FRIES: Fuzzing Rust Library Interactions via Efficient Ecosystem-Guided Target Generation

Xizhe Yin
xizheyin@smail.nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing 210023, China

Yang Feng*
fengyang@nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing 210023, China

Qingkai Shi
qingkaishi@nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing 210023, China

Zixi Liu
zxliu@smail.nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing 210023, China

Hongwang Liu
hongwangliu@smail.nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing 210023, China

Baowen Xu
bwxu@nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing 210023, China

Abstract

Rust has been extensively used in software development in the past decades due to its memory safety mechanisms and gradually matured ecosystems. Enhancing the quality of Rust libraries is critical to Rust ecosystems as the libraries are often the core component of software systems. Nevertheless, we observe that existing approaches fall short in testing Rust API interactions — they either lack a Rust ownership-compliant API testing method, fail to handle the large search space of function dependencies, or are limited by pre-selected codebases, resulting in inefficiencies in finding errors.

To address these issues, we propose a fuzzing technique, namely FRIES, that efficiently synthesizes and tests complex API interactions to identify defects in Rust libraries, and therefore promises to significantly improve the quality of Rust libraries. Behind our approach, a key technique is to traverse a weighted API dependency graph, which encodes not only syntactic dependency between functions but also the common usage patterns mined from the Rust ecosystem that reflect the programmer's thinking. Combined with our efficient generation algorithm, such a graph structure significantly reduces the search space and lets us focus on finding hidden bugs in common application scenarios. Meanwhile, an ownership assurance algorithm is specially designed to ensure the validity of the generated Rust programs, notably improving the success rate of compiling fuzz targets. Experimental results demonstrate that this technique can indeed generate high-quality fuzz targets with minimal computational resources, while more efficiently discovering errors that have a greater impact on actual development, thereby mitigating the impact on the robustness of programs in the Rust ecosystem. So far, FRIES has identified 130 bugs, including 84 previously unknown bugs, in 20 well-known latest versions of Rust libraries, of which 54 have been confirmed.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Rust; Library Testing; Fuzz Target Generation

ACM Reference Format:
Xizhe Yin, Yang Feng, Qingkai Shi, Zixi Liu, Hongwang Liu, and Baowen Xu. 2024. FRIES: Fuzzing Rust Library Interactions via Efficient Ecosystem-Guided Target Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680348

1 Introduction

In recent years, the Rust programming language has gained popularity due to its potential in safety [13, 20, 26, 34, 36] and has been widely applied in various domains such as system programming [3], network and concurrent programming [5], and secure programming [2], leading to the creation of many well-known projects. Despite the safety features provided by Rust, programs written in Rust still contain risks of memory safety issues or logical errors, which may crash the software and affect the stability of Rust programs [12, 23, 27, 30]. Interestingly, most of the crashes in Rust programs originate from Rust libraries [35]. As reported, RustSec [7], the Rust Security Advisory Database, received an average of over 150 severe bugs in Rust libraries per year from 2020 to 2022. Therefore, ensuring the quality of Rust libraries and uncovering hidden errors within them is a critical task.

However, testing Rust libraries presents the following three challenges, which existing approaches fail to handle well.

*Yang Feng is the corresponding author.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680348

C1 *Narrowing down the search space with low overhead while maintaining flexibility*, from the *huge* number of potential API invocation sequences to *reflect errors in the real development environment*, thus *saving computational resources*.

C2 *Producing long, diverse, and complex Rust API interactions*, which, in comparison to simple and short ones, are more likely to trigger potential errors.

C3 *Conforming to Rust ownership constraints* even when generating complex programs, i.e., the targets. Violating the constraints will invalidate the generated targets.

First, classical fuzzing tools, e.g., AFL [6], specialize in generating and mutating inputs of primitive types like integers. However, to effectively test functions in a library, which often require composite types, e.g., structure, as their inputs, a new technique must be able to recognize other related functions to construct these inputs. To this end, researchers have proposed a few unit testing techniques, e.g., RustyUnit [33], SyRust [31], and many others not for Rust [9, 19, 29, 32]. These techniques can synthesize valid programs, but are unable to mutate inputs, making it difficult to trigger bugs. They often focus on testing a small number of functions in the same module and ignore API interactions across different modules throughout the library. Thus, they generate simple interactions for only a few functions. Moreover, the synthesis of effective targets often relies on a lot of manual configuration. Thus, the challenges, C1 and C2, are not well handled.

To test Rust API interactions effectively, some fuzzing techniques, e.g. RULF[23], create coarse-grained relationships among Rust API functions and blow up an API dependency graph. Despite the effectiveness in some application scenarios, due to the large search space of the dependency graph, it has to limit the length of each API invocation sequence (path obtained from the dependency graph) to a constant three. In other words, it's so hard for them to detect bugs that have to be triggered by involving more than three functions, which, however, are very common as demonstrated in our evaluation. Thus, C1 and C2 are not sufficiently handled, either.

In addition to techniques specially designed for Rust, some techniques, such as FUDGE [11], FuzzGen [21], APICraft [37], UTopia [22] and Winnie [24], can synthesize API interactions to test programs that are written in common languages like C/C++. However, on the one hand, many of them only test API interactions that occur in a set of pre-selected code repositories and, thus, may miss bugs that are out of the scope of the codebases. On the other hand, they do not take the ownership mechanism of Rust into consideration and, thus, will generate a number of invalid testing targets when directly used for Rust programs. To summarize, all the challenges, C1, C2, and C3, are not well handled.

Rust's open, community-driven development approach and advanced dependency management system form the backbone of its library ecosystem. Rust uses Cargo, the official package manager, to simplify dependency management and make dependencies between programs clear and accessible. Inspired by the above work and facts, in this paper, we propose FRIES to alleviate the three challenges, expecting to improve the testing effectiveness and efficiency of Rust libraries. Specifically, we build a *weighted* API dependency graph that encodes syntactic constraints and the API usage patterns in the Rust ecosystem to reduce the search space. Combined with the graph, our proposed efficient generation algorithm with type checking lets us focus on finding hidden bugs in common application scenarios, significantly improving the quality of fuzz targets. To avoid violating Rust's ownership constraint, we arm the target generation approach with an ownership assurance algorithm, which covers concurrent references and borrowings of data to ensure the validity of the generated more complex API invocation sequences. Finally, the generated API invocation sequences are converted into executable programs parameterized by primitive-type inputs. As such, we feed inputs generated by conventional fuzzers like AFL++ [16] for testing.

We implement FRIES as a fuzzing tool and detect a total of 84 previously unknown bugs on 20 open-sourced popular Rust libraries from different domains in total. To evaluate FRIES, we compare it with several state-of-the-art techniques for testing libraries, including two current testing techniques for Rust libraries. The experimental results demonstrate that FRIES can effectively generate long API invocation sequences and cover a large number of function dependency edges and API functions with very low overhead. Besides, FRIES can generate API invocation sequences containing many more functions in almost linear time, which can increase the richness of function calls and trigger deep API bugs. Moreover, our bug case analysis further confirms that the long API invocation sequences generated by FRIES containing usage patterns mined from the ecosystem can detect potential defects in API functions. In summary, the main contributions of our work are as follows.

- We propose an API invocation sequence generation algorithm with type checking that generates high-quality Rust interactions with low overhead based on a *weighted* API dependency graph.
- We use Rust's Middle-level Intermediate Representation i.e. MIR to analyze the usage patterns of API in the corpus to synthesize API invocation sequences that influence the Rust ecosystem, narrowing down the huge search space.
- We present a Rust ownership assurance technique that significantly improves the validity of more complex Rust interactions, thus enhancing the testing performance.
- We implement FRIES as a prototype tool and conduct experiments on 20 popular Rust libraries to show its capability of detecting bugs and merits over existing methods.

2 Motivating example

In this section, we illustrate the limitations of previous works with an example of an API sequence generated by FRIES, which triggers a real bug in a Rust library.

An Example of a FRIES-generated API Sequence. The function `test(...)` in Listing 1 shows a testing target generated by FRIES. By feeding inputs into the target by fuzzing tools like AFL++ [16], it triggers an unknown bug (confirmed by the developers) in the library, `xi-core-lib`, a core library of an editor written in Rust.

The target contains two key data structures, `SelRegion` and `Selection`. The former is a region we select in an editor and the latter is a set of selected regions. Line 6 initializes a `Selection`, `v2`, using the region created at Line 5. Line 7 and Line 8 create two regions, which are respectively added to `v2` at Line 9 and Line 10

via two different functions. The bug can be triggered only when we consecutively add two identical regions to the `Selection` and the second region needs to be added distinctly, just like Line 9 and Line 10 in the target with `pos1=pos2`, meaning `v3=v4`.

```
1. use xi_core_lib::selection::SelRegion;
2. use xi_core_lib::selection::Selection;
3. // an input to trigger the bug, test(6, 24, 68, 68);
4. fn test(start:usize, end:usize, pos1:usize, pos2:usize) {
5.     let v1: SelRegion = SelRegion::new(start, end);
6.     let mut v2: Selection = Selection::new_simple(v1);
7.     let v3: SelRegion = SelRegion::caret(pos1);
8.     let v4: SelRegion = SelRegion::caret(pos2);
9.     Selection::add_region(&mut v2, v3);
10.    Selection::add_range_distinct(&mut v2, v4);
11. }
```

Listing 1: A target generated by FRIES, which triggers a bug.

Limitations of Techniques Not Designed for Rust. Many techniques not designed for Rust often utilize existing codebases, such as FUDGE[11], APICraft [37], Winnie [24], etc. They usually extract code snippets directly from codebases for testing or combine them into new targets. Thus, they hardly generate targets with API interactions beyond the codebases, which limits testing effectiveness. Regarding the example, existing code usually employs a validity check before using `add_range_distinct(...)`, as shown in Line 2 of Listing 2, which narrows program state space that can be explored. Since existing works will directly use the real clip in Listing 2 as a fuzz target, it cannot generate a target that calls `add_region(...)` and `add_range_distinct(...)` with two same regions, i.e., $v3$ and $v4$. Thus, the generated target misses the bug.

```
1. Selection::add_region(&mut v2, v3);
2. if v3 != v4  { Selection::add_range_distinct(&mut v2, v4); }
```

Listing 2: Real-world code with restricted program space.

And, since this kind of method is not specially designed for Rust, it may generate a number of invalid testing targets when used to test Rust libraries, thus degrading the testing performance. For example, given the first two code snippets in Listing 3, both of which obey Rust's ownership constraint, existing works may crossover or join them to generate a new testing target.

```
   // snippet 1
1. let v3: SelRegion = SelRegion::caret(...);
2. Selection::add_region(..., v3);
   // snippet 2
3. let v3: SelRegion = SelRegion::caret(...);
4. Selection::add_range_distinct(..., v3);
   // The generated program by joining snippet 1 and 2
5. let v3: SelRegion = SelRegion::caret(...);
6. Selection::add_region(..., v3);
7. Selection::add_range_distinct(..., v3);
```

Listing 3: Existing methods may violate Rust ownership.

The generated code in Lines 5-7 of Listing 3 does not follow Rust's ownership constraint. This is because the ownership of $v3$ has been transferred to the function `add_region(...)` and its lifecycle ends after the function returns. Hence, the variable $v3$ becomes invalid when we call `add_range_distinct(...)`, and the target even cannot be compiled. So, C1, C2, and C3 are not handled well.

Limitations of Techniques for Rust API Testing. The current state-of-the-art technique for Rust fuzz testing encodes the relationships among API functions in a library at a coarse-grained level [23]. Thus, given that there are often many functions in a library, the search space of the graph is too large such that one cannot afford an exhaustive exploration of the graph to generate testing targets.

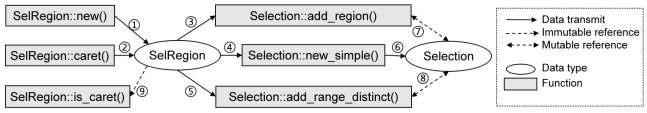

Figure 1: Dependencies between functions.

For example, the library `xi-core-lib` contains 212 public functions and the corresponding API dependency graph may contain 212^m paths to explore, where m is the length of API invocation sequences. To reduce the search space, they are armed with a key observation that "most bugs can be reproduced by calling library APIs within three times" [23], which, however, may let us miss a lot of bugs like the one shown in Listing 1. However, as explained before to trigger the bug in Listing 1, we need to call APIs five times. Thus, due to the relatively high graph traversal complexity, they cannot generate such a long and complex API sequence for testing and fail to find the bug.

Another work [31] synthesizes effective targets by encoding the semantic rules of Rust as constraints and solving the constraints. It encodes the function signatures and their potential inputs into SAT formulas and solves them. And it gets the models and translates them into testing targets. However, due to scalability, it can test only a few functions (up to 15) of the total library. For example, it can cover only 7% functions of library `xi-core-lib`, and a large number of inter-module interactions are left out. Additionally, the algorithm encodes labor-intensive constructed input templates into constraints, which disables input mutation and thus limits the exploration of state space. For example, the bug shown in 1 requires specific inputs to be triggered. Thus, current techniques designed for Rust API testing cannot handle C1 and C2.

FRIES addresses these limitations by mining programming patterns from the Rust ecosystem, constructing *weighted* API dependency graphs, and synthesizing API sequences using an efficient search algorithm with an ownership assurance algorithm, whose key insight is introduced in the next section.

3 FRIES in a Nutshell

Our approach addresses all three challenges via three main techniques as discussed below.

Mining API Patterns to Embrace Scalability. The first step of FRIES is to create a graph for all functions in the target Rust library. Traversing the graph allows us to find different paths, i.e., API invocation sequences, to generate the testing targets. Figure 1 shows a part of the graph containing the functions used in our motivating example, i.e., Listing 1. In the graph, a rectangle node represents a function and an ellipse node is a data type. A solid arrow from a function to a data type, e.g., ①, means the function returns a value of that type. A solid arrow from a data type to a function, e.g., ③, means the function accepts a value of the type as its input and transfers its ownership, and further usage isn't allowed. A one-way double-line arrow, e.g., ⑨, also input a value of a type to a function, but it means an immutable reference. This means that the function temporarily borrows the variable, and cannot change the value of the variable. And a two-way double-line arrow, e.g., ⑧, means a mutable reference.

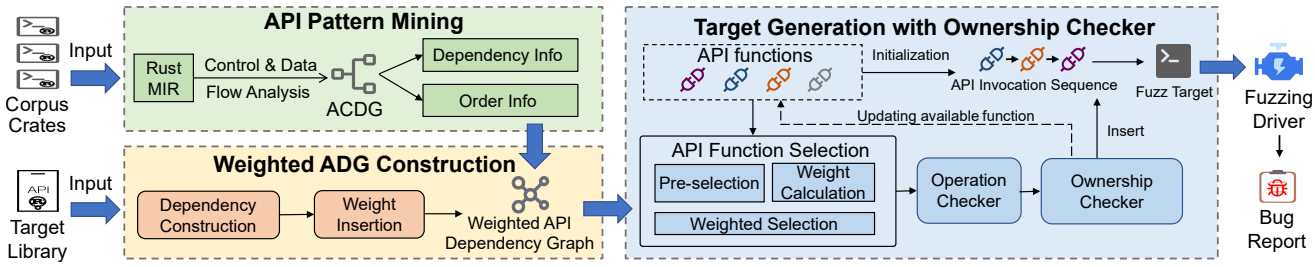

Figure 2: The workflow of FRIES.

Due to the large number of functions in a library, such a graph could be too large to be exhaustively explored for testing target generation. To reduce the search space, unlike RULF that limits the path length of graph traversal, we mine the API usage patterns, e.g. data or control flow information, from existing software written in Rust and, according to the pattern, assign weights to the edges, forming a weighted API dependency graph. Such weights allow us to focus on finding hidden bugs in common API interactions, thereby significantly reducing the search space, which handles C1.

Using Efficient Selection Algorithm to Ensure Diversity. Based on the weighted API dependency graph described above, we apply a semi-randomized sequence generation algorithm that generates API invocation sequences that have a higher probability of occurring in data and control flow of the real program mined, but not always.

For instance, the information we mined suggests that the dependencies, ①→⑨ and ②→⑨ are less frequently used. As such, we can focus on using other API dependencies to generate the testing target and, thus, have more chances to generate the target in Listing 1. More detailed design can be found in Section 4.

It is noteworthy that, unlike FUDGE or APICraft, we don't only use the pre-selected codebases to find more important API interactions to test, and do not lose the chance of generating testing targets beyond the pre-selected codebases. Combined with efficient sequence generation algorithms, we can synthesize diverse and complex sequences, thus handling C2.

Ownership Assurance for Target Generation. When generating API invocation sequences as per the paths of the weighted API dependency graph, we also advocate a list of ownership assurance rules to improve the validity of the generated targets. These rules include avoiding using moved variables, avoiding concurrent use cases of borrowings, etc., which will be detailed in the next section. For instance, since the edge ③ in Figure 1 stands for transferring the ownership of the variable, FRIES will not use the variable of the type `SelRegion` after passing it to the function `add_region(...)`. In comparison, since the edge ⑦ denotes a mutable reference, as shown in Listing 1, after passing the variable $v2$ into the API `add_region(...)`, we can still use it at Line 10 and pass it into the function `add_range_distinct(...)` as fuzz target. FRIES solved C3 by applying ownership assurance.

4 Approach

Figure 2 shows the overall workflow of FRIES. To capture the practical API usages, FRIES first collects corpus crates from the Rust ecosystem and then performs static analysis on Rust MIR to mine their usage patterns, which is described in Section 4.1. Next, in Section 4.2, we introduce how to analyze the target library and

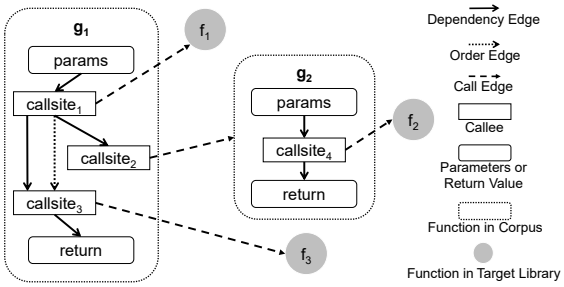

Figure 3: An example of API Control & Dependency Graph.

further combine the mined patterns to construct a weighted API dependency graph, which is the core of our approach to narrow the search space for C1. Then, we propose an efficient generation algorithm with type and ownership checking for Rust that generates high-quality API invocation sequences, which solves the challenges C2 and C3. We introduce the detailed implementation of FRIES generating sequences in Section 4.3, and our ownership assurance in Section 4.4. Finally, FRIES converts the sequences into executable programs and tests the target library with fuzzing tools.

4.1 Mining on MIR to Narrow Search Space

To generate fuzz targets with practical API usages, we first analyze the Rust MIR of corpus collected from the ecosystem to mine their usage patterns as guidance. Specifically, we gather Rust programs, potentially depending on the target library, to serve as corpus crates and construct an API Control & Dependency Graph to effectively mine practical API usages.

We define the API Control & Dependency Graph (ACDG) as an abstract representation of function order relationships and data dependencies in the corpus, which implies information about how the functions in the target library are used by programmers in real projects. Formally, ACDG is regarded as a directed graph $G = (Fn_{corpus}, Fn_{lib}, CE, DE, OE)$. Below is an explanation of the graph:

- $Fn_{corpus} = (Variables, Callsites)$ represents functions defined in the corpus crate, including variables in it, representing data flow. $Callsites$ represents call points that imply control flow.
- Fn_{lib} represents the functions in the target library. Our goal is to find the dependency pairs and order pairs in Fn_{lib}.
- $CE = \{(s_1, t_1), (s_2, t_2), \cdots\}$ is the set of Call Edges, where the source $s_i \in Fn_{corpus}$, and the target $t_i \in (Fn_{corpus} \cup Fn_{lib})$.
- $DE = \{(s_1, t_1), (s_2, t_2), \cdots\}$ represents the set of Dependency Edges, where the source s_i and target t_i represents a parameter or the return value of the functions in $Fn_{corpus} \cup Fn_{lib}$.

Table 1: Transformation patterns of recursive match

Operation	Transformation pattern	Statement
Direct	X→ X	I = O
Ref	X → &X X → &mut X	I = &O I = &mut O
Pointer	X → *const X X → *mut X	I = &O as *const I = &O as *mut
Deref	&X | &mut X | *const X | *mut X → X	I = *O
Wrap	X → Option<X> X → Result<X, E>	I = Some(O) I = Ok(O)
Unwrap	Option<X> → X Result<X, E> → X	I = if let Some(i) = O {i} else {exit} I = if let Ok(i) = O {i} else{exit}

- $OE = \{(s_1, t_1), (s_2, t_2), \cdots\}$ represents the set of Order Edges, where the source and target function $s_i, t_i \in Fn_{lib}$.

An example of ACDG is depicted in Figure 3, with g_1 and g_2 representing functions from the corpus, while the target library includes functions f_1, f_2, and f_3. There are 3 call sites in g_1 and 1 in g_2.The $callsite_3$ may be called after $callsite_1$ in the control flow. The data flows with dependency edges. Therefore, there are four call edges: CE ={ ($callsite_1$, f_1), ($callsite_2$, g_2), ($callsite_3$, f_3), ($callsite_4$, f_2) }. There are four dependency edges in g_1: DE = { ($params$, $callsite_1$), ($callsite_1$, $callsite_2$), ($callsite_1$, $callsite_3$), ($callsite_3$, $return$) }. And there is one order edge: OE = { ($callsite_1$, $callsite_3$) }.

We perform inter-procedural data flow analysis based on the MIR, which is the intermediate code representation from the Rust compilation process. To construct the ACDG, we traverse the assignment statements in MIR to find direct data dependencies. In the assignment statement, the variables on the left-hand side (LValue) depend on the variables on the right-hand side (RValue), which can be a binary operator (binaryOp), a reference (Ref), etc. It means the data flows from the right to the left. For the case where RValue is a function call, there may be extra data dependencies on the arguments in the function call , so we go inside the called function to explore the inter-procedural dependencies. Therefore, we iteratively propagate the parsed dependency relationships because the dependency relationship has transitivity, thus constructing ACDG.

Based on ACDG, we further mine the frequency of dependencies and call orders of functions of the target library. Regarding functions of the target library in Figure 3, the function f_1 is called before another function f_3 is called, without other functions called along the path from f_1 to f_3, so the function pair (f_1, f_3) represents an order relation in the control flow. Similarly, it can be parsed out that two function pairs (f_1, f_2), and (f_1, f_3) have a dependency relationship. By parsing a large number of corpus crates, the dependency and order pairs of the target library may be repeated, thus, we record their frequencies as the mined API usage patterns. If a specific dependency or order of a function pair is frequent, it implies that such usage is common in practical development, and such a pattern should be given higher weight to generate fuzz targets.

4.2 Weighted ADG Construction for Searching

To obtain the syntactic constraints between all functions of the target library, we transform the source code using rustc [8] into a simplified HIR [4], a high-level representation of Rust code, from which we extract function signatures. Based on the parameters and return value types of two functions, we determine whether they

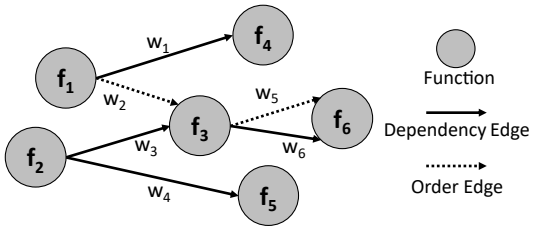

Figure 4: Weighted API Dependency Graph.

can construct syntactically compliant dependencies. Formally, we derive the abstraction dependency between functions in the target library, which is defined as follows:

Definition 4.1 (Function Abstraction Dependency). Given two functions f and g defined in a library, if the return value of f can be transformed as one of the parameters of g, then g depends on f, and (g, f) belongs to the API abstraction dependency.

Referring to the existing work [23], we summarize the patterns of data type transformations in Rust syntax, as shown in Table 1. For a given function's return value O and another function's parameter I, we recursively perform pattern matching to generate the corresponding transformation statement. If O can be converted to I through a series of conversion statements, then these two functions have an API abstraction dependency. For example, suppose we want to convert type T_1 into type T_2. If T_1 and T_2 are identical types, the operation is direct, allowing the direct passing of T_1 to T_2. When T_2 satisfies the pattern $\&X$, a call statement $T_2 = \&T_2'$ is generated, and then T_2' replaces T_2, recursively pattern matching T_1 and T_2' until encountering direct operation. If no pattern satisfies, then there is no dependency between them.

Based on the function abstraction dependency, we further employ the usage patterns mined, i.e. frequencies, as weights and construct weighted API dependency graphs. Specifically, the weighted API dependency graph (WADG) is denoted as $WADG = (N, E_D, E_O)$, where N represents the set of all public functions in the target library, and E_D and E_O are the frequencies of dependency and order pairs between the two functions, respectively, which we mine from the ecosystem in Section 4.1. In $WADG$, the greater weight of the order edge (f, g) implies that calls such as f followed by g occur more frequently in the control flow of real programs, and similarly, the dependency edge (f, g) implies that the return value of f tends to be readily used by g in real programs.

In Figure 4, we present an example illustrating the parsed public functions from the target library, resulting in a set of functions, denoted as $N = \{f_1, f_2, f_3, f_4, f_5, f_6\}$. The graph depicts four dependency edges denoted as E_D = { (f_1, f_4, w_1), (f_2, f_3, w_3), (f_2, f_5, w_4), (f_3, f_6, w_6) } and two ordering edges represented as E_O = { (f_1, f_3, w_2), (f_3, f_6, w_5) }. Among them, all the functions' dependency pairs are obtained through the target library analysis, while the order pairs are constructed through the corpus analysis, and all the weights on the edges are collected from the mined patterns. Some functions' dependency pairs may not exist in the corpus, then the weight of the corresponding edge is set to 0. The weighted API dependency graph can be applied as guidance for selecting functions when generating API invocation sequences, which is introduced in Section4.3.

4.3 Target Generation with Type Check

Algorithm 1 describes the process of generating API invocation sequences. It takes a weighted API dependency graph *WADG*, the maximum number of functions *maxlen* in each sequence, and the maximum number of sequences *maxsize* as inputs, which of both can be set manually. FRIES constructs a new API invocation sequence in each iteration and adds it to the output sequence set S_{out} (Line 2-18). For each API invocation sequence generated, FRIES first selects a function whose parameters are of the basic type as the start function (Line 3). Then, FRIES iteratively adds functions to the sequence until the sequence reaches *maxlen* (Lines 5-17). When selecting a function each time, we calculate the probability $p = 1/(len(seq) + C)$, where C is a constant, to determine whether to select a start function, ensuring a higher probability of selecting a start function at the beginning of *Seq* and prepare available variables for subsequent functions (Lines 6-8).

Algorithm 1: Selecting functions to generate sequences

Input: the graph WADG, Seuqence length threshold *maxlen*, function number threshold *maxsize*
Output: Final API invocation sequence S_{out}

1 $S_{out} \leftarrow \emptyset$;
2 **while** len(S_{out}) < *maxsize* **do**
3 $f_{start} \leftarrow$ PollingSelectStart();
4 $Seq, F_{avai} \leftarrow [f_{start}], \{f_{start}\}$;
5 **while** len(*Seq*)<*maxlen* **do**
6 **if** StartFuncOrNot(len(*Seq*)) **then**
7 *Seq*.add(SelectStart());
8 **continue**
9 $f \leftarrow$ GetAvailableFunc (F_{avai});
10 **for** *each node* f_i *in* $N_{adj}(f)$ **do**
11 $w(f_i) \leftarrow$ CalWeight(*WADG*, f,f_{last},f_i);
12 $\vec{w}_f(N_{\text{adj}}(f)) \leftarrow$ NormWeights($\vec{w}(N_{adj}(f))$);
13 $f_s \leftarrow$ WeightedSelect ($N_{\text{adj}}(f)$, $\vec{w}_f(N_{\text{adj}}(f))$);
14 **if** OperationCheck(*Seq*, f_s) *fails* **then**
15 Retry API selection once;
16 *Seq*.add(f_s);
17 $F_{avai} \leftarrow$ OwnershipAssurance(*Seq*);
18 S_{out}.add(*Seq*);
19 **return** S_{out}

To enable the potential use of the return values by other functions, we maintain a set of functions F_{avai} in *Seq* that have a return value and then choose a random function f from it (Line 9). To use the return value of f, we obtain a set of functions $F_{\text{adj}}(f)$ in the WADG graph that depend on f. Then, we calculate the dependency weight of each function in $F_{\text{adj}}(f)$ with f and select a function f_s based on weights (Lines 10-13). The construction and computation of the weight matrix are described in detail in *Weight Normalization*. After selecting f_s, we apply a composite type checking to determine whether to add f_s to *Seq* (Lines 14-15), which is illustrated in Section *Composite Type Check* in detail. Finally, FRIES inserts the selected f_s into *Seq*, and to ensure that the generated sequence can be compiled, we perform ownership assurance and update the functions available F_{avai} in *Seq* (Lines 17), which is introduced in Section 4.4. After reaching the *maxlen* threshold, FRIES finally returns the output set of API invocation sequences S_{out} (Line 19).

Weight Normalization. When selecting a function to add to the sequence, we first randomly select an available function f_x with a return value from the generated sequence *Seq* and obtain the set of functions $F_{\text{adj}}(f_x)$ in the WADG that depend on f_x. Each function $f_i \in F_{\text{adj}}(f_x)$ has a dependency with f_x, thus which function is chosen depends on the dependence frequency between f_x and f_i, and the order frequency of the last function f_{last} in *Seq* with f_i. Formally, the weight of $F_{\text{adj}}(f_x)$ is expressed as follows:

$$w_i = w(f_x, f_i) = E_D(f_x, f_i) + E_O(f_{last}, f_i) \tag{1a}$$

$$\boldsymbol{W} = [w_1, w_2, \ldots, w_n] \tag{1b}$$

To mitigate potential data imbalances caused by varying occurrence frequencies of different paired data in the corpus, we calculate the adjusted weight vector $\boldsymbol{W'}$ by applying a natural logarithm adjustment to each w. This is computed as $w'_i = \ln(w_i + e)$. Furthermore, we scale the weights, as presented in the equation 2.

$$w''_i = C + \frac{w'_i - \min(\boldsymbol{W'})}{1.0 + \max(\boldsymbol{W'}) - \min(\boldsymbol{W'})} \tag{2}$$

where $C \in (0, 1)$ is a constant, and $\max(\boldsymbol{W'})$ and $\min(\boldsymbol{W'})$ are the maximum and minimum values of $\boldsymbol{W'}$, respectively. The probability of each function being selected is the proportion of their weight to the total weight of the vector. Nodes with higher weights are more likely to be selected.

Composite Type Check. In Section 2, we explained that functions distinguish between mutable and immutable parameters. The value of an API-specific data structure may be modified after the execution of a function mutably borrows it. These mutable functions could provide more possible values for other functions by altering the value of internal data structures. To expand more program space, we set up a type-checking rule where mutable functions have a higher probability of being selected when they appear closer to the beginning of the sequence. Therefore, the probability of f_s being selected and added to the *Seq* is calculated as follows:

$$P(f_s) = \begin{cases} 1 - \frac{1}{3}\frac{len}{maxlen} & , f_s \text{ is a mutable function} \\ \frac{2}{3} + \frac{1}{3}\frac{len}{maxlen} & , \text{Others} \end{cases}$$

We control the probability of mutable functions passing the check, which monotonically decreases with the sequence length. Practice has shown that the probability is appropriate and beneficial in finding bugs without sacrificing performance or flexibility.

Target Generation. FRIES uses the analyzed function signatures to transform the generated API invocation sequence into fuzz targets that can be compiled. When a parameter of a function in the sequence is a primitive type, we generate conversion functions to convert byte arrays into different primitive types as function parameters. When the parameter depends on the return value of another function, we assign the return value to a variable and use the transformation statements in Table 1 to actualize the process of converting the return value into the parameter. At the same time, we include the target library as a dependency and write it into the "dependencies" section of the manifest, so that functions can be called. After generating the fuzzing targets, we compile them and use fuzzers like AFL++ to perform fuzz testing.

4.4 Rust Ownership Assurance

To enforce memory and concurrency safety features in Rust programming language, we design a series of ownership assurance rules to guarantee the validity of the generated sequences. Since ownership-related errors may be triggered in complex targets, FRIES conducts thorough ownership assurance to eliminate compilation errors regarding ownership. When function f_s is selected, FRIES checks if adding this function may violate any ownership rules of Rust. If the return value of a function is used by f_s and may violate ownership rules, it is removed from F_{avai}, thus avoiding triggering potential compiler errors related to ownership. Specifically, we design four ownership assurance rules as follows.

(1) Avoid using moved variables. If function `f` moves the return value of function `new` in the sequence, then FRIES removes function `new` from F_{avai} to avoid using variables returned by the function `new` in subsequent functions.

(2) Avoid concurrent use cases of borrowing. For the current *Seq*, suppose a function `new` returns an API-specific data structure as an intermediate variable *var*, the function `f` $\in$ *Seq* has already mutably borrowed *var* and returns its reference.

(2.1) Avoid concurrent mutable borrows of the same variable. If the selected function f_s requires the same mutable reference as *var*, then FRIES removes f from F_{avai}. This is because another function may take the return value of f as input, which could cause concurrent mutable borrows of the same variable.

(2.2) Avoid concurrent mutable and immutable borrows of the same variable. If the selected function f_s requires the same reference as *var* and borrows it immutably, then FRIES removes f from F_{avai}. This is because having a mutable and another immutable borrow of the same variable concurrently is not allowed. Similarly, if function f immutably borrows *var* and f_s needs to mutably borrow it, f should also be removed from F_{avai}.

(2.3) Avoid moving borrowed variables. Regardless of whether *var* is borrowed mutably or immutably by f, if f_s requires to take the ownership of *var*, then FRIES remove f from F_{avai}. In the case of synthesizing linearly aligned sequences, this check is comprehensive and does not trigger the corresponding compilation error.

```
1. struct S;
2. fn new() -> S {...}
3. fn f(s: &mut S) -> &mut S {...}
4. fn g(s: &mut S) {...}
5. fn t(s: &mut S) {...}
6. fn test_multi_mut() {
7.     let mut s:S = new();           //create s
8.     let mut_ref:&mut S = f(&mut s);//create mutable borrow of s
9.     g(&mut s);                     // double mutable borrow
10.    t(mut_ref);                    // Compilation Error
11. }
```

Listing 4: An example of triggering ownership errors.

The code snippet in Listing 4 shows an example that triggers an ownership error. The function `new` creates a struct *S*, and functions `f(...)`, `g(...)`, and `t(...)` take a mutable reference of `S` as a parameter, while `f` returns that mutable reference. After calling `f`, there exists a mutable borrow of *s*. When calling `g` with `"&mut s"` as a parameter, there are two mutable borrows simultaneously. If function `t` is called with the variable *mut_ref*, it triggers a compilation error. To avoid generating such invalid sequences, FRIES records that *mut_ref* points to the memory location where *s* lies, and when calling function `g`, it detects the existence of two mutable references, and thus removes function `f` from F_{avai}, ensuring that the variable *mut_ref* is not used in later functions.

5 Evaluation

Implementation. FRIES is implemented as a Rust program and relies on the rust compiler rustc [8] for static analysis. In the corpus crate analysis, to ensure sufficient information is obtained, we analyzed up to 200 corpus crates for each target library. To obtain corpus crates, leveraging Rust's open-source ecosystem, we retrieve potential reverse dependencies of the target library from the *Dependents* section on *crates.io* [1], or from GitHub.

In the target generation phase, we set the number of sequences generated by FRIES for each library to 50, and practical evaluation has shown that a high API coverage can be achieved with 50 sequences. To ensure that the generated sequences have complex interactions, we set the maximum threshold for sequence length to 15. In the fuzzing phase, for each library, we employ AFL++ [16] as the fuzzing tool and fuzz for up to 4 hours to explore FRIES' bug finding capability. AFL++ continuously feeds byte arrays into the fuzz targets, and our generated function templates convert them into primitive types.

Testing Subjects. Our experimental dataset consists of 20 well-known high-quality open-source Rust libraries from *crates.io* [1]. The vast majority of the code in each library is written by Rust, which means that the functions in it have ownership restrictions and allow us to analyze MIR and HIR. The download counts of these libraries range from tens of thousands to billions, covering various domains. The functionality of these Rust libraries includes networking, data processing, text processing, graphics and user interfaces, scientific computing, application development, and more. We classify these Rust libraries into three categories based on the number of public functions: Small (0-80 functions), Medium (81-200 functions), and Large (greater than 200 functions).

Baseline Approaches. To evaluate the effectiveness of FRIES, we compare it with existing fuzz target generation approaches and random strategy. RusyUnit and SyRust are designed for unit testing. SyRust also does program synthesis for Rust libraries, so we chose SyRust, as one of the baselines. We don't compare FRIES with APICraft, UTopia, and Winnie because they are not designed for Rust. However, from the approaches not designed for Rust, we have chosen the more representative FUDGE, which also uses ecosystems to synthesize programs. We implemented a Rust version of the FUDGE tool as a baseline. We also conducted an adequate comparison with RULF, the state-of-the-art fuzz target generation technique for Rust.

(1) SyRust. SyRust employs a semantic-aware approach to synthesize test cases. Due to scalability issues, SyRust can only test 15 APIs per library [31]. We crafted configuration files following the provided guidelines. We select the APIs in the most prominent submodule based on the recommended method, which also tends to be the most frequently used in the ecosystem.

(2) FUDGE. FUDGE extracts code snippets from the Google Codebase for generating fuzzing targets for C/C++. We reproduced Google's algorithm and extracted function sequences from the dependents of Rust libraries to generate sequences. We applied FRIES'

(a) The increasing trend of API coverage rate (b) Compilation Results for different tools (c) The increasing trend of generation time.

Figure 5: Effectiveness and Efficiency of FRIES, compared with other tools.

Table 2: Experiment results for the coverage and validity of generated fuzz targets.

Scale	Num of Covered Dependencies				API Coverage					Valid Ratio of Targets			
	FUDGE	RULF	Random	FRIES	FUDGE	SyRust	RULF	Random	FRIES	SyRust	RULF	Random	FRIES
Small (6)	39	108	108	**181**	0.35	0.38	**0.77**	0.75	**0.77**	**1.0**	0.99	0.74	**1.0**
Medium (6)	125	341	228	**821**	0.22	0.11	0.54	0.37	**0.58**	**0.99**	0.95	0.56	0.98
Large (8)	99	1,456	536	**2,349**	0.08	0.04	**0.52**	0.29	**0.52**	**0.99**	0.96	0.45	**0.99**
Average/Total	258	1,905	872	**3,351**	0.20	0.08	0.60	0.45	**0.62**	**0.99**	0.96	0.58	**0.99**

ownership checking algorithm to the Rust version of it to ensure that its compilability ratio is consistent with that of FRIES.

(3) RULF. RULF is a fuzz target generator for Rust libraries that uses BFS traversal and backtracking algorithms on the API dependency graph. According to the paper and their open-sourced code, the length of each sequence is set to 3 as default to ensure it performs at its best, and the number of sequences generated by each library is automatically terminated by RULF based on API coverage.

(4) Random. We also employ a random generation strategy without guidance and ownership checking as a baseline. Specifically, it randomly selects functions from public functions and then checks whether they can be added to the sequence. Other parameters are consistent with FRIES, for example, the length of each sequence is set to 15, and each library generates 50 invocation sequences.

Environment. Our experiment is conducted in a computer equipped with an Intel(R) Core(TM) i7-12700K processor and 32GB RAM, running Ubuntu 20.04 LTS.

5.1 Coverage and Validity

Table 2 presents a summary of evaluation results on the number of covered dependencies and API, and the validity of fuzz targets.

The number of covered dependencies indicates the complexity and flexibility of generated sequences, thus implying the possibility of triggering more bugs, which will prove to be more efficient in finding bugs in Section 5.3. As shown in Table 2, the covered dependency edges of FRIES are much more than other baselines. This is mainly because FRIES applies an ecosystem-guided semi-randomized algorithm for function selection, allowing for a higher possibility of selecting different function dependencies due to dynamically calculated weights, which facilitates improved coverage of dependency edges. On average, each library is parsed to obtain 1326 dependency pairs and 874 order pairs, and after de-duplication and counting, 109 and 249 different pairs are obtained, in which most functions are included, providing rich weighting information for constructing the WADG. We don't tabulate the dependencies covered by SyRust because it can synthesize so few functions (up to 15) that it only covers a few dependencies.

API coverage is the proportion of functions in the fuzz targets to all functions in the target library. FRIES achieved an API average coverage of 62% across the 20 libraries, surpassing other tools, including RULF which uses API-coverage-oriented algorithms. In practice, FRIES has covered almost all supported functions, while a few functions are not covered mainly due to advanced features.

FRIES focuses on covering more APIs with very few fuzz targets, which means consuming less computational resources during fuzzing. As shown in Figure 5(a), by generating only a small number of fuzz targets, FRIES achieves maximum coverage, whereas RULF needs to generate many more sequences to achieve similar coverage. However, due to the limitation of sequence length caused by the excessive spatiotemporal complexity of RULF, it covers functions much less efficiently than FRIES, thus more sequences are required for RULF to achieve high coverage. The random strategy performs worse, because it is not guided and checked, and selects many duplicate functions or generates invalid sequences. FUDGE has the lowest API coverage due to the coarse parsing granularity of the sequence parsing module. However, FRIES's analysis of data flow and control flow can extract all possible information at a finer granularity, making it more effective and flexible.

Table 3: Line coverage by FRIES, RULF, SyRust

Scale	FRIES		RULF		SyRust	
	Total	Per-target	Total	Per-target	Total	Per-target
Small (5)	**60.2**	**36.6**	58.3	19.1	10.4	5.8
Medium (5)	**41.6**	**16.7**	36.7	8.06	10.9	6.2
Large (6)	**28.9**	**9.6**	28.6	6.5	8.8	4.1
Average	**41.9**	**19.0**	40.1	10.9	9.9	5.1

Table 3 shows the average code coverage of each fuzz target and the overall coverage after running for one hour, due to the fact that the coverage grows minimally after one hour. We counted 16 libraries because certain ones could not be tested due to reasons such as incompatibility with instrumentation tools. `Total` is the total coverage of the library, while `Per-target` is the coverage

of the library by a single target. In terms of code coverage per target, FRIES is almost several times higher than RULF and SyRust, indicating that more program space can be explored per target. For overall library coverage, FRIES performs slightly better than RULF. This is because FRIES's overall API coverage is close to that of RULF, but due to its ability to generate complex interactions that produce sequences that RULF cannot, more search space may be explored. FRIES, on the other hand, performs significantly better than SyRust, both in each test case and overall, which may be because SyRust is unable to mutate inputs, and fails to explore more state space. Since FRIES can efficiently generate complex interaction, a single fuzz target FRIES generates can cover more functions, thus achieving higher code coverage, and thus detecting bugs more efficiently with fewer computational resources. For small libraries, compared to the other two tools, `Per-target` in FRIES is a large percentage of `Total`, probably because each target can contain more functions. We found that calling certain important functions may cover a large portion of the library, e.g., `url::Url::parse`, so the coverage may not grow linearly as more functions are included in each target.

As for the compilability, the fuzz targets generated by FRIES achieved a compilability of 99%, which benefits from our proposed ownership assurance. Through manual inspection, we found that compilation failures are often caused by function absence due to version incompatibility. Figure 5(b) demonstrates the compilability of fuzz targets generated by FRIES with and without ownership assurances. By comparing FRIES and FRIES-without-check, we can conclude that the ownership assurance rules are effective, increasing the compilability ratio from 58.0% to 98.6%. As library size increases, the percentage of sequences with complicated ownership mechanisms that can be compiled decreases, but the effectiveness of fuzz targets generated by FRIES remains constant. In practice, with a sequence length of 15, FRIES can improve RULF's synthesis effectiveness from less than 87% to 99%, which is because more complex sequences trigger more compiler errors about references. Because RULF generates sequences with a length of less than 3, in contrast, the sequences generated by FRIES have a length of 15, better reflecting the design of checks we designed for Rust. Similarly, our validity can be equated with SyRust, which is focused on synthesizing compilable programs.

5.2 Efficiency

Table 4 shows the time cost of analyzing corpus and generating sequences using FRIES and other baseline approaches. During corpus analysis, the average processing time cost is 4.59 seconds for each crate. Since corpus analysis is a one-time process and the analysis results can be used multiple times for generation algorithms, the time consumed in corpus analysis is acceptable.

Table 4: Experimental results for efficiency.

Scale	Analysis Time(s)		Generation Time(ms)			
	Total	Per-crate	FUDGE	RULF-50	Random	FRIES
Small (6)	5577	4.69	148	163155	289	**90**
Medium (6)	5398	4.50	1483	277338	1038	**126**
Large (8)	3883	4.55	**2914**	2335807	8796	3462
Average/Total	14859	4.59	4545	2776300	10123	**3678**

The termination condition of RULF to generate sequences is to reach the maximum API coverage, while FRIES and other baselines are fixed to generate 50 sequences in total. To ensure fairness, we modify RULF to the same termination condition. Nevertheless, the generation time of RULF is the longest due to the fact that it needs to iterate through all the public functions and check if they can be added to the sequence, which leads to a large time overhead. Similarly, the random strategy, which does not employ WADG as guidance to narrow the search space, also results in multiple function selection to satisfy the dependencies, leading to worse efficiency. FUDGE generates sequences directly based on the extracted corpus crates without additional function selection or checking, and is therefore more efficient than RULF and random. Among all baselines, FRIES has the most efficient generation efficiency and can guarantee sequence length and comparability.

To verify the relationship between sequence length and time complexity, we compare the time-consuming generation of sequences of different lengths by FRIES and RULF, which is shown in 5(c). For any size of libraries, FRIES takes less than 1.2 seconds to generate sequences with lengths from 1 to 15, which grows almost linearly and can be neglected for practical applications. In comparison, RULF has much higher time and space consumption. Due to the path explosion problem of RULF, the generation time exceeds one hour even with small sequence lengths. For larger Rust libraries, RULF usually takes several tens of minutes to generate fuzz targets with lengths of 3, and generating sequences with lengths more than 4 may take several hours or even days, which is unacceptable in terms of resource consumption. During our experiments with RULF, we even encountered crashes due to insufficient memory.

Overall, the experimental results show that FRIES can not only synthesize fuzz targets with higher dependency, API coverage and valid ratio, but also significantly reduces resource consumption in fuzz testing, improving the complexity and length of the generated sequences, outperforming current state-of-the-art techniques.

Table 5: Bugs detected by FRIES.

Library	Func	Crashes	OF	OB	EN	UC	OT	Bugs
hifitime	279	1,372	6	18	4	1	1	30
url	75	3	0	0	0	0	1	1
ratatui	227	245	14	0	0	0	0	14
regex-automata	270	9,870	0	1	0	0	0	1
time	212	1,278	10	0	0	0	0	10
regex	145	9,036	1	5	0	0	0	6
uni-seg	11	314	4	2	1	0	2	9
csv	114	201	2	0	0	0	0	2
tui	231	578	13	2	0	0	0	15
console	121	714	3	0	0	0	0	3
hyper	161	721	0	0	0	0	1	1
chrono	337	3,782	7	0	0	0	0	7
bytes	88	225	1	0	0	0	0	1
byteorder	12	286	0	8	0	0	0	8
regex-syntax	238	2,379	0	1	0	1	0	2
xi-core-lib	214	386	18	2	0	0	0	20
Total	2,735	31,390	79	39	5	2	5	130

[1] FRIES discovered bugs in 16 out of 20 libraries, with a total of 130 bugs, including 84 that were previously unknown.
[2] These bugs have been submitted to the corresponding library's issue in GitHub, and 54 of them have been confirmed or fixed.
[3] uni-seg is the library unicode-segmentation.

5.3 Bug Detection Effectiveness

We conduct fuzz testing for 20 Rust libraries to assess FRIES's bug-finding capability, shown in Table 5. In total, FRIES discovered a total of 130 bugs, including 84 previously unknown bugs. We

Table 6: Bugs Found by FRIES, RULF and SyRust

Tool	OF	OB	EN	UC	OT	Total	Unknown	Confirmed
FRIES	73	21	1	1	4	100	54	24
RULF	59	13	1	0	2	75	35	17
SyRust	0	0	0	0	0	0	0	0

have submitted the corresponding issues in Github, of which 54 bugs have been identified or fixed. The majority of the targets, 99%, passed compiler checks and they caused a total of 31,390 crashes. Among the 16 libraries where crashes occurred, 59.8% of the targets were able to trigger a crash. For all generated targets, the average number of crashes that can be triggered per target is 31.4, while the RULF is 6.1. And since a large number of these crashes appeared in duplicate stack traces in the source code, FRIES automatically removed the redundancy, resulting in 174 bug warnings being reported. After manual inspection, 74.7% of the warnings were recognized as real bugs. Excluding library hifitime, which cannot be measured by RULF, FRIES's accuracy is 77.5% compared to RULF's 72.1%.

Table 5 presents the classification and quantity of the bugs we discovered. The types of bugs detected include arithmetic overflow(OF), out-of-bounds(OB), encoding(EN), execution of unimplemented code(UC), and unwrap errors or others(OT). Most of the bugs FRIES found were due to integer overflow issues, accounting for 60.8% of the total. This is often a result of inadequate parameter validation and error handling by library developers. Notably, these errors can lead to critical software vulnerabilities and failures[15]. 30% of the identified bugs relate to array out-of-bounds issues, impacting program stability. The remaining 3 error types contribute to approximately 10% of the bugs. These include crashes due to encoding format discrepancies during encoding/decoding, encounters with unimplemented code leading to PANIC, and failures during the unwrap process. For the libraries with no bugs found, namely `http`, `textwrap`, `serde-json`, `semver`, they are mainly due to better robustness or the libraries are so small that they rarely have bugs, e.g., of the four, all but http are in the Small category.

Table 6 shows the bugs detected by FRIES, RULF and SyRust with the same total running time of the synthesis and testing. RULF's spatiotemporal complexity limitations prevented fuzz target synthesis for hifitime due to insufficient memory, hence the evaluation was based on 19 other libraries. FRIES was able to find 33% more bugs than RULF, which proves our approach is more effective in bug findings. As a technique that focuses on program synthesis rather than finding bugs, due to SyRust's scalability, it can test at almost 15 functions, and its inability to mutate inputs makes it difficult to trigger a bug. So no bugs were found by SyRust, which is almost consistent with the conclusion of the paper [31]. Since we have a generation algorithm with type checking and ecosystem guidance, we are able to more efficiently detect vulnerabilities that may have a greater impact on downstream programs in the ecosystem.

In following parts, we further discuss a few bug cases detected by our approach to show the effectiveness of FRIES.

Bug Case 1: The code in Listing 5 shows a minimized code snippet that triggers a bug in `xi-core-lib` library. It is worth noting that this code snippet, along with the code example listed in Section 2, originates from the same fuzz target generated by FRIES, with an invocation length of 15 in total, which demonstrates the role of long sequences generated by FRIES.

The purpose of this code is to add a selected region `SelRegion` to the selection state `Selection` and then retrieve the input within the specified range. The function `regions_in_range(...)` is used to provide a list of regions `&[SelRegion]` within a given range $[start, end)$. When encountering invalid input, such as `start=17` and `end=16` (where `start>end`), the function returns an empty array. However, an error "slice index starts at 2 but ends at 1" is triggered when `end=15`, resulting in a bug inside the library. In real programs, it is common to perform the action of adding a selected region `add_range_distinct(...)` before querying the selection state `regions_in_range(...)`. Adding a range is a mutable operation, while querying is an immutable operation. Based on patterns mined, the efficient algorithm with type checking allows FRIES to succeed in generating such an effective long bug-triggering target.

```
use xi_core_lib::selection::SelRegion;
use xi_core_lib::selection::Selection;
fn main() {
    let v1:SelRegion = SelRegion::caret(0);
    let mut selection:Selection = Selection::new_simple(v1);
    let v2:SelRegion = SelRegion::caret(16);
    Selection::add_range_distinct(&mut selection, v2);
    Selection::regions_in_range(&selection, 17, 15);
}
```

Listing 5: (Bug Case1) A fuzz target generated by FRIES.

Bug Case 2: The fuzz target generated by FRIES shown in Listing 6 triggered an unsatisfied constraint issue in Library `url`, which has been fixed after we reported it. The main purpose of this code snippet is to parse and process a URL. First, the function `parse(...)` is used to attempt parsing a string as a URL. Next, the function `join(...)` is used to concatenate variable $v1$ with a string. The error occurred within the `join(...)` function, where the private function `parse_path(...)` within the library is called incorrectly, resulting in an assertion and program crash.

Although this code snippet is relatively short, analysis based on the corpus crates reveals that the dependency relationship between `parse(...)` and `join(...)` appears frequently in the corpus programs, indicating a high real-world usage frequency and significant impact on the ecosystem. Therefore, FRIES could generate more of these dependency relationships in the fuzz targets, allowing the fuzzer to quickly identify corresponding bugs.

```
use url::Url;
let v1 = if let Ok(x) = Url::parse(r"fIlE:f\p:?../") {x}
            else { use std::process; process::exit(0); };
Url::join(&(v1), r"../r\\\\\m");
```

Listing 6: (Bug Case2) A fuzz target generated by FRIES.

This particular dependency occurred seven times among the 50 fuzz targets FRIES generated. Although RULF may also generate a similar fuzz target, it produces only one sequence containing this dependency at most guided by API coverage. In contrast, our approach enables more efficient identification of bugs that have a greater impact on the program ecosystem.

Bug Case 3: FRIES discovered an array out-of-bounds bug in the `regex-automata` library using the code snippet shown in Listing 7. This code utilizes the functions of the `regex-automata` library to perform regular expression automaton searches. It creates a DFA and a cache , as well as matches and their respective patterns. The `find_leftmost_rev_at(...)` function takes the previously created DFA, cache, and pattern as parameters to perform the matching. During the execution of the matching function, which involves a

6-level function invocation chain, an array out-of-bounds error is triggered in the code `self.cache.starts[index]` within the private function `get_cached_start_id(...)` defined in the library.

In this case, a complex sequence is essential, as the target function `find_leftmost_rev_at(...)` receives 3 different API-specific data structures, all of which need to be generated by other function calls. With such an effective function selection strategy and powerful flexibility, we successfully generated $v1$, $v3$, and $v4$. Then, based on the mined API usage patterns, FRIES successfully selected the `find_leftmost_rev_at` function and referenced these produced variables with API-specific data structures. The efficient algorithm enables the generation of such complex and lengthy sequences in almost linear time complexity, an achievement that is practically unimaginable for RULF to generate sequences longer than three. And It can produce cross-module interactions that Syrust cannot.

```
use regex_automata::hybrid::dfa::Cache;
use regex_automata::hybrid::dfa::DFA;
use regex_automata::HalfMatch;
let v1 = if let Ok(x) = DFA::always_match() { x }
            else { use std::process; process::exit(0); };
let v2 = HalfMatch::must(0, 0);
let mut v3 = Cache::new(&v1);
let v4 = regex_automata::HalfMatch::pattern(&v2);
DFA::find_leftmost_rev_at(&v1, &mut v3, Some(v4),
            &[124, 124, 123, 133, 144], 0, 0);
```

Listing 7: (Bug Case3) A fuzz target generated by FRIES.

5.4 Limitations

The experimental results show that FRIES performs well on diverse crates from different domains; however, there are some limitations of the current implementation of FRIES. Rust language has some complex language features, such as generics, macros, and traits that are not yet supported in FRIES, which naturally limits the API coverage to around 60%. Incorporating these features could make FRIES to cover more APIs and thus improve the bug detection efficiency. Our experiments demonstrate the effectiveness of FRIES's ownership assurance and complex interactions. The extensibility of its algorithms facilitates the addition of functionality and future integration with other technologies, thus expanding its use. The ownership mechanism including borrowing is the most important feature of rust, which we support, and whether or not it is satisfied determines whether or not it can be compiled. For corpus analysis, there may be room for further increases in algorithm complexity, but even so, our mining algorithms are fast enough to be tolerable in a production environment.

5.5 Threats to Validity

The effectiveness of FRIES relies on the representativeness of the corpus, thus the choice of corpus may introduce bias in the experiment result. To minimize the potential bias, in the experiment, we identified up to 200 programs per library to analyze, which forms a relatively large corpus. The experimental results show, on average, each library yielded over a hundred pairs of entirely distinct dependency and order pairs. The performance of FRIES and its bug detection effectiveness relies on the libraries' characteristics and usage scenarios, which may introduce a threat to validity. To alleviate this threat, we chose libraries of different domains and sizes from crates.io, and thus evaluated the performance of FRIES on diverse crates. The instability of the measurement tools introduces potential threats to validity. To address this, we manually filtered out extremely unreasonable data points based on predefined quality thresholds.

6 RELATED WORK

Testing or Static Analysis for Rust. There are also some efforts designed for testing Rust libraries. RustyUnit [33] is a search-based unit test generation tool for Rust. SyRust [31] is another tool for unit testing. It uses semantics-aware program synthesis algorithms to generate compilable test cases for Rust libraries. RULF [23] constructs graphs based on function signatures and generates API invocation sequences by traversing the graphs. Different from these approaches, our proposed approach incorporates API usage mining to generate valid sequences of complex and long interactions with low overhead. There are also static analyses for rust program vulnerability detection, such as MirChecker which detects potential runtime crashes and memory safety errors, etc., and SafeDrop which detects invalid memory allocations using path-sensitive data flow analysis.

API Interactions Testing for Other Languages. In earlier years, unit testing for Java was a hot topic of research, such as Korat [14], DiffGen [32], etc. And there has been several work [9, 18, 19] to improve on EVOSUITE [17] to guarantee the robustness of the java program. In addition, Pynguin [29] is a unit test generation technique for dynamically typed languages, i.e., Python. In recent years, there have been several works related to API fuzzing, which needs fuzzing targets. A survey [25] introduces a method for automatically generating fuzz targets and testing each API separately, which can only test functions with basic type parameters. RESTler [10] and Morest [28] are methods designed for RESTful API that use graph algorithms to generate appropriate sequences of API calls. Fudge [11] and FuzzGen [21] are fuzz target auto-generation tools proposed by Google. It utilizes client code from Google's vast code repository to generate candidate fuzz targets for libraries by intercepting code snippets. APICraft [37] mutates existing programs to generate fuzz targets for closed-source SDK libraries. UTopia [22] builds upon existing unit tests to generate new fuzzing targets through mutation. WINNIE bypasses irrelevant GUI code to test logic deep within the application to fuzz Windows applications. However, these methods are limited in scope and are unable to deal with Rust's ownership mechanism.

7 Conclusion

This paper proposes a fuzzing technique to test Rust libraries. It combines the syntax constraints and patterns mined in Rust ecosystems to generate the fuzz targets. The experimental results show that FRIES can efficiently generate high-quality fuzz targets, outperforming previous techniques. To date, FRIES has detected 84 significant and unknown bugs in Rust libraries.

Acknowledgments

We would like to thank anonymous reviewers for their constructive comments. This project was partially funded by the National Natural Science Foundation of China under Grant No. 62372225 and No. 62172209.

References

[1] [n. d.]. crates.io. https://crates.io/.

[2] [n. d.]. Firecracker MicroVMs. https://firecracker-microvm.github.io/. Accessed: 2023-07-24.

[3] [n. d.]. Redox OS. https://redox-os.org/. Accessed: 2023-07-24.

[4] [n. d.]. Rustc HIR on GitHub. https://github.com/rust-lang/rust/tree/master/compiler/rustc_hir. Accessed: Place the access date here.

[5] [n. d.]. Tokio. https://tokio.rs/. Accessed: 2023-07-24.

[6] 2023. American fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[7] 2023. A vulnerability database for the Rust ecosystem. https://rustsec.org/.

[8] Ongoing development. The Rust Programming Language. https://github.com/rust-lang/rust. Accessed on 2022-06-14.

[9] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 79–90.

[10] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 748–758.

[11] 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.

[12] 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.

[13] Thomas Ball and Sriram K Rajamani. 2017. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. https://www.microsoft.com/en-us/research/publication/few-billion-lines-code-later-using-static-analysis-find-bugs-real-world/.

[14] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated testing based on Java predicates. *ACM SIGSOFT Software Engineering Notes* 27, 4 (2002), 123–133.

[15] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2015. Understanding Integer Overflow in C/C++. *ACM Trans. Softw. Eng. Methodol.* 25, 1, Article 2 (dec 2015), 29 pages. https://doi.org/10.1145/2743019

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[17] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[18] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291. https://doi.org/10.1109/TSE.2012.14

[19] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2, Article 8 (dec 2014), 42 pages. https://doi.org/10.1145/2685612

[20] 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)*. 982–991. https://doi.org/10.1109/QRS57517.2022.00102

[21] Kyriakos K Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. Fuzzgen: Automatic fuzzer generation. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 2271–2287.

[22] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. 2023. UTopia: Automatic Generation of Fuzz Driver using Unit Tests. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2676–2692. https://doi.org/10.1109/SP46215.2023.10179394

[23] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. 2021. RULF: Rust library fuzzing via API dependency graph traversal. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 581–592.

[24] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*.

[25] 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.

[26] Steve Klabnik and Carol Nichols. 2019. *The Rust Programming Language* (2nd ed.). No Starch Press. https://doc.rust-lang.org/book/ch00-00-introduction.html

[27] 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.

[28] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. 2022. Morest: Model-Based RESTful API Testing with Execution Feedback. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1406–1417. https://doi.org/10.1145/3510003.3510133

[29] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated unit test generation for python. In *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12*. Springer, 9–24.

[30] 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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. https://doi.org/10.1145/3385412.3386036

[31] 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.

[32] Kunal Taneja and Tao Xie. 2008. DiffGen: Automated Regression Unit-Test Generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 407–410. https://doi.org/10.1109/ASE.2008.60

[33] Vsevolod Tymofyeyev and Gordon Fraser. 2022. Search-Based Test Suite Generation for Rust. In *International Symposium on Search Based Software Engineering*. Springer, 3–18.

[34] Shi Wang, Vivien Bono, Ana Maria Valdivia, and Miryung Kim. 2021. The Rust programming language: An empirical study on its use and adoption in the industry. *IEEE Transactions on Software Engineering* 47, 5 (2021), 1040–1064.

[35] Hui Xu, Zhuangbin Chen, Mingshen Sun, and Yangfan Zhou. 2020. Memory-Safety Challenge Considered Solved? An Empirical Study with All Rust CVEs.

[36] 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 Trans. Softw. Eng. Methodol.* 31, 1, Article 3 (sep 2021), 25 pages. https://doi.org/10.1145/3466642

[37] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. 2021. APICraft: Fuzz Driver Generation for Closed-source SDK Libraries. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2811–2828. https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-cen