ネットワークプロトコル実装における状態を考慮したテストケース最小化手法

西坂 龍太郎[1,a)]　加藤 和志[1,b)]　杉山 優一[1,c)]

概要：
ファジング技術の発展により，ソフトウェアのクラッシュを誘発するテストケースを自動かつ大量に生成可能になった．一方，膨大なクラッシュの中から真に有用なものを選別するトリアージ工程が，実運用上のボトルネックとなっている．特に，テストケースからクラッシュ再現に不要な要素を除去する最小化は，後続の解析作業の効率と精度を左右する重要な工程である．しかし，ネットワークプロトコル実装では既存の最小化手法は有効に機能しない．プロトコル実装における多くのクラッシュは特定のメッセージ列による状態遷移に依存するため，単一メッセージを前提とする既存手法では再現に必須な状態遷移まで破壊してしまう．そこで本研究では，プロトコルの状態遷移に着目した二段階テストケース最小化手法を提案する．本手法は，第一段階で応答コードを基に構築した状態遷移グラフを用いて不要なメッセージを削除し，第二段階で残存メッセージから不要なバイトを削減する．これにより，クラッシュの再現性を担保しつつ，トリアージの妨げとなる不要な状態遷移とコードパスを効果的に排除する．複数のプロトコル実装による評価の結果，提案手法はクラッシュ再現性を維持したままテストケースを削減することに成功した．また，状態遷移グラフを用いた手法では，状態遷移を必要とする複雑なテストケースに対して最小化に要する時間が，平均 36%短縮された．

キーワード：テストケース最小化，ファジング，ネットワークプロトコル，状態遷移グラフ，トリアージ

State-Aware Test Case Minimization for Network Protocol Implementations

Ryutaro Nishizaka[1,a)]　Kazushi Kato[1,b)]　Yuichi Sugiyama[1,c)]

Abstract:
The advancement of fuzzing techniques has enabled the automatic generation of numerous test cases that trigger software crashes. However, triaging these crashes has become a significant bottleneck. Test case minimization, the process of removing unnecessary elements, is crucial for the efficiency and accuracy of subsequent analysis. Unfortunately, existing minimization methods are often ineffective for network protocol implementations. Most crashes in these implementations depend on specific state transitions caused by a sequence of messages. Existing methods, which typically assume a single message as input, can destroy the essential state transitions required to reproduce the crash. To address this challenge, we propose a state-aware two-stage test case minimization method. The first stage leverages a state transition graph, constructed from response codes, to remove entire messages that are not essential for reproducing the crash. The second stage then performs byte-level reduction on the remaining messages. This approach effectively preserves crash reproducibility while eliminating irrelevant state transitions and code paths that complicate the triage process. Our evaluation on multiple protocol implementations demonstrates that the proposed method reduces test case size without compromising crash reproducibility. In addition, leveraging the state transition graph reduced the average minimization time to 0.64 times for state-dependent test cases.

Keywords: Test Case Minimization, Fuzzing, Network Protocol, State Transition Graph, Triage

1. はじめに

ソフトウェアの脆弱性検出において，ファジングは近年最も有効な自動化手法の一つである．特に AFL [1] に代表されるカバレッジ指向型ファジングは，コードカバレッジを活用して効果的なテストケースを生成することで，多数のクラッシュを短時間で発見できる．この手法は産業界での活用も進んでおり，Google の OSS-Fuzz は 2025 年 3 月時点で 1,000 以上のオープンソース・ソフトウェアを対象にファジングを実施し，13,000 件を超える脆弱性と 50,000 件を超えるバグの発見・修正に寄与している [2], [3].

一方で，ファジングは本質的に**クラッシュを発見するテストケースを生成するための技術**に留まり，その後の**トリアージ**は依然として人手に頼る場面が多く，実運用上のボトルネックとなっている．トリアージとは，クラッシュを引き起こすテストケースを対象に「最小化，重複除去，優先順位付け」などの分析を行う一連の工程を指す [4]．ファジングによって大量のクラッシュが発見されるほど，それに対するトリアージ処理の負荷は増大するため，その工程の自動化と効率化が重要な課題となっている．

トリアージの初期段階で中核となるのが**テストケース最小化**である．この工程は，クラッシュの再現に不要なノイズを除去して原因となる最小限のテスト入力を抽出するものであり，後段のトリアージ工程に直接的な効果をもたらす．代表的なアルゴリズムとして **Delta Debugging** [5] が知られている．それは，(i) テスト入力の部分的削除と，(ii) クラッシュの再現性を確認，という二つの試行を繰り返すことで，バグの再現に必要不可欠な最小限の入力へと体系的に削減する．このアプローチは，AFL が提供する `afl-tmin` などのツールにも採用され，様々なプログラムに対するトリアージに広く利用されてきた．

しかし，ネットワークプロトコル実装のような内部状態を持つソフトウェアのバグに対しては，Delta Debugging に代表される既存の最小化手法は有効に機能しない．プロトコル実装における多くのクラッシュは，特定のメッセージ列による状態遷移に依存する．例えば，RTSP 実装である `Live555` の脆弱性である CVE-2019-7314 [6] は，`SETUP` リクエストによってセッションを確立した後，特定の `PLAY` リクエストを含むメッセージ列を送信することで初めて顕在化する．そのため，Delta Debugging のステップ (i) において，テスト入力内の一部を盲目的に削除すると，クラッシュの再現に必須な状態遷移を容易に破壊してしまう．

本研究ではこのような課題に対処するため，**状態遷移を利用する二段階テストケース最小化手法**である「`afl-stmin`」を提案する．本手法は，ネットワークプロトコル実装の応答コードが，テスト対象の状態と相関を持つという知見に基づく [7]．図 1 に示すように，提案手法は，まず第一段階として，テスト対象の応答から状態遷移グラフを構築し，クラッシュに至るパスに寄与しないメッセージを優先的に削除する．続く第二段階では，残存したメッセージ列内の各ペイロードに対し，バイト単位の Delta Debugging を適用することで，不要なデータを削減する．

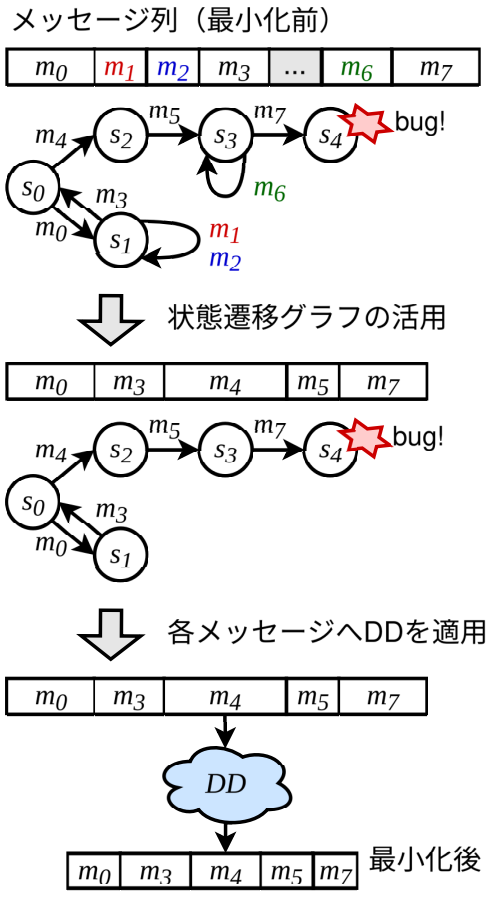


図 1: `afl-stmin` の全体像.

本研究の主な貢献は以下の通りである:

- ネットワークプロトコル実装に対し，状態遷移グラフを利用して最小化の効率を高める新たなテストケース最小化手法 `afl-stmin` を提案した．
- メッセージレベルでの優先順位付けと，バイトレベルでの削減を組み合わせた二段階の最小化アルゴリズムを設計・実装した．
- 既知の脆弱性を含む 2 つのプロトコル実装のバグを対象とした評価実験を通じて，提案手法が削減率と実行時間の観点からベースライン手法と比較してどのような特性を持つかを明らかにした．

2. 背景

本研究は，ネットワークプロトコル実装のバグを引き起こすテストケースに対して入力最小化の新たなアプローチ

[1] 株式会社リチェルカセキュリティ
Ricerca Security, Inc.
[a)] ryutaron@ricsec.co.jp
[b)] kazushik@ricsec.co.jp
[c)] yuichis@ricsec.co.jp

を提案する．その背景として本節では，プロトコル実装とそのバグについての特徴を § 2.1 で説明する．続く § 2.2 では，ファジングによって発見されたクラッシュの悪用可能性を評価するトリアージについて概説し，その主要な段階である入力最小化と重複するテストケースの除去，悪用可能性評価について説明する．最後に，入力最小化の代表的な手法として知られる Delta Debugging について § 2.3 で説明する．

2.1 プロトコル実装とそのバグの特徴

プロトコル実装は，クライアントとサーバー間の通信手続きを処理することを目的として設計されている．FTP, RTSP, DTLS などのプロトコルは，単一のリクエスト／レスポンスではなく，複数のメッセージのやり取りを通じてセッションを確立し，状態遷移を伴いながら動作する．この性質から，プロトコル実装の多くはステートフルであり，入力メッセージが到着する順序やタイミング，サーバー内部の状態によって，同じメッセージであっても異なる動作，応答を示す．

このステートフル性により，プロトコル実装における多くのバグは，内部状態と密接に結び付いて発生するという特徴をもつ．単一のリクエストでバグが顕在化するケースは限定的であり，多くの場合はサーバーの内部状態を特定の状態に遷移させた上で，特定のメッセージを送信するというメッセージ列としての条件が揃うことで初めて脆弱性が露呈する．

このように，バグの発生条件は内部状態に強く依存しており，単純な入力変異型のファジングでは十分に探索できない場合が多い．そのため，プロトコルファジングにおいては，単一の入力単位ではなく，メッセージ列全体とそれがもたらす状態遷移の探索を考慮したアプローチがとられている．代表的なプロトコルファザーである AFLNet [7] のような手法が応答コードを状態オラクルとして活用し，状態カバレッジを指標に入力列を生成しているのは，この依存性を考慮した結果である．

2.2 トリアージ

ファジングは，ヌルポインタ参照や Use-after-Free など，さまざまな種類のクラッシュを発見できる．しかし，クラッシュ時の入力にはファザーが生成する多量のノイズが含まれており，根本原因解析の障壁となっている．

このような困難を軽減するために行われる工程がトリアージである．トリアージでは，ファザーによって発見されたクラッシュ入力を整理・簡約し，最終的に悪用可能性の評価や根本原因の調査を効率的に進められるようにする．図 2 にその工程を示す．また，各工程の詳細は次の通りである．

- **入力最小化**：入力最小化はクラッシュの再現性を保ちながら，入力に含まれるクラッシュに関係ないノイズ部分を除去する工程である．ファジングで生成された入力はランダム性が高く，クラッシュの再現に不要な部分を含むことから，不要なデータを削除して最小限の入力に簡約する．これによって入力から本質的な部分だけを抽出し，バグの解析が容易になる．入力最小化の代表的な手法として Delta Debugging が知られており [5]，続く § 2.3 節で詳細を説明する．
- **重複するテストケースの除去**：この工程は，同一原因から生じたクラッシュを同一グループに分類し，重複するテストケースを排除する工程である．クラッシュの根本原因が同一であっても，入力の内容や実行時のスタックトレースはしばしば異なることから，単純にクラッシュ件数を数えるだけでは，実際のバグの数を正確に把握することは難しい．このようなクラッシュを分類し，ユニークなクラッシュのみを残すことで，解析対象が削減されて本質的なケースに対処することが可能となる．
- **悪用可能性評価**：これらの工程を経て，解析すべきクラッシュのサイズと総数を小さくなった段階で悪用可能性を評価する．この工程では発生したクラッシュによるメモリ破損がどのようなタイプかを解析，その影響を推定する．特に，読み書きが発生するアドレスとその内容をユーザーが制御可能な場合や，関数ポインタのような実行アドレスを制御できるような場合は深刻なバグであると判断される．

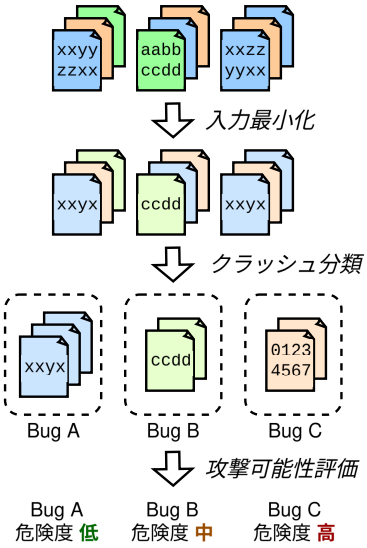


図 2: トリアージにおける一連の工程の概略図．

図 2 に示したように，入力最小化は最初に行われる工程であり，その成否が後続の重複除去や悪用可能性評価の効率に直接影響する．すなわち入力最小化が不十分であると，同一原因のクラッシュ分類や悪用可能性評価を適切に行うことが困難となる．そのため，トリアージ全体の基盤として入力最小化が重要な役割を担っている．

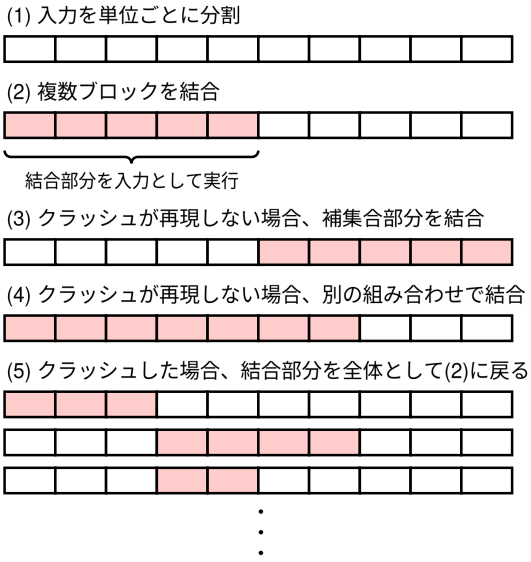


図 3: Delta Debugging のアルゴリズムの概略図

2.3 Delta Debugging に基づくテストケース最小化

Delta Debugging は，入力最小化の代表的な手法であり [5]，ソフトウェアテストやバグ解析の分野で広く用いられている．提案時点ではプログラムの失敗原因を特定するための一般的な手法として設計され，失敗を引き起こす入力やコード断片を効率的に絞り込むことを目的としていた．その性質から入力最小化アルゴリズムとしても採用されており，例えば，オープンソースのファザーである AFL では，Delta Debugging を取り入れたテストケース最小化ツールとして，`afl-tmin` を提供している．

Delta Debugging の基本的な考え方は，与えられた入力をブロック単位で分割し，各部分を削除または保持した形でプログラムを実行し，クラッシュが再現されるかを判定することである．アルゴリズムは分割統治に基づいて設計されており，初期では入力を大きく分割し，クラッシュを再現できる部分的な入力が存在すればそれを残してさらに細かく分割を繰り返す．逆に，部分的な入力でクラッシュが再現されない場合は，分割粒度を調整して探索範囲を徐々に絞っていく．この過程を繰り返すことで，クラッシュに本質的に寄与する最小限の入力が抽出される．図 3 にアルゴリズムの概略図を示す．

Delta Debugging に対する改善は広く試みられており，最小化する対象の構造や意味を考慮する手法が提案されている．構文木を利用した階層的削減 (Hierarchical Delta Debugging, HDD) [8] は，入力を構文的に意味のある単位に分割し，整合性を保ちながら不要部分を削除する．これを発展させた C-Reduce [9] は，C 言語のソースコードを対象に多数の文法ルールを適用して最小化を実現しており，コンパイラ開発現場で実用的に利用されている．さらに Perses [10] は文法ベースの削減と Delta Debugging を統合することで，汎用的かつ効率的な最小化を実現した．

近年では，Delta Debugging の効率や探索戦略を改善する研究も行われている．ProbDD [11] は，分割単位の要素がバグを引き起こす確率をこれまでのテスト結果から判断するという確率モデルを用いて削減効率を高める手法を提案した．また，Weighted Delta Debugging [12] は入力要素のサイズに対応した重み付けを導入し，削減対象の優先度を調整することで探索コストの削減を実現した．

3. モチベーション

テストケース最小化の代表的な手法である Delta Debugging は汎用性が高い一方で，ステートフルなプロトコル実装の入力に対してそのまま適用すると，アルゴリズムがプログラムの内部状態を考慮しないことによる効率面の問題が生じる．本節ではこの問題を 2 種類の遷移に注目して論じる．

Delta Debugging は，入力の一部を削除してプログラムを実行し，クラッシュが再現されるかという二値の結果のみをフィードバックとして利用する．この状態を無視したアプローチは，ステートフルなプロトコル実装のバグにおいて次の 2 つの問題を引き起こす．

第一に，バグの前提条件となる状態遷移を破壊する試行が生じるという問題がある．例えば，「ログイン後に特定の操作を行う」ことで初めて発生するバグに対し，前提となるログインに必要なメッセージの削除も他のメッセージと同等に試みる．この試行はクラッシュの再現に失敗するが，それがなぜ失敗したのかを考慮しない．結果として，状態遷移に不可欠なメッセージの削除を何度も試行してしまい，探索空間の大部分を無駄に消費する．

第二に，冗長なメッセージを効率的に削減できないという問題がある．入力列には，同じ状態に再度遷移するだけのメッセージや，バグとは無関係な状態遷移を引き起こすメッセージなど，クラッシュに寄与しない多くの冗長なメッセージが含まれる．状態を認識できるならば，こうしたメッセージを優先的に削除候補とすることができるが，状態を考慮しない Delta Debugging はこれらのメッセージを区別できない．

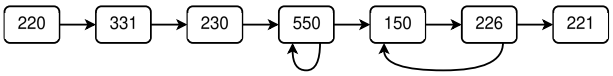
以上のような状態を考慮しないことに起因する非効率性が，本研究で解決を目指す課題である．

4. 手法

本研究では，実行時の SUT における状態遷移グラフを利用した二段階の入力最小化手法を用いて § 3 で提起した課題を解決する手法 `afl-stmin` を提案する．本節では，導入として § 4.1 で本手法を適用する上で重要な参考情報となる SUT の状態遷移グラフについて説明する．これを踏まえて，§ 4.2 ではメッセージ列とバイト列という 2 つの対象に対して段階的に削減を試みる本手法のメインアイデアについて詳細を述べる．§ 4.3 では実装のソースコード

```
220 FTP 2.0.1 ready.
USER user
331 Username ok, send password.
PASS password
230 Login successful.
RETR test.txt
550 No such file or directory.
DELE example.txt
550 No such file or directory.
STOR test.txt
150 File status okay.
226 Transfer complete.
LIST
150 File status okay.
226 Transfer complete.
QUIT
221 Goodbye.
```

(a) FTP サーバーからの応答例



(b) 図 4a に対応する状態遷移グラフ

図 4: FTP サーバーの応答例とそれに対応する状態遷移グラフ

について触れる．

4.1 応答コードに基づく状態遷移グラフ

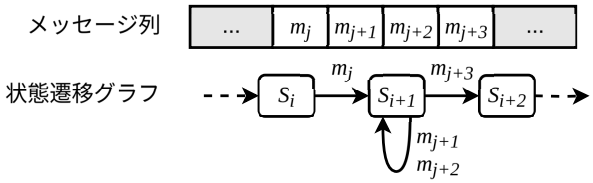
リクエストに対するレスポンスとして応答コードを含むメッセージを得られるような SUT では，応答コードから対応する状態が推定できる [7]．この性質を用いて，入力メッセージ列の送信ごとに応答コードを取得することで，SUT の状態遷移を追跡することが可能になる．本手法では，入力最小化においてメッセージ列を SUT に入力する際に，このような手順を用いて状態遷移グラフを構築し，削除するメッセージの優先順位付けに用いている．図 4 に FTP サーバーからの応答例とそれに対応する状態遷移グラフを示す．

4.2 状態遷移に着目した二段階テストケース最小化

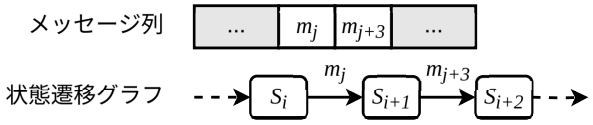
本手法におけるテストケースの最小化手順は二段階で構成される．第一段階では SUT に与える入力をメッセージ列とみなし，クラッシュに寄与しないメッセージを削除する．この段階では，実行時に取得した状態遷移グラフを参考に削除するメッセージの優先順位を付けることで効率的な最小化を行う．メッセージの総数を最小化した後の第二段階では，各メッセージ内のペイロードバイト列に対して Delta Debugging を適用して不要なバイトを削除する．

4.2.1 手法 1：状態遷移を発生させないメッセージの削除

第一段階では，メッセージレベルの Delta Debugging を



(a) 手法 1 適用前のメッセージ列と状態遷移グラフ



(b) 手法 1 適用後のメッセージ列と状態遷移グラフ

図 5: 手法 1 の適用前後におけるメッセージ列と状態遷移グラフの変化

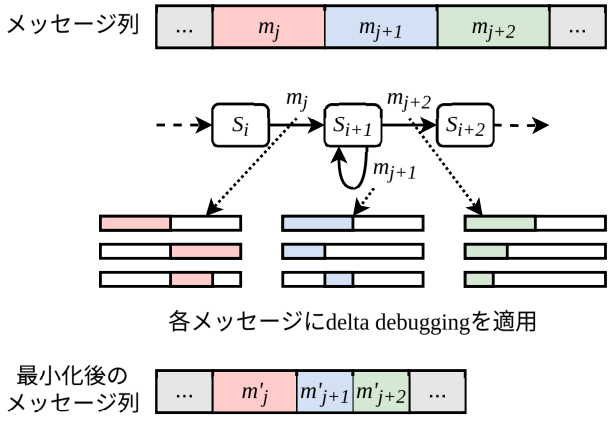


図 6: 手法 2 による入力最小化の概念図

行う過程において，§ 4.1 で導入した状態遷移グラフを実行時に取得し，その情報からクラッシュに寄与しない遷移を特定して優先的に削除する．クラッシュに至るまでの経路に含まれない遷移や，別の状態ではなく同一の状態に移るような遷移はクラッシュに寄与しない可能性が高い．このような遷移に対応するメッセージを優先して削除することで最小化効率の向上を図っている．図 5 に，同一状態へ遷移するメッセージを削除する際のメッセージ列と状態遷移グラフの変化を示す．

4.2.2 手法 2：メッセージ内の Delta Debugging

手法 1 では最小化において削除する対象の単位をメッセージとしたが，1 つのメッセージに含まれるデータに対しても最小化の余地がある．例えば，ある状態において特定のコマンドを受け取った際に発生するバグにおいて，重要な要素は状態への遷移とコマンドであり，メッセージに含まれるペイロードデータに大きな意味は無いと考えられる．このような不要なデータを除去するために，メッセージごとのデータに対して Delta Debugging を施す．図 6 に手法 2 の概念図を示す．

4.3 実装

`afl-stmin` は約 1000 行の C 言語によって実装されている．また，プロトコルごとのメッセージ送受信や応答コードの取得は，AFLNet のコードを利用している．

5. 評価

本節では § 4 で提案した手法を実装し，既知の脆弱性を引き起こすクラッシュに対して適用した結果を評価する．評価の上で設定した研究課題は次の 2 つである．

- **RQ1：`afl-stmin` はどの程度入力を小さくできるか**
- **RQ2：`afl-stmin` はどの程度高速に動作するか**

比較のためのベースラインとして，手法 1 と手法 2 のどちらも用いず，メッセージを 1 つずつ削除しながらクラッシュの有無を確認するだけの簡易的な Delta Debugging を設定した．このベースラインに加えて手法 1 を適用する場合と手法 1 と 2 の双方を適用する場合で研究課題を検証する．なお，Delta Debugging を取り入れた既存の最小化ツールである `afl-tmin` は，単一のファイルを対象とするため，本実験のベースラインとして設定しなかった．

5.1 評価環境

本手法を評価するにあたって，同一の脆弱性を引き起こす入力を複数用意し，それに対して `afl-stmin` を適用した．表 1 は脆弱性の一覧である．

なお，これらの SUT は AddressSanitizer [14] やファジング用の計装を施したものを用いた．また，テストケース最小化にあたって脆弱性の再現が必要であるため，既存のコードに脆弱性を再現するパッチを当ててコンパイルしている．Live555 では AFLNet の GitHub リポジトリ[*1]に同梱されていたパッチを当てており，TinyDTLS ではオリジナルの実装 [15] をフォークしたファジング用の実装 [16] を使用した．

5.2 RQ1：`afl-stmin` はどの程度入力を小さくできるか

表 2 に各手法がベースラインと比較してどの程度入力を小さくしたかを示す．入力のサイズが n バイトから αn バイトになった場合の削減率を $1-\alpha$ と定義し，ベースラインと各手法間の削減率比を入力ごとに平均した．

ベースラインと手法 1 はメッセージ単位で削除を行うが，評価で用いた入力のメッセージ総数が少なく，ベースラインだけで十分な最小化が行われている．したがって，メッセージ単位で削除する手法 1 はベースラインと同様の結果となった．

手法 2 に関しては，#1, #2 の両方においてそれぞれ 8 %， 18 % の削減率上昇を確認した．手法 2 を適用するのに要した時間に対して，この上昇比率が適切であるかどうかは §5.3 で議論する．

[*1] `https://github.com/aflnet/aflnet`

5.3 RQ2：`afl-stmin` はどの程度高速に動作するか

表 3 に，各手法とベースラインにおける実行時間比を入力ごとに平均した結果を示す．

手法 1 を適用したことによる実行時間の増減は #1 と #2 で対照的な結果となった．#1 では最小化に要する時間が 0.64 倍に短縮され，性能向上が見られた一方，#2 では 2.35 倍の時間を要している．#1 は状態遷移を必要とするバグであることから，状態遷移グラフを用いたメッセージの優先順位付けが効果的であったと考えられる．一方，#2 は状態遷移を必要とせず，1 つのメッセージで発生するバグであり，状態遷移グラフを用いた削減は大きな効果を発揮しないステップとなるため，その分がボトルネックになったと推察される．

手法 2 を取り入れた場合は，現実的な時間で最小化が終了するものの，大幅な速度低下が確認された．§ 5.2 より，削減率の向上が 10〜20% に留まっていることを考慮すると，効率面で課題が残る結果となった．手法 2 における Delta Debugging はメッセージ単位に対して施したものと同様に，1 バイトごとにクラッシュの有無を判定するという簡易的なものであるため，このアルゴリズムをより効率が良いものとすることで改善する余地がある．

6. 関連研究

本節では，`afl-stmin` の関連研究として，プロトコル実装に対するファジングの研究を挙げる．

初期のアプローチとしては，既存のファザーをネットワーク通信に対応させるものが挙げられる．例えば，AFL をソケット通信可能にした AFLnwe [17] は，そのシンプルさからベースラインとして広く用いられている．また，boofuzz [18] のような生成ベースのフレームワークは，プロトコルのメッセージ構造を定義することで体系的な入力生成を可能にするが，これらはサーバーの内部状態を積極的に追跡・利用する機構は持たない．ファジングのスループット向上に焦点を当てた研究もあり，SnapFuzz [19] は，高速なスナップショット機構をネットワークアプリケーションに適用するアーキテクチャを提案した．

近年は，プロトコルのステートフル性に対処するため，サーバーの内部状態を推定し，ファジングのフィードバックループに組み込む研究が進められている．その代表例が AFLNet [7] であり，サーバーの応答コードを状態の代理的な指標とみなし，コードカバレッジと状態カバレッジの双方を最大化するように探索を誘導する．本稿における状態遷移グラフの構築とその利用は，AFLNet のアイデアに着想を得ている．

AFLNet 以降，状態推定の精度と効率を高めるための研究が数多く行われている [20]．StateAFL [21] は，サーバー

表 1: 評価に用いる脆弱性の一覧

ID	SUT	脆弱性タイプ	状態遷移を必要とするか	CVE 番号
#1	Live555	Use-After-Free	必要	CVE-2019-15232 [13]
#2	TinyDTLS	Global-Buffer-Overflow	不要	(採番未確認)

表 2: ベースラインを 1.00 とした各手法の削減率平均の比較．ID は表 1 で示したバグに対応する．

ID	ベースライン	手法 1	手法 1&2
#1	1.00	1.00	1.08
#2	1.00	1.00	1.18

表 3: ベースラインを 1.00 とした各手法の実行時間平均の比較．ID は表 1 で示したバグに対応する．

ID	ベースライン	手法 1	手法 1&2
#1	1.00	0.64	75.00
#2	1.00	2.35	86.58

プロセスにおけるメモリ領域のスナップショットをハッシュ化することで，応答コードよりも精度の高い状態識別を実現した．静的解析を用いるアプローチもあり，SGFuzz [22] は，状態変数が列挙体（enum）で実装される傾向にある点に着目し，ソースコードから状態変数を静的に抽出して探索をガイドする．同様に NSFuzz [23] も，コンパイル時の静的解析によってイベントループや状態変数を認識し，ファザーに状態フィードバックを提供する．

また，プロトコル実装に対するファジングのベンチマークとして ProFuzzBench が知られている [24]．ProFuzzBench は AFLnwe, AFLNet, StateAFL の 3 ファザーによるファジングを様々なプロトコルと SUT に対応させ，コードカバレッジを計測するベンチマークプロジェクトである．

7. 今後の展望と課題

本研究では，状態遷移グラフを活用した二段階テストケース最小化手法を提案し，プロトコル実装における入力最小化の新たなアプローチを示した．評価から得られた知見は，今後の研究において以下の方向性と課題を示唆している．

本手法で用いた状態遷移グラフは実行時に得られる情報を使っており，プロトコル仕様や実装に従う状態遷移グラフとは完全に一致しないと考えられる．したがって，事前にこれらの情報を与える，または SUT への計装などによって実行時に追加の情報を取得することで状態遷移グラフの精度を高め，より効率的な最小化を実現する余地がある．

また，本論文における評価では，単一のメッセージや数回の状態遷移で発生する単純な脆弱性を用いており，手法 1 に関してはベースラインに対する削減率の優位性を示すには至らなかった．よって，評価に用いる脆弱性として，より複雑なバグを追加することを検討している．さらに，対象となる SUT とプロトコルを増やし，汎用性を検証することも考えている．

8. おわりに

本論文では，ネットワークプロトコル実装のファジングによって発見されたクラッシュのトリアージを効率化するため，状態遷移に着目した二段階のテストケース最小化手法 `afl-stmin` を提案した．汎用的な手法である Delta Debugging が状態遷移を必要とするバグに対して非効率的であるという課題に対し，状態遷移グラフを用いたアプローチを導入した．

本手法は，実行時に取得した状態遷移グラフの情報に基づき，まず不要なメッセージを優先的に削除する第一段階と，その後，各メッセージ内の不要なバイト列を削除する第二段階で構成される．この二段階の手法により，クラッシュに本質的に寄与する最小限の入力を効率的に抽出することを目指した．

評価実験により，本手法の有効性と課題が明らかになった．第一段階の手法は，状態に依存するバグのテストケース最小化を高速化する可能性を示唆した．さらに，第二段階の手法は，ベースラインと比較して 10〜20 %の削減率向上に貢献した．一方で，第二段階の処理は大幅な実行時間の増加を招き，効率面で改善の余地が残る結果となった．

本研究は，ネットワークプロトコル実装における入力最小化に対し，実行時の SUT の状態という新たな視点を導入し，その有効性を示したことに意義がある．今後は，各段階における最小化アルゴリズムの最適化の実装と，複雑な状態遷移を伴うバグや他のプロトコル実装におけるバグを追加したより広範な評価を検討している．

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

参考文献

[1] Zalewski, M.: American Fuzzy Lop-Whitepaper, *https://lcamtuf.coredump.cx/afl/technical_details.txt.* (2016).

[2] Serebryany, K.: OSS-Fuzz - Google's continuous fuzzing service for open source software, *USENIX Security Symposium* (2017).

[3] google: google/oss-fuzz: OSS-Fuzz - continuous fuzzing for open source software., `https://github.com/google/oss-fuzz`.

[4] Manès, V. J. M., Han, H., Han, C., Cha, S. K., Egele,

M., Schwartz, E. J. and Woo, M.: The Art, Science, and Engineering of Fuzzing: A Survey, *IEEE Transactions on Software Engineering*, Vol. 47, No. 11, pp. 2312–2331 (2021).

[5] Zeller, A. and Hildebrandt, R.: Simplifying and isolating failure-inducing input, *IEEE Transactions on Software Engineering*, Vol. 28, No. 2, pp. 183–200 (online), DOI: 10.1109/32.988498 (2002).

[6] NIST: NVD - CVE-2019-7314, `https://nvd.nist.gov/vuln/detail/CVE-2019-7314`.

[7] Pham, V.-T., Böhme, M. and Roychoudhury, A.: Aflnet: A greybox fuzzer for network protocols, *IEEE International Conference on Software Testing, Validation and Verification (ICST)*, pp. 460–465 (2020).

[8] Misherghi, G. and Su, Z.: HDD: hierarchical Delta Debugging., Vol. 2006, pp. 142–151 (online), DOI: 10.1145/1134307 (2006).

[9] Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C. and Yang, X.: Test-case reduction for C compiler bugs, *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, New York, NY, USA, Association for Computing Machinery, p. 335–346 (online), DOI: 10.1145/2254064.2254104 (2012).

[10] Sun, C., Li, Y., Zhang, Q., Gu, T. and Su, Z.: Perses: syntax-guided program reduction, *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, New York, NY, USA, Association for Computing Machinery, p. 361–371 (online), DOI: 10.1145/3180155.3180236 (2018).

[11] Wang, G., Shen, R., Chen, J., Xiong, Y. and Zhang, L.: Probabilistic Delta debugging, *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, New York, NY, USA, Association for Computing Machinery, p. 881–892 (online), DOI: 10.1145/3468264.3468625 (2021).

[12] Zhou, X., Xu, Z., Zhang, M., Tian, Y. and Sun, C.: WDD: Weighted Delta Debugging , *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, Los Alamitos, CA, USA, IEEE Computer Society, pp. 1592–1603 (online), DOI: 10.1109/ICSE55347.2025.00071 (2025).

[13] NIST: NVD - CVE-2019-15232, `https://nvd.nist.gov/vuln/detail/CVE-2019-15232`.

[14] Serebryany, K., Bruening, D., Potapenko, A. and Vyukov, D.: AddressSanitizer: A Fast Address Sanity Checker, *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, USENIX Association, pp. 309–318 (online), available from ⟨https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany⟩ (2012).

[15] contiki ng: contiki-ng/tinydtls: A version of tinyDTLS that is refactored to be more easy to use "standalone" (e.g. without bindings to a specific IP-stack)., `https://github.com/contiki-ng/tinydtls`.

[16] assist project: assist-project/tinydtls-fuzz: A version of tinyDTLS that is refactored to be more easy to use "standalone" (e.g. without bindings to a specific IP-stack)., `https://github.com/assist-project/tinydtls-fuzz`.

[17] thuanpv: aflnwe, `https://github.com/thuanpv/aflnwe`.

[18] Pereyda, J.: boofuzz: Network Protocol Fuzzing for Humans, `https://boofuzz.readthedocs.io/en/stable/`.

[19] Andronidis, A. and Cadar, C.: SnapFuzz: high-throughput fuzzing of network applications, *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, New York, NY, USA, Association for Computing Machinery, p. 340–351 (online), DOI: 10.1145/3533767.3534376 (2022).

[20] Meng, R., Pham, V.-T., Böhme, M. and Roychoudhury, A.: AFLNet Five Years Later: On Coverage-Guided Protocol Fuzzing, *IEEE Trans. Softw. Eng.*, Vol. 51, No. 4, p. 960–974 (online), DOI: 10.1109/TSE.2025.3535925 (2025).

[21] Natella, R.: StateAFL: Greybox fuzzing for stateful network servers, *Empirical Softw. Engg.*, Vol. 27, No. 7 (online), DOI: 10.1007/s10664-022-10233-3 (2022).

[22] Ba, J., Böhme, M., Mirzamomen, Z. and Roychoudhury, A.: Stateful Greybox Fuzzing, *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, USENIX Association, pp. 3255–3272 (2022).

[23] Qin, S., Hu, F., Ma, Z., Zhao, B., Yin, T. and Zhang, C.: NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing, *ACM Trans. Softw. Eng. Methodol.*, Vol. 32, No. 6 (online), DOI: 10.1145/3580598 (2023).

[24] Natella, R. and Pham, V.-T.: ProFuzzBench: a benchmark for stateful protocol fuzzing, *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, New York, NY, USA, Association for Computing Machinery, p. 662–665 (online), DOI: 10.1145/3460319.3469077 (2021).