by our higher-order function support accounts for 10% of the APIs in the entire benchmark set.

Fuzzing and Synthesis Together. The bugs identified by Crabtree (in integer-encoding and oxide-bpf) require both particular API calls and particular inputs for them. This makes the use of fuzzer-generated inputs in conjunction with synthesis important for triggering bugs.

8 Limitations and Future Work

While Crabtree addresses the scalability and feature limitations of prior work, there remains avenues for future exploration. One major performance bottleneck of Crabtree is its repeated call to the Rust compiler. Another performance bottleneck is Miri, which limited our fuzzing time. As future work, we will explore alternatives to reduce interaction with the Rust compiler and Miri so more tests and input corpus can be executed.

The evaluation showed that Crabtree cannot generate tests to reach APIs, whose input types have not been seen by Crabtree. RULF implements a tool to collect simple types to alleviate this problem. As future work, we will investigate a more sophisticated and complete tool for collecting types needed to call APIs.

Both Crabtree and prior works like SyRust and RULF test for crashes only. We do not generate assertions to check program-specific properties. This includes library-specific assertions and properties of the Rust language like invariants around Foreign Function Interfaces. We will investigate how to use tests generated by Crabtree for property-based testing. In particular, we are interested in customizing the synthesis process based on properties under test, so they are more likely to identify property violations.

For libraries with few APIs and simple input types, such as parsers, Crabtree is slower and generated more tests than RULF. Because it prioritizes new types, Crabtree prefers making new API calls to deeply fuzzing a short API sequence. As a consequence, it takes longer to find bugs quickly found by RULF. We plan to adopt a more flexible testing strategy to prioritize generating new sequences or fuzzing differently based on characteristics of the library under test.

Though guided by new type discovery when extending API call sequences, Crabtree, unlike RULF, only grows API sequences forward, and does not target reaching all APIs quickly a priori. This significantly increased the number of test cases generated to reach the diverse set of short API call sequences that RULF generates using a graph search. However, Crabtree can eventually get there and overcome more complex typing constraints. We plan to incorporate what RULF does better and seed the synthesis with short sequences early on.

Several Rust language features remain unsupported. While we support trait implementation on generic types, using generics and lifetime variables inside associated types are not supported. Crabtree does not generate any multi-threaded or async tests. async functions are not supported and thus higher-order functions involving `async`, such as callbacks, are not supported. Likewise, any trait involving `async` is not supported.

The API type specification and traits data are automatically derived using `rustdoc` [35], Rust's official documentation system. `rustdoc` provides API type signatures as well as hierarchies of traits required to synthesize well-typed Rust code. While we assume this information to be correct, any errors will cause Crabtree to generate ill-typed programs when using that API or trait. This occurred exactly once during our evaluation: while running `sparsey`, the API data missed default type variable instantiations, leading to a small number of test cases. We are interested in developing an automated validation tool for the specification against the Rust compiler.

9 Related Work

There are many fuzzing tools, such as the well-known coverage-guided fuzzers AFL [50] and LLVM libFuzzer [25], that have achieved considerable success in bug finding and vulnerability detection. Other works aim to extend the effectiveness of fuzzing, e.g., through symbolic execution techniques [30, 34, 39]. Crabtree is agnostic to the choice of fuzzers and may be used with these AFL alternatives. All these fuzzers require significant manual effort in the form of well-defined fuzz targets, which are sequences of invocations. Several works study automated generation of fuzz targets. For instance, Fudge [5] and FuzzGen [19] build fuzz targets from client projects, but rely on these projects to be high quality. Another tool, RESTler [4], generates fuzz targets for RESTful APIs, which are sequences of requests and responses of specific data format, as opposed to API call sequences. Finally, GraphFuzz [17] tests APIs written in C and C++ by mutating a graph representation of APIs equipped with awareness of pointer lifetimes. These automated fuzz harness generators have similar goals as Crabtree, which is customized to work with Rust type system.

Existing approaches for automatically testing Rust programs are mainly based on fuzzing. AFL.RS is a fuzzer for Rust which is built on AFL++ (based on AFL [50]). LLVM libFuzzer also provides coverage-guided evolutionary fuzzing with Rust bindings [25]. RULF is a fuzzing tool that can automatically generate a set of fuzz targets, based on a given API specification of a Rust library, and integrate them with AFL++ for fuzzing [21]. Building on a similar search algorithm as RULF, RPG [49] and Zhang et. al. [51] add support for polymorphism and traits. RPG keeps a dictionary of valid type variable instantiations while Zhang et. al. internally model trait hierarchies like Crabtree. However, neither tool generates closures necessary to synthesize tests with higher-order function calls. SyRust focuses on Rust polymorphism and generates type-checkable tests by applying semantic-aware synthesis [41]. SyRust has limited scalability and can only handle a few APIs in a library at a time. Crabtree combines fuzzing and synthesis, supporting more Rust features than RULF, RPG, Zhang et. al. [51], and SyRust. While we implemented Crabtree as an extension to SyRust, it is possible to apply some of the techniques on other tools. Trait and higher-order function support can extend RULF, RPG, or Zhang et. al., all of which generate tests with API chaining by allowing more ways to chain APIs. Type and coverage search heuristics are less applicable for RULF because it generates finitely many tests, but RPG and Zhang et. al. can be extended with these heuristics to prioritize test-relevant API chains over others.

RustyUnit [44] performs search-based test suite generation for Rust, using a genetic algorithm. It generates API calls by mutating and combining previous API sequences. These mutations and combinations ensure that the resulting program is also well-typed. However, RustyUnit supports fewer Rust features than Crabtree with polymorphism limited to common standard library types and flat traits without higher-order functions.

Also related are techniques on automated unit test generation, such as Randoop [31] or the testing approach in [46], that is based on Java PathFinder. These techniques generate sequences of method invocations, to be used for testing. However, these techniques cannot be used directly on

Rust because of Rust typing constraints. The techniques developed in Crabtree are applicable to other languages with complex types: type and coverage heuristics are generally applicable. Trait and higher-order function support applies to any language with these features such as Java with its interfaces and higher-order functions in C++.

Rust verification tools like Verus [23], Aeneas [18], Creusot [11], Prusti [47], Flux [24], and RustHornBelt [27] can all check programs for panic freedom in addition to verifying more complex properties. While most of the former tools don't handle unsafe code targeted by Crabtree, model-checkers like Kani [45] and symbolic execution tools like RVT [33] can be used to check for memory safety. To use these tools for libraries, the analyst must write harnesses to call the library functions. While some harness synthesis approaches have been proposed [8, 40], harness writing in general requires high manual effort. API sequences generated by Crabtree may be used by harness-based verifiers to exhaustively check for crashing inputs.

Finally, for synthesizing Rust programs without considering testing, RusSOL [14] generates Rust programs from a separation logic (SL) specification by searching through proof tree of SL rules that translate into a program once the SL predicates are resolved. Both RusSOL and Crabtree can generate inhabitant Rust expressions given a type signature. However, type inhabitation is not enough for testing since first we don't know the type signatures of high-coverage test cases and second, multiple test sequences can have the same type signature, but different test coverage. Crabtree performs both type inhabitation *and type and call sequence search*, which leverages heuristics like coverage and type utility to automatically identify useful types and library API call sequences.

Generating well-typed programs is useful for testing compilers as well. Several approaches have been proposed for testing compilers and type checkers such as Dewey et. al. [12] for Rust and Thalia [38] for JVM languages like Scala . These approaches generate code in semantic-aware ways for finding bugs that require the code to pass early checks of a compiler. While we share with them some of the techniques such as using search to generate well-typed code, we differ in the intent and thus these works are not guided by the type or coverage feedback from code under test.

10 Conclusion

We developed Crabtree, a testing tool for Rust APIs that employs program synthesis techniques to generate well-typed API sequences. Crabtree improves upon the synthesis algorithms of prior works by combining synthesis with fuzzing through a type- and coverage-guided search algorithm that intelligently grows test programs and input corpus towards higher coverage and diverse types. To the best of our knowledge, our tool is the first to model Rust traits and generate tests for arbitrary traits and higher-order functions. Our evaluation of Crabtree shows that Crabtree is more scalable than SyRust and can test libraries with features not supported by prior works.

Acknowledgments

We thank the anonymous reviewers for comments and questions that refined our paper. This work was funded by the National Science Foundation under grant No. 1901136 and the Amazon Research Award, Fall 2022 CFP. Yoshiki Takashima was supported by the Prabhu and Poonam Goel Graduate Fellowship 2021-2022 and Chanhee Cho by the Kwanjeong Educational Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the sponsors.

References

[1] 2018. *CVE-2018-1000810.* https://nvd.nist.gov/vuln/detail/CVE-2018-1000810

[2] 2022. Fuzzing Scripts For Rust libraries with afl.rs. https://github.com/Artisan-Lab/Fuzzing-Scripts original-date: 2021-03-01T05:05:18Z.

[3] 2023. *The Rust community's crate registry.* https://crates.io/

[4] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *Proceedings of the International Conference on Software Engineering (ICSE).* IEEE Press, 748–758.

[5] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. FUDGE: Fuzz Driver Generation at Scale. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE).* ACM, 975–985. https://doi.org/10.1145/3338906.3340456

[6] Daniel L. Berre and Anne Parrain. 2010. The Sat4j library, release 2.2. *J. Satisf. Boolean Model. Comput.* 7 (2010), 59–6.

[7] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2021. *Handbook of Satisfiability - Second Edition.* Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press. https://doi.org/10.3233/FAIA336

[8] Twain Byrnes, Yoshiki Takashima, and Limin Jia. 2024. Automatically Enforcing Rust Trait Properties. In *Verification, Model Checking, and Abstract Interpretation: 25th International Conference, VMCAI 2024, London, United Kingdom, January 15–16, 2024, Proceedings, Part II* (London, United Kingdom). Springer-Verlag, Berlin, Heidelberg, 210–223. https://doi.org/10.1007/978-3-031-50521-8_10

[9] Kees Cook. 2022. *Pull request for Rust in Linux 6.1.* https://lore.kernel.org/lkml/202210010816.1317F2C@keescook/

[10] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[11] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering: 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24–27, 2022, Proceedings* (Madrid, Spain). Springer-Verlag, Berlin, Heidelberg, 90–105. https://doi.org/10.1007/978-3-031-17244-1_6

[12] K. Dewey, J. Roesch, and B. Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP (T). In *Proceedings of the International Conference on Automated Software Engineering (ASE).* IEEE Computer Society, 482–493. https://doi.org/10.1109/ASE.2015.65

[13] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL).* ACM, 599–612.

[14] Jonáš Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. 2023. Leveraging Rust Types for Program Synthesis. *Proc. ACM Program. Lang.* 7, PLDI, Article 164 (jun 2023), 24 pages. https://doi.org/10.1145/3591278

[15] Google. 2023. *Fuchsia.* https://fuchsia.dev/

[16] Google. 2023. *Honggfuzz.* https://github.com/google/honggfuzz

[17] Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: Library API Fuzzing with Lifetime-Aware Dataflow Graphs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22).* Association for Computing Machinery, New York, NY, USA, 1070–1081. https://doi.org/10.1145/3510003.3510228

[18] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 116:711–116:741. https://doi.org/10.1145/3547647

[19] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *Proceedings of the USENIX Security Symposium.* USENIX Association, 2271–2287.

[20] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the International Conference on Software Engineering (ICSE).* Association for Computing Machinery, 215–224.

[21] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. 2021. RULF: Rust Library Fuzzing via API Dependency Graph Traversal. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 581–592. https://doi.org/10.1109/ASE51524.2021.9678813

[22] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked Borrows: An Aliasing Model for Rust. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL).* ACM, Article 41, 32 pages.

[23] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 85:286–85:315. https://doi.org/10.1145/3586037

[24] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.* 7, PLDI, Article 169 (jun 2023), 25 pages. https://doi.org/10.1145/3591283

[25] libFuzzer. [n.d.]. https://github.com/rust-fuzz/libfuzzer

[26] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (Portland, Oregon, USA) *(HILT '14)*. Association for Computing Machinery, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

[27] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 841–856. https://doi.org/10.1145/3519939.3523704

[28] Mozilla. 2020. *Oxidation*. https://wiki.mozilla.org/Oxidation

[29] Mozilla. 2021. grcov v0.7.1. https://github.com/mozilla/grcov

[30] Yannic Noller, Rody Kersten, and Corina S. Pasareanu. 2019. Badger: Complexity Analysis with Fuzzing and Symbolic Execution. In *Software Engineering and Software Management, SE/SWM 2019, Stuttgart, Germany, February 18-22, 2019 (LNI, Vol. P-292)*, Steffen Becker, Ivan Bogicevic, Georg Herzwurm, and Stefan Wagner (Eds.). GI, 65–66. https://doi.org/10.18420/se2019-16

[31] Carlos Pacheco, Shuvendu Lahiri, Michael D. Ernst, and Thomas Ball. 2006. *Feedback-directed Random Test Generation*. Technical Report MSR-TR-2006-125. Massachusetts Institute of Technology. 14 pages. https://www.microsoft.com/en-us/research/publication/feedback-directed-random-test-generation/

[32] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 763–779.

[33] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. 2020. Towards making formal methods normal: meeting developers where they are. https://doi.org/10.48550/arXiv.2010.16345

[34] Herbert Rocha, Rafael Menezes, Lucas C. Cordeiro, and Raimundo Barreto. 2020. Map2Check: Using Symbolic Execution and Fuzzing. In *Tools and Algorithms for the Construction and Analysis of Systems*, Armin Biere and David Parker (Eds.). Springer International Publishing, Cham, 403–407.

[35] Rust Contributors. [n.d.]. What is rustdoc? - The rustdoc book. https://doc.rust-lang.org/rustdoc/what-is-rustdoc.html

[36] Rust for Linux. 2023. . https://rust-for-linux.com/

[37] Kosta Serebryany. 2016. Continuous Fuzzing with libFuzzer and AddressSanitizer. In *IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016*. IEEE Computer Society, 157. https://doi.org/10.1109/SecDev.2016.043

[38] Thodoris Sotiropoulos, Stefanos Chaliasos, and Zhendong Su. 2024. API-driven Program Synthesis for Testing Static Typing Implementations. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. Association for Computing Machinery.

[39] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[40] Yoshiki Takashima. 2023. PropProof: Free Model-Checking Harnesses from PBT. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1903–1913. https://doi.org/10.1145/3611643.3613863

[41] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. 2021. SyRust: Automatic Testing of Rust Libraries with Semantic-Aware Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 899–913. https://doi.org/10.1145/3453483.3454084

[42] The Mutants.rs Authors. 2022. Mutants.rs. https://github.com/sourcefrog/cargo-mutants

[43] The Rust Developers. [n.d.]. *Miri*. https://github.com/rust-lang/miri

[44] Vsevolod Tymofyeyev and Gordon Fraser. 2022. Search-Based Test Suite Generation forÂ Rust. In *Search-Based Software Engineering*, Mike Papadakis and Silvia Regina Vergilio (Eds.). Springer International Publishing, Cham, 3–18.

[45] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. 2022. Verifying Dynamic Trait Objects in Rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 321–330. https://doi.org/10.1145/3510457.3513031

[46] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. 2004. Test input generation with java PathFinder. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, George S. Avrunin and Gregg Rothermel (Eds.). ACM, 97–107. https://doi.org/10.1145/1007512.1007526

[47] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 2021. Modular specification and verification of closures in Rust. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 145:1–145:29. https://doi.org/10.1145/3485522

[48] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. 2022. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM Trans. Softw. Eng. Methodol.* 31, 1 (2022), 3:1–3:25. https://doi.org/10.1145/3466642

[49] Zhiwu Xu, Bin Zhang, Bohao Wu, Shengchao Qin, Cheng Wen, and Mengda He. 2024. RPG: Rust Library Fuzzing with Pool-based Fuzz Target Generation and Generic Support. *ICSE* (2024).

[50] Michał Zalewski. [n.d.]. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/

[51] Yehong Zhang, Jun Wu, and Hui Xu. 2023. Fuzz Driver Synthesis for Rust Generic APIs. http://arxiv.org/abs/2312.10676 arXiv:2312.10676 [cs].

Received 2024-04-06; accepted 2024-08-18