SyRust: Automatic Testing of Rust Libraries with Semantic-Aware Program Synthesis

Yoshiki Takashima
Carnegie Mellon University
Pittsburgh, PA, USA
ytakashi@andrew.cmu.edu

Ruben Martins
Carnegie Mellon University
Pittsburgh, PA, USA
rubenm@andrew.cmu.edu

Limin Jia
Carnegie Mellon University
Pittsburgh, PA, USA
liminjia@andrew.cmu.edu

Corina S. Păsăreanu
Carnegie Mellon University Silicon Valley,
NASA Ames Research Center
Moffett Field, CA, USA
pcorina@cmu.edu

Abstract

Rust's type system ensures the safety of Rust programs; however, programmers can side-step some of the strict typing rules by using the unsafe keyword. A common use of unsafe Rust is by libraries. Bugs in these libraries undermine the safety of the entire Rust program. Therefore, it is crucial to thoroughly test library APIs to rule out bugs. Unfortunately, such testing relies on programmers to manually construct test cases, which is an inefficient and ineffective process.

The goal of this paper is to develop a methodology for automatically generating Rust programs to effectively test Rust library APIs. The main challenge is to synthesize *well-typed* Rust programs to account for proper chaining of API calls and Rust's ownership type system and polymorphic types. We develop a program synthesis technique for Rust library API testing, which relies on a novel logical encoding of typing constraints from Rust's ownership type system. We implement SyRust, a testing framework for Rust libraries that automatically synthesizes semantically valid test cases. Our experiments on 30 popular open-source Rust libraries found 4 new bugs.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Software and its engineering → Software maintenance tools**; *Semantics*; *Software safety*; *Software reliability*.

Keywords: Rust, API Testing, Program Synthesis

PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454084

ACM Reference Format:
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 (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3453483.3454084

1 Introduction

Rust has gained popularity in a wide range of domains such as operating systems, embedded devices, and high-performance web frameworks [1]. A key ingredient of Rust's success is the ownership and variable lifetime system, which guarantees any program passing the type check is free of unsafe behaviors such as use-after-free, double-free, and data races. To recover some expressiveness, Rust also allows programmers to disable some of the safety checks for code segments labeled with the `unsafe` keyword. A common use of unsafe Rust is by libraries, which encapsulate unsafe code and provide abstractions to safe Rust. Allowing `unsafe` code has inevitably led to vulnerabilities in Rust applications. The Rust Standard Library was found to be vulnerable in 2018 [2] and recent surveys [28, 38] further demonstrate the danger of using unsafe Rust.

Effort has been made to help programmers mitigate the risks stemmed from `unsafe` code, for instance, coding guidelines [4] and the Stacked Borrows project that can determine if a Rust program exhibits buggy behavior [20]. Fundamentally, it is necessary for programmers to thoroughly test their libraries that include `unsafe` code. Unfortunately, such testing relies on programmers to manually construct test cases, which can be inefficient and ineffective.

Our goal is to automatically generate test cases for Rust libraries. The main challenges stem from Rust type system features such as polymorphism and variable ownership and lifetime restrictions that make Rust safe. The Rust compiler will reject a large portion of programs that are generated using traditional random search techniques like JCrasher [12], Randoop [27] and RESTler [7]. Instead, we take inspiration from the recent success of constraint solving based program synthesis techniques like H+ [15] and SyPet [14]. These techniques take in a set of API specifications and synthesize valid programs by solving a constraint formula that encodes the set of all bounded-length programs that can be built using the given APIs. In this paper, we propose novel, Rust-specific constraints that encode the above mentioned Rust type system features, allowing us to synthesize valid Rust programs with high probability.

Concretely, we develop a semantic-aware synthesis algorithm that encodes typing constraints including ownership and lifetime constraints by keeping track of the typing contexts and which variables are active and thus can be used as arguments to APIs at each program point. Programs that satisfy these constraints are most likely to pass Rust type checker. Further, this algorithm facilitates the generation of valid API call chains, which are necessary for exposing subtle bugs that would have been missed by testing one API at a time. To make use of APIs with polymorphic types effectively, we propose a hybrid technique to effectively concretize polymorphic object constructors while avoiding combinatorial explosions through early pruning. Furthermore, we develop an API refinement algorithm that leverages error reports provided by the Rust compiler. In the end, we achieve a synthesis algorithm that can effectively generate programs that make use of APIs with polymorphic types while only suffering from a few errors required to refine the API.

We implement the proposed techniques in SyRust, a scalable, synthesis driven, testing framework for Rust libraries. We evaluated SyRust on 30 popular open source Rust libraries, and found 4 bugs, including a double-free that requires a complex sequence of five calls to trigger. Furthermore, our evaluation shows that SyRust is highly effective at generating test programs while reducing compiler errors. Finally, we demonstrate that the semantic awareness and hybrid API refinement is critical to the success of SyRust by observing a significant increase in the number of compiler-rejected test cases when the respective features are turned off. We summarize our contributions below.

1. We propose a novel semantic-aware synthesis algorithm that generates valid Rust programs.
2. We improve the state-of-the-art API refinement synthesis technique to accommodate Rust-specific features by leveraging Rust compiler error messages.
3. We implement our synthesis algorithm in SyRust, an automatic testing framework for Rust libraries.
4. We evaluate SyRust on popular Rust libraries and demonstrate that it can generate semantically valid test cases that expose bugs.

Due to space constraints, many technical details are omitted and can be found in the companion technical report [32].

2 Background and Motivation

We discuss key features of the Rust type system and highlight challenges in automatic generation of valid Rust programs. Considering a simple test case shown in Figure 1 for Vec<T>, a vector data structure from the Rust standard library. The type context for each line is shown in the comments. The test function takes as arguments a String object (s: String) and a vector of Strings (v: Vec<String>). Line 3 *moves* v into mutable vector vm. Line 5 creates a mutable reference vr pointing to vm, using which s is pushed to the vector on line 8. Line 10 calls into_raw_parts, a function that destroys the vector and returns a tuple of size 3. Rust-specific terminology, such as moving and borrowing will be discussed below.

```
1   fn test(s :String, v :Vec<String>) {
2   //context{s:String,v:Vec<String>}
3   let mut vm = v;
4   //context:{s:String,vm:mut Vec<String>}
5   let vr :&mut Vec<String> = &mut vm;
6   //context:{s:String, vm:Vec<String>,
7   //         vr:&mut Vec<String>}
8   vr.push(s);
9   //context:{vm:mut Vec<String>,vr:&mut Vec<String>}
10  let _ = vm.into_raw_parts();
11  //context: {}
12  }
```

Figure 1. An example Rust program

This small well-typed program chains several API calls together and is very similar to a test case generated by SyRust that exposed a bug in bitvec, a bitvector library for Rust. For the rest of this section, we will use this example program to review the Rust type system and illustrate challenges in automatically generating such a test program.

Basic Typing and Subtyping Constraints Rust is statically typed: all variables are assigned a type at compile time and all uses of variables are consistent with their types. For example, if an API takes a string (String) as its argument, then only expressions that have the type String can be given as argument to the API. Rust's type system allows reference types like &String, which refers to a memory of the internal type String and is immutable. Programmers can explicitly make mutable references via &mut String. Rust also allows subtyping. For example, a mutable reference to a string &mut String may be used in place of immutable reference &String, but not the other way around. This means that our test case generating algorithm needs to keep track of typing contexts and reason about types and subtyping.

Polymorphism and API Specification Refinement The type variable T in the polymorphic type Vec<T> can be instantiated with concrete types. To generate test cases for APIs with polymorphic types, our algorithm needs to know how the type variable is instantiated and handle subsequent typing constraints. For example, the push operation seen above has type (&mut Vec<T>,T)→(). Our algorithm needs to match the concrete vector reference &mut Vec<String> with the polymorphic input type &mut Vec<T>, and then understand that the other input type should match the type previously used for instantiating T. Naive enumeration of concrete types to instantiate type variables is inefficient; polymorphic types can be instantiated with other polymorphic types (Vec<T>, Vec<Vec<T>>, Vec<Vec<Vec<T>>> and so on), leading to possible infinite instantiation loops. Type variables that are required to be instantiated with types implementing a trait, indicating supports for sets of methods, is yet another constraint we need to consider.

We define a notion of API refinement: a type signature that permits fewer inputs is considered more refined. For example, the API specification with input type Vec<String> is considered more refined than that with Vec<T>, as the former permits only vectors of strings. API refinement is the process of making the API's type signature more refined.

Variable Ownership and Lifetime Any value of non-primitive type, which is allocated on the heap, has *one* variable that owns it. This ownership can change hands from one variable to another, but the uniqueness of the owner must hold across the program. Guaranteeing unique ownership makes reasoning about when an object can be deallocated easy. A lifetime of a variable is a region in the code in which this variable can be used. If a variable is used, then the compiler ensures that this usage occurs within the lifetime of the variable. Rust lifetime checks rule out programs with memory errors that violate lifetime constraints. To enforce unique ownership, when the owner variable is used in a right-hand-side expression, Rust forcefully terminates the lifetime of that variable at that line, evaluates the expression, and the ownership of the resulting data is moved to the left-hand-side variable. For example, the lifetime of s is terminated at line 8; if we were to call vr.push(s); again on line 9, the program will no longer type check. Our algorithm needs to handle lifetime and ownership constraints.

Relationship Between Lifetimes Sometimes, a variable's lifetime depends on another variable. For example vr is a reference to vm. To avoid creating a potential use-after-free bug, the lifetime of vr must be strictly contained in the lifetime of vm. Swapping the last 2 lines of Figure 1 (shown below) yields an ill-typed program, because when vm is destroyed by into_raw_parts, vr is also removed from the type context.

```
//context:{s:String, vm:Vec<String>,
//          vr:&mut Vec<String>}
let _ = vm.into_raw_parts();
//context:{s:String}
vr.push(s); // vr not found
```

Borrowing and References The Rust langauge allows programmers to make references to a value, which is called borrowing (to contrast it with ownership). Rust enforces properties on refrences to rule out memory errors. These properties depend on whether the reference allows mutation of the memory location. To prevent data races, Rust enforces the rule that only one mutable reference can be active for any location in memory. The compiler will reject any attempts to borrow a mutable reference to a memory location while another mutable reference is active. For example, the following program attempts to borrow a second mutable reference vr2. This does not pass the Rust compiler.

```
//context{s:String,v:Vec<String>}
let mut vm = v;
//context:{s:String,vm:mut Vec<String>}
let vr :&mut Vec<String> = &mut vm;
//context:{s:String, vm:Vec<String>,
//          vr:&mut Vec<String>}
let vr2 :&mut Vec<String> = &mut vm;
//context: can't have both vr and vr2
vr.push(s);
```

Even if vr2 is an immutable reference, the program still causes a type error; because mutable and immutable references for one memory location cannot co-exist. A memory location can have many immutable only references. Our algorithm need to encode these constraints on references.

3 SyRust Overview

The architecture of SyRust is shown in Figure 3. SyRust implements an iterative approach and consists of three main components: a semantic-aware test case synthesis engine; an API specification refinement engine, and a test executor.

Synthesis engine The synthesis engine takes as inputs a code template and library API typing specifications and returns a set of test cases. The engine implements a novel semantic-aware synthesis algorithm that takes into consideration constraints such as those discussed in Section 2. The algorithm is explained in Section 4.

The API typing specifications are collected from the library to be tested. The code template is manually generated, one for each target the analyst intends to test. An example code template for the vector library is shown in Figure 2. It serves the following purposes. First, it indicates to the synthesizer where to insert the synthesized code via the (// INSERT) comment. Second, the main function provides inputs to the test function. Typically, these inputs are assigned types so that they can be used as arguments to the APIs to be tested.

```
fn test(s :String, v :Vec<String>) {
//INSERT
}

fn main() {
let s_0 : std::string::String = "".to_string();
let v_0 : Vec<String> = Vec::new();

test(s_0, v_0);
}
```

Figure 2. An example template for Vec<T>

Polymorphic API Refinement Many Rust library APIs and data structures use polymorphic types. To effectively test them, we implement a novel *hybrid* type variable instantiation scheme that eagerly instantiates a small subset of APIs at the beginning and uses compiler errors to lazily refine the remaining API specifications. If a compiler error is encountered when testing generated programs, the polymorphic API refinement module alters the API specification, so that the same error does not occur again. Based on the error, it first identifies the (polymorphic) API that causes it, then, a new API specification with the instantiated type is added to the API specifications to be used in the next round of synthesis. For example, `Vec::pop` returns `Option<T>`. Initially, we only have the polymorphic version. If we were to use this API with a vector of type `Vec<String>`, then we add a new API with `T` instantiated as `String` for both the inputs and outputs. APIs deemed unfixable will be prevented from being used by the synthesizer. Details of this refinement process are presented in Section 5.

Test Executor The test executor has two roles. First, it compiles the test cases and reports compiler errors. Second, it executes compiled test cases and reports undefined behavior. In Rust, undefined behavior is caused by an operation that lacks well-defined semantics such as dereferencing a pointer after freeing it. For our implementation, we leverage Miri [26], an interpreter for Rust that flags undefined behavior (e.g., dereferencing freed memory). Any tool that can satisfy the above two roles can be used.

4 Semantics-Aware Synthesis

In this section, we show how we can synthesize code that follows the Rust semantics described in Section 2.

4.1 Synthesis Algorithm

Algorithm 1 shows an overview of SyRust's synthesis algorithm. SyRust takes as input a set of API type signatures ($\mathcal{A}$), a code template ($\mathcal{T}$), and a maximum number of lines of code to synthesize (m). The goal of this procedure is to synthesize a collection of test programs that follow the Rust semantics and can be successfully compiled by the Rust compiler. To achieve this goal, we synthesize programs of increasing size by encoding the search space of all possible programs of size m that satisfy Rust's semantic properties as a SAT formula φ (line 3). Each model σ of φ can be translated to a concrete program $\mathcal{P}$ (line 6) that can be compiled and executed by the test executor (line 7). If a compiler error is encountered (line 8), then we analyze the error and detect which APIs were the cause of the compilation error in order to refine $\mathcal{A}$ such that this error does not occur in the future. This can occur for instance when an API is polymorphic and we did not instantiate it. Regardless of the result, we save the solution and resulting messages from test execution in a database $\mathcal{DB}$ (line 12). Finally, we block σ to avoid repetition (line 13) and repeat the process until all models are found and the formula becomes unsatisfiable.

Algorithm 1: High-Level Algorithm of SyRust

```
   Input  : API type signatures $\mathcal{A}$, code template $\mathcal{T}$,
            maximum number of lines of code $m$
   Output : database of programs and results $\mathcal{DB}$
1  Procedure SyRust($\mathcal{A}$, $\mathcal{T}$, $m$):
2      for $l \in 1, \ldots, m$ do
3          $\varphi \leftarrow \mathsf{genConstraints}(\mathcal{A}, \mathcal{T}, l)$
4          while $SATSolver(\varphi) \neq UNSAT$ do
5              $\sigma \leftarrow \mathsf{getModel}(\mathsf{SATSolver}(\varphi))$
6              $\mathcal{P} \leftarrow \mathsf{codeGen}(\sigma, \mathcal{T})$
7              $\mathcal{R} \leftarrow \mathsf{test}(\mathcal{P})$
8              if $isCompilerError(\mathcal{R})$ then
9                  $\mathcal{A} \leftarrow \mathsf{refine}(\mathcal{A}, \mathcal{R})$
10                 $\varphi \leftarrow \mathsf{update}(\varphi, \mathcal{A})$
11             end
12             $\mathcal{DB} \leftarrow \mathcal{DB} \cup \mathcal{R}$
13             $\varphi \leftarrow \varphi \land \neg\sigma$
14         end
15     end
16     return $\mathcal{DB}$
```

This section focuses on which properties need to be encoded in φ (genConstraints) to satisfy Rust semantics. For simplicity, in the remainder of this section, we assume that all API functions are not polymorphic. Section 5 describes in detail how we perform polymorphic API refinement (refine) and carry out hybrid instantiation to make APIs concrete.

4.2 Modeling Programs as SAT Formulas

The syntax of straight-line programs that are synthesized by SyRust is shown below. To simplify the synthesis process, we restrict the programs generated to not include branching and loops. However, our evaluation (Section 6) shows that even when only generating straight-line programs it is still possible to find complex memory bugs. Furthermore, the inputs to the functions (*Vars*) are restricted to variables only, and no expressions. This is fine as expressions can be computed in the prior lines, assigned to a variable, and used in the following lines.

$$\begin{aligned} Program &:= Line \mid Line; Program \\ Line &:= f(Vars) \mid \texttt{let}\ v : \tau' = f(Vars) \\ Vars &:= v_1, \ldots, v_k \end{aligned}$$

To build a SAT formula that represents the space of all possible programs of size l, we start by defining a set of Boolean variables that will be useful to model Rust semantics. These variables can be split into two main categories: *API variables* and *synthesis type-context variables*. API variables denote when an API function occurs in the program and synthesis

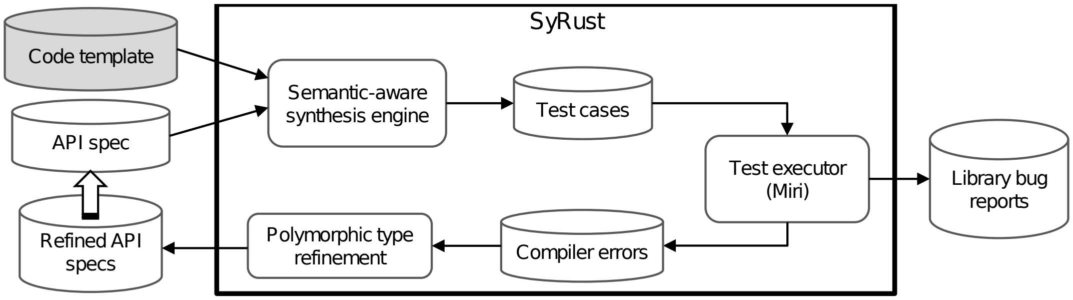


Figure 3. Overview of SyRust

type-context variables denote the universe of code variables that can be used in each line of the program.

For each API function $f \in \mathcal{A}$ we create $A_{f,i}$ Boolean variables with $1 \le i \le l$. We say that f is called on line i if $A_{f,i}$ is set to *true* in σ. Additional restrictions are added to ensure that for each line i *exactly one* $A_{f,i}$ is set to *true*.

To have valid programs, we must ensure that an API function $f : \tau_1 \times \ldots \times \tau_k \longrightarrow \tau'$ can only be called on line i if the program contains variables that have been declared in previous lines with types $\tau_1, \ldots, \tau_k$, respectively. We use a mapping from variable names (v) to types (τ) in *synthesis type context*. For any line, there is a synthesis type context before it, which is the set of variables and types available before the function call, and a synthesis type context after it, which is the set of variables and types available after the function on this line has been called.

We model synthesis type contexts as subsets of the Cartesian product between the set of variable names (S) and the set of types (T). S is the union of variables names in the template and the new variable names that may appear in each line of code $\{v_i \mid 1 \le i \le l\}$. T is the union of types in the template, with the output types from the APIs $\{\tau \mid f \in \mathcal{A}, f : * \longrightarrow \tau\}$. We define Boolean variables $V_{v,\tau,i}$ with $v \in S$, $\tau \in T$, and $1 \le i \le m$. We say that if variable v with type τ occurs in the synthesis type context of line i then $V_{v,\tau,i}$ is set to *true* in σ. Using these variables we can recursively construct the synthesis type context for each line. First, let us consider C_1, the synthesis type context for line 1. This is the synthesis type context before any APIs are called. This means the only variables available are those provided by $\mathcal{T}$. C_{i+1} must propagate every possible variable in C_i and include any new variables defined on line i. The following recurrence formalizes how we construct a synthesis type context for every line in the length-l program.

Definition 1 (Construction of Synthesis Type Context)**.**

$$\begin{aligned} C_1 &= \{V_{x,\tau,1} \mid x : \tau \in \mathcal{T}\} \\ C_{i+1} &= \{V_{x,\tau,i+1} \mid V_{x,\tau,i} \in C_i\} \cup \\ &\quad\ \{V_{x_{i+1},\tau',i+1} \mid f \in \mathcal{A}, f : * \longrightarrow \tau'\} \end{aligned}$$

We write C_l to denote the set of Boolean variables that occur in the synthesis type context on line l. We overload C_l as a mapping from variables to types, e.g., if $V_{v,\tau,l} \in C_l$ and $V_{v,\tau,l}$=1 then $C_l(v) = \tau$, and define the domain of the existing types of a context as $dom\ C_l = \{d \mid \exists\ \tau \text{ st. } C_l(d) = \tau\}$.

4.3 Basic Typing Constraints

We encode a set of rules that guarantees the correct usage of API functions over a set of variables. These rules are compatible with most imperative languages.

Rule 1. *If* $\mathcal{T}$ *provides the inputs* $\{i_1{:}\tau_1, \ldots, i_n{:}\tau_n\}$, *then* $\forall_{1 \le j \le n}\ C_1(i_j) = \tau_j$.

Rule 1 states that the available variables and types at the start of the synthesis are the ones in the template $\mathcal{T}$.

Rule 2. *Let* $Line_i$ = `let x : ` τ ` = f(...)`. $C_{l+1}(x) = \tau$ *and* $\forall v \in (dom\ C_l \cap dom\ C_{l+1}) \setminus \{x\}, C_{l+1}(v) = C_l(v)$.

Rule 2 describes how the synthesis type context is modified from line i to line $i+1$ with the call to function f in line i.

Next, we define a line l type checks with Context C_l, written $C_l \vdash Line_l \Rightarrow C_{l+1}$ as follows.

Definition 2 (Single-Line Type Check)**.** *Let* $f : \tau_1 \times \ldots \times \tau_k \longrightarrow \tau'$. *A line* l *of code containing* $f(v_1, \ldots, v_k)$ *type checks with Context* C_l *if all of the following conditions are satisfied.*

1. $v_1, \ldots, v_k \in dom\ C_l$
2. $\forall_{1 \le p \le k} C_l(v_p) \sqsubseteq \tau_p$
3. $\forall_{1 \le p < q \le k}\ compatibleTypes(C_l, v_p, \tau_p, v_q, \tau_q)$

We use the subtype operator ($\sqsubseteq$) instead of = to not miss generating valid programs, in particular for the following two aspects. First, Rust has subtyping for reference mutability: `&mut String` $\sqsubseteq$ `& String`. Second, by allowing type variables to match the broadest range of types (i.e., $\forall \tau, \tau \sqsubseteq$ `T`), the synthesis can generate code that uses polymorphic types; for instance `Vec`$\langle String \rangle \sqsubseteq$ `Vec`$\langle T \rangle$. However, if the same type variable occurs in multiple places, then it must be matched to compatible types. For example, for `Vec::push`, we must ensure `T` from first input `&mut Vec`$\langle T \rangle$ matches with a type that is compatible with what `T` matches in the second input. To this end, we use a compatibleTypes function that determines if a polymorphic match is compatible for every pair of polymorphic variables v_i, v_j. Note that if v_i and v_j are not polymorphic then they are always compatible. More discussions on polymorphic types are presented in Section 5.

We can inductively extend Definition 2 to define type checking for a multi-line program below.

Rule 3. *Let $Program = Line_0; \ldots; Line_{n-1}$ and $Line_i$ contain a call to f_i. Then there exists $C_1, \ldots, C_n$ such that for every $1 \leq i \leq n$, $C_i \vdash Line_i \Rightarrow C_{i+1}$.*

4.4 Ownership and Variable Lifetime

We show how to encode variable and reference lifetime constraints in Rust as described in Section 2.

4.4.1 Basic Variable Lifetimes. A variable v's lifetime, denoted L_v, is the set of *continuous* contexts in which v occurs:

Definition 3 (Variable Lifetime). *A lifetime for variable v that is created on line s and ends in line e is defined as follows: $L_v = \{C_l \mid s \leq l < e, v \in dom\ C_l\}$.*

Some types can be used at most once in a line. The algorithm needs to add the following condition to the compatibleType function presented in Definition 2. If the variables are the same, then the types must be a primitive type or static reference (&), the only type of reference that is allowed to be used in interleaving terms on the same line. That is,

Rule 4. *$compatibleTypes(C, v, \tau_i, v, \tau_j)$ with $i \neq j$ is true if and only if $\tau_i = \tau_j$ is a static reference.*

Further, non-primitive variable lifetimes need to be terminated upon use.

Rule 5. *Let v be a non-primitive variable used on the right-hand-side in $Line_i$, then v's lifetime terminates here. We require that $v \notin dom\ C_{i+1}$, and therefore $C_{i+1} \notin L_v$.*

4.4.2 Relationship between Lifetimes. Next, we encode rules for references. In particular, we need to define the notion of a variable outliving another one.

Definition 4 (Lifetime Containment). *A lifetime L_{v_1} contains another lifetime L_{v_2} if and only if $L_{v_2} \subseteq L_{v_1}$.*

Because our algorithm treats borrowing (`&` and `&mut`) as a special kind of API, the lifetime containment rule will be formalized as follows.

Rule 6. *Let $Line_i$ be a call (`&` or `&mut`) to borrow variable v to produce a reference v', which points to v, then $L_{v'} \subseteq L_v$.*

4.4.3 Paths and Lifetimes. Rule 6 alone does not capture all of the constraints in lifetime management; constraints on implicit movements of references are missing. The Rust type signatures are annotated with lifetime variables [3, 5] (e.g. `'a`). These annotations can be programmer-provided, or in some cases, the Rust compiler automatically annotates using Lifetime Elision rules [3].

Consider the option type `Option<T>`. Assuming it is not `None`, it serves as a wrapper for that can be unwrapped with `unwrap : Option<T> → T`. This carries no significance when `T` is not a reference. However, when T is a reference, we need to make sure that the output of type `T=` $\&mut\ \tau$ must be as a valid reference and all of the reference rules must be applied to it. To formalize implicit movements, we define *Path* to model data flow.

Definition 5. *(Path) A path is a sequence of variable-line pairs $(v_s, l_s), \ldots, (v_k, l_e)$ such that for every (v_i, l_j) the variable associated with l_{j+1} is not equal to v_i. Moreover, there exists a function call on $Line_j$ that propagates the lifetime of v_j to the new variable defined on $Line_j$.*

This rule formalizes the notion that we need to keep the lifetime relation even when the types change and content is moved out.

Rule 7. *For every path $(v_s, l_s), \ldots, (v_k, l_e)$, $L_v \subseteq L_s$ is enforced for every $v \neq s$ in the path.*

Note that the above rule is enforced using a post-processing check, rather than encoded in the formula. This is because this notion of connectivity may lead to a cubic number of additional SAT clauses [31] and it is more efficient to either solve it lazily or via a post-processing check.

4.4.4 Banned Operations and Lifetimes. Recall the example in Section 2 where creating a mutable reference (`vr2`) was prevented because another mutable reference (`vr`) was already active in the context. Our goal is to add constraints to prevent these kinds of programs from being synthesized. For any non-primitive variable with a reference borrowed on it the following 3 rules must be obeyed:

1. while `&mut` is active, prevent another `&mut`,
2. while `&mut` is active, prevent another `&`,
3. while *any* `&` is active, prevent another `&mut`.

In the last category, *any* is emphasized as it is possible to have multiple static references, but every static reference must be exclusive against mutable references.

Rule 8. *If $Line_i$ is a call to mutable borrow (`&mut`) v and the resulting mutable reference is stored in v', then $\forall Line_j$ such that $j > i$ and $v' \in dom\ C_{j+1}$, $Line_j$ cannot be a mutable borrow or static borrow of v.*

Rule 9. *If $Line_i$ is a call to static borrow `&` v and the resulting mutable reference is stored in v', then $\forall Line_j$ such that $j > i$ and $v' \in dom\ C_{j+1}$, $Line_j$ cannot be a mutable borrow of v.*

4.5 Connecting to SAT

We have presented the rules that we use to encode the semantics of Rust in our synthesis engine. We convert these rules into a Boolean Satisfiability formula and solve it using an off-the-shelf SAT solver. We refer the reader to Technical Report [32] for further details on the SAT encodings.

Getting back to Algorithm 1, the rules of this section correspond to what genConstraints generates. Once this formula is solved, we will have a truth assignment σ for the Boolean variables that correspond to a valid program in Rust. Using the one-to-one correspondence between the Boolean variables that represent the APIs and inputs with their code representation, the codeGen function parses the solution and builds the test case line-by-line.

4.6 Properties of the Synthesized Programs

Remark 1. *The programs synthesized by SyRust satisfy the requirements of the Rust compiler for the following properties.*

1. *The synthesized program type-checks (Rules 1, 2, 3).*
2. *Variable aliases are not allowed except for static references (Rule 4).*
3. *All non-primitive variables are uniquely owned (Rule 5).*
4. *While any references are active, the memory location cannot be moved or deallocated (Rules 6, 7).*
5. *For any given variable, there is at most 1 active mutable reference or an arbitrary number of active static references, but not both (Rules 8, 9).*

Even though the synthesized programs can still result in compilation errors, our evaluation (Section 6) shows that only a very small percentage of programs are rejected by the compiler and these errors are due to other factors not captured in the above properties, e.g., polymorphic errors that can be lazily fixed with the approach presented in Section 5.

4.7 Suppressing Redundancy

While the above synthesis rules are sufficient for generating valid programs that pass the compiler checks, we need additional rules to prevent the synthesizer from generating programs that are semantically equivalent to previously generated programs. The goal here is to be more efficient, as the set of valid programs is infinite and grows exponentially with respect to the number of lines. SyRust must effectively avoid generating programs that do not contribute to testing.

We do not aim to rule out every program that is semantically equivalent to a previously generated one, which in itself is computationally intensive. Instead, we define efficient constraints that we can add to the SAT formula to rule out programs containing obviously redundant operations. Such programs are semantically equivalent to a previously generated program of a shorter length.

```
1   fn redundant(s : String) {
2     let v_old : Vec<String> = Vec::new();
3     let mut v : Vec<String> = v_old;
4     //P_1...
5     let mut v2 : Vec<String> = v;
6     //P_2...
7   }
8
9   fn removed(s : String) {
10    let v_old : Vec<String> = Vec::new();
11    let mut v : Vec<String> = v_old;
12    //P_1...
13    //[v/v2]P_2...
14  }
```

Figure 4. An example redundant move operation.

4.7.1 Example of a Redundant Operation.

Let us consider the function redundant in Figure 4. Assume the comments P_1 and P_2 are arbitrary length straight-line code that is well typed with respect to the rest of the program. We observe that line 5 of the above program is redundant. Since v is already a mutable variable of type Vec<String>, there is no need to move it to v2. Furthermore, since this ownership movement does not alter the underlying memory, the function redundant is equivalent to the function removed. If we remove line 5 and substitute v for v2 in the code following it, the function redundant becomes the same program as function removed.

4.7.2 Characterizing Redundancy.

We consider a more general characterization of redundant operations based on the following three observations of Rust programs.

1. Moving an object via simple assignment does not alter its value.
2. Consecutively made unique references (&mut) to the same object can be replaced by a single one.
3. Making a reference to an object does not alter its value.

While we do not formally prove it, the soundness of these observations is supported by the semantics of Rust's primitive operations [20, 29, 35].

The above 3 observations yield corresponding syntactic characterizations of redundant operations. Figure 4 illustrates the pattern identified by the first observation, where the assignment on line 5 is redundant.

The second one yields the pattern shown in Figure 5. Mutable references vr and vr2 have to be unique, and thus creating the second reference vr2 to v in line 5 terminates the lifetime of the previous mutable reference vr. Removing line 5 and substituting vr for vr2 in P_2 yields a semantically equivalent program.

```
1   fn test(i : i32) {
2     let mut v : Vec<String> = Vec::new();
3     let mut vr : &mut Vec<String> = v;
4     //P_1...
5     let mut vr2 : &mut Vec<String> = v;
6     //P_2...
7   }
```

Figure 5. An example redundant unique reference.

The third observation yields the pattern: a line is considered redundant if it creates a reference that is never used.

We define the following three constraints over straight-line Rust programs that suppress the generation of programs containing the above mentioned redundant operations.

1. For any variable that is already mutable, do not move it to another mutable variable. This is encoded by negating the use of such variable by the special API that represents a move to a mutable variable.
2. The number of mutable borrows of any variable is at most once. This is encoded as a numerical inequality constraint over the actions on all given variables.
3. All references must be used at least once. This is again encoded as a numerical inequality over actions.

These constraints are used in conjunction with other rules during synthesis. The corresponding SAT formulas can be found in the Technical Report [32].

5 Hybrid API Refinement

In this section, we present our *hybrid* approach to instantiate polymorphic APIs. Rust allows users to specify that a type variable can only be instantiated with types implementing a given trait (i.e., types that support a set of functions). We also discuss how SyRust deals with traits here.

A type τ is concrete if it has no type variables, and polymorphic otherwise. APIs with only concrete inputs and output are concrete, and otherwise polymorphic. The following are the three most common types of polymorphic Rust APIs

1. No Input Polymorphism:
Example: $\texttt{Vec} :: \texttt{new}\ () \to \texttt{Vec}\langle T\rangle$
2. Polymorphic Input, Concrete Output:
Example: $\texttt{Vec} :: \texttt{push}\ (\texttt{\&mut Vec}\langle T\rangle, T) \to ()$
3. Polymorphic Input, Polymorphic Output:
Example: $\texttt{Vec} :: \texttt{pop}\ (\texttt{\&mut Vec}\langle T\rangle) \to \texttt{Option}\langle T\rangle$

While these categories are not exhaustive; it is certainly possible to write a function that has type variables in the inputs different from type variables in the output. In practice, we find this extremely uncommon.

Prior works dealt with polymorphism by either exhaustively instantiating type variables beforehand (purely eager) [14] or keeping all APIs polymorphic and refining them incrementally (purely lazy) [15]. Our observation is that neither purely lazy nor purely eager approaches are sufficient for Rust. Purely lazy approaches cannot synthesize types for no input polymorphism; while purely eager approaches result in too many incorrect APIs (see Section 7.1 for more details). Therefore, SyRust combines the two.

5.1 No Input Polymorphism

Let's start with the most simple case: a function with a polymorphic output and no inputs. These functions are often used as constructors in data structure libraries. Since the function has no inputs, we cannot infer the concrete output type automatically. The Rust compiler is often able to reason about the output by looking ahead (i.e. Resolve to `Vec<i32>` because the vector gets `i32` pushed onto it sometime later). Such information is not available to our synthesis algorithm that builds programs from the ground up. Instead, we opt to eagerly concretize the output type. We need to find concrete types to substitute type variables with. As we noted in Section 2, we must be careful as this space is infinitely large. The hybrid API refinement module mines concrete types from the API set and template. For example, if we collect `i32` and `u32` from the API and template, we add 2 variants of `Vec::new` where the output type is changed to $\texttt{Vec}\langle\texttt{i32}\rangle$ and $\texttt{Vec}\langle\texttt{u32}\rangle$ respectively. This combinatorial enumeration generates a large number of APIs, so it should be only sparingly used. We only use it for No Input Polymorphism.

The eager concretization also ignores trait annotations on type variables. This means some of the type instantiations are incorrect due to trait mismatches. Because the instantiated APIs are concrete, they cannot be refined further. SyRust removes fully concrete APIs that caused trait errors to avoid a combinatorial explosion of incorrect programs.

5.2 Polymorphic Inputs, Concrete Output

Let us now consider cases where a function has polymorphic inputs and concrete output. We only need to instantiate the inputs. For a majority of the cases, the subtyping strategy outlined in Section 4 works fine. However, this strategy fails when type variables in the input are annotated with traits that any matching type must support.

While it is certainly possible to support traits at the synthesis level, Rust's trait system is fairly complex. Traits can be polymorphic themselves, and a few complex traits are even defined on other traits. Instead of dealing with complex trait requirements, we use the compiler errors as feedback to refine specifications of polymorphic variables. When a type match fails because of mismatching traits, we refine the specification of the type variable to no longer match that particular type. The refinement is usually complete after a few rounds.

5.3 Polymorphic Inputs, Polymorphic Output

In the final category, let's consider, $\texttt{Vec} :: \texttt{pop} : \texttt{\&mut Vec<T>} \to \texttt{Option<T>}$, which takes a mutable reference of a vector and returns a wrapped value containing the vector's last element if such exists, and `None` if the vector is empty. Because this output may be used in later API calls, we must get the exact type of the output. We must do so without modifying the original API as we want to use to match other types later. Our solution is to duplicate the function for that set of inputs only.

We begin the process the same way as we deal with functions that have no polymorphic outputs. For example, the synthesis algorithm matches `&mut Vec<T>` with a concrete type such as `&mut Vec<i32>`. The synthesizer generates a test case where `Vec::pop` is used with the input of type `&mut Vec<i32>` and predicts the output type as `Option<i32>`. This test case is then passed to the test executioner, which attempts to compile and run the test case.

If the test case compiles successfully, we duplicate the function and fully concretize the duplicated API's inputs with the current input types. Since the program type-checked, the predicted output type is correct. We replace the output of the duplicated API with the prediction.

If the test case fails to compile, what happens next depends on what error we get. If the error is originating from the inputs to the API, then we perform the same refinement as in the previous section. Other errors are fixed directly. For example, if we get the message "expected `String`, got `Vec<i32>`" then we duplicate the API, concretize the inputs, and set the output type to `Vec<i32>`.

Because we are duplicating APIs, we must be careful to keep the duplicated API disjoint from the original. After duplicating the API, we add a constraint to the original API to prevent it from being used with the same combination of input types. We block combinations rather than individual input types because APIs like Hash Map's `get` function has multiple type variables involved, and blocking individual types will block too many potential candidates.

6 Implementation and Evaluation

We implemented our proposed approaches in SyRust, an automatic testing tool leveraging semantic-aware synthesis and type refinement. We evaluated it against Rust libraries on the following research questions.

RQ1. How effective is SyRust in generating valid Rust test cases and in finding bugs in Rust libraries?
RQ2. How do the various semantic-aware synthesis features (Section 4.4) help generate valid test cases?
RQ3. How does the polymorphic API refinement algorithm help find bugs?

6.1 Implementation

The synthesizer is implemented as a Java program that takes in a configuration file that specifies the APIs to be tested and a template to use in the tests, and generates test cases through constraint solving. We use Sat4J [10] to solve SAT formulae. The type refinement is also implemented in Java.

The test executioner uses `cargo` [30], the official build system for Rust. We run `cargo` with the `–message-format=json` flag to get compiler errors in JSON format, and send back the parsed data to the synthesizer. To detect bugs, we use Miri [26], a bug detection interpreter already integrated with `cargo`. Miri relies on the Rust compiler to generate the Mid-Level Intermediate Representation (MIR), which Miri then interprets. It flags any behavior that is considered undefined according to its dynamic semantic model for Rust [20]. Coverage given in Section 7.3 was taken using `grcov` [25]. We used `lcov` [23] and GNU Parallel [33] to post-process the coverage data.

The API type signatures used for synthesis are collected using a modified Rust compiler. We compile the target libraries with the modified compiler and derive the API type signatures. Since the total number of APIs is too large, a smaller subset is used in testing. Section 6.2 provides further details on how this selection is done for our experiments.

6.2 Library and API Selection

To select a sufficiently large and representative set of Rust libraries for our evaluation, we examine `crates.io`, the official library repository for Rust. We focus on Data Structure and Encoding categories. This is because these libraries are often used as the building blocks of other libraries and thus are highly critical to the Rust ecosystem. Furthermore, they are also more likely to contain `unsafe` code as they often perform low-level operations. From the two categories, we select the 30 libraries by download count, while making sure that the libraries have the following 2 properties.

1. **The library and all of its dependencies are written purely in Rust.** This is required as Miri cannot handle foreign functions.
2. **The library must also be API based.** Libraries consisting mostly of macros cannot be used as no type signatures are readily available for these libraries.

We also prioritize libraries that contain `unsafe` code. For details about library versions, popularity, and other features, we refer the reader to Technical Report [32].

For each library in the evaluation, we select a subcomponent to test. We pick the most prominent component in the library, but we randomly pick one if more than one component is prominent. For many libraries, the most prominent library is a data structure or function that represents the intended use case of the library. For example, we selected `bitvec::vec::BitVec` for the `bitvec` library as that was the data structure that represents it. Our component selections are shown in the Technical Report [32]. Then, we select a set of 15 APIs belonging to that subcomponent. We allow 2 APIs out of 15 to be manually selected to simulate the scenarios where the programmers want to test specific APIs. Then the rest of the APIs are chosen through weighted random selection. The weights depend on whether the API contains unsafe code or not. APIs that contain unsafe code are given 50% more weight than APIs that are completely free of unsafe code. To this set of 15, we add 3 default APIs that represent operations built into Rust: for assignment to mutable (`let mut x = y`) and 2 kinds of borrowing (`&, &mut`). In practice, we find that this number of APIs gives us a good trade-off between having a diverse set of APIs and synthesizing complex programs with multiple lines of code. With the APIs selected, we write one code template for each library. The effort of writing it is similar to that of a fuzz driver.

6.3 Experiment Setup

Since SyRust generates a large number of test cases, the majority of the computational resources is spent on running these test cases. Exploiting the fact that the test cases can be run independent of each other, we deployed 64 containers running the test executioner across 4 machines, each of them running an Intel i9-9900K with 128GB of RAM. On the other hand, we find that in practice, solving the constraint formulas is quite fast. Therefore, the synthesizer node is deployed on an Intel i7-6700K with 32GB of RAM. All nodes are running Ubuntu Linux, with the test execution nodes running 19.10 and the synthesizer running 18.04. We used the 2020-10-01 nightly build of Rust and Miri (Miri only comes with nightly builds). Nodes communicate using ZeroMQ, with the synthesizer node running using JeroMQ Java library version 0.5.0 and test executioners using PyZMQ version 19.0.0 as the respective driver. Due to memory constraints, we limit the size of the queue at 10,000 programs.

7 Evaluation Results

We present our evaluation results and answer RQ1-RQ3 here. Due to the lack of prior work in Rust library API testing, we are unable to provide an external baseline.

7.1 RQ1: New Bugs and Test Effectiveness

We ran SyRust on 30 libraries with a timeout of 10 hours per library. We excluded two libraries `cookie-factory` and `jsonrpc-client-core` from the result as they use first-class functions, not supported by our syntax. For libraries that overlap (e.g. `crossbeam-queue` and `crossbeam`), we test different components with no shared APIs. The results are summarized in Figure 6. The columns denote the library name, number of lines enumerated, number of test cases processed, the number of programs that were rejected by the compiler, followed by a percent-wise breakdown of the rejections into categories. The "Type errors" are caused by wrongly instantiated polymorphic types; "Lifetime & Ownership" is self-explanatory; "Miscellaneous errors" typically indicate errors in collected API type signatures (done in the experiment setup phase). Finally, Libraries that were flagged as buggy by SyRust are marked with a ★.

Overall, we observe that for most libraries, the portion of test cases rejected by the compiler is extremely low (often less than one percent) indicating the effectiveness of our tool in generating valid Rust programs. However, for certain libraries, the error rate is significantly higher than for the rest and this is due to features that are not supported yet in SyRust. These include an unsupported lifetime corner case involving anonymous parameterized lifetimes in libraries for which we had non-zero "Lifetime & Ownership" errors, and missing default values for type variables in `Petgraph`. We believe the former issue can be fixed with improved API collection and fixing the latter requires modifying the rules of Section 5 and Section 4 to accommodate default values of type variables. We leave these improvements to future work.

Miscellaneous errors, such as "expected n arguments, found j," are often caused by errors in API type signatures. However, the large number of miscellaneous errors in generic-array and hashbrown are "method not found" errors, which can be caused by both polymorphism and API issues. We chose to conservatively include them in Miscellaneous, but we suspect a large number of them are polymorphism induced.

Finally, we observe that some libraries have significantly fewer test cases synthesized than others. For most such libraries, there are simply not enough valid combinations to generate a large number of valid test cases. This means that the synthesis terminates early, with a small number of solutions found. These libraries also exhibit higher error rates as APIs become more refined towards the end of the run. One exception is dashmap, where the library was extremely slow to be interpreted by Miri. Therefore, only about half as many test cases can be executed within the 10 hour limit.

In total, SyRust found 4 previously unknown bugs in 3 different libraries, all of which were accepted by the library authors. These bugs are shown in Figure 7. The columns denote the library in which the bugs were found, the minimum number of lines required to trigger the bug, the type of bugs, and whether the authors accepted the bug. We will go over these noteworthy bugs in detail.

Bug ★1 is a memory leak in crossbeam-queue's `ArrayQueue` data structure that can be induced by initializing the `ArrayQueue` with a non-zero initial capacity. While this is a one-line trivial bug, it exposes problematic assumptions about internal memory layout, and the issue was cited in a vulnerability report [6].

Bug ★3 is a dereference-after-free bug in bitvec, a bitvector library. This bug occurs when deallocating a non-empty `BitBox` object, a fixed-length equivalent of `BitVec`.

The bug-inducing code in Figure 8 is particularly challenging to synthesize. First, note that it involves ownership movement through the `into_boxed_bit_slice` function. This means that the same call sequence cannot be triggered using a loop-based fuzzing harness because it would not pass compiler checks when the ownership moves inside the fuzz loop. Therefore, it can only be triggered using synthesis-driven approaches like ours. Second, the bitvector library uses extensive polymorphism and trait-driven programming. For example, the type `BitVec<Msb0, usize>` is really `BitVec<O,T>` with `O` and `T` instantiated with `Msb0` and `usize`. One must be careful in the instantiation because `BitVec<usize,Msb0>` will not pass compiler checks, since `usize` does not support the `BitOrder` trait (`Msb0` = Most significant bit is index 0). Finally, the sheer size of the bug-inducing case makes this a difficult bug to induce.

Finally, bugs ★2 and ★4 provide interesting insight into the operational semantics of Rust. These bugs may look benign,

Library	Max Test Case Length	# Synthesized	# Rejected (% of total)	Type Error (%)	Lifetime & Ownership (%)	Misc. (%)
smallvec	9	1225952	66 (< 0.01 %)	95.45 %	0.00 %	4.55 %
crossbeam-utils	5	1242990	703 (0.06 %)	58.61 %	2.99 %	38.41 %
bytes	7	1194703	43 (< 0.01 %)	93.02 %	0.00 %	6.98 %
slab	6	1229924	811 (0.07 %)	64.24 %	35.76 %	0.00 %
crossbeam-deque	6	1216076	957 (0.08 %)	100.00 %	0.00 %	0.00 %
generic-array	10	1216241	2088 (0.17 %)	1.29 %	0.00 %	98.71 %
crossbeam-queue ★1	5	1153651	19463 (1.69 %)	100.00 %	0.00 %	0.00 %
num-rational	4	1255485	4506 (0.36 %)	99.82 %	0.00 %	0.18 %
hashbrown	6	1122196	17649 (1.57 %)	7.25 %	0.54 %	92.21 %
crossbeam ★2	4	1231844	1367 (0.11 %)	98.76 %	0.88 %	0.37 %
petgraph	4	1318138	143347 (10.87 %)	100.00 %	0.00 %	0.00 %
im-rc	6	1226829	25531 (2.08 %)	95.06 %	0.06 %	4.87 %
bitvec ★3	7	1221730	120 (< 0.01 %)	100.00 %	0.00 %	0.00 %
ndarray	9	1188730	830 (0.07 %)	100.00 %	0.00 %	0.00 %
dashmap	7	660986	918 (0.14 %)	82.57 %	0.22 %	17.21 %
encoding_rs ★4	6	1233420	152 (0.01 %)	100.00 %	0.00 %	0.00 %
bstr	9	1207815	258 (0.02 %)	93.80 %	1.55 %	4.65 %
csv-core	6	14961	478 (3.19 %)	5.02 %	93.72 %	1.26 %
data-encoding	10	900509	136 (0.02 %)	64.71 %	23.53 %	11.76 %
encode_unicode	6	1238800	67 (< 0.01 %)	94.03 %	0.00 %	5.97 %
urlencoding	6	1139257	48 (< 0.01 %)	100.00 %	0.00 %	0.00 %
rmp-serde	6	11544	963 (8.34 %)	99.27 %	0.00 %	0.73 %
bytemuck	5	112030	19568 (17.47 %)	86.26 %	13.74 %	0.00 %
sval	10	86606	392 (0.45 %)	44.39 %	55.61 %	0.00 %
base16	6	1194409	78 (< 0.01 %)	100.00 %	0.00 %	0.00 %
cbor-codec	6	17292	656 (3.79 %)	36.59 %	63.41 %	0.00 %
hcid	5	1158079	100 (< 0.01 %)	100.00 %	0.00 %	0.00 %
utf8-width	4	1267697	168 (0.01 %)	100.00 %	0.00 %	0.00 %

Figure 6. This table gives rejection rates for all the tested libraries, and breaks down the rejections by categories (right 3 columns). For every row, the right 3 cells add up to 100 %, showing which kind of error was dominant in the compiler rejection. Libraries in which we found bugs are marked with a ★. For information about each bug, see Figure 7.

★	Bug Type	Min. Lines to Induce	Time to Discovery (s)	Accepted by Authors?
★1	Memory Leak	1	4.45	Yes
★2	Hanging Pointer	3	2850.3	Yes
★3	Use-After-Free	5	200.61	Yes
★4	OOB Pointer	4	238.99	Yes

Figure 7. Bugs Caught by SyRust

```
let x1_1: BitVec<Msb0,usize> = BitVec::with_capacity(0);
let mut x2_0  = x1_1;
let x3_0 = &mut x2_0;
x3_0.push(true);
let x5_0: BitBox<Msb0,usize> = x2_0.into_boxed_bitslice();
```

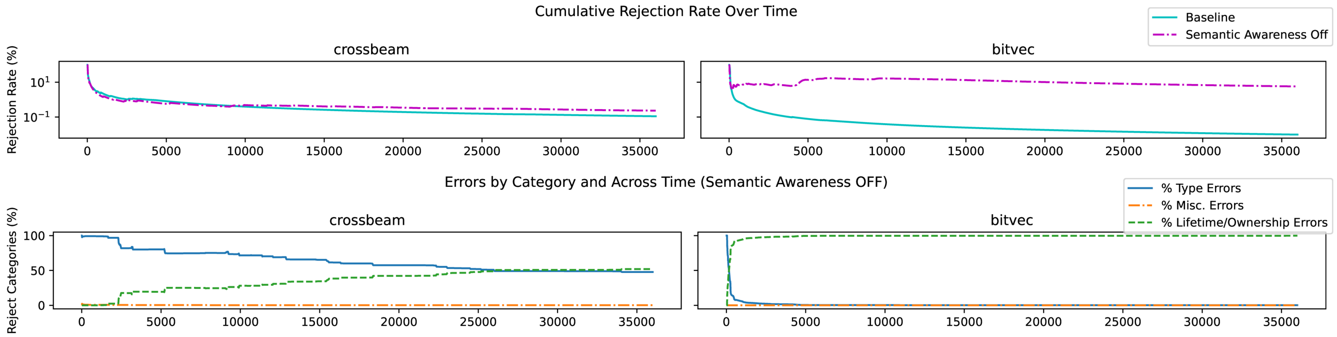
Figure 8. Buggy Case for bitvec

as simply having an out-of-bounds (OOB) or hanging pointer is not considered to be harmful as long as it is not dereferenced. However, Miri considers it to be buggy, because Rust's standard library provides the `std::mem::MaybeUninit` wrappers for dealing with pointers that may possibly be uninitialized. By not using `MaybeUninit`, the library risks becoming buggy if the compiler behavior changes at a later time. For example, if deallocation scheduling changes, it may cause a use-after-free bug because the compiler is not aware that this buggy pointer is pointing to.

Further discussions and the bug-inducing code are presented in the Technical Report [32].

7.2 RQ2 and RQ3: Effectiveness of SyRust Features

In RQ2 and RQ3, we will examine how SyRust's features contribute to its success in testing Rust libraries. RQ2 will compare fully featured SyRust against a variant of SyRust with semantic-aware constraints (Section 4.4) turned off; RQ3 will compare fully featured SyRust against a variant of SyRust



★	# Lines to Induce	Time to Discovery (s)	Increase in # Errors	Increase in # Lifetime/ Ownership Errors	Ownership Errors	Borrowing Errors
★2	2 (+0)	9879.8 (×3.47)	1475 (×2.08)	1467 (×123.25)	94.12 %	5.88 %
★3	4 (+0)	6007.07 (×30.25)	73460 (×613.17)	73454 (0 → 73454)	99.85 %	0.15 %

Figure 9. Selected libraries with semantic awareness turned off. The top row shows error rates regardless of kind. Blue is baseline and purple is with semantic awareness turned off. Second row of graphs show the portion occupied by each error kind. Green is Lifetime/Ownership, blue is type error, and orange is misc.

that employs a fully eager strategy to resolve polymorphic types instead of our hybrid API refinement (Section 5).

Before diving into the setup, we note a few restrictions on our evaluation. For RQ2, the semantic-aware constraints are highly dependent on each other. Turning off one will cause the rest of the constraints to behave incorrectly. We believe these constraints should be evaluated as a whole, leading to this on/off comparisons instead of fine-grained feature-for-feature comparisons. For RQ3, we cannot simply turn off the polymorphism strategy as our SAT encoding cannot reason about the infinite space of polymorphic type instantiations. A purely lazy strategy trivially fails as it cannot handle object constructors in Rust. This leads us to use a purely eager strategy from prior work [14] as a point of comparison.

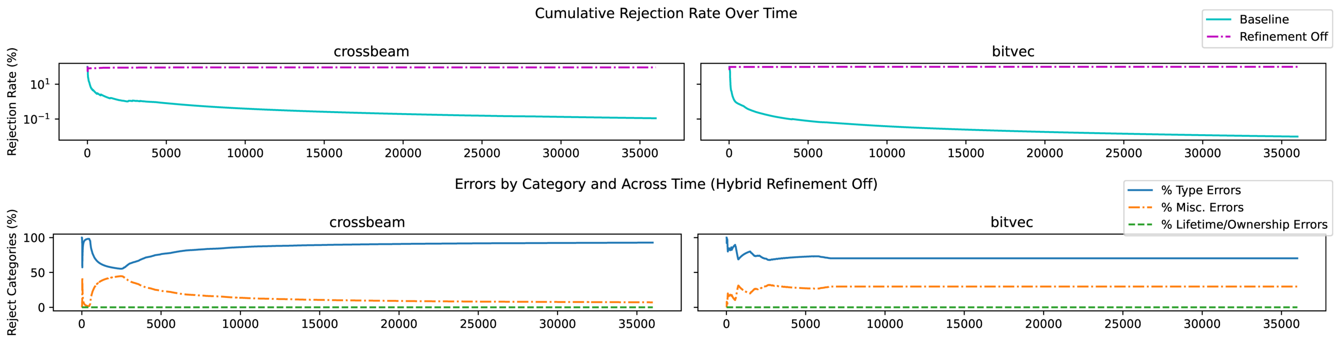
Using 2 of the 4 libraries that had previously unknown bugs, we attempt to replicate the results with the respective features turned off. We excluded `crossbeam-queue` (★1) and `encoding_rs` (★4) because the former is a 1-line bug, and the latter is extremely simple with respect to lifetime/ownership. Once again, we run every experiment for 10 hours. Our focus will be on the time-to-bug, as well as the rates of rejection by the compiler. We refer to the latter as the error rate.

Results are shown in Figure 9 and Figure 10. The top row of graphs plot the growth of errors over the execution of SyRust. The x-axis is time, from 0 to 10 hours. The y-axis is the percent of test cases up to that point that were rejected by the compiler, shown in logarithmic scale. This metric is cumulative, and the overall rejection rate is the y value of the rightmost point on the curve. Lower is better as it implies a low rejection rate. The dotted red line is SyRust with the respective feature turned off, and the solid blue is the baseline with all features. The second row is a breakdown of the errors by category, and across time. These graphs are only for executions with the respective feature turned off. The x-axis is again time, and the y-axis is the percent this error category occupies out of the total programs rejected by the compiler (in linear scale). The blue solid line corresponds to type errors, green dashed line corresponds to Lifetime/Ownership errors, and the orange dotted line corresponds to miscellaneous errors. This means the sum of the 3 lines is always 100%. The higher the line, the more dominant this particular error is. Finally, we provide a table with the size of the synthesized bug-inducing case, time-to-bug, error rates, error rates for the error category of interest, and the breakdown within the error category of interest. For the first 4 data points, we also provide the difference compared with the fully featured SyRust; the breakdown cannot be compared because the total error rate for that error category differs significantly.

We observe that turning off either semantic-awareness or hybrid polymorphic API refinement results in a significant increase in errors. There is a corresponding delay in bug discovery, with SyRust without hybrid polymorphic API refinement failing to find the bugs. Furthermore, these increases are driven by the error categories the respective features were meant to address. The negative effects are more severe for more complex API sets, with `bitvec` suffering larger increases in errors and time-to-bug than `crossbeam`. From this, we conclude that SyRust's features, semantic-aware constraints and hybrid API refinement, are critical to testing and finding bugs in Rust APIs, with more pronounced effects for more complex API sets.

7.3 Coverage

We further investigate the effectiveness of SyRust by measuring coverage attained by the test cases synthesized by SyRust and its variants. We measure line and branch coverage for both the subcomponent we tested (i.e., selected



★	# Lines to Induce	Time to Discovery (s)	Increase in # Errors	Increase in # Type Errors	Trait Errors	Polymorphism Errors	Misc. Errors
★2	Not Found	- (N/A)	4810914 (×3520.32)	4464091 (×3307.73)	0.00 %	95.11 %	4.89 %
★3	Not Found	- (N/A)	1150435 (×9587.96)	807844 (×6733.03)	100.00 %	0.00 %	0.00 %

Figure 10. Selected libraries with hybrid API refinement turned off. The top row shows error rates regardless of kind. Blue is baseline and purple is with refinement turned off. Second row of graphs show the portion occupied by each error kind. Green is Lifetime/Ownership, blue is type error, and orange is Misc.

APIs from) and the entire library. Since SyRust only uses 15 APIs, we expect to see a difference in the coverage between the entire library and the subcomponent. To identify coverage saturation point, coverage snapshots were taken at 900 second intervals. The results are shown in Figure 11, with bitvec denoted as BV and crossbeam denoted as CB. RQ1 is the fully featured SyRust; RQ2 is SyRust without semantic awareness; RQ3 is SyRust that has hybrid API refinement replaced by a purely eager strategy.

Library and RQ #	Component Line Coverage	Component Branch Coverage	Library Line Coverage	Library Branch Coverage
BV RQ1	76.67 %	34.88 %	61.87 %	31.95 %
BV RQ2	76.67 %	34.88 %	61.87 %	31.95 %
BV RQ3	0.5 %	17.39 %	26.82 %	28.89 %
CB RQ1	71.88 %	37.84 %	28.88 %	8.48 %
CB RQ2	71.23 %	37.67 %	28.24 %	8.48 %
CB RQ3	19.19 %	6.85 %	7.97 %	1.52 %

Figure 11. Library and Component Coverage

When looking at the component under test, fully featured SyRust has about 75% line coverage and 35% branch coverage in bitvec, with similar numbers reported for crossbeam. The low branch coverage is mainly caused by the lack of input mutations. The decrease of whole library coverage from component coverage is larger for crossbeam, because crossbeam is much larger than bitvec.

Comparing between the SyRust variants, we observe that the data for RQ1 and RQ2 are roughly identical. However, the coverage using tests generated by SyRust with all features turned on converges much faster than those from other variants. For bitcec, tests generated by the RQ2 variant with semantic awareness turned off took approximately 50% longer (2.25 hours) to reach saturation point than those generated by full-featured SyRust, which saturated after 1.5 hours. For crossbeam, tests generated by the RQ2 variant never reaches the same coverage. For crossbeam, tests generated by RQ1 SyRust saturated after 5.5 hours. Running RQ2 variant for a longer time (longer than the 10 hours that we allowed) would allow it to generate enough tests to reach coverage saturation. Finally, the RQ3 variant shows significantly worse coverage, demonstrating the necessity of our hybrid approach to polymorphic API specification refinement.

7.4 Limitations and Future Work

While our tool has been shown empirically to be capable of finding bugs, some limitations remain. Due to scalability issues, we are only able to use 15 APIs per library. Further, we do not mutating inputs to APIs. All of the above limit the tool's ability to exercise the library APIs. We believe addressing these limitations may allow us to achieve higher library coverage and find more bugs.

Some of the limitations are inherited from the Rust tooling. We expect the situation to improve as the Rust ecosystem evolves. Others are more fundamental and offer insight into a need for further work in this area. We discuss them below.

7.4.1 Closures and Asynchronous Functions. Recall that SyRust failed to generate valid test cases for libraries cookie-factory and jsonrpc-client-core. This is due to the lack of syntax support for closures. Closures in Rust are used to define anonymous functions that are treated as first-class objects. Because our synthesizer is limited to straight line code, defining new closures is not allowed. Because asynchronous functions rely on closures to perform callbacks, asynchronous APIs are off the table as well. Given some recent work in synthesis of asynchronous programs [9], we believe this will be an interesting area of research.

7.4.2 Inputs to the Test Program.

Currently, our test framework runs on user-provided input parameters, and we do not mutate them. Mutation of the inputs is likely to trigger more bugs, but we chose to focus on the orthogonal API problem for now. Because this problem may be addressed with Bounded Model Checking (BMC) techniques, we believe that extending our tool into a BMC tool in the style of CRUST [34] is an interesting path forward.

7.4.3 Optimal Scheduling of Tests.

We run the tests in whichever order the SAT formula was solved. This is not necessarily the best order, and we may be able to find bugs quicker if we prioritize certain programs. This is a well studied problem in the fuzzing community [37] and we expect that some of those techniques are applicable to our domain.

7.4.4 Miri Limitations.

As noted in Section 6, we use Miri [26] to run our test cases. While Miri will not miss any bugs, its precision comes at the cost of speed. Given all the checks performed, Miri is significantly slower than running the compiled code. In particular, some of the Stacked Borrows rules [20] results in cases where the loops of linear complexity require polynomial number of operations to interpret. This significantly limits our ability to test APIs that internally contain large loops and iterations.

8 Related Work

Automatic test case generation is useful for testing language interpreters and compilers [13, 16, 17, 39] and libraries [7, 12, 22, 24, 27]. Some mutate and piece together existing code segment [16, 19], some generate code from scratch [13, 39], and some like ours, generate API call chains [7, 12, 27].

A number of techniques have been proposed for API testing. Tools like RESTler [7] mutate call sequences [40]. FUDGE [8] and FuzzGen [18] generate fuzz drivers for C/C++ libraries from existing code bases. FUDGE uses recursive search to combine API calls. FuzzGen performs program analysis of API usage to generate A^{2}DG graphs that represent data and control dependencies and uses the graph to generates programs. Since these works are for C/C++, lifetime and ownership is not an issue.

Leveraging advances in constraint-solving, component-based program synthesis techniques [19] have been applied to many languages. Most closely related to ours, are SyPet [14] (Java) and H+ [15] (Haskell), which use constraint-solving-based technique to generate loop-free straight-line code. Both SyPet and H+ employ a graph-based encoding, which is unwieldy for encoding the lifetime and ownership of variables; and therefore SyRust does not use. To handle polymorphism, SyPet takes a completely eager approach to Java Generic Types. To support Haskell's polymorphism and partial evaluation, H+ starts by assuming every API takes and returns the universal polymorphic type and refines lazily from there. SyRust takes a hybrid approach. Since Rust has polymorphic struct constructors but not partial evaluation, we start with a mix of both concrete and polymorphic types and use the hybrid approach to refinement.

Dewey et al. [13] encode the Rust syntax and semantic constraint in Prolog [11] to generate code to test the Borrow Checker. Their lifetime and ownership constraints are roughly equivalent to ours. They generate programs with complex structures such as loop nesting and define and use of data structures. Because they aim to test the compiler, they only support polymorphism through a few standard library features like Box<T> and cannot be easily modified for API testing.

Several projects aim to formalize Rust's operational semantics [21, 29, 35], Stacked Borrows [20], being the most recent. Stacked Borrows is used as a correctness model in Miri [26], a component in SyRust to detect unsound behaviors.

Finally, Oxide [36] formalizes static checks of Rust. We could connect to it to formally verify the completeness of our constraints as future work.

9 Conclusion

We have proposed a semantic-aware synthesis algorithm for Rust that effectively synthesizes valid API usage across a wide range of libraries exercising complex language features like ownership and polymorphism. Our algorithm encapsulates Rust's compiler checks through a logical encoding and leverages a hybrid API refinement strategy to drive polymorphic APIs. We implement our algorithm in SyRust, a library testing framework for Rust. Our experiments demonstrate SyRust's ability to generate valid test cases for a wide range of libraries and find 4 confirmed bugs. In future work we plan to investigate whether coverage information can be used to improve the synthesis of valid test cases. We also plan to apply our framework to testing the Rust compiler and to formally prove the correctness of the synthesis.

Acknowledgments

This work was supported in part by the National Science Foundation via grant CCF-1901136. We would like to thank our shepherd Ben Hardekopf and the anonymous reviewers for helping improve our paper. We would also like to thank the anonymous artifact reviewers for helping improve our artifact and Darion Cassel for helping test our artifact before submission. Finally, we thank Ralf Jung for helping us understand Miri's output and Maverick Woo for maintaining the compute cluster on which our evaluation was done.

References

[1] 2018. *Built In Rust.* https://www.rust-lang.org/what/

[2] 2018. *CVE-2018-1000810.* https://nvd.nist.gov/vuln/detail/CVE-2018-1000810

[3] 2020. *Built In Rust.* https://doc.rust-lang.org/nomicon/lifetime-elision.html

[4] 2020. *Rust's Unsafe Code Guidelines Reference.* https://rust-lang.github.io/unsafe-code-guidelines/

[5] 2020. *Trait and lifetime bounds.* https://doc.rust-lang.org/reference/trait-bounds.html

[6] 2020. *Undefined Behavior in bounded channel.* https://github.com/RustSec/advisory-db/blob/main/crates/crossbeam-channel/RUSTSEC-2020-0052.md

[7] 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.

[8] 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

[9] Suguman Bansal, Kedar S. Namjoshi, and Yaniv Sa'ar. 2018. Synthesis of Asynchronous Reactive Programs from Temporal Specifications. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*. Springer International Publishing, 367–385.

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

[11] D.L. Bowen, L.H. Byrd, and William Clocksin. 1983. A portable Prolog compiler. In *D.A.I. research paper*. 9.

[12] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An Automatic Robustness Tester for Java. *Journal of Software: Practice and Experience (SPE)* 34, 11 (Sept. 2004), 1025–1050. https://doi.org/10.1002/spe.602

[13] 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

[14] 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.

[15] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program Synthesis by Type-Guided Abstraction Refinement. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 12:1–12:28. https://doi.org/10.1145/3371080

[16] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

[17] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the USENIX Security Symposium*. USENIX Association, USA, 445–458.

[18] 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.

[19] 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.

[20] 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.

[21] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, Article 66, 34 pages. https://doi.org/10.1145/3158154

[22] Y. Kim, Y. Kim, Taeksu Kim, Gunwoo Lee, Y. Jang, and M. Kim. 2013. Automated unit testing of large industrial embedded software using concolic testing. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 519–528.

[23] Linux Test Project. 2020. *lcov 1.15.* https://github.com/linux-test-project/lcov

[24] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. 2015. GRT: An Automated Test Generator Using Orchestrated Program Analysis. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 842–847.

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

[26] Scott Olson. [n.d.]. *Miri.* https://github.com/rust-lang/miri

[27] 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/

[28] 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.

[29] Eric Reed. 2015. *Patina: A formalization of the Rust programming language.* Technical Report. University of Washington.

[30] Rust Foundation. 2020. *cargo 1.47.0.*

[31] Takehide Soh, Daniel Le Berre, Stéphanie Roussel, Mutsunori Banbara, and Naoyuki Tamura. 2014. Incremental SAT-Based Method with Native Boolean Cardinality Handling for the Hamiltonian Cycle Problem. In *Proceedings of the European Conference on Logics in Artificial Intelligence (JELIA)*, Vol. 8761. Springer, 684–693.

[32] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina Pasareanu. 2021. *SyRust: Automatic Testing of Rust Libraries with Semantic-Aware Program Synthesis–Technical Report.* Technical Report. Carnegie Mellon University. https://doi.org/10.1184/R1/14356949

[33] O. Tange. 2011. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine* 36, 1 (Feb 2011), 42–47. http://www.gnu.org/s/parallel

[34] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. CRUST: A Bounded Verifier for Rust. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Press, 6 pages.

[35] F. Wang, F. Song, M. Zhang, X. Zhu, and J. Zhang. 2018. KRust: A Formal Executable Semantics of Rust. In *Proceedings of the International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE Computer Society, 44–51.

[36] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *CoRR* abs/1903.00982 (2019). arXiv:1903.00982 http://arxiv.org/abs/1903.00982

[37] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-Box Mutational Fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*. Association for Computing Machinery, 511–522.

[38] Hui Xu, Zhuangbin Chen, Mingshen Sun, and Yangfan Zhou. 2020. Memory-Safety Challenge Considered Solved? An Empirical Study with All Rust CVEs. arXiv:2003.03296 https://arxiv.org/abs/2003.03296

[39] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 283–294.

[40] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. Fuzzing APIs. In *The Fuzzing Book*. Saarland University. https://www.fuzzingbook.org/html/APIFuzzer.html Retrieved 2019-12-17 16:45:28+01:00.

A Target Library Statistics

Figure 12 provides a list of libraries used for RQ1, along with statistics about popularity, the presence of polymorphic APIs, the subcomponent that was targeted by the test, and finally, a revision hash of the library used.

B Bug Inducing Code

We will also provide source code for the bugs found and details surrounding the bug. For readability, the code presented is a simplified version of what is actually synthesized.

C Synthesis as Boolean SAT

In prior Section 4, we have built up a logical encoding that encapsulates the typing and ownership rules of Rust. In this section, we aim to provide the exact formulas of how we encode the SAT formula.

Let us begin by considering the exact layout of SAT variables. Recall from Section 4 that we had API variables $A_{f,i}$ whose value denotes the use of API f in line i, a synthesis context variables $V_{x,\tau,i}$ whose value denotes the availability of Rust program variable x of type τ in line i. We also carry over other notations from Section 4 such as the line count l

In addition to the former 2 categories, we define an auxiliary variable set called *Use Variables*. Use variable $U^{V_{x,\tau,i}}_{j,f}$ is true if and only if variable x, represented by $V_{x,\tau,i}$ is used as the jth input to f. We cannot forget the index j as some APIs may have multiple inputs of the same type.

The remainder of this section is provided in Figure 14. The table displays the exact SAT formulas with the left column associating each individual formula with the Rules in Section 4. For rules that were too primitive to be discussed explicitly, we label them as "Primitive." Rule 7 will not show up as it is checked via post-processing. Note that some constraints are pseudo-Boolean instead of SAT, but SAT4J supports pseudo-Boolean constraints natively [10].

We consider each row of the table.

1. $\forall_{1<i\le l} V_{x,\tau,i} \implies V_{x,\tau,i-1}$. This rules enforces the property that, if a variable is active in line i, then it must be active in the previous line. This property does not apply if a predecessors $i-1$ does not exist. This happens when the variable is freshly created on line $i-1$, or when $i=0$, in which case, this rule is not applicable.
2. $\forall_{1<i\le l} \sum_{f\in\mathcal{A}} A_{i,f} = 1$. This rule set enforces the property that only one function can be called on a given line. This is trivially required.
3. $\forall_{\substack{1\le i\le l \\ f\in\mathcal{A} \quad 1\le j\le |inputs\ f|}} U^{V_{x,\tau,i}}_{j,f} \implies V_{x,\tau,i}$. This rule enforces the property that if a function call on some line uses a variable, then, the used variable must exist first. Otherwise, the synthesized programs will contain references to unbound variables.
4. $\forall_{\substack{1\le i\le l \\ f\in\mathcal{A} \quad 1\le j\le |inputs\ f|}} U^{V_{x,\tau,i}}_{j,f} \implies A_{i,f}$. This rule is the dual of the previous rule, enforcing the property that, if a variable is used, then the function call that uses it must be active on the same line.
5. $\bigwedge_{x:\tau\in\mathcal{T}} V_{x,\tau,0}$. This rule forces true all variables that are passed to the test function by the template.
6. $\forall_{\substack{1\le i\le l \\ f:*\to\tau}} A_{i,f} \iff V_{x_{i+1},\tau,i+1}$. This rule links the function call and its output. A function call is on a line $\iff$ the output is true on the next line.
7. $\forall_{\substack{1\le i\le l \\ f:\tau_1,\ldots,\tau_2\to\tau}} A_{i,f} \implies \forall_{1\le j\le k} \left(\sum_{\tau\sqsubseteq\tau_j} U^{V_{x,\tau,i}}_{j,f} \right) \ge 1$. If a function call is made, then all of its inputs must be satisfied. This property is enforced by requiring that, for every input index, at least one use varaible in the corresponding set is true.
8. $\forall_{\substack{1\le i\le l \\ f:\tau_1,\ldots,\tau_2\to\tau}} \forall_{\substack{1\le j_1<k_2\le k \\ \tau\sqsubseteq\tau_j,\ \tau'\sqsubseteq\tau_{j'} \\ \neg\text{compatibleTypes}(C_i,x,\tau_j,x',\tau_{j'})}} U^{V_{x,\tau,i}}_{j,f} \implies \neg U^{V_{x',\tau',i'}}_{j',f}$. For every pair of use variables in which the usage is incompatible (discussed in Section 4), we enforce the rule that if one of the elements of the pair is true, then the other must be false. This is roughly equivalent of putting an XOR constraint for every said pair.
9. $\forall_{1<i\le l} \left(\sum_{f\in\mathcal{A} \quad 1\le j\le |inputs\ f|} U^{V_{x,\&mut\,\tau\,\mathrm{or}\,\tau,i}}_{j,f} \right) \le 1$. Normal types and mutable refernces can only be used once in line.
10. This rule enforces the property that standard objects (which are subject to Rust's ownership rules) can be used once, passed along to the next line, but not both (i.e. ≤ 1). Note that both mutable and immutable borrowing is excluded. They will be dealt with in the next 3 rules.
11. This rule enforces the property that all borrowed references outlive the object it refers to. In some cases (such as wrapper types around references), this rule is insufficient, so Rule 7 applies via post-processing.
12. This rule prevents another borrow while a mutable reference is active.
13. This rule is identical to the previous one, but for immutable references. Since we can have multiple immutable references to the same object, we allow immutable borrow operations, but deny mutable borrows

Library Name	Cat.	Total Downloads	Polymorphism	Subcomponent	Rev. Hash
smallvec	DS	21780282	Yes	smallvec::SmallVec	9ae7076
crossbeam-utils	DS	19491917	Yes	crossbeam_utils::atomic::AtomicCell	5a68889
bytes	DS	16302396	No	bytes::BytesMut	b7f7582
slab	DS	15575908	Yes	slab::Slab	e6b8676
crossbeam-deque	DS	15140300	Yes	crossbeam_deque::Injector	5a68889
generic-array	DS	12145172	Yes	generic_array::GenericArray	04fe34c
crossbeam-queue	DS	10081038	Yes	crossbeam_queue::ArrayQueue	5a68889
num-rational	DS	7250507	No	num_rational::Ratio	bb4c920
hashbrown	DS	6577360	Yes	hashbrown::HashSet	34c1189
crossbeam	DS	5645952	No	crossbeam::epoch::Collector	5a68889
petgraph	DS	4538136	Yes	petgraph::graph::Graph	397b9fc
im-rc	DS	916529	Yes	im::ordset::OrdSet	b586a96
bitvec	DS	799016	No	bitvec::vec::BitVec	293e670
ndarray	DS	684962	Yes	ndarray::ArrayBase	9cba023
dashmap	DS	465022	Yes	dashmap::DashMap	b2951f8
encoding_rs	EN	7344939	No	Decoder	8e3eee5
bstr	EN	5789836	No	bstr::BString	7f0ad15
csv-core	EN	4144518	No	csv_core::Reader::	70c8600
data-encoding	EN	2240282	No	data_encoding::Encoding	34d1f0e
encode_unicode	EN	1985895	No	encode_unicode::Utf8Char	47f8483
urlencoding	EN	1119712	No	urlencoding::	a86f1c4
rmp-serde	EN	816677	Yes	rmp_serde::	00eeadf
bytemuck	EN	727756	No	bytemuck	68ed5fe
sval	EN	414356	No	sval::stream::OwnedStream	c432b60
cookie-factory	EN	292900	No	cookie_factory	a935a81
base16	EN	133173	No	base16	a532182
cbor-codec	EN	108378	No	decoder::Decoder	ea76c0c
jsonrpc-client-core	EN	78992	No	example::ExampleRpcClient	4fde208
hcid	EN	75423	No	hcid::HcidEncoding	2caee15
utf8-width	EN	64822	No	utf8_width	938c0b2

Figure 12. Libraries selected from `cargo.io` for evaluation. Libraries were selected from 2 popular categories: data structures (DS) and encodings (EN). For some libraries, we tested a subcomponent. The import path of the item tested is in the subcomponent column.

to make immutable references exclusive from mutable references.

Code	Description

`fn buggy() {`
`  // bug inducing code`
`  let x1_0: ArrayQueue = crossbeam_queue::ArrayQueue::new(1);`
`}`
	crossbeam-queue (★1) has a deallocation bug when the internal capacity (specified in the constructor) is bigger than the number of elements. This is induced by simply creating an ArrayQueue object with initial allocation size > 1, and deallocating without putting any elements in it.
	The deallocation occurs by converting `Arrayqueue`'s internal memory into a `Vec`, and dropping the memory using `Vec`'s standard deconstructor. This means differing size will cause `Vec` to drop incorrect amount of memory, possibly leading to memory corruption [6].

`fn buggy() {`
`  // bug inducing code`
`  let x1_0: crossbeam::epoch::Collector = crossbeam::epoch::Collector::new();`
`  let x2_1: & crossbeam::epoch::Collector = & x1_0;`
`  let x3_0: crossbeam::epoch::LocalHandle = x2_1.register();`
`}`
	crossbeam (★2) creates a dangling pointer during deallocation. Like with the previous bug, simply having a dangling pointer is considered to be a bug unless it is done through specific methods (i.e. `MaybeUninit`)

`fn buggy() {`
`// inputs`
`let bool_0_0: bool = true;`
`let usize_0_1: usize = 0;`
`let bitvecordermsb0_0_2: bitvec::order::Msb0 = bitvec::order::Msb0;`
 	
`// bug inducing code`
`let x1_1: = bitvec::vec::BitVec::with_capacity(usize_0_1);`
`let mut x2_0:  = x1_1;`
`let x3_0: &mut = &mut x2_0;`
`x3_0.insert(usize_0_1, bool_0_0);`
`let x5_0: = x2_0.into_boxed_bitslice();`
`return;`
`}`
	bitvec (★3) has a use-after-free bug The above code creates a bitvector, then casts it as mutable and borrows it. Using the borrowed reference, it inserts a new bit into the bitvector. Finally, it converts the bitvector into BitBox, a data structure that serves as a rough approximation of a pointer into the heap. The deallocation of the BitBox triggers the use-after-free.
	This bug is notable because it cannot be triggered by a static fuzzing harness. This is because the `into_boxed_bitslice` operation moves value out of `bitvec`, preventing this from called within a loop of a fuzz harness driven using random input.

`fn buggy() {`
`  //inputs`
`  let s_0 : & [u8] = "";`
`  let b_0 : &mut [u16] = &mut Vec::with_capacity(0)[..];`
`  let t : bool = true;`
 	
`  // bug inducing code`
`  let x1_1 = encoding_rs::UTF_8.new_decoder_without_bom_handling();`
`  let x2_1 = mut x1_1;`
`  let x3_2 = &mut x2_1;`
`  let x4_5 = x3_2.decode_to_utf16_without_replacement(string, b_0, t);`
`}`
	encoding_rs (★4) creates a out-of-bound pointer. In Miri's dynamic semantics, creating a pointer out of bounds (not just dereferencing) is considered to be undefined behavior. This occurs when converting from UTF-8 to UTF-16 as the pointer scans an array to find alignment.

Figure 13. Bugs SyRust found.

Rule	Constraint
primitive	$\forall_{1<i\le l} V_{x,\tau,i} \implies V_{x,\tau,i-1}$
primitive	$\forall_{1<i\le l} \sum_{f\in\mathcal{A}} A_{i,f} = 1$
primitive	$\forall_{\substack{1\le i\le l \\ f\in\mathcal{A} \quad 1\le j\le \lvert inputs\ f\rvert}} U_{j,f}^{V_{x,\tau,i}} \implies V_{x,\tau,i}$
primitive	$\forall_{\substack{1\le i\le l \\ f\in\mathcal{A} \quad 1\le j\le \lvert inputs\ f\rvert}} U_{j,f}^{V_{x,\tau,i}} \implies A_{i,f}$
Rule 1	$\bigwedge_{x:\tau\in\mathcal{T}} V_{x,\tau,0}$
Rule 2	$\forall_{\substack{1\le i\le l \\ f:*\to\tau}} A_{i,f} \iff V_{x_{i+1},\tau,i+1}$
Rule 3	$\forall_{\substack{1\le i\le l \\ f:\tau_1,\ldots,\tau_2\to\tau}} A_{i,f} \implies \forall_{1\le j\le k} \left(\sum_{\tau\sqsubseteq\tau_j} U_{j,f}^{V_{x,\tau,i}} \right) \ge 1$
Rule 3	$\forall_{\substack{1\le i\le l \\ f:\tau_1,\ldots,\tau_2\to\tau}} \forall_{\substack{1\le j_1<k_2\le k \\ \tau\sqsubseteq\tau_j,\ \tau'\sqsubseteq\tau_{j'} \\ \neg\text{compatibleTypes}(C_i,x,\tau_j,x',\tau_{j'})}} U_{j,f}^{V_{x,\tau,i}} \implies \neg U_{j',f}^{V_{x',\tau',i'}}$
Rule 3	$\forall_{1<i\le l} \left(\sum_{f\in\mathcal{A} \quad 1\le j\le \lvert inputs\ f\rvert} U_{j,f}^{V_{x,\&mut\,\tau\,\mathrm{or}\,\tau,i}} \right) \le 1$
Rule 3, 4, 5	$\forall_{\substack{1<i\le l \\ isNonPrimitiveObj\ \tau \\ f\neq\texttt{borrow\_mut,borrow}}} \left(\sum_{f\in\mathcal{A} \quad 1\le j\le \lvert inputs\ f\rvert} U_{j,f}^{V_{x,\tau,i}} \right) + V_{x,\tau,i+1} \le 1$
Rule 6	$\forall_{\substack{1<i\le l \\ f=\texttt{borrow\_mut,borrow}}} U_{j,f}^{V_{x,\tau,i}} \implies \forall_{i<i_2\le l} V_{x_{i+1},(\&\text{ or }\&mut\ \tau),i_2} \implies V_{x,\tau,i_2}$
Rule 8	$\forall_{1<i\le l} U_{j,\texttt{borrow\_mut}}^{V_{x,\tau,i}} \implies \forall_{\substack{i+1<i_2\le l \\ f_2=\texttt{borrow\_mut,borrow}}} V_{x_{i+1},\&mut\ \tau,i_2} \implies \neg U_{j,f_2}^{V_{x,\tau,i_2-1}}$
Rule 9	$\forall_{1<i\le l} U_{j,\texttt{borrrow}}^{V_{x,\tau,i}} \implies \forall_{i+1<i_2\le l} V_{x_{i+1},\&mut\ \tau,i_2} \implies \neg U_{j,\texttt{borrow\_mut}}^{V_{x,\tau,i_2-1}}$

Figure 14. Exact SAT formulas used for the encoding.