RULF: Rust Library Fuzzing via API Dependency Graph Traversal

Jianfeng Jiang, Hui Xu, Yangfan Zhou

摘要

鲁棒性是 Rust 库开发的一个关键问题，因为如果开发人员只使用安全的 API，Rust 承诺不会出现未定义行为的风险。模糊测试是检验程序鲁棒性的一种实用方法。然而，由于模糊目标的缺失，现有的模糊测试工具并不能直接适用于库 API。模糊目标的逐案设计主要依靠人的努力，是劳动密集型的。针对这一问题，本文提出了一种新的自动模糊目标生成方法，通过 API 依赖图遍历来模糊测试 Rust 库。我们确定了库模糊测试的几个基本要求，包括模糊目标的正确性和有效性、高 API 覆盖率和效率。

为了满足这些要求，我们首先使用广度优先搜索和剪枝来寻找长度阈值下的 API 序列，然后反向搜索较长的序列以寻找未覆盖的 API，最后将序列集优化为集合覆盖问题。

我们实现了我们的模糊目标生成器，并在几个现实世界流行的 Rust 项目中使用 AFL++ 进行模糊实验。我们的工具最终为每个库生成 7 到 118 个模糊目标，API 覆盖率达到 0.92。我们以 24 小时为阈值测试每个目标，并从 7 个库中找到 30 个以前未知的错误。

关键词：模糊测试；程序合成；Rust

1 引言

Rust 语言是一种新兴的语言，它促进了内存安全特性，同时又不牺牲太多性能。它向开发人员承诺，如果他们不使用不安全的代码，他们的程序将不会遭受未定义的行为。同时，该语言包含了许多其他编程语言的新特性和最佳实践。由于这些优点，Rust 在近年来迅速流行起来，并被许多学术项目和工业项目采用。尽管 Rust 在语言级别提供了专门定制的机制来增强安全性，但在现有的 Rust 项目中仍然报告了许多严重的 bug。特别是，AdvisoryDB[1]和 Trophy-Case[2]是两个著名的公共存储库，在 Rust 项目中发现了数百个 bug。这些 bug 的一个有趣现象是，它们的大多数宿主程序都是库[3]。这样的错误只能在使用库 API 的特定用法组合程序时触发。因为 Rust 强调软件的安全性和健壮性，所以必须查找这些库 bug。不幸的是，我们仍然缺乏检查 API 鲁棒性的有效工具。例如，模糊测试[4]和符号执行[5]通常需要可执行目标; 对第三方库 API 的形式化验证[6]不能完全自动化。现有的模糊测试工具，如 AFL++[7]、HongFuzz[8]、LibFuzzer[9]等，在进行库模糊测试时都需要模糊目标，而模糊目标的编写主要依靠人工。Fudge[10]是最近为 C/C++ 程序提出的模糊目标生成器，它从谷歌代码库中提取代码片段来形成模糊目标。但其有效性很大程度上取决于库的使用情况，存在很大的局限性。为此，作者提出了一种基于给定库的 API 依赖关系图的自动生成模糊目标方法，将模糊目标视为 API 序列，通过在 API 依赖关系图进行 BFS 和后向搜索生成 API 调用序列，最终生成模糊目标。该方法摆脱了对代码库的依赖，具有一定的通用性。

2 相关工作

2.1 模糊目标生成问题的目标

正确性：合成的模糊目标应该能够被成功编译，即每个 API 的所有参数都应该被正确地确定。

API 覆盖率：对于库程序来说，每个 API 都可能包含 bug。因此，生成的模糊目标集应该涵盖尽可能多的 API。

高效性：生成的模糊目标集对于模糊测试应该是有效的。一方面，模糊目标的数量应该是可控的，因为模糊测试每个目标通常需要几个小时。另一方面，如果可能的话，我们应该避免在不同的模糊目标中使用相同的 API，因为多次模糊测试相同的 API 可能没有帮助。

有效性：生成的模糊目标应该对用于 bug 搜索的模糊工具友好。为了有效地模糊测试 API，生成的程序应该很小。否则，模糊测试工作将浪费在测试程序或其他 API 上，而不是目标 API 上。

2.2 API 依赖关系图的构建

如果一个 API 的返回值和另一个 API 的参数具有相同的类型，则可以合成一个有效的程序，该程序使用第一个 API 的返回值作为参数调用第二个 API。换句话说，两个 API 之间可能存在一种数据流依赖关系。非正式地说，API 依赖关系图是一个有向图，它捕获了 crate 中 API 之间所有可能的数据流依赖关系。

形式上，将 API 依赖关系图定义为四种结点的组合，分别是 API 结点、参数结点、生产者边和消费者边。根据参数类型和返回值类型，为参数结点与 API 结点之间添加生产者边或消费者边。此外，进一步定义两种特殊的 API 结点：不使用非基本类型参数的开始结点和结束结点。API 结点可以同时是开始结点和结束结点。一个 API 依赖关系图示例如图 1所示。

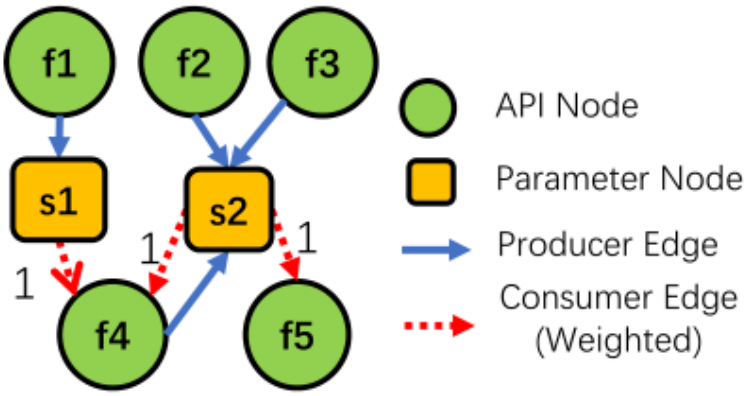

图 1: API 依赖关系图

2.3 API 序列生成

在 API 依赖关系图上，API 结点是可达的当且仅当它是一个开始结点或其所有必需参数结点都可达，并满足消费者边权值。类似地，如果至少有一个可以产生参数的 API 结点是可达的，那么参数结点是可达的。另外，对有效 API 序列的定义并不强制相邻 API 结点之间存在优先关系，这样，具有重复子序列或具有多个结束结点的序列也可以是有效的。

通过遍历 API 依赖关系图来生成有效的序列。根据模糊目标生成问题的目标，算法应该满足三个目标：API 覆盖率、高效性和有效性。为此，作者首先采用带剪枝的 BFS 方法获得一组 API 序列，该序列在给定长度阈值的情况下能够覆盖最大数量的 API；然后，通过反向搜索长度大于 BFS 阈值的包

含未覆盖 API 的有效序列。最后，将这两个集合合并为一个集合，并进一步细化它以去除冗余。

3 本文方法

3.1 本文方法概述

首先构造 API 依赖关系图，分为两个阶段。第一个阶段从源代码中提取所有公共 API 签名，然后，基于类型推理推断 API 之间的依赖关系，以构建依赖关系图。为类型推断定义调用类型：如果存在从类型 A 到类型 B 的调用类型，则 A 和 B 是相同类型或 A 可以转换为 B。根据调用类型、参数类型和返回类型，在参数结点和 API 结点之间添加边，生成 API 依赖关系图。

构建 API 依赖关系图后，通过遍历图生成有效序列，主要通过 DFS 和后向搜索进行遍历。同样，DFS 的遍历带有阈值和剪枝，在遍历过程中允许生成冗余的序列，以满足对同一函数多次调用获取相应数量参数的情况。再 DFS 遍历得到所有在阈值范围内可得到的序列后，进行剪枝操作，去除冗余序列。然后，使用后向搜索查找未被覆盖的 API，通过使用在 DFS 中生成的序列为其提供参数，以判断其是否能够加入到序列中。

最后，将 DFS 中得到的序列集与后向搜索得到的序列集进行合并操作，并进行优化，去除冗余序列。

3.2 API 序列的有效性

对于一个序列 $f_{n0}, \ldots$, 如果 f_{n0} 是一个开始结点，并且对于给定的子序列每个 $f_{ni}(0 < i \leqslant k)$ 都是可达的，则该序列是有效的。这是合成有效模糊目标的基本要求。例如，{f2, f5} 在图 1中是一个有效序列；{f1, f4} 无效，因为 s2 无法到达 f4。

3.3 调用类型定义

所有调用类型如表 1所示，其可以分为两类。前七个是通用调用类型（从直接引用到解引用原始指针），后三个是两个类型的特定调用类型，Result 类型和 Option 类型。Result 类型和 Option 类型是 Rust 中常用的枚举类型。Result 用于返回可恢复的错误，Option 用于表示可以为空的值。这两种类型用于包装普通类型，例如，u8 是 8 位整数类型，而 Option 可以是普通 u8 或 None。如果 Option 不是 None，可以通过 unwrap 操作来获取 u8 类型。

Call type	Example
Direct call	T→T
Borrowed reference	T→&T
Mutable borrowed reference	T→&mut T
Const raw pointer	T→*const T
Mutable raw pointer	T→*mut T
Dereference borrowed reference	&T→T
Dereference raw pointer	*const T→T
Unwrap result	Result<T, E>→T
Unwrap option	Option<T>→T
To option	T→Option(T)

表 1: 用于类型推断的调用类型

3.4 DFS 阈值的确定

如表 2所示，本文在 smallvec 库上测试了阈值长度为 2 到 5 时算法的性能，发现当阈值为 2 时，DFS 能够覆盖的 API 数太少，使得过多的工作交给了后向搜索。阈值从 3 到 5，算法运行时间越来越长。观察生成的序列发现，大部分序列长度为 2 到 3，这就说明序列长度 3 以后的遍历大部分都是无效遍历，严重影响了算法的效率。同时阈值设定为 3 时，API 覆盖率高且时间开销小，因此选定 3 为 DFS 的阈值。

Threshold	Time	Cover APIs
2	726	0.75
3	542	1.00
4	6018	1.00
5	185141	1.00

表 2: 阈值测试

3.5 合并和优化

获得 DFS 的序列集和后向搜索得到的序列集后，将两个集合合并，再进行优化。优化的目标是选择提供等效 API 覆盖率的序列的最小子集。这通常是一个集合覆盖问题 (SCP)[11]，它是一个 NP 完全问题。

为了解决这个问题，采用贪心算法[12]。算法选择一个对 APi 覆盖率贡献最大的序列，知道覆盖所有 API。基于以下规则，算法倾向于选择 A 而不是 B：

- A 覆盖比 B 更多的新结点；
- A 和 B 覆盖等量的新结点，但 A 覆盖更多新的边；
- A 和 B 覆盖等量的新结点和边，但 A 比 B 更短；

如果根据以上规则有多个贡献相等的序列，从其中随机选择一个。以图 1为例，假设 {f3} 和 {f3, f5} 是两个候选序列进行选择。如果 f5 或从 f3 到 s2 (f5 的参数节点) 的生产者边没有被覆盖，选择 {f3,f5}，因为它对覆盖的贡献更大。否则，选择 {f3}，因为它更短。

4 复现细节

4.1 与已有开源代码对比

API 依赖图的构建代码参考原论文代码编写。对于 API 序列的构建，原论文采用 BFS 的策略遍历 API 依赖关系图生产 API 序列。而本文则采用 DFS 的策略遍历 API 依赖关系图，与 BFS 不同，DFS 实现上更为复杂。借助队列，BFS 可以很方便获取当前要添加 API 的指定长度的序列，而 DFS 需要进行回溯，但 API 序列并不以链表的方式实现，而是以数组的方式实现，回溯并不方便，需要借助 tmp_sequence 数组来记录当前遍历序列的各个长度的子序列，以及一个 tmp_len 变量来记录当前遍历序列的长。通过这两者的组合实现回溯。另外，BFS 一层层的增加结点，每一层的结点在搜索下一层可添加 API 时只会搜索所有 API 一次，而 DFS 由于回溯的原因，需要记录每一层结点进入下一层结点时遍历到的 API 位置，回溯时从该位置继续遍历。否则，重复遍历会导致重复添加相同的 API，从而无法退出循环。

DFS 实现的伪代码如下所示：

Procedure 1 DFS with Pruning

Input: A graph *G(FN, PAR, PE, CE),* Sequence length threshold *max_ len*
Output: A set of sequences S_{DFS}
Init:
  $S_{DFS} \leftarrow \emptyset$;
  $S_{new} \leftarrow Vec(max\_len + 1,\ empty\_sequence)$;
  no_more ← *false;*
  len ← 0*;*
DFS with Pruning:
while !no_more **do**
  $seq \leftarrow S_{new}$ [*len*];
  if *EndNodeTest(seq)* **then**
    $len \leftarrow len - 1$;
    continue;
  end
  for fn ***in*** $G \rightarrow FN$ **do**
    if *ReachabilityTest(G, seq, fn)* **then**
      new_seq ← *seq.append(fn);*
      $len \leftarrow len + 1$;
      S_{new} [*len*] ← new_seq;
      S_{DFS}.append(new_seq);
    end
  end
  if *NoNewSeq()* **then**
    $no\_more \leftarrow true$;
  end
  if *CanRecall()* **then**
    $len \leftarrow len - 1$;
  end
end
for seq ***in*** S_{DFS} **do**
  if *RedundancyTest(seq)* **then**
    S_{DFS}.*remove(seq);*
  end
end

4.2 实验环境搭建

安装 Ubuntu18.04

在 Ubuntu 中，安装 Rust

curl –proto ’=https’ –tlsv1.2 -sSf https://sh.rustup.rs | sh

cargo install –force afl

安装需要的依赖，其它依赖，后续提示需要什么就安装什么

apt-get install build-essential ninja-build python3 cmake

进入工作目录，拉取 1.59 版本的 rust 源代码

cd Rust-Lib-Testing/src

git clone -b 1.59.0 https://github.com/rust-lang/rust.git rust –depth 1

进入 rust 目录，将 rust 引入为项目的子模块（最后一步需要较长时间）

cd rust

git submodule init

git submodule update

运行项目包里的构建脚本，构建程序（需要较长时间）

cd ..

./scripts/build.sh init

运行脚本编译程序

./scripts/build.sh fast

4.3 界面分析与使用说明

首先运行项目主目录中的脚本配置环境变量

source ./scripts/init_env.sh

然后进入到要测试的库目录中，运行运行工具，这里以 smallvec_v1.7.0 为例

cd tests/smallvecs_ v1.7.0

rustgen init

执行 rustgen init 后，会有如图 2所示界面让你选择要测试的 Crate，一般选择 1。

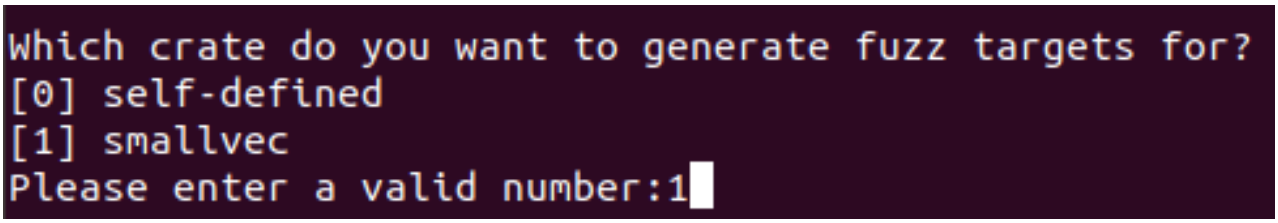

图 2: 选择测试的 Crate

运行结果如图 3所示。

图 3: 运行结果示意图

表中各项的含义：

TN: Total Nodes，被测试 Crate 中 API 总数；

IN: Involved Nodes，被覆盖 API 个数；

CN: Covered Nodes，API 覆盖率；

UTN: Unsafe Total Nodes，unsafe API 总数；

UCN: Unsafe Covered Nodes，unsafe API 的覆盖率；

TE: Total Edges，API 依赖关系图中边的总数；

IE: Involved Edges，被覆盖的边的个数；

CE: Covered Edges, 边覆盖率；

UTE: Unsafe Total Edges, 与 unsafe API 有关的边的总数；

UIE: Unsafe Involved Edges, 被覆盖的与 unsafe APi 有关的边的个数；

UCE: Unsafe Covered Edges, unsafe API 有关边的覆盖率；

total test sequences: 生成并测试的序列总数；

Total Time Used: 遍历 API 依赖关系图生成 API 序列并去除冗余的过程消耗的时间；

4.4 创新点

在 API 序列的生成方法上，原文使用了广度优先遍历（BFS）和后向搜索。本文则选用了深度优先遍历（DFS）和后向搜索。此外，本文还实现了广度优先遍历加深度优先遍历加后向搜索的方法。

5 实验结果分析

本文在 smallvec_v1.7.0、libc_v0.2.112、cc_v1.0.72 这三个 Crate 上进行了测试，实验结果如下图，其中左边是本文的 DFS+BackWardSearch 方法的结果，右边是作者的 BFS+BackWardSearch 方法的结果。

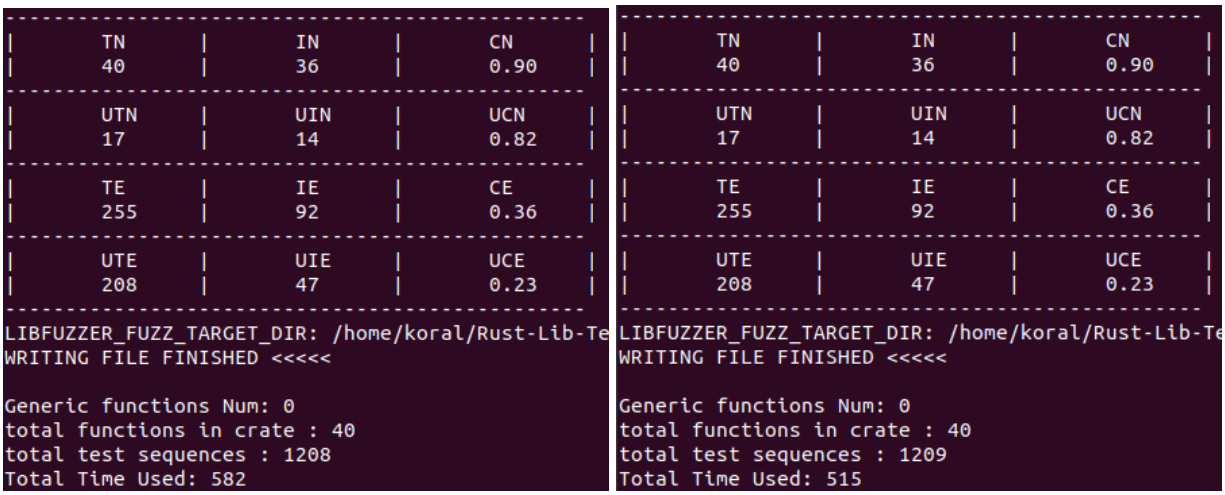

图 4: smallvec 测试结果

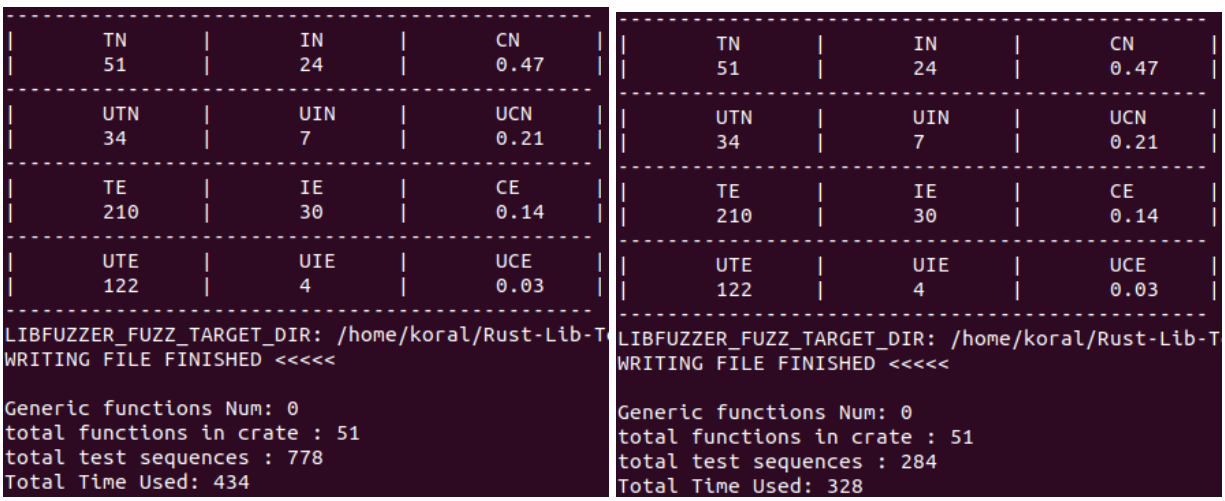

图 5: libc 测试结果

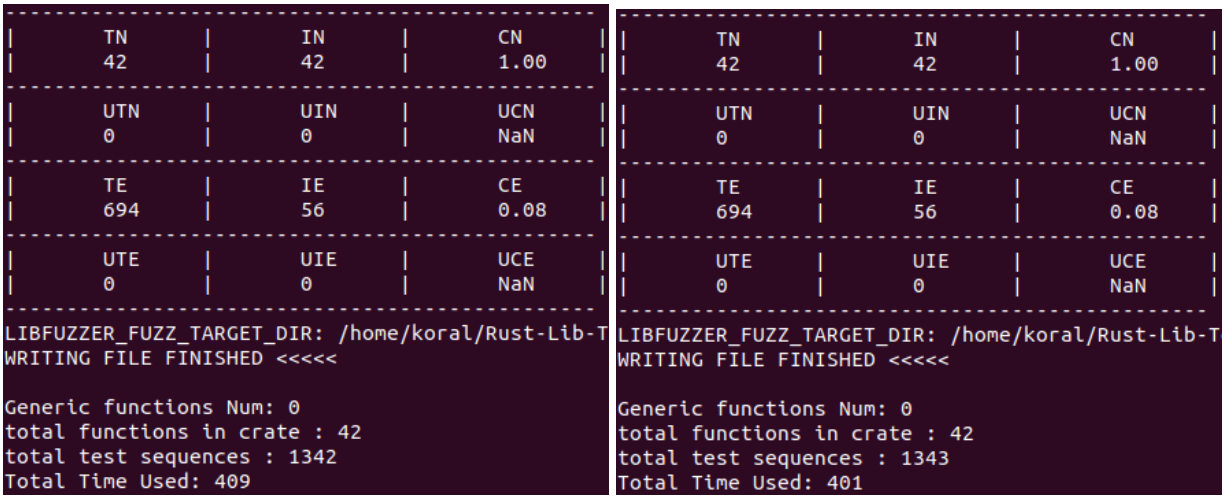

图 6: cc 测试结果

Crate	API Coverage	Test Sequences	Generate Sequences	Time
cc_v1.0.72	1.00|1.00	1342|1343	38|38	409|401
libc_v0.2.112	0.47|0.47	778|284	27|26	434|328
smallvec_v1.7.0	0.90|0.90	1208|1209	50|50	582|515

表 3: 实验结果 (DFS|BFS)

从实验结果可以看到,两种方法在三个库测试中的 API 覆盖率是相同的,但是 DFS+BackwardSearch 消耗的时间远远大于 BFS+BackwardSearch。从代码实现的角度看，在 BFS 中，以序列长度作为遍历条件，每次从结果队列集中选择的符合当前要遍历长度的序列 push 进缓存队列中即可，然后遍历缓存队列中的序列，查找能够添加到序列中的 API 加入到序列中生成新序列并加入到结果队列集中。而在 DFS 中，由于序列不是以链表的方式实现的，所以数组记录当前被遍历到的序列的各个长度的子序列已实现回溯。同时，为了实现回溯，需要设置多个信号来协同完成。另外，为了满足同一个函数被多次调用来生成多个同一种类型的参数的情况，需要允许 API 被多次扫描，由于深搜的特性，这样的操作导致了算法耗时很多。为了解决这个问题，引入对某个深度的序列遍历结束的位置的记录，回溯到该深度时可以跳过前面的序列，以减少耗时。但同时也引入了很多条件判断的耗时以及实现复杂度。

总的来说，BFS 比起 DFS 更适用于当前场景。此外，本文还尝试先找到所有开始结点，再从这些开始结点深度搜索的方式，但是并没有带来性能上的改善。

6 总结与展望

作者提出的模糊目标生成方法想法很好，充分运用了静态分析，通过库自身的 API 相互提供参数，实现了通用性，避免了像 Fudge 那样对代码库的依赖。同时，很好地根据 Rust 的特性设计了调用类型，在实现 API 依赖关系图的构建上很方便。

虽然通过库自身的 API 相互提供参数以实现对 API 的调用是十分巧妙的构思，从实验结果看也不错，但依然有可能存在因为某些特殊调用而没有被扫描进序列的 API。为了解决这个问题，需要进一步分析未被覆盖 API 的调用类型以及这些 API 的特点。

除此之外，DFS 的性能应该还能够继续提升，通过简化代码，减少各种信号变量的使用，能够进一步降低运行时间。特别是回溯的实现，以更巧妙的方式实现将能够很好的提升性能。

参考文献

[1] R. S. C. WG. ”Rustsec advisory database”[Z]. https://github.com/RustSec/advisory-db. 2021.

[2] R. F. Authority. ”Trophy case”[Z]. https://github.com/rust-fuzz/trophy-case. 2021.

[3] XU H, CHEN Z, SUN M, et al. Memory-safety challenge considered solved? an empirical study with all rust cves[J]. arXiv preprint arXiv:2003.03296, 2020.

[4] LIANG H, PEI X, JIA X, et al. Fuzzing: State of the art[J]. IEEE Transactions on Reliability, 2018, 67(3): 1199-1218.

[5] CADAR C, SEN K. Symbolic execution for software testing: three decades later[J]. Communications of the ACM, 2013, 56(2): 82-90.

[6] JUNG R, JOURDAN J H, KREBBERS R, et al. RustBelt: Securing the foundations of the Rust programming language[J]. Proceedings of the ACM on Programming Languages, 2017, 2(POPL): 1-34.

[7] Z. Michal. ”American fuzzy lop”[Z]. https://lcamtuf.coredump.cx/afl/. 2015.

[8] Google. ”Honggfuzz”[Z]. https://github.com/google/honggfuzz. 2020.

[9] L. team. ”libfuzzer –a library for coverage-guided fuzz testing”[Z]. https://llvm.org/docs/LibFuzzer.html. 2020.

[10] BABIĆ D, BUCUR S, CHEN Y, et al. Fudge: fuzz driver generation at scale[C]//Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2019: 975-985.

[11] GAREY M R, JOHNSON D S. Computers and intractability[J]. A Guide to the, 1979.

[12] CHVATAL V. A greedy heuristic for the set-covering problem[J]. Mathematics of operations research, 1979, 4(3): 233-235.