

ファジングにおける希少到達領域の 特性に関する実証的研究

岩橋 涼介^{1,a)} 加藤 和志^{1,b)} 杉山 優一^{1,c)}

概要：

ファジングは強力なソフトウェアテスト手法であるが、その探索効率はマジックバイトやチェックサムに代表される「Fuzz Blocker」によってしばしば阻害される。これまでの研究は特定の Fuzz Blocker を克服することで性能を向上させてきたが、最先端のファザーでも長時間の実行でカバレッジ増加が停滞する問題が依然として残る。これは、既存手法では扱い切れていない「未解決 Fuzz Blocker」の残存を示唆するが、その体系的な特定は容易ではない。主因は、テスト環境の不備に起因する要因と、ファジング・アルゴリズムの改善によって真に解決すべき要因を切り分ける方法が確立していない点にある。本研究は、アルゴリズム改善の対象となる Fuzz Blocker を系統的に抽出すべく、多数回の独立試行に基づき到達確率が極めて低い非ゼロの「希少到達領域」を統計的に検出する方法論を提案する。本手法は、環境設定の不備に起因して理論上到達不能な領域を除外し、現行アルゴリズムが苦手とする解析価値の高い箇所のみを選別する。予備評価として、本手法を実プログラムに適用し、希少到達領域の出現割合を定量化するとともに、ケーススタディによりその特性を示す。得られた知見は、「次に解くべき Fuzz Blocker」の同定を支援し、今後のファジング・アルゴリズム設計の基盤となりうる。

キーワード：ファジング, Fuzz Blocker, 希少到達領域, カバレッジ停滞

An Empirical Study on the Characteristics of Rarely Reached Regions in Fuzzing

RYOSUKE IWAHASHI^{1,a)} KAZUSHI KATO^{1,b)} YUICHI SUGIYAMA^{1,c)}

Abstract:

While fuzzing is a powerful software testing technique, its efficiency is often impeded by program-specific constraints (Fuzz Blockers) such as magic bytes and checksums. Prior work has addressed particular Fuzz Blockers, yet even state-of-the-art fuzzers still exhibit coverage plateaus during long-running campaigns. This suggests the persistence of "unresolved Fuzz Blockers" that remain intractable to existing techniques, but systematically identifying them is challenging, especially when confounding environmental causes (e.g., harness deficiencies) are mixed in. We propose a methodology to systematically surface Fuzz Blockers that warrant algorithmic solutions. The key idea is to statistically detect rarely reached regions whose reachability probability is extremely low but nonzero by aggregating outcomes from many independent fuzzing trials. By construction, regions that are unreachable due to environmental issues are filtered out, isolating valuable targets where current algorithms struggle. A preliminary evaluation on real-world programs quantifies the prevalence of these regions and characterizes their properties through case studies. Our findings provide a practical basis for identifying "the next Fuzz Blockers to solve", informing future fuzzer design.

Keywords: Fuzzing, Fuzz Blocker, Rarely Reached Regions, Coverage Plateau

1. はじめに

ファジングは、入力生成・変異と実行を反復することでプログラムの状態空間を自動探索し、脆弱性を発見する強力なソフトウェアテスト手法である [1]。Google の OSS-Fuzz [2] や syzbot [3] に代表される大規模な実運用がその有効性を裏付けており、導入の容易さと高いスケーラビリティから、現代のソフトウェア開発におけるセキュリティ確保の標準技術となっている。

しかし、その探索は **Fuzz Blocker** と呼ばれるプログラム固有の特性によってしばしば妨げられる [4]。典型例として、マジックバイトやチェックサム、フィールド間の整合性といった厳しい入力制約が挙げられる。古典的なファザーはランダムな変異に依存するため、これらの制約を課すプログラム内の分岐を突破することは極めて困難である。

この問題に対し、比較条件の扱いを改善する手法 [5]、入力の構造を活用する手法 [6]、あるいはシンボリック実行を統合する手法 [7] など、多数の先行研究がアルゴリズムの改良によって Fuzz Blocker の克服を試みてきた。それでもなお、**ファジングの探索が著しく阻害される場面は少なくない**。長時間のベンチマーク実行ではカバレッジ増分の頭打ちが報告されており [4]、図 1 が示すように、最先端のファザーですら複数プログラムで同様の停滞に直面する。これは、既存手法では効果が不十分な「**未解決 Fuzz Blocker**」が依然として残存していることを示唆する。

しかし、こうした未解決 Fuzz Blocker を体系的に解析する方法論は未だ確立されていない。例えば、Fuzz Introspector [8] は未到達コードを可視化するが、その「原因」までは区別できない。そのため、アルゴリズムの改善で解消すべき要因と、テスト環境の不備に起因する要因が同列に列挙されてしまう。近年の研究では、よくテストされたライブラリにおいて後者が多数を占めると報告されており [4]、アルゴリズム設計に直結する示唆を得るにはノイズが大きすぎる。結果として、「次に解くべき Fuzz Blocker」の同定自体が困難となっている。

本研究はこの問題に対し、**ファジング・アルゴリズムの改善によって解決すべき Fuzz Blocker のみを特定する新たな方法論を提案する**。本手法では、多数回の独立したファジング試行に基づき、**到達確率が極めて低いがゼロではない「希少到達領域」に着目する**。これらの領域は、少なくとも一度は到達が観測されるという事実から、テスト環境の不備が主因ではなく、まさに入力により制御可能でありながら既存アルゴリズムでは突破が困難な、解析価値の高い Fuzz Blocker の存在を示唆する。本手法は、この到達確

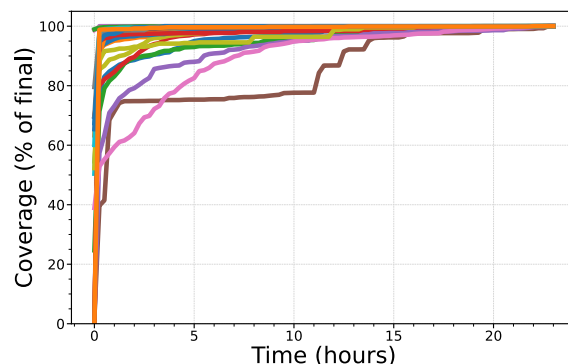


図 1: 典型的なファジングのカバレッジ推移^{*1}。各曲線は、特定のベンチマークに対する結果を示しており、最終到達カバレッジで正規化されている。

率を用いて解析対象を絞り込む。

提案手法の有効性を確認するため、本論文では次の二点に焦点を当てた予備評価を行う。第一に、希少到達領域が実プログラムにどの程度の割合で存在するかを定量化し、本手法が解析対象の絞り込みに有効であるかを検証する。第二に、検出された領域のケーススタディを通じてその特性を分析し、未解決 Fuzz Blocker の体系的な分類に向けた初期の知見を提示する。

本論文の貢献は以下の通りである：

- 多数回の独立試行に基づき希少到達領域を特定し、アルゴリズム改善の対象となる Fuzz Blocker を分析するための新たな方法論を提案する。
- 予備評価により、希少到達領域が実プログラムに一定数存在することを実証し、本手法が解析対象の選別に有効であることを示す。
- ケーススタディを通じて、複数の希少到達領域が単一の根本原因に依存する構造を明らかにし、この知見が将来の効率的な Fuzz Blocker の分析やファジング・アルゴリズムの改善に繋がる可能性を提示する。

2. 背景

2.1 ファジング

ファジングは、自動化されたソフトウェアテスト手法の一つであり、ファザーはそれを実行するプログラムである [1]。ファザーは入力生成・変異と、それをテスト対象プログラム (PUT) に与えて実行するという二つのステップを繰り返す。この過程で、クラッシュやハングアップなどの異常挙動を観測し、バグや脆弱性を発見する。

脆弱性の発見はプログラムが脆弱な**状態**に遷移することで達成される。ここでいう状態とは、メモリやレジスタなど実行時に定まる内部構成要素の値の組であり、その集合を**状態空間**と呼ぶ。プログラムの状態は与えた入力に応じ

¹ 株式会社リチェルカセキュリティ
Ricerca Security, Inc.

a) ryosukei@ricsec.co.jp

b) katok@ricsec.co.jp

c) yuichis@ricsec.co.jp

^{*1} 本図は 2024 年 7 月 23 日時点の FuzzBench [9] レポート <https://www.fuzzbench.com/reports/2024-07-23-aflpp/index.html> における AFL++ [10] のデータを基に作成した。

て遷移するため、外部から与えるあらゆる入力を広く試すことにより、到達できる状態の範囲を押し広げ、結果として脆弱な状態への到達可能性を高める。

しかし、広大な状態空間をランダムに探索するだけでは、プログラムの深部に存在する脆弱な状態へ到達することは極めて困難である。そこで現代のファザーの多くは、探索を効率化するために、プログラムの実行時に得られるフィードバック情報を活用する [1]。最も代表的なフィードバックはコードカバレッジである。この方法では、プログラムの実行を監視し、ある入力がこれまで未到達だったコード領域を通過した場合、その入力は有望なシードとして保存する。そして、保存された入力を起点にさらなる変異を加えることで、ランダムな入力では到達できない、より深いコード領域への探索を可能にする。このようにフィードバックを活用することで、ファザーは状態空間の探索を最適化し、脆弱性の発見確率を高めている。

2.2 ファジングの阻害要因

ファジングはプログラムの状態空間を効率的に探索できる一方で、その探索はしばしば特定のプログラム構造や実行環境に起因する様々な要因によって妨げられ、カバレッジの停滞を引き起こす。本論文では、このような探索を困難にさせるプログラム上の特性や要因を総称して **Fuzz Blocker** と呼ぶ。Fuzz Blocker はその特性から二つに大別され、ファジング・アルゴリズムの改善によって解決される **入力依存** のものと、ハーネスや実行環境といったファジングのセットアップに起因する **入力非依存** のものがある。

2.2.1 入力依存の Fuzz Blocker

入力依存の Fuzz Blocker は、プログラムが特定の入力形式やフィールド間の整合性を要求する箇所に存在する。単純な変異ではそれらの条件を満たすことが困難なため、探索が妨げられる。典型例は次のようなものである：

- **マジックバイト**: ファイルフォーマットの先頭や特定のコードブロックの開始位置に要求される固定のバイト列である。 n バイトの完全一致比較では、ランダムな入力一致する確率は 256^{-n} となる。例えば、`if (input == 0xDEADBEEF) {...}` のような 4 バイトの比較を、完全にランダムな試行で突破の j は稀である。
- **チェックサム**: 入力全体または一部の整合性を検証する値である。例えば、`if (crc32(input.data) == input.crc32) {...}` のように、入力データ (`input.data`) から計算した値と、入力内の別のフィールド (`input.crc32`) を比較する。マジックバイトのような固定値との比較とは異なり、チェックサムの検証では入力の一部を変更すると期待される検証値も変化してしまう。入力内の複数フィールドが連動するため、条件を満たす入力の生成をより一層困難にする。

```
1 int LLVMFuzzerTestOneInput(  
2     const uint8_t *data, size_t len  
3 ) {  
4     ctx_t *ctx = ctx_new();  
5     // 本来は以下のコードで機能を有効化が必要がある  
6     // ctx_enable_unknown_chunks(ctx);  
7  
8     parse_image(ctx, data, len);  
9  
10    ctx_free(ctx);  
11    return 0;  
12 }
```

(a) ハーネス側のコード。変数 `ctx->allow_unknown_chunks` を真に設定する初期化関数の呼び出しが欠落している。

```
1 int parse_image(  
2     ctx_t *len, const uint8_t *buf, size_t len  
3 ) {  
4     if (!ctx->allow_unknown_chunks) {  
5         return OK;  
6     }  
7  
8     // ハーネスで機能を有効化した場合のみ到達可能  
9     if (is_unknown_chunk(buf, len)) {  
10        return handle_unknown_chunk(ctx, buf, len);  
11    }  
12    return OK;  
13 }
```

(b) PUT 側のコード。変数 `ctx->allow_unknown_chunks` が真の場合にのみ、内部の処理が実行される。

図 2: libpng におけるハーネスの不備に起因する入力非依存 Fuzz Blocker の例。(a) のハーネスで初期化関数が呼び出されていないため、(b) の PUT 内の分岐条件が常に偽となり、その先のコード領域が到達不能になっている。

2.2.2 入力非依存の Fuzz Blocker

入力非依存の Fuzz Blocker は、入力データの内容とは無関係に、ファジングのセットアップに起因する阻害要因である。そのため、ファジング・アルゴリズムを改良しても原理的に解決することはできない。

- **ハーネスの不備**: ハーネスとは、ライブラリのようにそれ単体では実行できないプログラムをテスト対象とする際、ファザーと PUT を橋渡しするためのラッパーコードである。ハーネスの記述に、初期化関数の呼び出し忘れや API 設定の欠落といった不備があると、広範囲のコードが到達不能となる。図 2 は、画像処理ライブラリである libpng において、ハーネス側での機能有効化が欠けているために PUT 内の特定の分岐が常にスキップされる例である。具体的には、初期化が行われないことで変数 `ctx->allow_unknown_chunks` が常に偽となり、入力内容に関わらず分岐が早期に終了してしまうことで、ファザーはこれより奥のコードを探索できなくなってしまう。この場合、ハーネス側の関数に初期化呼び出し `ctx_enable_unknown_chunks(ctx)` を追加することで、当該コードパスは到達可能となる。
- **実行環境の不備**: テスト対象の実行に必要な外部ファイル、設定、権限、あるいはネットワーク資源が不足

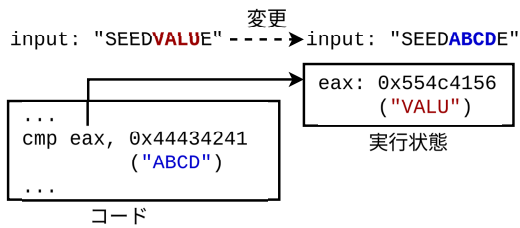


図 3: REDQUEEN のアプローチ概念図。実行時の比較命令を監視し、期待される値を抽出して新しい入力を生成する。

している場合も、関連するコードパスは実行されない。例えば、ファジング実行環境では通常、設定ファイルは常に正しいパスに配置され、適切な読み取り権限が与えられている。そのため、ファイルが存在しない、あるいは権限がないといったエラーを処理するためのコードパス（エラーハンドリングやフォールバック処理など）は、ファザーがどんな入力を生成しても決して実行されず、恒常的に未到達となる。

2.3 ファジングの性能改善とその限界

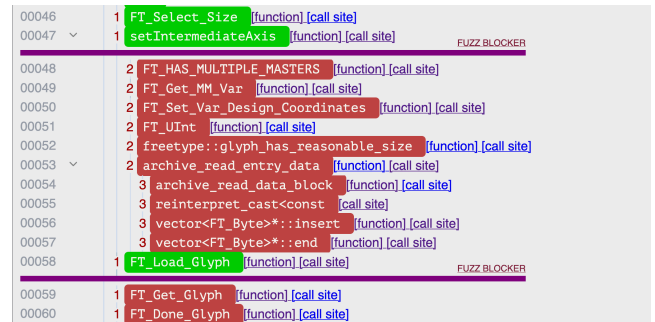
Fuzz Blocker を克服するためのアプローチは、その性質によって大別される。入力非依存の Fuzz Blocker は、主にハーネスの修正や実行環境の整備といったセットアップ面の改善で解消される。こうした手作業のコストを抑えるため、セットアップの自動化を目指す研究もある（§6）。

一方、入力依存の Fuzz Blocker に対しては、その特性に応じた専用の解決戦略が数多く提案されてきた [5], [6], [7], [11]。例えば REDQUEEN [5] は、マジックバイトやチェックサムに起因する分岐を、動的テイント解析を用いずに低オーバーヘッドで突破する手法である。図 3 に示すように、REDQUEEN は実行中の比較命令（`cmp` など）のオペランドを監視し、期待される値（例：ABCD）を抽出する。次に、その比較に寄与した入力バイト範囲を特定し、該当部分を抽出した値で書き換えた新しい入力（例：SEEDABCDE）を生成する。これにより、ランダムな変異と比較して、効率的に制約の厳しい比較条件を突破できる。

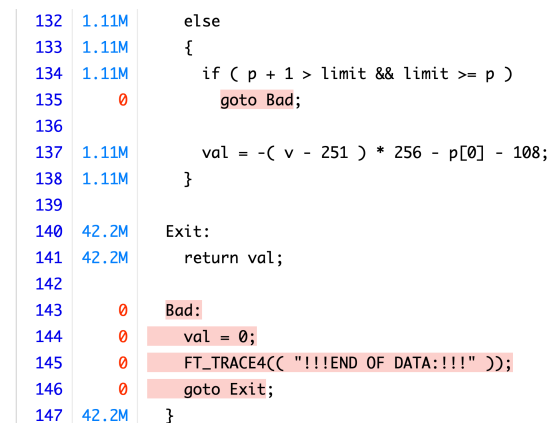
しかし、既に全ての Fuzz Blocker を解決できているわけではない。多くの標準的なベンチマークにおいて、最先端のファザーでさえも長時間実行すると、新たなコードカバレッジの増加が鈍化してしまう [4]。この事実は、既存の高度な手法では突破が困難な Fuzz Blocker が、依然として数多く残存している可能性を示唆している。

3. モチベーション

§ 2.3 で述べたように、最先端のファザーであっても長時間の実行でカバレッジの向上が停滞する問題が存在する。この性能停滞は、その特性が未解明な Fuzz Blocker が残存していることに起因すると考えられるため、カバ



(a) コールツリーレベルの可視化例。赤色でハイライトされている関数呼び出しは、実行されなかったことを示す。



(b) ソースコードレベルのカバレッジ可視化例。実行回数が 0 回である未到達のコードブロックが赤くハイライトされる。

図 4: Fuzz Introspector による可視化レポートの例。

レッジをさらに向上させるには、探索を阻害している Fuzz Blocker を正確に抽出し、その特性を理解することが不可欠である。本章では、そのための既存アプローチである Fuzz Introspector [8] を紹介し、なぜそのアプローチだけでは不十分なのか、その課題を明らかにする。

3.1 Fuzz Introspector による未到達領域の可視化

Fuzz Introspector は、カバレッジ向上の阻害要因を特定するために、未到達の関数やコード領域を列挙・可視化するフレームワークである [8]。図 4 に示すように、ファジング中に到達できなかった領域をレポートとして提示し、どの分岐や関数がボトルネックとなっているかを視覚的に把握できる。例えば、図 4a のコールツリー表示では、実行された関数呼び出し（緑色）と実行されなかった関数呼び出し（赤色）が明示され、探索がどこで停滞しているかを大局的に理解できる。また、図 4b のカバレッジ表示では、ソースコード上で未到達の行（赤色）と各行の実行回数が見られ、より詳細な分析が可能となる。

Fuzz Introspector の利点は、静的に到達可能と見なせる範囲と動的に実際に到達した範囲を突き合わせ、「到達できるはずなのに到達していない」箇所を機械的に抽出

できる点にある。その手順は次の通りである。

- (1) コールグラフや制御フローグラフの情報から、到達可能なコード領域の集合 S を推定する。^{*2}
- (2) 実際のファジング実行の結果から、到達済み領域の集合 C を収集する。
- (3) 差集合 $U = S \setminus C$ を計算し、 U を「到達すべきなのに未到達」の候補として抽出する。

このアプローチにより、ビルドオプション等で無効化され、そもそも到達不能な関数といったノイズは、 S の構築段階で予め排除される。その結果、解析者は真に注目すべき未到達領域の候補 U の分析に集中できる。

3.2 未解決 Fuzz Blocker の特定の課題

Fuzz Introspector は未到達領域を効率的に抽出する有用な基盤であるが、その出力からファジング・アルゴリズムの改善対象となる入力依存の Fuzz Blocker を特定するには、二つの大きな課題が存在する。

第一の課題は、Fuzz Introspector が報告する未到達領域の候補が、膨大かつ異種の要因が混在したリストである点にある。ファジングのカバレッジが 100% に達成することは稀であるため、未到達領域の集合 K は肥大化することが多い。実際に Fuzz Introspector のレポートにおいて、libpng では 220 個^{*3}、libxml2 では 824 個^{*4}もの未到達関数が報告されている。しかし、この膨大な候補群には、**入力依存**（アルゴリズムの改善対象）と、**入力非依存の Fuzz Blocker**（アルゴリズム改善の観点ではノイズ）という、性質の全く異なる二つの要因が混在している。本研究の目的であるアルゴリズム改善に繋がる前者だけを見極めることは、このノイズの存在によって一層困難になる。

第二の課題は、この問題を解決するために提案されている**原因の自動分類における不確実性**である。Fuzz Introspector を補うため、テイント解析を用いて未到達の原因が入力依存かを自動分類する手法も提案されている [4]。しかし、このアプローチはテイント解析が本質的に抱える精度上の問題により、分類の信頼性が損なわれる。

このように、未到達領域を網羅的に列挙するアプローチ (Fuzz Introspector) と、その原因を自動分類するアプローチ (テイント解析) は、アルゴリズムの改善対象となる Fuzz Blocker を効率的に特定するには不十分である。この現状が、次章で述べる我々の方法論の動機となっている。

4. 方法論

§ 3.2 で述べたように、Fuzz Introspector は未到達領域を網羅的に列挙できる一方で、その原因を自動で区別できない。その結果、出力には主に (a) **入力依存の Fuzz Blocker**（ファジング・アルゴリズムの改善により克服可能なもの）と、(b) **入力非依存の Fuzz Blocker**（アルゴリズムの改善では原理的に解決不可能なもの）が混在する。

§ 2.3 で論じたように、近年のファジングではカバレッジの停滞が問題となっている。本研究は、この停滞をアルゴリズムの改善によって解消することに関心があるため、(a) 入力依存の Fuzz Blocker の分析に焦点を当てる。これらの Fuzz Blocker はアルゴリズム自体の性能限界を示すものであり、その特性を理解し克服することが、ファジング技術をさらに発展させる上で不可欠である。しかし、Fuzz Introspector の出力には (b) がノイズとして大量に混在するため、(a) の特定自体が困難となっている。そこで本研究は、解析対象を (a) へと絞り込むため、**未到達の有無ではなく到達頻度の低さ**に基づいて候補を選別する。

4.1 希少到達領域の定義

本研究は Fuzz Introspector が主に対象とする「一度も到達しない領域」ではなく、**到達確率はゼロではないが極めて低い領域**に着目する。PUT のベーシックブロックの集合を B とし、同一条件下で独立な N 回のファジング試行を行ったとき、各コード領域 $b \in B$ と各試行 $i \in \{1, \dots, N\}$ について

$$X_i(b) := \begin{cases} 1 & b \text{ に少なくとも一度到達したとき,} \\ 0 & \text{それ以外.} \end{cases}$$

と定義する。 N 回の試行中に b に到達した試行回数は

$$K(b) := \sum_{i=1}^N X_i(b)$$

であり、コード領域 b への経験的到達率は

$$\hat{p}(b) := \frac{K(b)}{N} \quad (0 \leq \hat{p}(b) \leq 1)$$

と定義する。

閾値 $\theta \in (0, 1]$ を固定し、

$$\mathcal{H}_\theta := \{b \mid 0 < \hat{p}(b) \leq \theta\}$$

を**希少到達領域**（到達困難なコード領域）の集合と定める。ここで条件から $\hat{p}(b) = 0$ のケースを除外している点が重要である。なぜなら、一度も到達しなかった領域には、本研究の関心対象である入力依存の Fuzz Blocker だけでなく、環境やハーネスの不備に起因する原理的に到達不能な領域がノイズとして混入するためである。到達率がゼロでは

^{*2} 間接呼び出しや関数ポインタを介した呼び出しの静的解析は本質的に困難であるため、この集合 S は完全ではなく、本来到達可能な領域を過小評価する可能性がある。

^{*3} https://storage.googleapis.com/oss-fuzz-introspector/libpng/inspector-report/20250819/fuzz_report.html

^{*4} https://storage.googleapis.com/oss-fuzz-introspector/libxml2/inspector-report/20250819/fuzz_report.html

ない領域に限定することで、このようなノイズを設計上排除し、解析対象をアルゴリズムの改善によって解決すべき Fuzz Blocker へと絞り込むことができる。これにより、未到達領域に含まれる大量のノイズを回避しつつ、解析すべき候補を現実的な規模にまで縮約することが可能となる。

4.2 解析方法

Fuzz Blocker の特定は、以下の手順で行われる：

- (1) **多数回の独立したファジング試行の実施：**各 PUT に対し、同一の設定で N 個のファザーインスタンスを独立に実行する。これにより、各コード領域への到達が確率的な事象として観測可能になる。
- (2) **カバレッジデータの収集と集計：**各インスタンスの実行が終了した後、それぞれのカバレッジ情報を収集する。そして、全てのインスタンスの結果を統合し、プログラム内の各コード領域（本研究では基本ブロックを単位とする）が、全 N インスタンス中、何回到達されたかを記録した到達頻度マップを作成する。
- (3) **希少到達領域の特定：**そのマップから、到達率が閾値 θ 以下の領域を「希少到達領域」として抽出し、それらを Fuzz Blocker の候補リストとして出力する。

本研究では、カバレッジデータの収集基盤として Fuzz Introspector の計装機能を利用し、上記の集計および特定処理を行うためのカスタムスクリプトを実装した。

5. 予備評価

5.1 研究課題

本論文では提案手法の有効性を検証するため、以下の 2 つの研究課題 (RQ) を設定した。

- **RQ1: 希少到達領域 (到達困難なコード領域) はどの程度の割合で存在するのか？** 本手法で定義する希少到達領域が、プログラム全体の中でどの程度の割合で存在するのかを定量的に評価し、本手法が解析対象の絞り込みや優先度付けに有効であるかを検証する。
- **RQ2: 希少到達領域はどのような特性を持つか？** 希少到達領域として検出されたコードの構造的・意味的な特徴を、ケーススタディを通じて分析する。

5.2 評価設定

本評価では、テスト対象プログラム (PUT) として、OSS-Fuzz に統合されているものの中から、代表的なプログラムである freetype と libxml2 を選択した。再現性を確保するため、初期シードは空とし、ビルド方法は OSS-Fuzz で提供されているデフォルト設定をそのまま利用した。

ファザーには、広く利用されている libFuzzer [12] を採用した。ファジングはその確率的な性質上、少数の試行結果では性能が大きく変動することが知られている [13]。このランダム性の影響を緩和するため、本評価では各 PUT

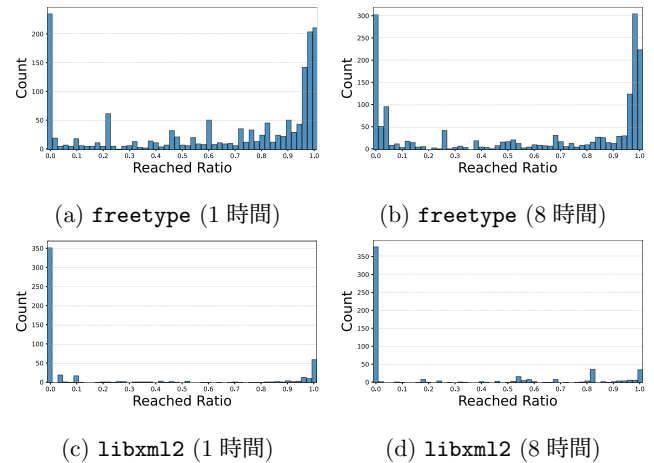


図 5: freetype と libxml2 におけるコード領域の到達率分布。X 軸は到達率 (128 インスタンス中、当該領域に到達した割合)、Y 軸は該当するコード領域の数を示す。左端 (到達率 0.0) は一度も実行されなかった領域の数を表す。視認性のため、到達率が 1.0 は図示していない。

に対して 128 個のインスタンスを独立に実行し、その多数の試行結果から希少到達領域を特定した。

本評価では、§ 4.1 で定義した希少到達領域を特定するための閾値として $\theta = 1/N$ を用いる。インスタンス数は 128 であるため、これは 128 回の試行のうちちょうど 1 回だけ到達した領域 ($K(b) = 1$) が希少到達領域として抽出されることを意味する。なお、閾値は経験的比率に固定する必要はなく、二項分布の信頼区間 (例: Clopper-Pearson 法) の片側上限に基づいて統計的に設定することも可能であるが、本論文では簡潔さのため $\theta = 1/N$ を採用する。

5.3 RQ1: 希少到達領域の割合

図 5 は、freetype と libxml2 における各コード領域の到達率分布を、それぞれ 1 時間および 8 時間実行時点で示している。両 PUT とも、到達率 0.0 に集中しやすく、同時に 1.0 近傍にも多くの領域が分布する傾向が見られる。

本手法で定義した希少到達領域 (到達率 $\theta = 1/128$) に注目すると、1 時間の実行では、freetype で 6 個 (全領域の 2.0%)、libxml2 で 9 個 (同 12.2%) が該当した。8 時間実行時点では、freetype で 25 個 (6.7%)、libxml2 で 1 個 (2.7%) であった。この結果は、Fuzz Introspector 等が報告する膨大な未到達領域の中から、**本手法を用いることで人手で十分に解析可能な規模まで解析対象を絞り込めることを示唆している**。また、実行時間を 1 時間から 8 時間に延長すると、到達率 0.0 の領域が減少し (未到達だった領域の一部が少なくとも一度は到達される)、同時に 1.0 近傍の領域が増加し、全体として右方向ヘシフトする。

5.4 RQ2: 希少到達領域の特性

本節では、希少到達領域の具体的な性質を明らかにす

```

1  FT_Error cff_parser_run(
2      CFF_Parser parser,
3      int* p, // ユーザーが制御可能な変数
4      int* limit
5  ) {
6      ...
7      while(p < limit) {
8          int code = *p++;
9          ...
10         switch(code) {
11             ...
12             case CODE_VSTORE:
13                 // 希少到達領域 (1)
14                 varRegionList =
15                     malloc(sizeof(varRegionList));
16                 cffface->family_name =
17                     parse_family(&p);
18                 ...
19             }
20         }
21     }
22
23     void cff_vstore_done() {
24         ...
25         if (varRegionList) {
26             // 希少到達領域 (2)
27         }
28         ...
29     }
30
31     FT_ERROR cff_face_init() {
32         ...
33         if (cffface->family_name) {
34             // 希少到達領域 (3)
35         }
36         ...
37     }

```

図 6: 検出した freetype における Fuzz Blocker の一例.

るため、8 時間の実行における freetype の結果を対象にケーススタディを行った。分析対象として、8 時間実行後のデータから抽出された 25 個の希少到達領域を調査した。

まず、これらの領域のコードを分析したところ、その一部が典型的な Fuzz Blocker である**定数比較**に起因していることが判明した。図 6 に示す cff_parser_run 内の switch (code) {...} はその一例である。この条件分岐に入るためには、ユーザー制御のポインタ p から読み出された code が CODE_VSTORE と一致する必要があるが、これは § 2.2.1 で述べたマジックバイトの比較そのものである。

次に、25 個の希少到達領域間の関係を調査したところ、**単一の根本的な Fuzz Blocker に依存する希少到達領域群が存在する**という、もう一つの重要な特性が明らかになった。図 6 の例では、switch (code) {...} が上流の Fuzz Blocker として機能する。希少到達領域 (1) を突破できたインスタンスでは、switch 内で varRegionList の確保と cffface->family_name の設定が行われ、続いて cff_vstore_done 内の if (varRegionList)、cff_face_init 内の if (cffface->family_name) が真となり、希少到達領域 (2) と (3) が実行される。実際、これ

らは同一のインスタンスにおいて同時に到達しており、上流の CODE_VSTORE 比較の成否に強く依存することを示す。

この結果は、複数の Fuzz Blocker が検出されたとしても、それらの根本原因がより上流にある単一の Fuzz Blocker へ集約されうること示唆する。したがって、Fuzz Blocker を個別に列挙するだけでなく、到達依存関係を解析して根本原因単位でグループ分けすることにより、さらに人手解析のコストを大幅に削減できる可能性がある。

6. 関連研究

6.1 ハーネスの改善

ファジングのセットアップ不備に起因する入力非依存の Fuzz Blocker も大きな課題となっている。その代表例が、テスト対象を適切に呼び出すためのハーネスの記述に起因するものである。ハーネスの作成は属人的な作業に依存することが多く、初期化関数の呼び出し漏れなどによって、広範囲のコードが意図せず到達不能になる場合がある。

この課題に対し、近年ではハーネスの自動生成に関する研究が行われている [14], [15]。これらの研究は、プログラムのソースコードやバイナリを静的・動的に解析し、ライブラリ関数を呼び出すための適切なハーネスを自動生成することで、ハーネス作成に伴う人為的な誤りを削減する。

本研究もファジングの阻害要因を扱う点で共通しているが、アプローチの対象が異なる。これらの研究がハーネスというファジング環境の「設定」に起因する入力非依存の Fuzz Blocker に焦点を当てているのに対し、本研究はファザーの「探索能力」の限界に起因する入力依存の Fuzz Blocker に焦点を当てている。

6.2 静的解析による Fuzz Blocker の予測

静的解析を用いて到達困難なコード領域を特定し、ファジングの探索を効率化する手法がある [16], [17]。これらは、ファジングを実行する前に、ソースコードやバイナリの静的解析から Fuzz Blocker となりうる箇所を予測し、探索を誘導することを目的とする。例えば、VUzzer [16] は静的解析を用いて各基本ブロックの重要度を計算し、到達が困難と予測される箇所を優先的に探索する。

このような手法は、実際にファジングを行うことなく阻害要因を推測できる利点がある。しかし、静的解析による予測は必ずしも正確ではなく、実際には問題なく通過できる箇所を誤って到達困難と判定したり、逆に真の Fuzz Blocker を見逃したりする可能性がある。これに対し、本研究のアプローチは根本的に異なる。我々は、多数回のファジング試行という動的な実行結果に基づいて、経験的に到達が困難であった領域を特定する。これにより、静的解析の不確実性を排除し、より信頼性の高い Fuzz Blocker を抽出することを目指している。

7. 今後の課題と展望

本研究は、提案手法の有効性を示すための予備評価に留まっている。今後の重要な課題として、まず**大規模評価の実施**が挙げられる。本評価では限られた数のプログラムを対象としたが、提案手法の一般性を確立するためにはより多様なプログラムを対象とした評価が不可欠である。

次に、本研究のケーススタディ (§ 5.4) で得られた知見を発展させ、**希少到達領域の依存関係解析**が重要な課題となる。予備評価では、複数の希少到達領域が単一の根本的な Fuzz Blocker に依存している可能性が示唆された。この知見に基づき、今後は希少到達領域間の依存関係を静的または動的に解析し、根本原因ごとに自動でグループ化する手法を開発する。これにより、人手による解析対象を個々の領域から根本原因となる Fuzz Blocker に絞り込むことができ、解析コストの大幅な削減が期待できる。

最後に、これらの解析結果を**ファジング・アルゴリズムへ応用**することも重要な展望である。例えば、依存関係の根本原因となる Fuzz Blocker を特定できれば、その領域の探索を優先するようにスケジューリングを最適化できる。このように、本研究は、未解決 Fuzz Blocker の特性を明らかにするだけでなく、それを克服するための次世代ファザーの開発に直接繋がる知見を提供する。

8. おわりに

ファジングにおけるカバレッジ停滞の要因である「未解決 Fuzz Blocker」の特定は、アルゴリズム改善に繋がらないノイズとの区別が困難であった。本論文は、この課題に対し、多数回の独立したファジング試行から得られる経験的な到達確率に基づき、「希少到達領域」を特定する新たな方法論を提案した。本手法は、環境要因による到達不能な領域を設計上排除することで、解析価値の高い入力依存の Fuzz Blocker へと対象を効果的に絞り込む。予備評価では、希少到達領域が実際のプログラムに一定数存在することを確認し、本手法が解析対象の絞り込みに有効であることを実証した。さらに、検出された領域のケーススタディは、根本原因の特定による効率的な分析や、将来のファジング・アルゴリズム改善に繋がる重要な知見をもたらした。

謝辞 本研究の一部は国立研究開発法人新エネルギー・産業技術総合開発機構 (NEDO) の委託事業「経済安全保障重要技術育成プログラム／先進的サイバー防御機能・分析能力強化」(JPNP24003) によるものである。

参考文献

[1] Manès, V. J. M., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J. and Woo, M.: The Art, Science, and Engineering of Fuzzing: A Survey, *IEEE Transactions on Software Engineering*, Vol. 47, No. 11, pp. 2312–2331

(2021).
[2] Google: OSS-Fuzz, <https://google.github.io/oss-fuzz>.
[3] Google: syzbot, <https://syzkaller.appspot.com/upstream>.
[4] Gao, W., Pham, V.-T., Liu, D., Chang, O., Murray, T. and Rubinstein, B. I.: Beyond the Coverage Plateau: A Comprehensive Study of Fuzz Blockers (Registered Report), *Proceedings of the International Fuzzing Workshop (FUZZING)*, p. 47–55 (2023).
[5] Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R. and Holz, T.: REDQUEEN: Fuzzing with Input-to-State Correspondence, *The Network and Distributed System Security Symposium (NDSS)*, pp. 1–15 (2019).
[6] Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.-R. and Teuchert, D.: NAUTILUS: Fishing for deep bugs with grammars., *The Network and Distributed System Security Symposium (NDSS)*, p. 337 (2019).
[7] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C. and Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution., *The Network and Distributed System Security Symposium (NDSS)*, pp. 1–16 (2016).
[8] Google: Fuzz Introspector, <https://introspector.oss-fuzz.com>.
[9] Metzman, J., Szekeres, L., Maurice Romain Simon, L., Trevelin Sprabery, R. and Arya, A.: FuzzBench: An Open Fuzzer Benchmarking Platform and Service, *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, p. 1393–1403 (2021).
[10] Fioraldi, A., Maier, D., Eißfeldt, H. and Heuse, M.: AFL++ : Combining Incremental Steps of Fuzzing Research, *USENIX Workshop on Offensive Technologies (WOOT)* (2020).
[11] Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.-H., Song, Y. and Beyah, R.: MOPT: Optimized mutation scheduling for fuzzers, *USENIX Security Symposium*, pp. 1949–1966 (2019).
[12] Serebryany, K.: libFuzzer—a library for coverage-guided fuzz testing, *LLVM project*, p. 34 (2015).
[13] Klees, G., Ruef, A., Cooper, B., Wei, S. and Hicks, M.: Evaluating fuzz testing, *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 2123–2138 (2018).
[14] Babić, D., Bucur, S., Chen, Y., Ivančić, F., King, T., Kusano, M., Lemieux, C., Szekeres, L. and Wang, W.: FUDGE: fuzz driver generation at scale, *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, p. 975–985 (2019).
[15] Ispoglou, K., Austin, D., Mohan, V. and Payer, M.: FuzzGen: Automatic fuzzer generation, *USENIX Security Symposium*, pp. 2271–2287 (2020).
[16] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C. and Bos, H.: Vuzzer: Application-aware evolutionary fuzzing, *The Network and Distributed System Security Symposium (NDSS)*, pp. 1–14 (2017).
[17] Böhme, M., Pham, V. and Roychoudhury, A.: Coverage-based Greybox Fuzzing as Markov Chain, *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 1032–1043 (2016).