RPG: Rust Library Fuzzing with Pool-based Fuzz Target Generation and Generic Support

Zhiwu Xu
CSSE, Shenzhen University
Shenzhen, China

Bohao Wu
CSSE, Shenzhen University
Shenzhen, China

Cheng Wen*
Guangzhou Institute of Technology, Xidian University
Guangzhou, China

Bin Zhang
CSSE, Shenzhen University
Shenzhen, China

Shengchao Qin*
Fermat Labs, Huawei
Hong Kong, China

Mengda He
Fermat Labs, Huawei
Hong Kong, China

ABSTRACT

Rust libraries are ubiquitous in Rust-based software development. Guaranteeing their correctness and reliability requires thorough analysis and testing. Fuzzing is a popular bug-finding solution, yet it requires writing fuzz targets for libraries. Recently, some automatic fuzz target generation methods have been proposed. However, two challenges remain: (1) how to generate diverse API sequences that prioritize unsafe code and interactions to reveal bugs in Rust libraries; (2) how to provide support for the generic APIs and verify both syntactic and semantic validity of the fuzz targets to enable more comprehensive testing of Rust libraries. In this paper, we propose RPG, an automatic fuzz target synthesis technique to support Rust library fuzzing. RPG uses a pool-based search to generate diverse and unsafe API sequences, and synthesizes fuzz targets with generic support and validity check. The experimental results demonstrate that RPG enhances both the quality of the generated fuzz targets and the bug-finding ability through pool-based generation and generic support, substantially outperforming the state-of-the-art. Moreover, RPG has discovered 25 previously unknown bugs from 50 well-known Rust libraries available on *Crates.io*.

ACM Reference Format:
Zhiwu Xu, Bohao Wu, Cheng Wen, Bin Zhang, Shengchao Qin, and Mengda He. 2024. RPG: Rust Library Fuzzing with Pool-based Fuzz Target Generation and Generic Support. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639102

1 INTRODUCTION

Rust is a promising programming language that aims to provide memory safety without sacrificing performance [27, 41]. Nevertheless, security bugs may still be present in Rust software due to logic errors, unsafe code blocks, or external dependencies [19, 26, 34]. An interesting phenomenon of these bugs is that most of their host programs are libraries [24]. Note that the Rust libraries are commonly used in software development to write code that is safe, efficient, and expressive across different applications and platforms. Therefore, assuring the safety and robustness of the Rust libraries remains paramount because of their expanding and diverse ecosystem.

Existing studies on Rust widely utilize static analysis (*e.g.*, Rudra [4], SafeDrop [8], MirChecker [31]) or dynamic analysis (*e.g.*, Miri [35] and UnsafeFencer [22]) to detect bugs. Note that a library cannot be executed as a standalone program; instead, it is invoked through another application, which must provide the correct calling context to invoke the library functions. This makes it difficult to directly apply existing static and dynamic analysis techniques due to the lack of calling contexts.

Fuzzing is a dynamic testing technique that involves passing random or semi-random inputs to a software system and observing its behavior for crashes or anomalies [18, 47–50]. To test a library, a fuzzer requires fuzz targets that can execute certain codes within a library [3, 29, 43, 52]. Inputs can then be passed into this target for further testing. Unfortunately, writing fuzz targets remains a primarily manual exercise, a major hindrance to the widespread adoption of library fuzzing. Several recent studies, including RULF [24], RustyUnit [44] and SyRust [42], propose to automatically synthesize a fuzz/test target for testing to detect bugs. However, existing studies often prioritize API or code coverage, neglecting the detection of bugs caused by suspicious usage of library functions, such as invoking functions with unsafe code or involving multiple functions that manipulate the same data. Additionally, these studies either lack proper support for generics or only offer limited support, leading to inadequate testing capabilities. The main challenges in this context are as follows: (1) how to generate diverse API sequences that prioritize unsafe code and interactions to reveal more bugs in Rust libraries; (2) how to provide support for the generic functions and synthesize valid fuzz targets, conforming to the strong type system [2, 14, 33, 46] and the ownership-based memory management scheme [25, 28, 32, 38], to enable more comprehensive testing of Rust libraries.

To address the aforementioned problems, we present RPG[1], an automatic fuzz target synthesis technique to support Rust library fuzzing. RPG consists of two key contributions: (1) a novel pool-based method to generate diverse API sequences that prioritize unsafe code blocks and interactions; (2) fuzz target synthesis with

*Corresponding authors: Cheng Wen and Shengchao Qin.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639102

[1]The acronym of **R**ust **P**ool-based fuzz target **G**eneration.

generic support and validity checks. Specifically, RPG primarily leverages static analysis to obtain rich type information and metadata available in Rust libraries and constructs an API dependency graph. It then generates diverse API sequences by prioritizing unsafe code and dependencies through a pool-based generation. To support calling generic functions, RPG performs type inference among API dependencies with the support of a parameter provider containing library-defined data types and commonly used data structure types. RPG performs additional checks (i.e., a move-borrow check and a generic declaration check) on the API sequences to ensure syntactic and semantic validity, that is, the sequences do not contain compiler errors related to mutability, moving, borrowing, and generic declaration. Lastly, the API sequence set is synthesized into the corresponding fuzz targets that can be utilized by a popular fuzzer (*e.g.*, AFL++ [15] in our experiments) to detect bugs.

We have implemented a prototype of RPG and performed a thorough evaluation on 50 popular Rust libraries from *crates.io*. The obtained results demonstrate that RPG is able to generate valid fuzz targets, achieving an API coverage rate of 71.8% and a dependency coverage rate of 11.1%. By utilizing fuzz targets generated by RPG for further fuzzing, RPG is able to detect more bugs compared to the state-of-the-art approaches (*i.e.*, RULF [24], Miri [35] and Rudra [4]). These bugs include 25 previously unknown bugs, confirmed by the library maintainers.

Our main contributions are summarized as follows:

- We propose a novel pool-based method to generate diverse API sequences that expose bugs in Rust libraries. By prioritizing both unsafe APIs and API dependencies, we construct an API pool that enables us to generate API sequences capable of uncovering more bugs.
- We provide generic support and validity checks to synthesize more fuzz targets. We propose a local type inference approach that considers API dependencies and utilizes a preset parameter provider. In addition, we incorporate a move-borrow check and a generic declaration check to ensure the validity of API sequences. This comprehensive approach enables us to generate more fuzz targets for thorough testing of Rust libraries.
- We implement and evaluate RPG on 50 popular Rust libraries. Our experimental results show that RPG enhances both the quality of the generated fuzz targets and the bug-finding ability through pool-based generation and generic support, substantially outperforming the state-of-the-art.

2 BACKGROUND AND MOTIVATION

2.1 The Rust Programming Language

Rust is a systems programming language designed for memory safety and concurrency. With a strongly-typed and compiled nature, Rust's strict type system and ownership system enforce rules to ensure memory safety.

In Rust, each value (*e.g.*, a variable, string, or array) has a unique owner (the variable binding), which determines the lifetime of the value. Ownership is moved between owners, making the value no longer accessible from its original binding [28]. Rust also uses a borrowing mechanism [25, 38], which allows values to be temporarily borrowed without invalidating the ownership. References are either mutable or immutable, and the key rule is "no mutable aliasing". As long as a value is read-only, multiple references are safe. When a value is writable, only one mutable reference is allowed. These restrictive rules for moving and borrowing ensure memory safety. Rust does not require reference counting or garbage collection since resources' lifetimes are bounded by the objects' lifetimes. The Rust compiler deallocates resources automatically when their owners go out of scope. All of this is done at compile time, thus introducing no runtime overhead.

Although Rust's ownership rules and lifetime system ensure the safety of code, developers need to use unsafe Rust code to achieve higher performance or perform low-level operations. Unsafe code is marked using the "unsafe" keyword, which can violate Rust's safety guarantees and result in undefined behavior [39, 51]. By testing unsafe code first, potential security vulnerabilities can be discovered earlier, helping to improve the security and quality of Rust libraries.

2.2 Motivating Example

We give a motivating example to illustrate the challenge of testing and finding bugs in the Rust library. Specifically, we illustrate our observation which motivates the design of RPG.

Listing 1 shows a variant example from Rust *std* library [39], containing an implementation of the `Queue` data structure. This library defines a generic struct called `Queue` that can store values of any type `T`. It has a field called `qdata` that is a vector of type `T`. It also implements some functions (*a.k.a*, APIs) for the `Queue` struct, such as `new()`, `push()`, `pop()`, `peek()`.

- `new()`: This function creates a new `Queue` instance with an empty vector as `qdata`.
- `push()`: It takes a mutable reference to `self` and an item of type `T` as parameters. It adds the item to the end of the `qdata` vector using the function `push()` of `Vec`.
- `pop()`: This function takes an immutable reference to `self` as a parameter and calls the function `remove()` on the vector to remove the first `i` elements, by first converting a reference to the vector into a raw mutable pointer.
- `peek()`: It takes a reference to `self` as a parameter. It uses raw pointers and unsafe code to get a mutable reference to the first element of the `qdata` vector and return it as `Some(&mut *raw)`, if the `qdata` vector is not empty. Otherwise, it returns `None`.

This data structure uses unsafe code to manipulate the `qdata` vector without checking its bounds or ownership. This can lead to memory errors or undefined behavior if used incorrectly.

Current static analysis tools such as Rudra [4], SafeDrop [8], and MirChecker [31], as well as dynamic analysis tools like Miri [35], were unable to detect any defects in this particular data structure. This is due to the lack of a calling context and the absence of defects within a single function. Several existing Rust library testing techniques, such as SyRust [42] and RULF [24], automatically synthesize test/fuzz targets for dynamic testing to detect bugs. However, they are unable to effectively produce sequences of API calls that trigger vulnerabilities. On the one hand, they mainly focus on the coverage of a single API instead of API sequences, thus cannot detect any bugs that arise from interactions within a complex API sequence. On the other hand, the `Queue` could store values of any type `T`, yet

Listing 1: Unsafe Queue

```rust
pub struct Queue<T> {
  qdata: Vec<T>,
}
impl<T> Queue<T> where T:std::fmt::Display, T:Clone {
  // Create a Queue
  pub fn new() -> Self {
    Queue { qdata: Vec::new() }
  }
  // Add item to the Queue
  pub fn push(&mut self, item: T) {
    self.qdata.push(item);
  }
  // pop the top i item from the Queue
  // And free the pointer
  pub fn pop(&self, usize i) {
    let l = self.qdata.len();
    if l > i {
      for n in 0 .. i + 1 {
        let raw = &self.qdata as *const Vec<T> as *mut
            Vec<T>;
        unsafe { (*raw).remove(0); }
      }
    } else { None }
  }
  // Get item from Queue and get pointer
  pub fn peek(&self) -> Option<&mut T> {
    if !self.qdata.is_empty() {
      let raw = &self.qdata[0] as *const T as *mut T;
      unsafe { Some(&mut *raw) }
    } else { None }
  }
}
```

Listing 2: An API Sequence that Triggers the Use-After-Free

```rust
fn test() { /* API Seq: Queue::new() // T -> String
                        Queue::push()
                        Queue::peek()
                        Queue::pop()
                        Use() */
  let mut q: Queue<String> = Queue::new();
  q.push(String::from("hello"));
  let e = q.peek().unwrap();
  q.pop(0);
  println!("{}", *e); }// <-use-after-free
```

these techniques like RULF do not support generating API calls involving generic types.

However, a use-after-free vulnerability exists in the library because the function `pop()` releases the first element of the `Queue`, while the function `peek()` retains a pointer to this element. This can lead to potential use-after-free memory safety issues if the pointer is subsequently used. Listing 2 shows an API sequence that triggers the vulnerability. The functions `new()` and `push()` are required prerequisites for the subsequent callings. `peek()` retrieves a pointer to the first element of the Queue, which is then removed and freed by `pop()`. Further use of the pointer obtained by `peek()` can then result in a use-after-free vulnerability[2].

[2]Due to the encapsulation of unsafe code blocks within the safe function, these unsafe code blocks in `Queue` are not visible to the user, which indicates that the library is responsible for this vulnerability.

2.3 Design Inspiration

The motivation behind this paper is to design a library fuzzing method that can both detect bugs resulting from complex API sequence interactions and accommodate strict Rust features, like generic types. Our approach can easily generate an API sequence, such as the one shown in Listing 2, and then synthesize it into a valid fuzz target, enabling us to easily detect the use-after-free bug.

Compared to the existing works, we have made advances and improvement in the following three ways:

(1) Unsafe APIs and Dependencies. Prioritizing the testing of unsafe code blocks can expose hidden vulnerabilities in Rust libraries more effectively as such code often exceeds Rust's safety limitations and can result in potential security risks. Therefore, it is necessary to pay more attention to APIs containing unsafe code blocks (referred to as unsafe APIs), such as `pop()` and `peek()` in listing 1. Moreover, Existing approaches mainly focus on API coverage, and thus cannot detect bugs that arise from interactions within a complex API sequence. In this paper, we will prioritize testing the unsafe API and the API interactions. Firstly, we construct a labeled version of the API dependency graph to capture the presence of unsafe code and the interactions between APIs. Secondly, we attempt to initiate the generation process using unsafe APIs, according to the dependency graph. Finally, we give priority to unsafe APIs or APIs with more dependencies if there are multiple suitable options.

(2) Pool-based Generation and Validity Checks. To ensure the comprehensiveness and accuracy of testing, it is necessary to pay attention to the generation of fuzz/test targets, especially to consider the generation of long sequences to cover all possible scenarios. In this paper, we propose a novel pool-based sequence generation method and several sequence checks to effectively solve the above problems. Firstly, we construct an API pool, where an API is consumed from the API pool if it is generated, and the number of (occurrences of) each API depends on the presence of unsafe code and its number of potential interactions. Secondly, we introduce a heuristic algorithm to select APIs from the pool, and the greater the number of API, the higher the priority to be selected. If the proportion of the consumed APIs exceeds an exploration threshold, the pool will be reset to facilitate the generation of longer API sequences. Finally, we perform additional checks on the API sequences to ensure syntactic and semantic validity.

(3) Generic Support. Generics parameterize data structures and functions to enable different types can reuse the same code, which is widely employed in library and framework development in Rust [12, 53]. However, existing automated methods for generating fuzz targets are ineffective in calling generic functions. Handling generics is challenging as it requires inferring various suitable concrete types for generic type parameters and ensuring their consistency in an API sequence with the least cost.

We propose a scheduling method for calling generic functions. Specifically, during sequence generation, we employ type class constraints to limit the input and output types of each generic function. We then utilize a type inference engine that incorporates a parameter provider to infer the concrete types of generic parameters and transmit them to other functions. Throughout the sequence, the

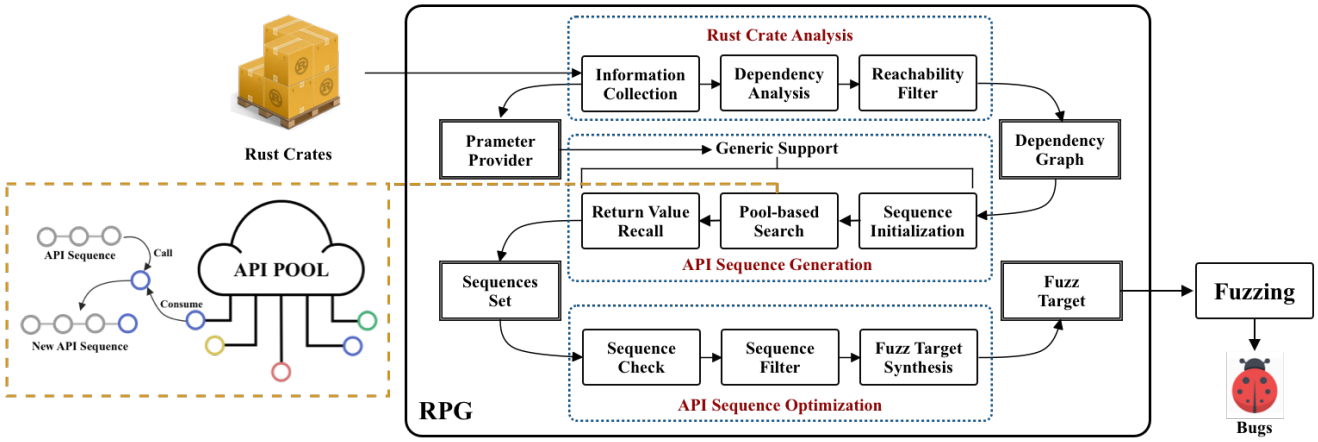


Figure 1: Workflow of Our Approach RPG

type of the generic parameter remains consistent. Our approach enhances polymorphic handling and generates more comprehensive tests.

3 APPROACH

3.1 Overall Framework

Our RPG adopts a novel pool-based fuzz target generation approach for Rust libraries. As illustrated in Figure 1, the main workflow of RPG consists of the following steps:

- *Rust Crate Analysis*: RPG utilizes static analysis to extract information on functions and data types from Rust libraries (*a.k.a.* crates), based on which an API dependency graph and a parameter provider are constructed for a given library. The graph records the information of functions (*i.e.*, APIs) that can be invoked in the library, which is then used to guide the API sequence generation; and the provider contains the data types defined in the library as well as a number of commonly used data structure types, which is then used to provide type candidates for generic functions.
- *API Sequence Generation*: this step is crucial in RPG, as an API sequence directly reflects the quality of its corresponding fuzz target. RPG starts with a sequence set, created with consideration of unsafe APIs, and generates API sequences based on the API dependency graph as well as an API pool, which consists of the currently available APIs, which may have multiple occurrences. In particular, when working with generic functions, RPG takes the data types from the provider and maintains the consistency of generic type parameters to ensure validity.
- *API Sequence Optimization*: To ensure the syntactic and semantic validity of the fuzz targets, RPG employs a move-borrow check and a generic declaration check to remove invalid API sequences. RPG also performs sequence filtering to obtain a minimal set of sequences that cover the most APIs and their dependencies. Finally, the remaining API sequences are synthesized individually, yielding fuzz targets.

3.2 Rust Crate Analysis

This section presents our analysis on Rust libraries, which is comprised of three components, namely, Information Collection, Dependency Analysis, and Reachability Filter.

Information Collection. RPG leverages rustdoc [11] to extract relevant information from Rust libraries. Specifically, we gather a comprehensive list of the public APIs present in the library, including their respective names, inputs and outputs. Moreover, we conduct static analysis to detect any unsafe code in each API. Additionally, to facilitate the invocation of generic APIs, we also collect data structure types defined in Rust libraries, since these types are more likely to be compatible with the generic functions in their corresponding library. Coupled with a number of commonly used data structure types, we construct a parameter provider, which is then used to generate type candidates for generic functions.

Definition 3.1 (Dependency). Given two functions f and g, if the return value of f can be used as the i-th non-basic parameter of g, then there is a dependency from f to g with respective to the i-th parameter.

Dependency Analysis. We characterize the interactions between functions as their dependency relationship, which is defined in Definition 3.1. However, we determine the dependency relationships via the input and output types of APIs conservatively. If the output type of an API f is compatible with one of the input types of another API g, then we label a dependency between f and g. Table 1 shows all the compatible cases (referred to as dependency kinds) between the two involved types of two APIs, which are grouped into three levels, namely, Basic Dependency, Composite Dependency and Wrapper Dependency. Basic Dependency signifies a normal dependency or a generic dependency without any reference or wrapping operations (*e.g.*, the return value of f may be directly used as an input for g), and it is essential to Composite Dependency and Wrapper Dependency. Note that, only non-basic types are considered for dependency analysis, since values of basic types (*e.g.*, *Int*, *Boolean*) can be automatically generated during fuzz testing.

Table 1: Dependency Kinds Grouped by Type Compatibility

Dependency Level	Dependency Kind	Pattern
Basic Dependency		
	Normal Dependency	$C \Rightarrow C$
	Generic Dependency	$C \Rightarrow T$ or $T \Rightarrow T$ or $T \Rightarrow C$
Composite Dependency		
	Immutable Ref	$A \Rightarrow \&A$
	Mutable Ref	$A \Rightarrow \&mutA$
	Constant Raw Pointer	$A \Rightarrow$ const $*$ A
	Mutable Raw Pointer	$A \Rightarrow$ mut $*$ A
	Deref	$\&A \Rightarrow A$
Wrapper Dependency		
	Unwrap Result	Result<A,E> $\Rightarrow A$
	Unwrap Option	Option<A> $\Rightarrow A$
	To Option	$A \Rightarrow$ Option<A>
	To Box	$A \Rightarrow$ Box<A>

C: (Non-Basic) Concrete Type; E: Error Type; T: Generic Type; A: Any Type

On the other hand, the other two levels denote a dependency that needs either a reference operation or a wrapping operation, and may be applied multiple times (*e.g.*, Option<A> $\Rightarrow A \Rightarrow \&mutA$). Specifically, for generic function calls, we consider a type variable is compatible with another type if their trait bounds are locally satisfiable, without checking the global consistency of generic type parameters, which will be checked during API sequence generation. Based on these dependencies, an API dependency graph is constructed.

Considering the motivating example in Listing 1, the dependency graph for `Queue` library is shown in Figure 2, where each node represents an API, including its non-basic input types, output types, name, trait bounds for generic types, and whether unsafe code is present (annotated in red). Meanwhile, each edge captures the type of dependency between two APIs, as well as the dependency kind. Any candidates from the parameter provider for generic types are also highlighted in red boxes.

Reachability Filter. If some parameters of an API could not be provided, then this API cannot be invoked. Consequently, such APIs are not generated during our API sequence generation. Thus, there is no need to traverse those APIs. Filtering them out can significantly enhance the efficiency of the generation process. To achieve this, we introduce the following definitions and employ the classic reachability analysis on graphs to filter the unreachable APIs out.

Definition 3.2 (Starting API). An API is said to be a starting API if all its inputs are of basic types.

Definition 3.3 (Reachable API). An API is said to be reachable if (*i*) it is a starting one, or (*ii*) every its required non-basic parameter has at least one reachable dependency.

3.3 API Sequence Generation

Our API sequence generation method involves two steps: Sequence Initialization and Pool-based Generation. Prior to presenting our generation, we introduce the following definitions.

Definition 3.4 (Calling Sequence). A calling sequence of an API represents the sequence of APIs that are required to provide all the non-basic parameters for the API to be called successfully. An API may have multiple calling sequences.

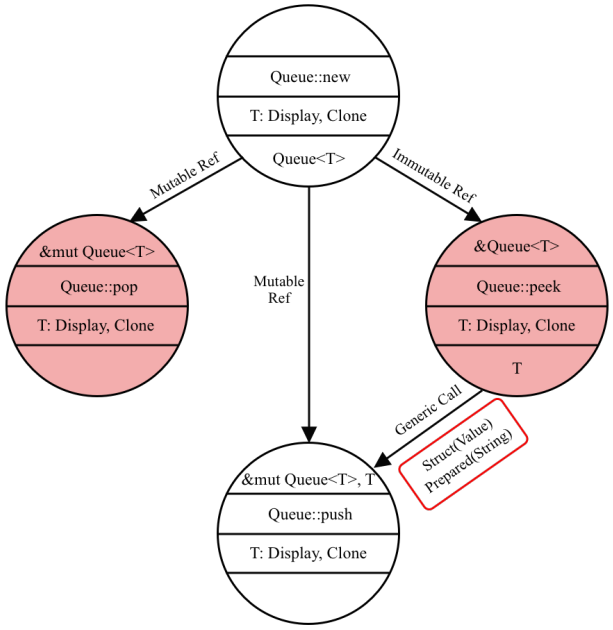


Figure 2: Dependency Graph for Queue with Generic Support

Definition 3.5 (API Depth). The depth of an API is defined as the minimum length of its calling sequences.

Definition 3.6 (API Pool). An API pool is a collection of APIs, with each API potentially having multiple occurrences. Whenever an API is successfully called, it is consumed from the API pool, that is, its quantity is decreased by one.

Sequence Initialization. Our pool-based sequence generation is optimized based on the breadth-first search (BFS). However, the breadth-first search always starts with the starting APIs, so some unsafe APIs may not be generated. To address this issue, RPG incorporates the original starting APIs with the expanded calling sequences of the unsafe APIs to form the starting set for the BFS search. This enhances both the breadth of the traversal search and the importance of the unsafe APIs. The generation of calling sequences for an (unsafe) API is primarily based on the backtracking approach on the dependency graph. The process involves continuously identifying APIs that are capable of satisfying the input parameters of the target API and adding them to the calling sequence until all non-basic parameters in the calling sequence can be satisfied. Once this is achieved, the calling sequence is considered to be generated successfully.

Pool-based Generation. When invoking an API in a sequence, each of its input parameters is assessed according to three criteria to determine if the API can be invoked. Firstly, if the parameter is of a basic data type, it is provided through a fuzzing mechanism. Secondly, the parameter is examined against the return values of the previously-called APIs to check whether a callable dependency exists between them and the target API. Finally, for generic type

Algorithm 1: Pool-based Sequence Generation

Input: Dependency graph G, API pool P_{API}, starting sequence set S_{Init}
Output: Sequences set S_{RPG}

```
1  S_RPG ← S_Init ;
2  S_Depth[0] ← S_Init ;
3  depth ← 0 ;
4  while true do
5      foreach s in S_Depth[depth] do
6          foreach f in G do
7              if f can be called by s and f ∈ P_API then
8                  new_s ← s.call(f) ;
9                  P_API.consume(f) ;
10                 S_Depth[depth + 1].push(new_s) ;
11     if P_API.isExplorationSufficient(depth) then
12         P_API.reset() ;
13     if S_Depth[depth + 1].isEmpty() then
14         break ;
15     S_RPG.append(S_Depth[depth + 1]) ;
16     depth ← depth + 1 ;
17 return S_RPG ;
```

parameters, a compatible data type is chosen from the parameter provider to satisfy the call. If any of the above cases meets the input parameter requirements, the parameter is deemed as satisfied. Once all API parameters are satisfied, the API can be called[3].

To maximize the number of APIs and their dependencies that can be covered, RPG imposes limits on the number of each API to be called. Intuitively, for APIs in a larger library (*i.e.* a larger API dependency graph), they can be called more frequently. Moreover, RPG prioritizes both the unsafe APIs and the API dependencies, so that an API that is unsafe or with more potential dependencies may be called more frequently. Based on these, we construct an API pool, wherein the number of an API indicates the maximum time it can be called. That is to say, the greater the API number, the more frequently the API will be called. The original number of an API is determined by the information of both the Rust library it belongs to and the API itself, as shown in Formula (1). The first factor, the *basic call volume* (BCV), is determined by the API dependency graph of the Rust library, which is calculated using Formula (2), where α and β are constants, N is the number of reachable APIs, and M is the number of API dependencies. The second factor, the *function call volume* (FCV), depends on a coefficient and the basic call volume, as shown in Formula (3). The coefficient I is calculated based on whether unsafe code or potential dependencies are present: (*i*) I is initialized to 0; (*ii*) I is increased by 1, if the API involves unsafe operations; (*iii*) I is increased by n, if n parameters of the API are of non-basic types; (*iv*) I is increased by 2, if the return type of the API is non-basic.

$$CV = BCV + FCV \tag{1}$$

[3]Beside the type checking, it should also pass the move-borrow checking, which will be presented in Sec 3.4.

$$BCV = 1 + \alpha \cdot \tanh\left(\frac{NM}{\beta}\right) \tag{2}$$

$$FCV = I \cdot BCV \tag{3}$$

Algorithm 1 illustrates the workflow of the pool-based generation. The algorithm takes the dependency graph, the API pool, and the starting sequence set as input, which are obtained beforehand. The primary portion of the pool-based generation is highlighted in lines 5-10, which examines whether there is an API that can be called by the current sequence and whether the API exists in the API pool, and if so, extends the sequence by consuming the corresponding API from the API pool. After the sequence generation at the current depth is complete, in order to explore the possibility of generating longer/deeper sequences, RPG runs an exploration sufficiency test. When the proportion of consumed APIs exceeds the exploration threshold of the current depth (lines 11-12), RPG resets the API pool as the original setting. The calculation of the exploration threshold Φ is done through Formula (4), where γ is a constant. If no new sequence is generated at the current depth, it indicates that the API pool has been depleted or there is no API that can be called, and thus the algorithm terminates (lines 13-14).

$$\Phi = log_{\gamma}(depth + 1) \tag{4}$$

Generic Support. To support calling generic functions, RPG performs a lightweight type inference to infer the concrete types for generic parameters locally. Specifically, RPG maintains a hash map of generic signatures and concrete types to ensure the consistency in an API sequence, and applies different processes for different general dependencies given in Table 1 to update the hash map:

(1) Both types in a generic dependency are generic (*i.e.*, $T \Rightarrow T$). RPG first examines whether both the generic types in the call have already matched any data types. (*i*) If both have been matched, RPG then checks whether these two matched types are compatible. (*ii*) If only one has a matched type, RPG checks whether the matched type meets the trait bounds of the other generic type. (*iii*) If neither generic type has a matched type, RPG selects a data type that meets the trait bounds of both generic types from the parameter provider. RPG prioritizes the data type that implements the most traits in such situations. If any of the above conditions are met, the call is considered successful.

(2) One type in a generic dependency is generic and the other is concrete (*i.e.*, $T \Rightarrow C$ or $C \Rightarrow T$). RPG first examines whether the generic type have already matched any data type. (*i*) If matched, and the matched type is compatible with the concrete type, the call is considered successful. (*ii*) If not matched, RPG then checks whether the concrete type meets the trait bounds of the generic type. If so, the call is also successful.

(3) If the above process fails, RPG will select a data type from the parameter provider that meets all the involved trait bounds and is compatible with all associated types.

Return Value Recall. To exploit memory-related vulnerabilities, RPG attempts to use the values returned by the APIs in a generated sequence, which is termed the return value recall. The return value recall can be considered as a simple case of the parameter check for a target API, where only some special non-generic functions are considered. For instance, if the return value satisfies the `Display` or `Debug` trait, an immutable reference can be used

to call it via the `println!` macro. However, recalling these return values is prohibited if they have been moved, or doing so violates the borrowing rules. The return value recall performs a reverse move-borrow check (see Sec 3.4) on the return value of each called API to determine whether it can be used before searching for a calling function. Moreover, to alleviate borrow rule violations caused by alias references, an alias analysis of the return values and inputs of APIs is conducted in their corresponding implementations.

Example. Consider the motivating example in Listing 1 and its dependency graph in Figure 2. We assume that the size of API pool is sufficient during the sequence generation process. The starting set comprises three distinct API sequences: `Queue::new`, `Queue::new`→`Queue::peek`, and `Queue::new`→`Queue::pop`. For the sake of simplicity, we pick sequence s_0: `Queue::new` as our starting sequence in this example.

At depth 1, when invoking `Queue::push`, s_0 satisfies only the input of the Queue type. As the generic type item cannot be provided by s_0, we select String as the input from the parameter provider. Since the String type is a preset data type, its value can be obtained from the input of the fuzzer. This results in the sequence s_1: `Queue::new`→`Queue::push`.

At depth 2, s_1 can use the return value of `Queue::new` to perform an immutable reference call of `Queue::peek`. The resulting sequence is s_2: `Queue::new`→`Queue::push`→`Queue::peek`.

At depth 3, an additional basic data type `usize` is required when invoking `Queue::pop` to set the number of items to pop. This results in the sequence s_3: `Queue::new`→`Queue::push`→`Queue::peek`→`Queue::pop`.

At this stage, the sequence generation process based on the pool is completed. Then we perform the return value recall on the sequence s_3, yielding the sequence: `Queue::new(RC)`→`Queue::push`→`Queue::peek(RC)`→`Queue::pop`, where `RC` denotes the return value recall. The return value recall is helpful, as the UAF vulnerability can only be triggered after using the return value of `Queue::peek`.

3.4 API Sequence Optimization

This section presents several API sequence optimization strategies to obtain valid fuzz targets efficiently.

Sequence Check. Due to the complexity of Rust's syntax, particularly its unique ownership system, the use of the return value of an API (as an input for another API) may cause syntactic errors. To maximize validity of the generated sequences, RPG proposes two essential checks to eliminate invalid sequences from the sequence set: a move-borrow check and a generic declaration check.

Algorithm 2 illustrates the procedure for the move-borrow check. The algorithm first analyzes the data flows between the reference inputs and outputs for each API in the sequence (line 4), which is used to update the life cycles for reference variables. Then it examines each dependent (*i.e.*, non-basic) input of each API in the sequence based on ownership and borrowing rules, and simulates the moving (lines 9-10) and borrowing cycles (lines 11-18) of each variable. Finally, the overlapping cycles between moving and borrowing or mutable and immutable references are checked (lines 19-20). If there are overlapping cycles that violate the rules, the sequence is considered a syntactic error and is therefore removed from the sequence set.

Algorithm 2: Move-Borrow Check

```
   Input: API sequence seq
   Output: True or False
1  move_map ← new HashMap();
2  mut_map ← new HashMap();
3  immut_map ← new HashMap();
4  ref_map ← dataRefFlowAnalysis(seq);
5  for func in seq do
6      for input in func.dep_inputs() do
7          from_idx ← getDependency(func, input, seq);
8          to_idx ← index(func, seq);
9          if isMoved(from_idx, to_idx, input) then
10             move_map.insert(from_idx, to_idx);
11         else if isMutRef(from_idx, to_idx, input) then
12             life_lst ← mut_map.getOrInsert(from_idx, []);
13             life_lst.insert((to_idx, to_idx));
14             update life w.r.t. from_idx and ref_map to to_idx;
15         else if isImmutRef(from_idx, to_idx, input) then
16             life_lst ← immut_map.getOrInsert(from_idx, []);
17             life_lst.insert((to_idx, to_idx));
18             update life w.r.t. from_idx and ref_map to to_idx;
19     if !check(move_map, mut_map, immut_map) then
20         return False;
21 return True;
```

The generic declaration check first examines whether every generic type parameter of a sequence already has a matched concrete type in hash map maintained for the sequence. If not, it will select a data type from the parameter provider that meets all the involved trait bounds. Then it inspects the hash map and generates the declaration for the relevant data types, thereby avoiding compilation errors.

Sequence Filter. After successfully passing the prior sequence checks, the resultant sequence set can be synthesized into fuzz targets. However, generating too many targets may reduce the efficiency of fuzz testing, due to the potential redundancy of APIs and their dependencies. Therefore, to mitigate this potential redundancy and optimize testing results, RPG employs greedy sequence filtering to obtain a minimum sequence set. The filtering process involves selecting the most desirable sequence continuously, considering the following criteria, ranked by importance:

(1) The maximum number of new (unsafe) APIs;
(2) The maximum number of new (unsafe) dependencies;
(3) The longer sequence.

Fuzz Target Synthesis. Similar to RULF [24], RPG utilizes the information of an API sequence to generate the code necessary to achieve a fuzz target. RPG starts with an empty *Cargo.toml* file, and subsequently links the target Rust library and the fuzzing library *AFL* using the extern crate. RPG then defines a *main* function to call the target APIs in the sequence one by one based on their

Listing 3: Fuzz Target Example For Queue

```
1  // _param0, _param1: provided by fuzzer.
2  fn fuzz_fn(_param0: &str ,_param1: usize) {
3    let mut _local0: Queue<String> = Queue::new();
4    Queue::push(&mut _local0, _param0.to_string());
5    let _local1 = Queue::peek(&(_local0));
6    Queue::pop(&(_local0) ,_param1);
7    println!("{:#?}", _local1);
8    println!("{:#?}", _local0);
9  }
```

signatures. If the return value of an API is required by other APIs, the 'let' construct is generated. Moreover, the 'mut' tag is added if the return value will be modified. If there are wrapper calls, auxiliary functions are utilized to unwrap the underlying objects. The 'unsafe' tag is required for calling unsafe APIs. Lastly, RPG considers the parameters of basic and prepared types as inputs for fuzz testing and generates various values for them.

Example. Listing 3 shows the final synthesized fuzz target of the sequence `Queue::new(RC)` → `Queue::push` → `Queue::peek(RC)` → `Queue::pop`. There are three dependencies between the four APIs, all of which are related to the variable _local0, which is returned by the first API `Queue::new`. The first dependency occurs in the second API `Queue::push`, which is a mutable reference. Therefore, there is a mutable lifetime (4, 4) (the line of `Queue::push`). There is also an immutable lifetime (5,5), as the second dependency occurs in `Queue::peek`. The third dependency, which occurs in `Queue::pop`, extends the immutable lifetime from (5,5) to (5,6). It is clear that the mutable lifetime and the immutable lifetime do not overlap. There are two RCs. The first RC corresponds to an immutable reference of _local1 (line 7). So there is an immutable lifetime (7,7) for _local1, which extends the immutable lifetime of _local0 to (5,7), since there is a data flow from _local0 to _local1 (line 5). Similarly, the second RC further extends the immutable lifetime of _local0 to (5,8). The move-borrow check also passes here. If the move-borrow check for a RC fails, then simply remove it. Finally, note that fuzzing the input of `Queue::pop` is useful for triggering the UAF vulnerability, as it can only be triggered by a value of 0. This corresponds to the POC presented in Listing 2.

4 EVALUATION

The experiments conducted in this study pursue the answers to the following research questions:

RQ1. What is the quality of the fuzz targets generated by RPG?
RQ2. What are the individual contributions of the various components in RPG toward generating fuzz targets?
RQ3. Can RPG outperform the state-of-the-art approaches in terms of bug-finding ability?

4.1 Evaluation Setup

4.1.1 Evaluation Dataset. The dataset used in the experiments comprises 50 Rust libraries, which are selected from Rust's official crate host *crates.io*, based on their popularity, size, functionality, and the presence of unsafe code or generic functions. Figure 3 shows the statistics of the dataset, revealing that the dataset covers a broad

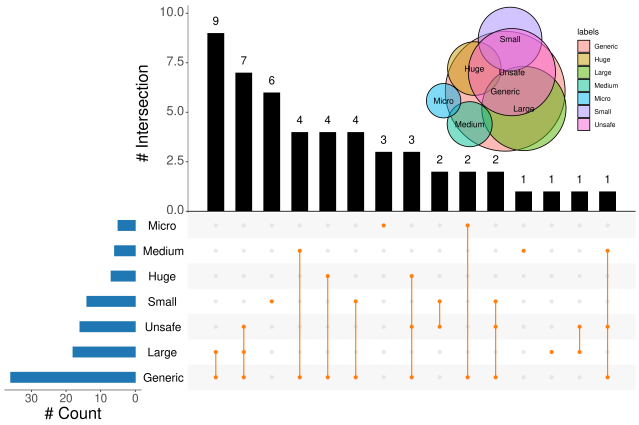


Figure 3: Statistics of Dataset

range of library sizes. The sizes of the libraries are measured by calculating the lines of code (LoC) present in the public functions of each library. The libraries are categorized into five groups based on their sizes, that is, Micro (0-1,000 LoC), Small (1,000-5,000 LoC), Medium (5,000-10,000 LoC), Large (10,000-50,000 LoC), and Huge (more than 50,000 LoC).

4.1.2 Baselines. We compared RPG against a state-of-the-art fuzz target generation technique, namely RULF [24]. For comparison, we selected various kinds of representative static and dynamic analysis techniques as competitors[4], such as Miri [35] and Rudra [4], as they are widely used in practice. We also include the results obtained by different versions of RPG without a particular component.

4.1.3 Configuration Parameters. After conducting numerous experiments and analyses, we have determined the values of α, β, and γ to be 3, 200, and 5, respectively. These values were selected to strike a trade-off between target generation time and API coverage. For Rudra, we use its default configuration. Miri runs under the built-in test cases of the project. For fuzz targets generated by RULF and RPG, the fuzz targets generated by RPG and RULF were further performed fuzzing using AFL++ [15]. We determined the maximum duration of fuzz testing by considering the number of targets. Specifically, we set a timeout of 2 hours for cases with less than 40 targets, whereas for instances with over 200 targets, the timeout was extended to 6 hours. For scenarios with the number of targets falling within the range of 40 to 200, the timeout duration followed a linear transition between 2 and 6 hours. Additionally, to mitigate the impact of randomness on the experimental outcomes, we conducted each set of experiments 3 times and used the average of the results as the final value.

[4]We encountered challenges while attempting to run SyRust [42] and RustyUnit [44] on our dataset. Unfortunately, we faced compatibility issues with the Rust version when installing RustyUnit. Furthermore, most of the programs generated by SyRust were rejected by Miri. Upon inspecting the rejected programs for 14 libraries affected by bugs, we found that some were rejected due to the uninitialized generic type T, while others failed due to unwrapped results. In addition, there were a few programs that passed the Miri test, but the reasons for their rejection during the generation process remain elusive. We noted that the API sequences generated by SyRust are relatively simple and may not be effective in detecting bugs. Further discussion can be found in Section 5.

4.1.4 Test Oracle. Panic is an error-handling mechanism in Rust that allows the program to abort and provide a log of what caused the panic. When a panic occurs during fuzzing, it indicates an unexpected behavior or a bug in the program. AddressSanitizer (ASan) is a tool used to detect memory-related bugs, such as buffer overflows, use-after-free errors, and uninitialized memory accesses. By combining Panic and AddressSanitizer as test oracles, we can effectively identify and diagnose potential issues.

4.1.5 Experiment Infrastructure. We conducted all experiments on a machine with an Intel Xeon Gold 6132 Processor (56 cores, 2.60GHz) and 256GB of RAM running 64-bit Ubuntu LTS 20.04.

4.2 Quality of Fuzz Targets (Q1)

The effectiveness and quality of the fuzz targets generated by RPG on 50 Rust libraries were evaluated by computing various statistics such as the API coverage rate, API dependency coverage rate, sequence generation time, and the number of targets for each library. In Table 2, a summary of these statistics is presented according to the five groups classified by library sizes. And Table 3 shows statistical results of various methods for sequence generation (at the moment, concentrating on RULF [24] and RPG is sufficient).

API coverage represents the ratio of APIs included in the sequence set to the total number of APIs offered in the library. RPG achieved API coverage rates ranging from 57% to 100% for the analyzed 50 Rust libraries, with an overall coverage rate of 72%. In contrast, the RULF achieves an API coverage rate of only 47.3%. This indicates that RPG is capable of significantly improving the quality of fuzz targets generated through pool-based generation and generic support. We conducted a manual inspection of the uncovered APIs. We found that those APIs were not covered because they require support for advanced Rust syntaxes, such as macros and closures. For instance, the `macro_rules!` feature enables the creation of declarative macros that ensure code portability across various platforms. Another instance is closures, which are function-like constructs that capture and manipulate the surrounding state.

Dependency coverage signifies the ratio of dependencies included in the sequence set to the total number of dependencies present in the API dependence graph of the library. The achieved coverage rate of dependencies for RPG is 11%, which is significantly higher than RULF's rate of 4.84%. This can be attributed to two reasons. Firstly, comprehensive coverage of dependencies is challenging, especially for generic ones. RPG's support of generic functions leads to a dramatic increase in the number of dependencies (the improvement of generic support will be further discussed in *Q2*). Secondly, as mentioned in Section 3.2, a generic dependency is labeled when the trait bounds of the generic types are locally satisfiable, meaning that either the trait bounds may not be satisfied globally or the concrete type could not be provided by the parameter provider.

The number of generated fuzz targets was higher in RPG than RULF (2121 VS. 923), as shown in Table 3. This achievement of RPG can be attributed to the pool-based generation and generics support (with additional 745 generic targets). Moreover, by prioritizing unsafe APIs, RPG generate a larger number of unsafe targets compared to RULF (317 VS. 127), resulting in an improvement of approximately 150%. Table 2 also presents the number of fuzz targets generated by RPG and the number of successfully compiled targets. The success rate of RPG in target compilation ranges from 87.3% to 100%, with an average of 94.7%. Noted that compilation failures during fuzzing mainly occur due to private function calls and incompatible features. For instance, the fuzz targets generated for the *serde* library could not complete the compilation process due to the presence of the `private` feature. These results demonstrate the validity of the fuzz targets generated by RPG.

Table 2: Quality of Fuzz Targets generated by RPG

Scale	API Coverage (%)	Dependency Coverage (%)	Generation Time (ms)	Generated Targets	Compiled Targets
Micro (5)	12 (100%)	0 (-)	206 (41)	7	7 (5)
Small (14)	176 (96%)	161 (31%)	2481 (177)	156	156 (116)
Medium (6)	125 (95%)	174 (40%)	1992 (332)	125	124 (118)
Large (18)	549 (57%)	955 (7%)	128363 (7131)	840	733 (639)
Huge (7)	630 (80%)	1269 (15%)	375114 (53588)	993	988 (704)
Total	1492 (72%)	2559 (11%)	508156 (10163)	2121	2008 (1582)

Table 3: Results for Different Sequence Generations

	RULF	RPG-1	RPG-2	RPG-3	RPG
API Coverage	47.3%	61.6%	64.2%	53.7%	**71.8%**
Dependency Coverage	4.84%	10.14%	10.42%	7.11%	**11.11%**
Generation Time (ms)	305634	1406828	410659	**109576**	508156
Unsafe API	54	81	77	63	**82**
Target Number	923	1822	2125	1523	2121
Unsafe Target	127	245	249	270	317
Generic Target	0	751	737	0	745

* RPG-1: RPG without pool-based generation, RPG-2: RPG without unsafe calling sequences, RPG-3: RPG without generic support

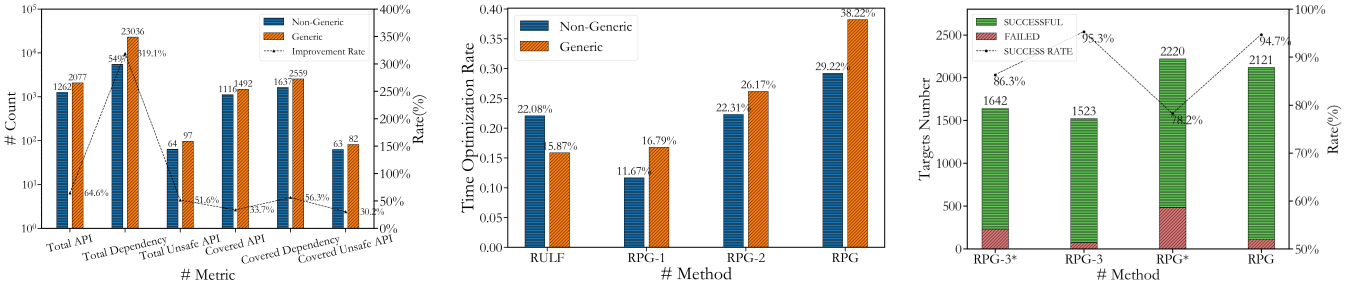
Generation time refers to the duration of the API sequence generation, which includes building the API dependency graph. On average, RPG took 10,163ms to generate API sequences for all 50 libraries, with a total time of around 508,156ms. The duration of sequence generation increases proportionately with the library size. When analyzing the 7 largest libraries, RPG required approximately 375,114ms, accounting for 74% of the total time for all 50 libraries.

Overall, we conclude that RPG produces valid API sequences with little redundancy, while maintaining satisfactory API and dependency coverage rates, as well as optimizing the validity and generating time of the fuzz targets.

4.3 Improvement of Various Components (Q2)

We conducted ablation experiments to study the effects of various components on the quality of fuzz targets.

Sequence Generation Components. Table 3 presents the results of the fuzz targets generated by different sequence generation methods. Specifically, RPG-1 represents RPG without pool-based generation, RPG-2 denotes RPG without unsafe calling sequences, and RPG-3 refers to RPG without generic support. Without the pool-based generation, the sequence generation time increases significantly, making it impossible to complete the generation task within the allotted 10-minute time limit (taking more than 20 minutes for RPG-1), particularly for large-scale libraries. Moreover, the pool-based generation is also able to explore a wider range of APIs and dependencies and generate more unsafe targets, due to the prioritization of unsafe APIs and dependencies. While eliminating



(a) Results for RPG with/without generic support

(b) Improvements of Reachability Analysis for different generation methods

(c) Compilation result for RPG with/ without(∗) sequence checking

Figure 4: Improvement of Various Components in RPG

Table 4: Results for Different Tools

Scale	RPG				RULF				Miri				Rudra	
	Success	Targets(Finished)	Time(hour)	Bugs	Success	Targets(Finished)	Time(hour)	Bugs	Success	Test(Finished)	Time(hour)	TP/FP	Success	TP/FP
Micro(5)	5	7(5)	2.02	0	2	4(2)	4.01	0	5	67(65)	0.03	0/2	5	0/2
Small(14)	14	156(116)	80.22	7	12	93(62)	71.22	3	13	1364(1240)	28.97	0/40	14	0/2
Medium(6)	6	124(118)	22.28	1	5	39(30)	3.78	0	6	408(258)	47.29	0/21	6	1/4
Large(18)	18	733(639)	390.61	26	17	305(262)	228.02	23	15	2099(1631)	23.9	1/20	15	0/2
Huge(7)	7	988(704)	545.12	24	6	440(261)	143.04	16	4	342(341)	48.39	1/5	7	1/5
Total	50	2008(1582)	1040.25	58	42	881(617)	450.07	42	43	4280(3535)	148.58	2/88	47	2/15

unsafe calling sequences improves the efficiency of sequence generation, it also limits the exploration of unsafe APIs. Compared with RPG-1, RPG-2 identifies four fewer unsafe APIs.

Generic Support. While RPG-3 has a shorter sequence generation time, its lack of generic support leads to reduced API and dependency coverage rates. Additional information on the effects of generic support is available in Figure 4(a). Without generic support, RPG identifies 1262 APIs and 5497 dependencies. On the other hand, providing generic support results in a significant increase in the number of identified APIs and dependencies, by 64.6% and 319.06%, respectively. Additionally, enabling generic support enables RPG to cover a wider range of APIs and dependencies, exhibiting improvements of 33.7% and 56.3%, respectively.

Reachability Analysis. Figure 4(b) demonstrates the efficiency improvements resulting from the use of reachability analysis for various API sequence generations. By filtering out unreachable APIs and dependencies, reachability analysis minimizes the time needed for the pool-based generation. Consequently, RPG's efficiency is significantly enhanced, with a noticeable improvement of 38.22%. These results clearly demonstrate the effectiveness of reachability analysis in improving the efficiency of API sequence generation.

Sequence Checking. Figure 4(c) illustrates the impact of sequence checking on the validity of fuzz targets. In RPG and RPG-3, the absence of sequence checking results in a significant decrease in the compilation success rate of fuzz targets, from 94.7% to 78.3% and from 95.3% to 86.3%, respectively. The decline is more significant in RPG than RPG-3. primarily due to generic declaration errors stemming from generic support.

4.4 Bug-finding ability (Q3)

We applied three representative tools, RULF, Miri, and Rudra to the 50 libraries to compare performance. RULF was incorporated with the sequence checker and fuzz target synthesizer of RPG to ensure that the fuzz targets generated by RULF align with RPG and are valid when using a newer version of Rust. Experimental results are available in Table 4.

Compared with the other tools, RPG exhibits superior library testing and bug detection capabilities. More specifically, RPG can successfully test all 50 libraries, while RULF, Miri and Rudra can respectively test only 42, 43 and 47 libraries. RULF failed mainly due to its inability to generate an API sequence. On the other hand, Miri's testing was subject to timeouts or disruptive interferences, and Rudra faced difficulties in completing an analysis due to library dependencies. Moreover, despite more testing time, RPG generates 1.28× more fuzz targets than RULF. This is due to its prioritization of unsafe APIs and APIs interactions, as well as its support of generic functions. While the test cases of Miri are provided by library developers, rather than automatic generation. In terms of bug detection, RPG outperforms the other tools by identifying a total of 58 bugs. RULF ranks second with a total of 42 bugs. Miri[5] and Rudra, on the other hand, reports 90 and 17 bugs respectively, however, only two of them can be classified as true bugs through manual inspection.

Table 5 presents the detailed information regarding the crashes detected by RULF and RPG in the 14 libraries affected by bugs, where T indicates the total number of targets that trigger crashes

[5]Miri functions as a platform-independent interpreter, so the program has no access to most platform-specific APIs or FFI. Consequently, Miri may report an unsupported error, which we classify as a false error.

Table 5: Crashes Detected by RPG and RULF

Crate Name	RPG							RULF						
	T	UT	GT	C	UC	GC	UA	T	UT	GT	C	UC	GC	UA
ryu	1	0	0	31	0	0	0	2	0	0	57	0	0	0
byteorder	1	1	0	1	1	0	4	8	4	0	8	4	0	4
autocfg	8	0	0	8	0	0	0	0	0	0	0	0	0	0
bumpalo	8	8	0	19	19	0	12	5	5	0	13	13	0	11
mio	3	0	0	6	0	0	0	3	0	0	6	0	0	0
unicode-segmentation	2	0	0	34	0	0	0	3	0	0	35	0	0	0
fixedbitset	32	8	0	52	18	0	10	15	8	0	23	14	0	8
idna	1	0	0	10	0	0	0	1	0	0	21	0	0	0
csv	25	1	3	117	5	15	1	3	0	0	11	0	0	0
bytes	58	31	10	158	94	29	49	23	23	0	70	70	0	33
syn	3	0	3	4	0	4	0	0	0	0	0	0	0	0
chrono	147	0	23	453	0	113	0	41	0	0	110	0	0	0
time	44	0	44	49	0	49	0	4	0	0	4	0	0	0
regex	14	0	0	3842	0	0	0	4	0	0	1143	0	0	0
Total	347	49	83	4784	137	210	76	112	40	0	1501	101	0	56

* T means **T**arget, U means **U**nsafe, G means **G**eneric, C means **C**rash, A means **A**PI calling.

Table 6: Bugs Found by RPG and RULF

Crate Name	Total Bugs	RPG						RULF						Confirmed Bugs
		A	OOM	OOR	U^1	U^2	OTHER	A	OOM	OOR	U^1	U^2	OTHER	
ryu	1	-	-	-	-	-	1	-	-	-	-	-	1	0
byteorder	6	-	-	6	-	-	-	-	-	2	-	-	-	0
autocfg	1	-	-	-	1	-	-	-	-	-	-	-	-	0
bumpalo	2	-	1	-	1	-	-	-	1	-	1	-	-	1
mio	1	-	1	-	-	-	-	-	1	-	-	-	-	0
unicode-segmentation	13	7	-	-	2	4	-	6	-	-	2	3	-	13
fixedbitset	2	-	2	-	-	-	-	-	1	-	-	-	-	0
idna	2	2	-	-	-	-	-	2	-	-	-	-	-	0
csv	2	-	2	-	-	-	-	-	1	-	-	-	-	0
bytes	5	1	2	-	-	-	1	1	2	1	-	-	1	0
syn	1	-	-	-	1	-	-	-	-	-	-	-	-	0
chrono	8	7	-	-	-	1	-	3	-	-	-	1	-	8
time	5	5	-	-	-	-	-	5	-	-	-	-	-	0
regex	10	2	-	8	-	-	-	1	-	6	-	-	-	3
Total	**59**	24	8	14	5	5	2	18	6	9	3	4	2	25

* A means **A**RITH errors, U^1 means **U**NWRAP errors, U^2 means **U**TF-8 errors.

during the fuzz testing, UT (GT, resp.) indicates the number of unsafe (generic, resp.) targets that trigger crashes, C indicates the total number of crashes triggered by the targets, UC (GC, resp.) indicates the number of crashes triggered by the unsafe (generic, resp.) targets, UA indicates the number of unsafe API callings among all the (unsafe) targets. The results demonstrate that RPG outperforms RULF by generating more targets that trigger more crashes. More specifically, RPG generates 22.50% more unsafe targets and explores 35.71% more unsafe API callings compared to RULF. Furthermore, with generic support, RPG generates 83 additional generic targets, resulting in 210 crashes.

Table 6 provides additional details on the bugs detected by RULF and RPG, where A denotes arithmetic errors (ARITH), OOM denotes out-of-memory errors, OOR denotes out-of-range access, U^1 denotes attempts to unwrap None or Err (UNWRAP), and U^2 denotes problems with UTF-8 string handling (UTF-8). The results demonstrate that RPG outperformed RULF in detecting errors for each library, with the exception of the bytes library. Notably, RPG identified all the errors that RULF detected, except for a single instance of OOM in the bytes library. That is to say, RPG detected 17

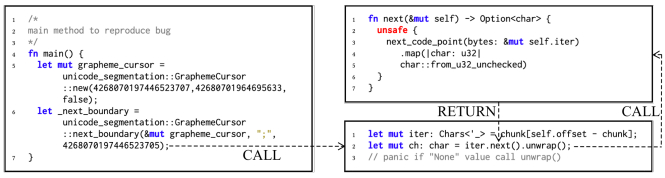


Figure 5: Code to reproduce bug in `unicode-segmentation`

unique bugs, while RULF detected only one unique bug. Moreover, 25 bugs were previously unknown and were confirmed by the library maintainers. RPG is capable of detecting all 25 bugs, while RULF could only detect 16. More detailed information on these bugs, as well as the corresponding fuzz targets and test inputs, are available on our website[6].

Case Study. To highlight the factors contributing to RPG's superiority, we present two case studies. These two specific errors were only be found by RPG in our experiment, thanks to its pool-based generation and fuzz target synthesis methods.

The first case comes from the `unicode-segmentation` crate, as depicted in Figure 5, and exhibits a bug that triggers an unexpected panic. The program enters a panicked state due to a subtraction overflow within the unsafe code block, leading to an erroneous index calculation that causes the `next_boundary` method in `GraphemeCursor` to return `None`. By prioritizing unsafe APIs, RPG generated an API sequence related to the unsafe code block and promptly identified the issue. However, RULF failed to detect this bug, possibly due to spending excessive time on other APIs, which made it difficult to discover this particular bug within a limited time period.

The second case study, derived from the `regex` crate, is illustrated in Figure 6. The bug in question arises in the function `pos` as a result of inadequate overflow verification during arithmetic operations. Consequently, this triggers a panic within the function `get` and yields an unexpected positional value. In scenarios where the index is invalid, including overflow scenarios, the expected outcome is `None`. The alteration made in the presented figure effectively resolves this issue. It is a significant challenge to trigger this bug through API calls. This process involves constructing precise input data and executing three API calls, with each call having complex dependencies and undergoing strict type checks. However, existing techniques like RULF are inadequate for handling such scenarios. In contrast, our approach, which prioritizes API dependencies and incorporates stringent type checking and validation, facilitates the generation of valid fuzz targets.

5 RELATED WORK

Existing works have demonstrated that Rust libraries and applications may contain security bugs [13, 16, 30, 39, 51, 54]. We discuss and differentiate three kinds of methods: static analysis, tests generation, and fuzzing—which are combined in our work.

Static Analysis. Existing static analysis techniques [20, 21, 36] usually perform bug detection on either Rust MIR or LLVM IR generated by the Rust compiler. Rudra [4] is a static analyzer that utilizes

[6]RPG's website: https://sites.google.com/view/rust-rpg.

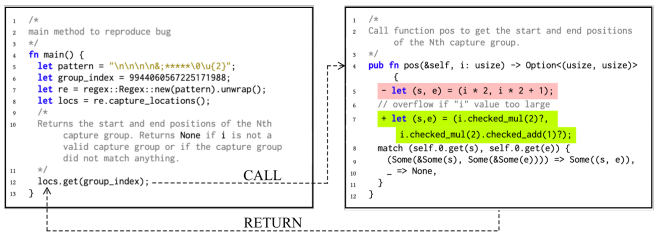


Figure 6: Code to reproduce bug in Regex crate

the type system and the borrow checker to identify typical undefined behaviors through three types of bug patterns. SafeDrop [8] focuses on the deallocation of dynamic memory during runtime and detects memory corruptions by performing alias analysis and taint analysis on the Rust MIR. MIRAI [7] is an abstract interpreter that performs symbolic execution for Rust. It enables users to add annotations and utilizes the SMT solver Z3 [9] to prove the safety of Rust programs. MIRChecker [31] detects potential runtime crashes and memory-safety errors by using constraint-solving on MIR. Different from these static analysis techniques that directly search for bugs, RPG employs static analysis to construct an API dependency graph and a parameter provider.

Fuzz Target Generation. Since RPG is a fuzz target generation technique designed to support Rust library fuzzing, we will detailed review related research on fuzz target generation for Rust libraries. RULF [24] is a technique that can automatically generate fuzz targets from an API specification of a Rust library. It utilizes breadth-first search with pruning to generate API sequences within a specified length threshold. Additionally, RULF synthesizes multiple fuzz targets to use for AFL++ fuzz testing with ease. However, it disregards the significance of unsafe APIs and ignores the API interactions. Furthermore, RULF lacks the ability to analyze and fuzz generic components effectively. SyRust [42] aims to generate compilable tests for Rust libraries using a semantic-aware synthesis algorithm. It generates the API sequences in an enumeration-like manner. Additionally, it employs a hybrid approach to independently instantiate polymorphic APIs from the API set and template provided by users in a combinatorial enumeration manner and removes the fully concrete APIs that caused trait errors after a few rounds. Although it enables SyRust to generate a large number of tests within a reasonable amount of time, it lacks fast bug detection. Furthermore, SyRust targets only a few APIs in a crate at a time and does not mutate the input value of each test, making it less directly applicable to fuzzing. RustyUnit [44] is a search-based tool for automatically generating unit tests for Rust programs. It utilizes the DynaMOSA algorithm [37] to optimize tests for code coverage and handles generic types randomly in most times without explicitly checking the trait bounds. However, both SyRust and RustyUnit target only a few APIs in a crate at a time and do not mutate the input values of each test, making them less directly applicable to fuzzing.

Moreover, there exists a line of fuzz target generation techniques in other programming languages, such as Fudge [3], JFD [6], and FuzzGen [23]. However, these techniques are language-specific and cannot be directly applied to Rust programming language, which has different syntax, semantics, and features.

Fuzzing. Fuzzing [17, 55] has been successfully used to detect vulnerabilities in Rust applications [1], Rust libraries [24, 50], and the Rust compiler [10]. RPG mainly focuses on synthesizing fuzz targets rather than on improving the fuzzing algorithm. In theory, the fuzz targets generated by RPG can easily be used as verification hardness for verification tools such as Kani Rust Verifier [45] and SMACK [5, 40], with some engineering efforts required. Our future work will explore further how to utilize RPG for verification tasks.

6 CONCLUSION

This paper presents a novel approach called RPG for automated fuzz target generation in Rust libraries. By proposing a pool-based generation, a local type inference and several validity checks, we successfully addressed the challenges of generating diverse valid fuzz targets with generic support. The experimental results validate the effectiveness and practicality of RPG in discovering various bugs of Rust libraries. RPG will be beneficial for developing more secure and reliable Rust libraries and ensuring the safety of Rust applications that rely on these libraries.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their constructive comments. This work was supported in part by the National Natural Science Foundation of China (Nos. 62372304, 61972260, 62302375, 62192734), the China Postdoctoral Science Foundation funded project (No. 2023M723736), and the CCF-Huawei Populus Grove Fund (No. CCF-HuaweiFM202205).

REFERENCES

[1] Mohammadreza Ashouri. 2020. RUSTY: A Fuzzing Tool for Rust. In *Annual Computer Security Applications Conference*.
[2] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. 2019. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.
[3] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 975–985.
[4] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 84–99.
[5] Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. 2018. Verifying Rust programs with SMACK. In *Automated Technology for Verification and Analysis: 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings 16*. Springer, 528–535.
[6] Zhuo Chen and Yongjun Wang. 2021. JFD: Automatic Java Fuzz Driver Generation. In *Proceedings of the 2021 5th International Conference on Electronic Information Technology and Computer Engineering*. 862–867.
[7] MIRAI Contributors. 2021. MIRAI: Rust mid-level IR Abstract Interpreter.
[8] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2022. SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *ACM Transactions on Software Engineering and Methodology* (2022).
[9] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 337–340.
[10] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust typechecker using CLP (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 482–493.
[11] Mihnea Dobrescu-Balaur and Lorina Negreanu. 2015. Enhancing RUSTDOC to allow search by types. *Studies in Informatics and Control* 24, 2 (2015), 221–228.

[12] Matthias Erdin, Vytautas Astrauskas, and Federico Poli. 2019. *Verification of rust generics, typestates, and traits.* Ph. D. Dissertation. Master's thesis, ETH Zürich.

[13] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust used safely by software developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 246–257.

[14] Jonáš Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. 2023. Leveraging Rust Types for Program Synthesis. In *Proceedings of the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1414–1437.

[15] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*. 10–10.

[16] Kelsey R Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L Mazurek. 2021. Benefits and drawbacks of adopting a secure programming language: rust as a case study. In *Symposium on Usable Privacy and Security*.

[17] Patrice Godefroid. 2020. Fuzzing: Hack, art, and science. *Commun. ACM* 63, 2 (2020), 70–76.

[18] Taotao Gu, Xiang Li, Shuaibing Lu, Jianwen Tian, Yuanping Nie, Xiaohui Kuang, Zhechao Lin, Chenyifan Liu, Jie Liang, and Yu Jiang. 2022. Group-based corpus scheduling for parallel fuzzing. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1521–1532.

[19] Shuang Hu, Baojian Hua, and Yang Wang. 2022. Comprehensiveness, Automation and Lifecycle: A New Perspective for Rust Security. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 982–991.

[20] Shuang Hu, Baojian Hua, Lei Xia, and Yang Wang. 2022. CRUST: Towards a Unified Cross-Language Program Analysis Framework for Rust. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 970–981.

[21] Baojian Hua, Wanrong Ouyang, Chengman Jiang, Qiliang Fan, and Zhizhong Pan. 2021. Rupair: towards automatic buffer overflow detection and rectification for Rust. In *Annual Computer Security Applications Conference*. 812–823.

[22] Zhijian Huang, Yong Jun Wang, and Jing Liu. 2018. Detecting unsafe raw pointer dereferencing behavior in rust. *IEICE TRANSACTIONS on Information and Systems* 101, 8 (2018), 2150–2153.

[23] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*. 2271–2287.

[24] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. 2021. RULF: Rust Library Fuzzing via API Dependency Graph Traversal. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 581–592. https://doi.org/10.1109/ASE51524.2021.9678813

[25] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.

[26] Ralf Jung, Jacques-Henri Jourdan, and Robbert Krebbers. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

[27] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *Commun. ACM* 64, 4 (2021), 144–152.

[28] Shuanglong Kan, Zhe Chen, David Sanan, Shang-Wei Lin, and Yang Liu. 2018. An Executable Operational Semantics for Rust with the Formalization of Ownership and Borrowing. *arXiv preprint arXiv:1804.07608* (2018).

[29] Matthew Kelly, Christoph Treude, and Alex Murray. 2019. A case study on automated fuzz target generation for large codebases. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–6.

[30] Hao Li, Filipe R Cogo, and Cor-Paul Bezemer. 2022. An Empirical Study of Yanked Releases in the Rust Package Registry. *IEEE Transactions on Software Engineering* 49, 1 (2022), 437–449.

[31] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2021. MirChecker: detecting bugs in Rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2183–2196.

[32] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2022. Detecting Cross-language Memory Management Issues in Rust. In *Computer Security–ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III*. Springer, 680–700.

[33] Peiming Liu, Gang Zhao, and Jeff Huang. 2020. Securing unsafe rust programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 234–245.

[34] 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*. 841–856.

[35] Scott Olson. 2016. *Miri: an interpreter for Rust's mid-level intermediate representation.* Technical Report. Technical report.

[36] Wanrong Ouyang and Baojian Hua. 2021. *R*: Towards Detecting and Understanding Code-Document Violations in Rust. In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 189–197.

[37] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.

[38] David J Pearce. 2021. A lightweight formalism for reference lifetimes and borrowing in Rust. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 1 (2021), 1–73.

[39] 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 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 763–779.

[40] Zvonimir Rakamarić and Michael Emmi. 2014. SMACK: Decoupling source language details from verifier implementations. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 106–113.

[41] Eric Reed. 2015. Patina: A formalization of the Rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02* (2015), 264.

[42] 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*. 899–913.

[43] Chi Thien Tran and Shamil Kurmangaleev. 2021. Futag: Automated fuzz target generator for testing software libraries. In *2021 Ivannikov Memorial Workshop (IVMEM)*. IEEE, 80–85.

[44] Vsevolod Tymofyeyev and Gordon Fraser. 2022. Search-Based Test Suite Generation for Rust. In *Search-Based Software Engineering: 14th International Symposium, SSBSE 2022, Singapore, November 17–18, 2022, Proceedings*. Springer, 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*. 321–330.

[46] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. 2018. Krust: A formal executable semantics of rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, 44–51.

[47] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 999–1010.

[48] Cheng Wen, Mengda He, Bohao Wu, Zhiwu Xu, and Shengchao Qin. 2022. Controlled concurrency testing via periodical scheduling. In *Proceedings of the 44th International Conference on Software Engineering*. 474–486.

[49] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 765–777.

[50] Hui Xu. 2022. Rust Library Fuzzing. *IEEE Software* 39, 5 (2022), 105–108.

[51] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R Lyu. 2021. Memory-safety challenge considered solved? An in-depth study with all Rust CVEs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–25.

[52] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. 2021. IntelliGen: Automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 318–327.

[53] Yizhou Zhang, Matthew C Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C Myers. 2015. Lightweight, flexible object-oriented generics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 436–445.

[54] Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. 2022. Learning and programming challenges of rust: A mixed-methods study. In *Proceedings of the 44th International Conference on Software Engineering*. 1269–1281.

[55] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36.