情報処理学会論文誌 Vol.62 No.3 877–890 (Mar. 2021)

ミドルウェアに対する Coverage-based Greybox Fuzzingの適用

伊藤 弘将[1,a)] 松原 豊[1] 高田 広章[1]

受付日 2020年5月22日, 採録日 2020年12月1日

概要：近年，自動ソフトウェアテスト手法の一種である fuzzing が注目されており，その効果や効率を向上させるための研究が数多く発表されている．しかし，多くの既存研究では，比較的シンプルなベンチマークを対象に，単一のインタフェースに対し単一の入力を与えた際の探索結果により，評価が行われている．これらのベンチマークとは異なる，より複雑なソフトウェアを対象とした場合，効果的に fuzzing を行うための研究は，あまりなされていない．本研究では，複数のインタフェースから入力系列が与えられ内部状態が変化する，より複雑なソフトウェアとして，ミドルウェアを取り上げ，その開発者がより効果的に fuzzing を適用するためのプロセスを提案する．提案手法では，テスト対象への情報の流入路として，テスト対象が提供するインタフェースに加え，外部コンポーネントがテスト対象へ提供するインタフェースを考慮する．さらに，依存関係を持つインタフェースに対して同時に fuzzing を実行することで，探索範囲の拡大を試みる．既存の組込み向け TCP/IP スタックを用いたケーススタディでは，開発者らが実施した fuzzing の結果と，提案手法に基づく fuzzing の結果を，ラインカバレッジと検出されたクラッシュの数を指標として比較した．結果として，ラインカバレッジを最大で 7 割程度向上させ，新たなバグを 9 種類検出した．

キーワード：fuzzing，セキュリティテスト，ソフトウェアテスト，ミドルウェア，オープンソースソフトウェア

How to Test Middleware with Coverage-based Greybox Fuzzing

Hiromasa Ito[1,a)] Yutaka Matsubara[1] Hiroaki Takada[1]

Received: May 22, 2020, Accepted: December 1, 2020

Abstract: Fuzzing is one of the most popular automated software testing methods. Many researchers have published in the fuzzing related fields. However, most of them have evaluated their proposals with simple benchmarks, which have explored target programs with a value from a single interface. For more complicated targets, how to do that has not been proposed so far. In this research, we propose a new effective method based on coverage-based greybox fuzzing. The method targets middleware, which take some inputs from a lot of interfaces and change internal states of themselves. In order to increase code coverage of fuzzing, we 1) target every interface from where other system components input some data to the target program, and 2) take care of dependencies between their interfaces. We apply our proposed method to a TCP/IP stack for embedded systems and compare line coverage and the number of found crashes between the fuzzing campaigns of the developers and ours. As a result, our fuzzing campaigns covered up to 70% more codes than developers and discovered 9 zero-day bugs.

Keywords: fuzzing, security testing, software testing, middleware, open-source software

1. はじめに

近年開発されるソフトウェアは大規模・複雑化の一途をたどっており，国内のソフトウェア開発においてテストが

[1] 名古屋大学大学院情報学研究科
Graduate School of Informatics, Nagoya University, Nagoya, Aichi 464–8601, Japan
[a)] itoh@ertl.jp

占める工数は約 4 割にのぼっている [1]．テスト工程に多くのリソースが割かれることは，真に創造的な活動にあてられるリソースの減少や，開発全体にかかる時間の増加を引き起こす．ゆえに，ソフトウェアテストの自動化は早急に実現すべき課題である．

自動化されたソフトウェアテストの一手法として，fuzzing（fuzz testing）がある．fuzzing は，テスト対象のソフトウェアに対しさまざまな入力を与えて実行することで，テスト対象が任意の入力に対してクラッシュしないことをテストする．テスト対象がクラッシュすればテストは失敗，クラッシュしなければテストは成功，というように，テストの期待結果（テストオラクル）を単純化することで汎用性の高い自動テストを実現しており，近年では fuzzing によって多くのバグや脆弱性が検出されている．

研究領域においても fuzzing に対する注目度は高く，fuzzing によるテストをより効率的かつ効果的に実行するための手法が，数多く提案されている．Klees ら [2] は，fuzzing に関する多数の論文について，評価に用いられているベンチマークの調査を実施している．Klees らが調査した関連論文 32 本においてベンチマークとして使用されている実世界プログラム上位 10 種とその使用論文数を**表 1** に示す．表 1 に示したベンチマークは，以下のような共通の性質を持つ．

- 入力系列によって変化する内部状態を持たない．
- インタフェース間に相互作用が存在しない．

このような性質を満たさない，より複雑なソフトウェアに対して fuzzing を行う場合，多くの論文における評価のように単一のインタフェースから fuzzing を行っていては，探索範囲が大幅に制限されてしまう可能性がある．

本研究では，表 1 に示したベンチマークとは異なる性質を持つソフトウェアとしてミドルウェアを取り上げ，より効果的に fuzzing を適用するためのプロセスを提案する．ミドルウェアは，オペレーティングシステム（OS）やハードウェアとアプリケーションの中間に位置するソフトウェアである．複雑に相互作用する多数のインタフェースを持ち，さまざまな外部コンポーネントと連携しながら動作する．また，その内部状態は，外部コンポーネントから与えられる入力系列によって断続的に変化する．

表 1 fuzzing 関連論文 32 本においてベンチマークとして使用されている実世界プログラム上位 10 種

Table 1 The top 10 real-world programs most frequently used as benchmarks in 32 fuzzing papers.

テスト対象	使用論文数
`libpng`	9
`libjpeg`	7
`libpcap`	5
`libpoppler`	5
`tcpdump`（`libpcap`）	5
`binutils`	4
`djpeg`（`libjpeg`）	4
`ffmpeg`	4
`gif2png`（`libpng`）	4
`mupdf`	4

提案手法では，テスト対象に対する情報の流入路として，テスト対象が提供するインタフェースに加え，外部コンポーネントがテスト対象へ提供するインタフェースを考慮する．さらに，fuzzing の対象とするインタフェース間の依存関係に着目することで，生成されるテストケースの複雑化やテスト効率の悪化を抑えつつ，fuzzing による探索範囲の拡大を試みる．ケーススタディとして，提案手法を既存の組込み向け TCP/IP スタック lwIP へ適用し，コードカバレッジと検出されたクラッシュの数を指標として，提案手法の有効性を評価する．

本研究の具体的な貢献は以下のとおりである．

- 多くの既存研究で用いられているベンチマークとは異なる性質を持つソフトウェアとしてミドルウェアを取り上げ，その開発者がより効果的に fuzzing を適用するためのプロセスを提案した．
- 既存研究においては言及されていない，外部コンポーネントが提供するインタフェースを，fuzzing の対象インタフェースとして考慮した．さらに，インタフェース間の依存関係を解析することで，より複雑な不具合の検出と，fuzzing による探索範囲の拡大を試みた．
- 組込み向け TCP/IP スタック lwIP を対象としたケーススタディにおいて，既存の結果と比較してラインカバレッジを平均で 2 割から 3 割，最大で 7 割程度向上させた．さらに，開発者らが認知していなかった新たなバグを検出し，不具合報告とテストドライバの提供を行うことで，広く利用されているオープンソースソフトウェア（OSS）の品質向上へ貢献した．

本論文は，以下のように構成される．1 章（本章）では，本研究の背景，ならびにその目的と貢献について述べた．続く 2 章では，本研究の技術的背景として fuzzing について概説する．3 章において提案手法を述べ，4 章ではケーススタディにおける手法の適用とその結果を述べ，考察を行う．その後，5 章において関連研究に対する本研究の位置付けと差分を示し，6 章をまとめとする．

2. Fuzzing

2.1 概要

fuzzing は，ソフトウェアテストの一手法である．Miller ら [3] が行った UNIX 用ユーティリティプログラムの信頼性テストに端を発しており，近年では，数多くのバグや脆弱性が fuzzing によって検出されている．

fuzzing の基本的な概念図を**図 1** に示す．原理は非常に単純であり，

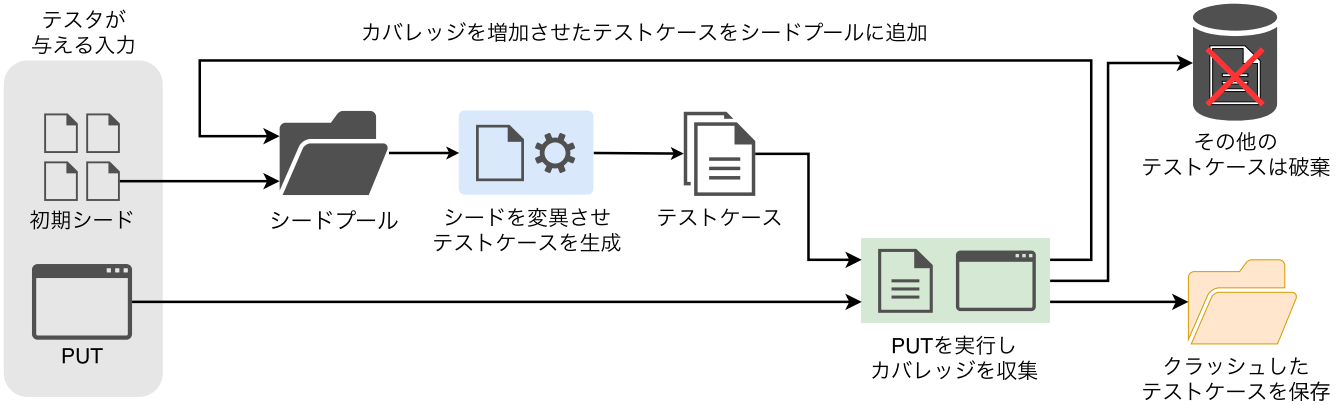

図 2 Coverage-based Greybox Fuzzing のアーキテクチャ
Fig. 2 The architecture of Coverage-based Greybox Fuzzing.

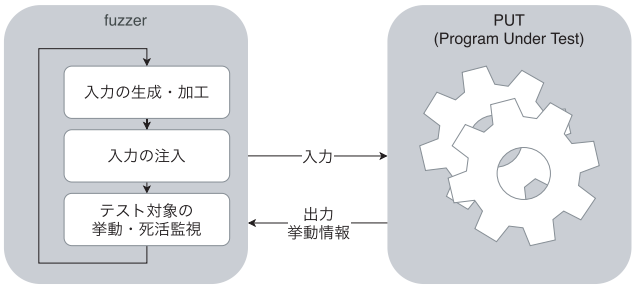

図 1 fuzzing の概念図
Fig. 1 A concept of fuzzing.

(1) 入力の生成・加工
(2) 入力の注入
(3) テスト対象の挙動・死活監視
という 3 つのステップを高速に繰り返し，テスト対象プログラム（Program Under Test; PUT）のテストを行う．これらの処理を自動で実行し fuzzing を行うプログラムを，一般に fuzzer と呼ぶ．入力の生成・加工では，何らかのモデルに基づいて入力を生成，あるいは，既存の入力を変異させることで，新たな入力を作成する．入力の注入では，PUT–fuzzer 間に存在する何らかの通信路を介して，新たな入力を PUT へ注入する．PUT の挙動・死活監視では，与えた入力に対する PUT の出力や振舞いを観測し，テストの成否を判断する．成否判断のためのテストオラクルとしては，暗黙的テストオラクル（implicit test oracle）の一種である「PUT が異常終了しない」という条件が用いられることが多い [4]．この判定には，PUT プロセスの終了コードや，生存確認パケットに対する応答の有無などが用いられる．

2.2 Coverage-based Greybox Fuzzing

Coverage-based Greybox Fuzzing（CGF）は，DeMott ら [5] により提案された，fuzzing の派生手法の一種である．fuzzing に対して，コードカバレッジによるフィードバックを導入し，より広範囲の探索と，より多くのバグの検出を試みる．近年では，American Fuzzy Lop（AFL）[6] や libFuzzer [7] といった CGF を実行する fuzzer の有効性が経験的に認知され，多くの実世界プログラムのバグや脆弱性の検出に利用されている．

CGF のアーキテクチャを**図 2** に示す．本論文では，CGF における既知の入力をシードと呼び，シードの集合をシードプールと呼ぶ．また，テスタが fuzzer に対してあらかじめ与えるシードを初期シードと呼ぶ．CGF では，シードプールから取り出したシードを変異させ，新たな入力を生成する．生成した入力を与えて PUT を実行し，コードカバレッジを収集する．与えた入力によってカバレッジが増加した場合には，その入力をシードプールへ追加する．「カバレッジを増加させた入力を変異させると，新たなコードがカバーされる可能性が高い」という仮説に基づきシードプールを更新していくことで，ランダムに生成された入力よりも効率的かつ効果的に PUT を探索する．

3. 提案手法

3.1 方針

本研究では，既存研究で用いられているベンチマークとは異なる性質を持つソフトウェアとしてミドルウェアを取り上げ，より効果的に fuzzing を適用するためのプロセスを提案する．

対象とするミドルウェアは，既存研究のベンチマークとは異なる以下の性質を持つ．

- 入力系列によって変化する内部状態を持ちうる．
- インタフェース間に相互作用が存在しうる．

これらの性質は，既存研究の評価において行われているような，単一のインタフェースに対する単一の入力による fuzzing において，探索範囲を限定する要因となる．この場合，探索範囲を拡大する素朴な方法は，複数のインタフェースに対して同時に fuzzing を行うことである．しかし，複数のインタフェースに対して同時に fuzzing を行うことは，あわせてテストケースの複雑化を引き起こし，ひ

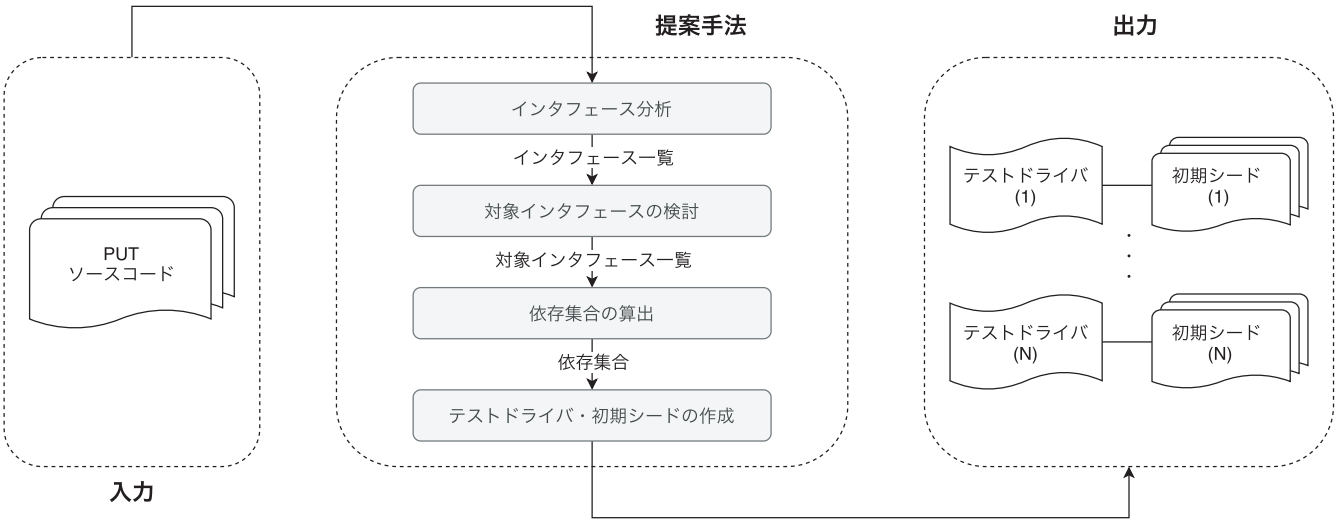

図 3 提案手法の流れ
Fig. 3 The process of our proposed method.

いては不具合修正の困難化につながる．そのため，必要最小限のインタフェースに対してのみ，同時に fuzzing を実行することが望ましい．そこで，本研究における提案手法では，fuzzing を実行するインタフェースの間に存在する依存関係に着目し，依存関係が存在するインタフェースに対してのみ同時に fuzzing を実行することで，これを実現する．本研究が着目する依存関係の詳細については，3.4 節で述べる．

提案するプロセスの流れを，**図 3** に示す．提案手法は，PUT のソースコードを入力として，fuzzing によるテストを実行するためのテストドライバと，それに対応する初期シードを出力する一連のプロセスである．手法は主に以下の 4 フェーズからなる．

(1) インタフェース分析
(2) 対象インタフェースの検討
(3) 依存集合の算出
(4) テストドライバ・初期シードの作成

従来のプロセスとの大きな違いは，インタフェース分析フェーズにおいて，PUT が利用する外部コンポーネントのインタフェースを含めた点，そして，依存集合の算出フェーズを設け，インタフェース間の依存関係を考慮した点である．

3.2 インタフェース分析

インタフェース分析フェーズでは，PUT と外部コンポーネント間のインタフェースを列挙する．本研究におけるインタフェースは，PUT が外部コンポーネントに対して提供しているインタフェース（**図 4** 上側）だけでなく，PUT が利用する外部コンポーネントのインタフェース（図 4 下側）を含む．本フェーズ終了後には，PUT と外部コンポーネント間のインタフェース一覧が出力される．

提案手法は，テスト対象とするミドルウェアの開発者に

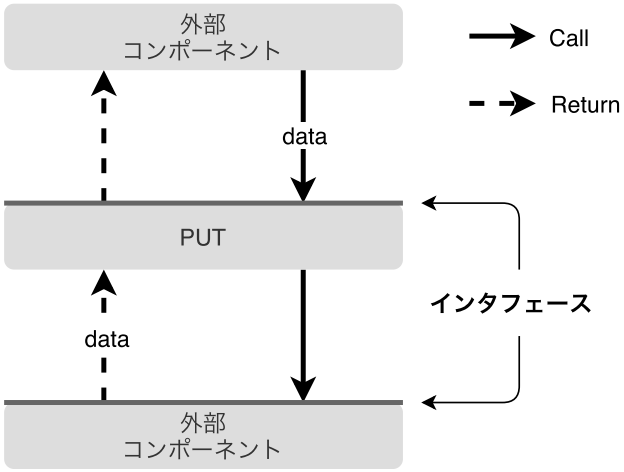

図 4 PUT と外部コンポーネント間のインタフェース
Fig. 4 Interfaces between a PUT and other components.

よって適用されることを想定している．そのため，開発に際して詳細なドキュメントが作成されている場合には，それらを参照することで，人手によりインタフェースを列挙することが可能と考えられる．また，参照可能なドキュメントなどが存在しない場合には，ソースコードの簡単な静的解析によって，大部分のインタフェースの抽出が可能である．例として，C 言語で記述されたミドルウェアであれば，ヘッダファイルを解析することで，PUT が提供するインタフェースを収集することが可能である．同様に，コールグラフを解析することで，PUT が利用する外部コンポーネントのインタフェースを収集することが可能である．

3.3 対象インタフェースの検討

本研究では，fuzzing において PUT へデータを入力するインタフェースを，対象インタフェースと呼ぶ．対象インタフェースの検討フェーズでは，インタフェース分析フェーズで列挙したインタフェースのうちから，対象イン

```
interfaceA()
{
  [...]
  if (interfaceB() == hoge)
  {
    [...]
  }
  [...]
}

interfaceB()
{
  [...]
  return hoge;
}
```

(a) 陽な依存関係

```
state_t state;

interfaceA()
{
  [...]
  if (state == hoge)
  {
    [...]
  }
  [...]
}

interfaceB()
{
  [...]
  state = hoge;
  [...]
}
```

(b) 陰な依存関係

図 5 依存関係が存在するインタフェースの例
Fig. 5 Examples of a pair of dependent interfaces.

タフェースを決定する．本フェーズ終了後には，対象インタフェースの一覧が出力される．

原理的には，非信頼の外部コンポーネントから PUT へデータが流入するインタフェースはすべて対象インタフェースとなりうる．しかし，最終的にどのインタフェースを対象インタフェースとすべきかは，ケースバイケースである．

3.4 依存集合の算出

3.4.1 インタフェース間の依存関係

本論文では，2 つのインタフェース間に存在する依存関係として，陽な依存関係と陰な依存関係を定義する．陽な依存関係，陰な依存関係にあるインタフェースの例をそれぞれ**図 5** (a)，(b) に示す．これらの依存関係にあるインタフェースに対し独立に fuzzing を行っていても，探索可能な範囲は限定的となるため，同時に fuzzing を実行することが望ましい．

定義 1（陽な依存関係）．あるインタフェース A の呼び出しによって駆動される PUT のコードを考える．そのコード内で他のインタフェース B の呼び出しが行われ，インタフェース B から流入するデータにコードの振舞いが依存する場合，インタフェース A はインタフェース B に「陽に依存している」という．

定義 2（陰な依存関係）．あるインタフェース A の呼び出しによって駆動される PUT のコードを考える．そのコード内で読み出されるメモリ領域に対して他のインタフェース B から書き込みが行われる場合，インタフェース A はインタフェース B に「陰に依存している」という．

3.4.2 算出方法

依存集合は，以下のステップにより算出される．

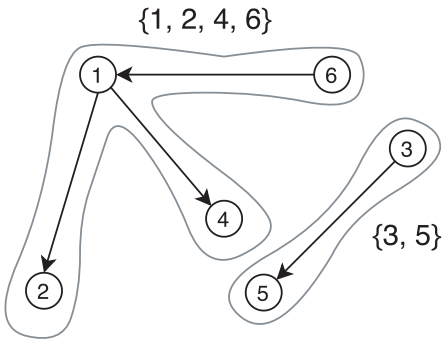

図 6 依存関係グラフとその弱連結成分の例
Fig. 6 An example of interface dependency graph and weakly connected components of that.

(1) 依存関係グラフの作成
(2) 弱連結成分の抽出

依存関係グラフの作成

対象インタフェース間における依存関係グラフは，対象インタフェースをノードとする有向グラフである．依存関係グラフの例を**図 6** に示す．ノード間の有向辺は，依存元の対象インタフェースを表すノードから依存先の対象インタフェースを表すノードへの向きを持つ．陽な依存関係，あるいは陰な依存関係を持つインタフェースを表すノード間にエッジを引くことで，依存関係グラフが得られる．

弱連結成分の抽出

得られた依存関係グラフから，弱連結成分を抽出する．ある有向グラフにおいて任意の 2 頂点間に準路が存在するとき，その有向グラフは弱連結グラフである．また，ある有向グラフにおける弱連結成分とは，その有向グラフにおける極大な弱連結部分グラフである．図 6 の場合，弱連結成分は 2 つ存在する．それぞれの弱連結成分から，それを構成する対象インタフェースを要素とする集合を抽出し，これを依存集合と呼ぶ．図 6 の例であれば，$\{1,2,4,6\}$，$\{3,5\}$ がそれぞれ依存集合となる．

3.5 テストドライバ・初期シードの作成

3.5.1 テストドライバ

前フェーズで抽出した依存集合ごとに，テストドライバを作成する．テストドライバは，fuzzer が生成した入力を読み込み，PUT に入力するプログラムである．fuzzer は単一のデータストリームを入力として生成するため，テストドライバがその内容を解釈し，対象インタフェースから PUT へ適切に入力を与える必要がある．

テストドライバから PUT へデータを入力するコンポーネントは，対象インタフェースの性質により，ドライバとスタブに大別できる．図 4 上側のように，対象インタフェースが PUT の提供するインタフェースである場合は，ドライバを作成する．ドライバでは，PUT が提供する API を呼び出し，fuzzer が生成したデータを PUT へ入力する．

一方，図 4 下側のように，対象インタフェースが外部コンポーネントの提供するインタフェースである場合は，スタブを作成する．スタブでは，インタフェースの外部仕様を模擬し，fuzzer が生成したデータを呼び出し元の PUT へと入力する．

3.5.2 初期シード

前フェーズで抽出した依存集合ごとに，初期シードを作成する．2.2 節で述べたとおり，初期シードは，テスタが fuzzer に対して与えるシードである．CGF を実行する fuzzer は，fuzzing の開始時において，与えられた初期シードを変異させることで入力の生成を開始する．

一般に，CGF による探索効率は，初期シードに大きく依存することが知られている．CGF の効果をより高めるためには，PUT の正常系で処理することができ，より深いコードまで到達可能な初期シードをいくつか与えることが望ましいとされている [8], [9], [10]．

4. ケーススタディ

4.1 概要

提案手法の有効性を評価するため，既存の OSS を対象とした提案手法の適用と評価実験を実施した．対象とした OSS は，C 言語で記述された組込み向け TCP/IP スタック lightweight IP（lwIP）である．数多くの組込みシステムベンダにおける採用実績があり，近年ではさまざまなものの IoT 機器化も手伝って，個人開発者から企業まで幅広く利用されている OSS である．

lwIP の開発者らは，AFL により lwIP に対する fuzzing をすでに実行しており，その際に利用されたテストドライバと初期シードが一般に公開されている [11]．本研究では，開発者らが公開しているテストドライバと初期シードを用いた fuzzing の結果と，提案手法に則って作成したテストドライバと初期シードを用いて実行した fuzzing の結果を定量的な指標で比較することで，提案手法の有効性を評価する．

4.2 提案手法の適用

4.2.1 インタフェース分析

本実験においては，公開されている lwIP のドキュメント，ならびに，`cflow` [12]，`grep` [13] などの静的解析ツールを利用し，人手によってインタフェースの列挙を行った．また，**図 7** に示すような，lwIP の一般的なユースケースにおけるシステムを想定し，各インタフェースから lwIP へ流入するデータを特定した．lwIP には主に 3 つの外部コンポーネントからデータが流入する．アプリケーション，デバイスドライバ，OS である．それぞれと lwIP 間のインタフェースからは，以下のようなデータが流入する．

アプリケーション

- TCP/IP を利用して送出するデータ
- コネクションの制御値

デバイスドライバ

- 受信したネットワークパケット
- 仮想 NIC の制御値

OS

- OS が管理するシステムリソースに関するデータ（例：システム時刻）

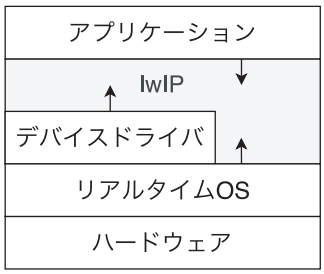

図 7 lwIP を用いた典型的なシステム
Fig. 7 A typical system with lwIP.

表 2 評価実験における対象インタフェース
Table 2 The target interfaces in the experiments.

インタフェース名	外部コンポーネント	入力するデータ
`ethernet_input`	デバイスドライバ	Ethernet フレーム
`sys_now`	OS	システム時刻
`tcp_write`	アプリケーション	TCP セグメントのペイロード
`udp_send`	アプリケーション	UDP セグメントのペイロード

4.2.2 対象インタフェースの検討

lwIP は，ユーザが必要に応じてさまざまな環境に移植して利用することを前提として設計されており，高い移植性を持つソフトウェアである．このようなミドルウェアの開発者は，つねに信頼可能な外部コンポーネントからデータが流入することを保証できない．そのため本実験においては，外部コンポーネントはすべて非信頼であると仮定した．

3.3 節において述べたように，非信頼な外部コンポーネントからデータが流入しうるインタフェースは fuzzing の対象となりうるが，実際にどのインタフェースを対象インタフェースとするかは，ケースバイケースである．lwIP の場合，自身が提供しているインタフェースに限定した場合でも，100 以上のインタフェースが存在している．本実験においては，OS，デバイスドライバ，アプリケーション，といった外部コンポーネントごとに，主要なインタフェースを 1 つ以上選出し，対象インタフェースとした．

具体的に選出した 4 種類の対象インタフェースを，**表 2** に示す．本実験における対象インタフェースの選定のねらいは，lwIP が実際のシステムに組み込まれて利用される際の典型的なユースケースにおいて発生しうる不具合を検出することである．lwIP を搭載したシステムに対しては，外部から任意のタイミングで Ethernet フレームが到着し，lwIP を利用する TCP/UDP のアプリケーションがそれに対して応答を行う，という挙動が想定される．外部から到着

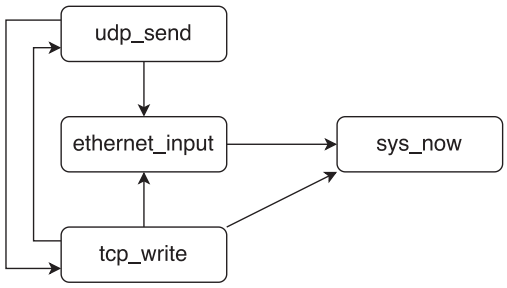

図 8 評価実験における対象インタフェース間の依存関係グラフ
Fig. 8 The interface dependency graph in the experiments.

する Ethernet フレームに起因する不具合を捕捉するため，Ethernet フレームの受信インタフェース `ethernet_input` を対象インタフェースとした．また，Ethernet フレームの到着タイミングは任意であるという実世界の性質を反映するため，OS が lwIP に対してシステム時刻を提供するインタフェース `sys_now` も，あわせて対象インタフェースとした．さらに，アプリケーションからは，受信データなどに基づき任意のデータが送出される可能性がある．データの送出にともなう不具合を捕捉するため，TCP の出力インタフェース `tcp_write`，ならびに UDP の出力インタフェース `udp_send` を，対象インタフェースとした．

4.2.3 依存集合の算出

表 2 に示した 4 つの対象インタフェースについて，依存関係を調査した．調査した依存関係をもとに作成した依存関係グラフを**図 8** に示す．

受信された L2 フレームは，その内容により lwIP の状態を変化させ，アプリケーションの出力処理に影響を与える．さらに，TCP や UDP は下位レイヤにおける出力機構を共有しているため，出力処理は互いに影響を及ぼし合う．すなわち，`ethernet_input` や `tcp_write`，`udp_send` の間には，陰な依存関係が存在する．

OS が lwIP へ提供するシステム時刻は，通信の信頼性を担保する TCP プロトコルの振舞いに影響を与える．すなわち，TCP の出力インタフェースである `tcp_write` や TCP セグメントをペイロードとして持つ Ethernet フレームを処理する `ethernet_input` は，システム時刻を返すインタフェースである `sys_now` との間に，陽な依存関係が存在する．

図 8 に示した依存関係グラフの弱連結成分から依存集合を算出すると，4 つすべての対象インタフェースを要素に持つ集合が得られる．ゆえに，評価実験においては 4 つの対象インタフェースに対し同時に fuzzing を実行することとした．

4.2.4 テストドライバ・初期シードの作成

テストドライバ

算出した依存集合に対して作成したテストドライバのアーキテクチャを**図 9** に示す．テストドライバには，Ethernet ドライバと TCP/UDP アプリケーションを模したドライバと，UNIX 向け実装をベースとしたシステム時刻取得用インタフェース `sys_now` のスタブを実装した．ドライバは lwIP のインタフェースを呼び出すことで，スタブは lwIP から呼び出されることで，データの入力を行う．テストドライバは fuzzer が生成した入力をバッファに保存しており，テストドライバ内に実装されたドライバとスタブは，それを共有している．必要に応じてバッファから入力を取り出し，lwIP へと入力する．

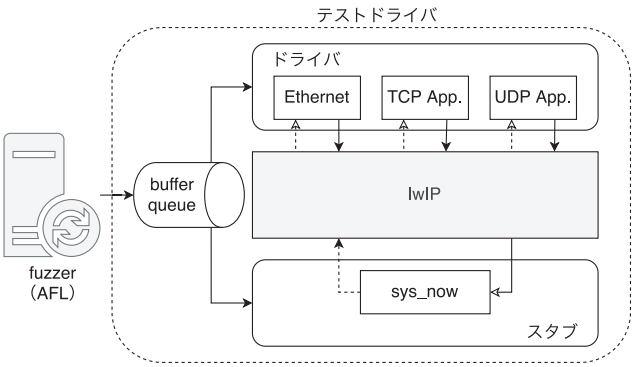

図 9 テストドライバのアーキテクチャ
Fig. 9 The architecture of the test driver.

表 3 作成した初期シード
Table 3 The initial seeds we made.

プロトコル	初期シードの個数	初期シードの内容
ARP	2	ARP リプライ，リクエスト
ICMPv4	1	エコーリクエスト
ICMPv6	3	MAC アドレス通知，DAD 問合せ，ルータ広告
UDPv4	2	擬似サーバおよびクライアント宛データ
なし	1	文字列 “hello, world”
TCPv4	17	擬似クライアントおよびサーバと通信を行い，TCP 状態機械の各状態まで到達する入力系列
TCPv6	1	擬似サーバと通信を行い，コネクションを確立する入力系列

初期シード

作成した初期シードに関する情報を，**表 3** に示す．基本的に，プロトコルごとに正常な通信が可能な数個の初期シードを作成した．ただし，TCP については，その他のプロトコルよりも多くの初期シードを作成した．これは，TCP が CGF により探索しにくい以下のような性質を持つことに起因する．

(1) 状態を持つプロトコルである．
(2) 状態遷移の条件が厳しい．

TCP は状態を持つプロトコルであり，状態を遷移させるような入力系列を与えなければ十分に探索できない．しかしながら，TCP は通信の信頼性を担保するプロトコルであり，シーケンス番号・確認応答番号と呼ばれる仕組みに基づく非常に厳密な遷移条件を持つ．このような厳密な遷移条件をうまく満たし，より深いコードへと到達可能な入力系列を生成することは，プロトコルに関する知識や，出力の観測・フィードバック機構をまったく持たない純粋な CGF では困難である．そこで評価実験においては，TCP

表 4 実験設定
Table 4 The experimental configurations.

実験 ID	*default/single*	*default/multi*	*proposal*
テスト環境	Ubuntu 18.04 on Intel®Core™i9-9960X CPU @ 3.10 GHz		
PUT	lightweight IP ver 2.1.0 RC1		
fuzzer	American Fuzzy Lop ver 2.52b		
実行時間	24 時間		
試行回数	30 回		
初期シード	6 種類		27 種類
対象インタフェース	`ethernet_input`	`ethernet_input`	`ethernet_input`, `sys_now`, `tcp_write`, `udp_send`
入力の解釈	単一の Ethernet フレーム	Ethernet フレームの系列	Ethernet フレーム，システム時刻， アプリケーションの入力データからなる系列

コネクションの状態機械に関する情報をもとに，それぞれの状態に到達可能な入力系列を初期シードとして作成し，fuzzer へ与えることとした．なお，TCP ヘッダはシーケンス番号や確認応答番号以外にも多くのフィールドを持つ．ゆえに，状態機械に基づいて作成した初期シードをもってしてもすべてのコードがカバーされることはなく，依然として fuzzing の余地がある．

4.3 評価実験

4.3.1 概要

評価実験では，公開されているテストドライバおよび初期シードを用いた fuzzing の結果と，提案手法に基づいて作成したテストドライバおよび初期シードを用いた fuzzing の結果を，2 つの定量的な指標で比較する．第 1 の指標は，ラインカバレッジである．この指標は探索範囲の広範さを数値化するものであり，この値を比較することで，どちらがより PUT を広範に探索したかを知ることができる．第 2 の指標は，検出したクラッシュの数である．この指標は，fuzzing による探索の効果を数値化するものであり，この値を比較することで，どちらがより効果的に PUT を探索したかを知ることができる．

表 4 に示す 3 種類の実験設定で fuzzing を実行した．テスト環境，PUT，fuzzer および fuzzing の実行時間は，すべての実験設定で統一した．本論文では，既存のテストドライバと初期シードを用いる実験を *default* とし，提案手法に則って作成したテストドライバと初期シードを用いる実験を *proposal* とする．また，既存のテストドライバでは，fuzzer が生成した入力を単一の Ethernet フレームとして解釈するか，複数の Ethernet フレームが連なった系列として解釈するかをテスタが選択できる．本論文では，単一の Ethernet フレームとして解釈する場合の実験を *default/single* とし，Ethernet フレームの系列として解釈する場合の実験を *default/multi* とする．

4.3.2 結果

ラインカバレッジ

ラインカバレッジの計測対象は，lwIP のコアコンポーネントを含むディレクトリ `lwip/src/core/` 内に存在するファイルとした．ディレクトリ内には 37 のソースコードファイルが存在し，その SLoC（Source Lines of Code）は約 20,000 行である．なお，ラインカバレッジの計測においては，外部コンポーネントから呼び出さない限り実行されない API やユーティリティ関数のみが記載されているファイルは除外しており，計測対象の総ファイル数は 18，その SLoC は約 13,000 行である．

表 5 提案手法の適用によるカバレッジ増加量
Table 5 Some statistical values of coverage improvement.

比較対象	Min. [%]	Max. [%]	Avg. [%]	Med. [%]
default/single	0.00	72.53	28.25	29.28
default/multi	0.00	64.48	21.80	11.54

各設定で 24 時間の fuzzing を 30 回実行し，得られたラインカバレッジの平均値を**図 10** に示す．同図のエラーバーは，95%のブートストラップ信頼区間である．また，提案手法の適用による各ファイルのカバレッジ増加量について，その最小値，最大値，平均値，中央値を**表 5** に示す．いずれの設定と比較してもラインカバレッジが低下したファイルは存在せず，最大で 7 割程度の増加が観測された．

なお *proposal* については，未到達となったコードを関数レベルで分析し，その要因を調査・分類した．未到達関数の 9 割は今回対象インタフェースとしていない API やユーティリティ関数であり，関数レベルでは到達可能領域を網羅的に探索できていることが判明した．

検出されたクラッシュの数

各設定で fuzzing を行った結果検出されたクラッシュの数を**図 11** に示す．また，各設定で 30 回 fuzzing を行った結果検出されたクラッシュの総数を**表 6** に示す．なお，同一箇所において発生したクラッシュを 1 つとしてカウントしている．*default/single* ではクラッシュは 1 つも検出されなかった．*defualt/multi* では各試行において 0 から 2 つ程度のクラッシュが検出され，30 回の試行で検出されたクラッシュの総数は 3 であった．*proposal* では各試行において 3 から 6 つ程度のクラッシュが検出され，30 回の試行で

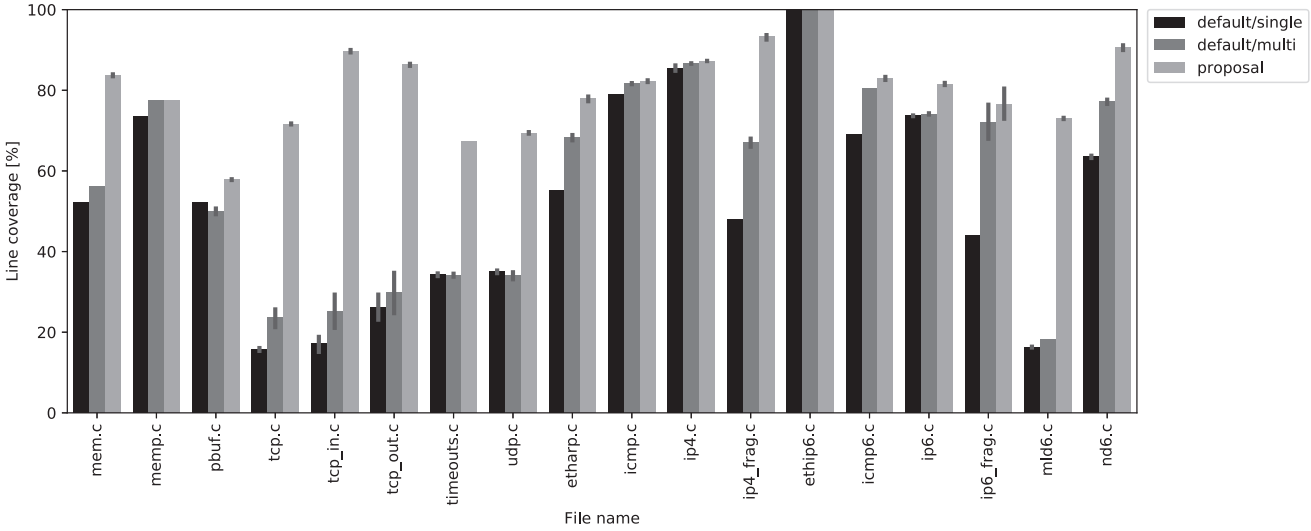

図 10 各ファイルの平均ラインカバレッジ
Fig. 10 The averages of the line coverage of each file in the experiments.

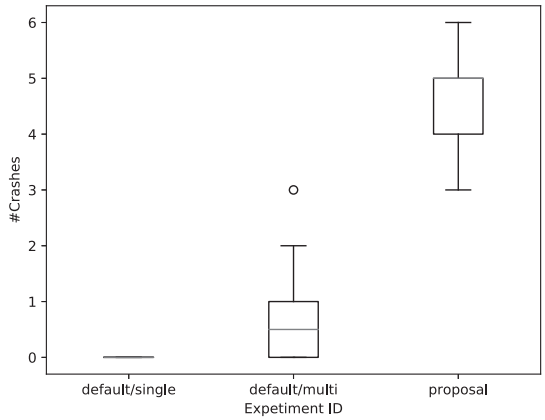

図 11 各試行で検出されたクラッシュ数
Fig. 11 The number of crashes found in each trial.

表 6 検出されたクラッシュの総数
Table 6 The total number of crashes found in each experiment.

実験 ID	検出されたクラッシュの総数
default/single	0
defualt/multi	3
proposal	9

検出されたクラッシュの総数は 9 であった．あわせて，各設定で検出されたクラッシュの包含関係を**図 12** に示す．*default/multi* と *proposal* では，検出されたクラッシュのうち 1 つが共通していた．

proposal で検出されたクラッシュについて，原因調査を行った．調査の結果判明したクラッシュの発生箇所，クラッシュの原因と考えられる事象，その事象を発生させるデータが流入したアタックベクタを**表 7** に示す．*proposal* で検出されたクラッシュでは，その原因となる異常な値の流入は，すべてネットワークインタフェース，すなわち，`ethernet_input` から入力された Ethernet フレームによるものであった．一方，クラッシュの発生箇所については，異常な入力を直接的に処理する入力処理に限らず，何らかのデータを出力する出力処理や，時間経過によるタイムアウトの処理を行うタイマ処理など，多岐にわたっていた．

なお，本実験で作成したテストドライバ，ならびにクラッシュを発生させる入力については，lwIP の開発者らへ提供のうえ，バグレポートを投稿している．クラッシュの検証ならびに修正などの対応判断は開発者らに委ねているが，すでに修正が行われている不具合 [14], [15] や，修正の方針を検討中の不具合 [16], [17] が存在している．また，作成したテストドライバは，lwIP の正式なテストドライバの 1 種として開発のメインストリームへマージされた [18]．

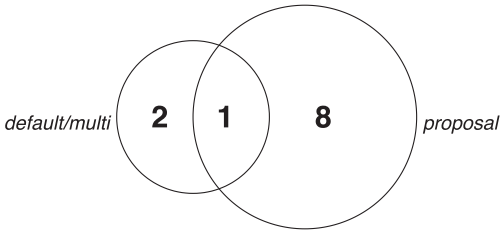

図 12 各設定で検出されたクラッシュの包含関係
Fig. 12 The relative coverage of finding crashes.

表 7 *proposal* で検出されたクラッシュ
Table 7 Crashes found in *proposal*.

ID	理由・原因	クラッシュの発生箇所	アタックベクタ
001	暗黙の仮定に基づいた型変換	入力処理	ネットワーク
002	暗黙の仮定に基づいた型変換	入力処理	ネットワーク
003	IPv6 ヘッダに対する不十分な validation	出力処理	ネットワーク
004	TCP ヘッダに対する不十分な validation	出力処理	ネットワーク
005	TCP ヘッダに対する不十分な validation	出力処理	ネットワーク
006	エラー処理におけるメモリの二重解放	タイマ処理	ネットワーク
007	一貫性のない後処理	タイマ処理	ネットワーク
008	暗黙の仮定に基づいた型変換	タイマ処理	ネットワーク
009	TCP ヘッダに対する不十分な validation	タイマ処理	ネットワーク

4.4 考察

4.4.1 提案手法の有効性

本実験では，提案手法を適用した場合の方が，ラインカバレッジ，検出されたクラッシュの数ともに優れた結果となった．この結果は，少なくとも lwIP という PUT に対しては，開発者らが実行した一連の fuzzing プロセスよりも，提案手法のプロセスの方が，より fuzzing による探索範囲を広範にし，効果的に fuzzing を適用可能であることを示している．

検出された不具合についても，提案手法のみが検出した不具合は，既存手法では検出が不可能なものと思われる．クラッシュを発生させる直接的な原因となった入力は特定のインタフェースから与えられたものであっても，その他のインタフェースに対してさまざまな入力を与えて駆動しない限り発生しないようなクラッシュは存在しうる．このようなクラッシュの発生手順を**図 13** に示す．具体的には，以下のような手順を踏む．

(1) あるインタフェースから，クラッシュを引き起こしうる入力が与えられ，その入力が状態として PUT に保持される．
(2) 他のインタフェースから，入力が与えられる．
(3) 他のインタフェースから与えられた入力を処理する際に，状態として保持されている入力を参照することで，クラッシュが発生する．

このようなクラッシュは，依存関係にある複数のインタフェースに対して入力を与えることで発生するものであり，提案手法での検出が期待される．

一方で，図 12 に示すように，既存手法のみで検出された不具合が存在することは，無視できない点である．この要因としては，主に以下の 2 つが考えられる．なお，これらの要因は排他的なものではなく，いずれか一方の要因によるものと断定することは困難である．

(1) インタフェースあたりの探索試行回数の減少
(2) テストドライバの差異

第 1 の可能性について考察する．本実験では，すべての実験条件において同一の時間を与えて fuzzing を行った．

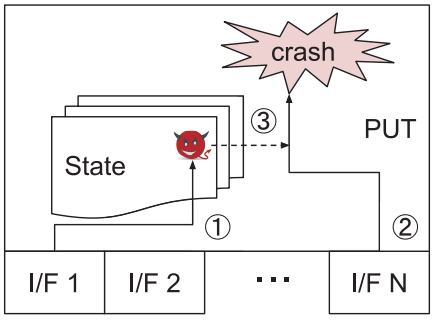

図 13 依存関係にあるインタフェースからの入力によるクラッシュの発生
Fig. 13 A crash due to inputs from dependent interfaces.

複数のインタフェースに対して同時に fuzzing を実行する *proposal* では，探索空間が拡大しているため，単一のインタフェースに対してのみ fuzzing を行う場合と比較して，インタフェースあたりの探索試行回数は減少していると考えられる．*default/multi* は `ethernet_input` のみを対象としており，*proposal* は同インタフェースを対象インタフェースに含んでいるため，原理的には，*default/multi* において検出可能な不具合は，*proposal* においても検出可能なものと思われる．ゆえに，テスト時間を延ばすことにより，既存手法のみで検出された不具合が，提案手法でも検出される可能性がある．ただし，現実的な状況下では，テスト工程にかけられる工数が限られている場合も少なくない．限られた時間内で，重要度の高いインタフェースに対する優先的なテストが求められる場合は，提案手法における対象インタフェース数を削減することや，既存手法を併用することも検討すべきである．

第 2 の可能性について考察する．*default* で用いたテストドライバと，*proposal* で用いたテストドライバには，いくつかの実装上の差異が存在する．主な差異は，以下の 2 点である．

- TCP/UDP アプリケーションを模したドライバとの疎通を図るため，初期シードの送信元エントリを ARP テーブルに追加している．
- `sys_now` への入力によりシステム時刻の経過が発生するため，それにともなうタイムアウトの確認処理を呼び出している．

これらの差異が，一部不具合の未検出につながった可能性は否定できない．この可能性の分析については，次節において述べる．

4.4.2 議論

本実験では，表 2 に示した 4 つのインタフェースを対象インタフェースとして，lwIP に対する fuzzing を試みた．本実験における対象インタフェース選定のねらいは，4.2 節において述べたとおりである．実際に，既存手法と比較して探索範囲を拡大し，新たな不具合を検出できたことから，本実験における対象インタフェースの選定は，目的に対して妥当であったと考えられる．一般に，それぞれの対象インタフェースによって駆動されるコードが異なる場合には，より多くのインタフェースを対象とすることで，探索範囲の拡大が期待できる．そのため，fuzzing による探索範囲を厳密に最大化するという目的の下では，最適な対象インタフェースは検討の余地がある．ただし，提案手法にのっとりインタフェース間の依存関係を考慮した場合でも，対象インタフェースの増加による探索範囲の拡大は，後述するテストケースの複雑さや，テスト効率とのトレードオフであることを留意すべきである．

本実験において検出されたクラッシュが，lwIP を搭載した実世界のシステムにおいて再現することは，検証してい

ない．本実験では外来の Ethernet フレームだけでなく，アプリケーションが送出する入力や，システム時刻も fuzzing の入力として与えているため，実システム上で再現するためには，非常に詳細なシステム制御が必要となる．しかしながら，本実験において各クラッシュを発生させた一連の入力系列は，すべてデータとして保存されており，テストドライバが動作可能な環境上であれば，容易に再現や動作の検証が可能である．また，4.2 節において述べたとおり，lwIP の開発者の視点から考えると，本実験における対象インタフェースはすべて，任意のデータが流入しうるものである．そのため，本実験において検出された不具合が，実世界において再現することのない不具合であるとは考えにくい．

前節では，既存手法のみによって検出された不具合が存在した要因について，2 つの可能性を提示した．このうち，第 2 の可能性については，実際に，*proposal* のテストドライバから該当する処理を取り除いたものをそれぞれ用意し，評価実験と同様の条件のもとで，それぞれ 1 回の fuzzing 試行を行った．その結果，タイムアウトの確認処理を呼び出さなかった場合において，既存手法のみで検出された 2 種類の不具合のうち，1 種類の検出が確認された．しかし，この結果のみから，タイムアウト確認処理の呼び出しが特定の不具合を完全に隠匿したと結論づけるのは早計である．たとえば，以下のような可能性が考えられる．

- タイムアウト確認処理の呼び出しの有無により，特定の不具合の発生しやすさが変化した．
- タイムアウト確認処理の呼び出しの有無により，fuzzer による探索の指向が変化した．

また，一般に，greybox fuzzer の入力生成にはある程度のランダム性がある．追加実験はただ 1 度の試行にとどめており，より厳密な議論を行うためには，より多くのデータを収集する必要がある．詳細な分析やさらなる追加実験については，今後の課題とする．

本論文においては，適用プロセスを含めた提案手法の具体的な論証に焦点を当て，lwIP という単一の対象へ適用した結果を示したため，本手法の適用可能性や効果が一般的なものであるかについては疑問の余地が残る．しかしながら，提案手法は具体的なミドルウェアの性質に依存しないきわめて一般的な要素から構成されており，任意のミドルウェアに対して適用可能であると考えられる．また，対象がミドルウェアである以上，少なくとも上位レイヤならびに下位レイヤに存在する外部コンポーネントとの相互作用は発生すると考えられる．ゆえに単一のインタフェースや一方のレイヤからのみ fuzzing を行っていた場合，提案手法にのっとることで PUT をより広範に探索できることが大いに期待される．

一方，提案手法を適用することで，テストケースが複雑化するというデメリットも存在する．3.5 節で述べたとおり，fuzzer は単一のデータストリームをテストケースとして生成する．fuzzing の実行中はテストドライバがそれを解釈し PUT へ入力するが，fuzzing の実行後にその結果を分析する過程では，人がその入力を解釈する必要がある．その際，単一のインタフェースから入力を与えた場合と比較して，複数のインタフェースからいくつもの入力を与えた場合のテストケースは，より解釈が困難なものとなりうると考えられ，クラッシュを発生させるテストケースが存在すると判明しても，その原因究明が困難となる．しかしながら，fuzzing，特に CGF の場合，クラッシュを発生させる入力が 1 つでも発見されれば，同一のクラッシュを発生させる異なるテストケースを生成することは比較的容易である．実際，評価実験で使用した fuzzer である AFL は，同一の箇所でクラッシュが発生する入力を探索する crash exploration という機能を備えている．このような機能を活用することで，原因究明の困難さを緩和することができると考えられる．

5. 関連研究

5.1 fuzzing における探索範囲の拡大

fuzzing による探索範囲を拡大することを目的とし，さまざまな研究が行われてきた．CGF に限って言及すると，2 つのアプローチに大別できる．

最適化アプローチ 最適化アプローチは，主に CGF におけるシードの変異を何らかの方法で最適化することで，より短時間の fuzzing で広範囲を探索可能にすることを試みるアプローチである．主要な研究としては，Böhme らによる AFLFast [19]，AFLGo [20]，Lemieux らによる FairFuzz [21]，Gan らによる CollAFL [22]，Nagy らによる UnTracer [23]，Chen らによる Angora [24]，Yue らによる LearnAFL [25] などがあげられる．

組合せアプローチ 組合せアプローチは，CGF と何らかの手法を組み合わせることで，CGF による探索を支援する手法である．主要な研究としては，Stephens らによる Driller [26]，Rewat らによる VUzzer [27]，Wang らによる SAFL [28]，Aschermann らによる REDQUEEN [29]，Yun らによる QSYM [30] などがあげられる．

本研究における探索範囲拡大のアプローチは，fuzzing 研究の本流であるこれらのいずれとも異なる．1 章において述べたとおり，これらの研究では比較的シンプルな対象をベンチマークとして手法の評価を行っている．一方，本研究ではそれらのベンチマークに用いられているプログラムとは異なる性質を持つミドルウェアに研究のスコープを絞っている．ミドルウェアは既存研究のベンチマークとは異なり，多様な外部コンポーネントと複雑に相互作用しながら動作する．本研究はそのようなプログラムの開発者が fuzzing を実行する際，より探索範囲を拡大するための一手法を提案するものである．

5.2 インタフェース間の依存関係を考慮した fuzzing

本研究と同様に，ライブラリなど，複数のインタフェースを持つプログラムに対して fuzzing を行う場合，インタフェース間の依存関係に着目すべきであると主張している既存研究はいくつか存在する．

Han ら [31] は，ordering dependence と value dependence という API 間の依存関係を定義している．前者は，一方の API は必ず他方の API の後に呼び出されるべきであるという呼び出し順序の依存関係であり，後者は，一方の API の返り値が他方の API の引数となるべきであるという値の依存関係である．両者は確かに API 間に存在する妥当な依存関係であると考えられる．本研究において，Han らの主張する依存関係は，テストドライバの作成者によって適切に満たされることを前提としている．また，本研究では，Han らの考慮していない依存関係を新たに定義し，fuzzing における探索範囲の拡大に活用した．

Pailoor ら [10] は，OS に対して効果的な fuzzing を実行するための初期シードを自動生成する研究を行った．その際，OS が提供するシステムコール間の依存関係として，explicit dependency と implicit dependency を定義している．explicit dependency は Han ら [31] の研究における value dependence と同様であり，implicit dependency は本研究における陰な依存関係の定義と同様である．システムコールをインタフェースとした上位レイヤからのみ fuzzing を実行しており，本研究のように，下位に存在する外部コンポーネントからの入力を考慮していない．

Jang ら [32] は，ユニットテストをベースとし，C/C++ ライブラリの fuzzing に必要なテストドライバと初期シードを自動生成する研究を行った．ライブラリが提供する API 間の依存関係として，本研究における陰な依存関係と同様のものに着目しており，それにより生じる API 呼び出しの順序制約にも言及している．ただし Pailoor ら [10] と同じく，下位に存在するコンポーネントからの入力を考慮していない．また，ユニットテストをベースとしているため，1 度の fuzzing の実行において対象とする API はただ 1 つであり，複数の API に対して fuzzer が生成した入力を与えた際の挙動はテストしていない．

既存研究において言及される依存関係はさまざまだが，いずれも PUT が提供する API のみをデータが流入するインタフェースとして扱っており，外部コンポーネントが提供するインタフェースの呼び出しによるデータの流入については言及されていない．本論文におけるケーススタディのように，非信頼な外部コンポーネントの管理するデータが流入しうるインタフェースである場合には，そのようなインタフェースも fuzzing の対象とすべきであり，PUT が提供するインタフェースと同様にインタフェース間の依存関係を議論すべきである．

6. おわりに

本研究では，多くの既存研究で用いられているベンチマークとは異なる性質を持つソフトウェアとしてミドルウェアを取り上げ，その開発者がより効果的に fuzzing を適用するためのプロセスを提案した．

提案手法では，対象とするインタフェース間の依存関係に着目し，fuzzing による探索範囲を可能な限り拡大することを試みた．また，PUT が提供するインタフェースのみに着目するのではなく，外部コンポーネントが提供するインタフェースからの情報流入まで含め，fuzzing の対象とするインタフェースをより一般化することで，より複雑な不具合の検出を試みた．

組込み向け TCP/IP スタック lwIP を対象としたケーススタディでは，開発者らがすでに実行した Ethernet フレームのみによる fuzzing と比較して，提案手法を適用することによりカバレッジを平均で 2 割から 3 割，最大で 7 割程度向上させることに成功した．また，開発者らが認知していなかった新たなバグを検出し，不具合報告とテストドライバの提供を行うことで，品質向上へ貢献した．

本論文に記載したケーススタディでは，CGF による TCP プロトコルの探索困難性を，初期シードを多く与えることで解決している．初期シードの作成はテスタに知識や労力を要求するものであり，不要であることが望ましい．そのため，入力系列を生成しつつ，その系列に含まれる単一の入力に対し，PUT の出力に基づいた制約を適用可能であるような fuzzer の試作や評価が，今後の課題と考えている．また，4.4 節において議論したように，既存手法によってのみ検出された不具合が存在した要因の詳細な分析や，提案手法の一般性の実証も，同様に今後の課題である．

参考文献

[1] 独立行政法人情報処理推進機構社会基盤センター：ソフトウェア開発データ白書 2018–2019 (2018).

[2] Klees, G., Ruef, A., Cooper, B., Wei, S. and Hicks, M.: Evaluating Fuzz Testing, *Proc. 2018 ACM SIGSAC Conference on Computer and Communications Security* (*CCS '18*), pp.2123–2138, ACM Press (2018).

[3] Miller, B.P., Fredriksen, L. and So, B.: An Empirical Study of the Reliability of UNIX Utilities, *Comm. ACM*, Vol.33, No.12, pp.32–44 (1990).

[4] Barr, E.T., Harman, M., McMinn, P., Shahbaz, M. and Yoo, S.: The Oracle Problem in Software Testing: A Survey, *IEEE Trans. Softw. Eng.*, Vol.41, No.5, pp.507–525 (2015).

[5] DeMott, J., Enbody, R. and Punch, W.: Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing, *BlackHat and Defcon* (2007).

[6] Zalewski, M.: American fuzzy lop (online), available from ⟨https://lcamtuf.coredump.cx/afl/⟩ (accessed 2020-03-17).

[7] LLVM Developer Group: libFuzzer – A library for coverage-guided fuzz testing. – LLVM 10 documen-

tation (online), available from ⟨https://llvm.org/docs/LibFuzzer.html⟩ (accessed 2020-03-17).
[8] Rebert, A., Cha, S.K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., Franco, C.I., Científicas, I. and Brumley, D.: Optimizing Seed Selection for Fuzzing, *23rd USENIX Security Symposium* (*USENIX Security 14*), pp.861–875, USENIX Association (2014).
[9] Wang, J., Chen, B., Wei, L. and Liu, Y.: Skyfire: Data-Driven Seed Generation for Fuzzing, *2017 IEEE Symposium on Security and Privacy* (*SP*), pp.579–594, IEEE (2017).
[10] Pailoor, S., Aday, A. and Jana, S.: MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation, *27th USENIX Security Symposium* (*USENIX Security 18*), pp.729–743, USENIX Association (2018).
[11] lwip-devel mailing list: lwip-devel – Fuzzing the lwIP stack (online), available from ⟨http://lwip.100.n7.nabble.com/Fuzzing-the-lwIP-stack-td26515.html⟩ (accessed 2020-04-20).
[12] GNU Project: GNU cflow (online), available from ⟨https://www.gnu.org/software/cflow/⟩ (accessed 2020-09-14).
[13] GNU Project: Grep – GNU Project – Free Software Foundation (online), available from ⟨https://www.gnu.org/software/grep/⟩ (accessed 2020-09-14).
[14] lwIP Project: lwIP – A Lightweight TCP/IP stack – Bugs: bug #57374, Assertion "this needs a pbuf in one piece!" failed [Savannah] (online), available from ⟨https://savannah.nongnu.org/bugs/?57374⟩ (accessed 2020-09-11).
[15] lwIP Project: lwIP – A Lightweight TCP/IP stack – Bugs: bug #57377, Assertion "pbuf_free: p->ref > 0" failed [Savannah] (online), available from ⟨https://savannah.nongnu.org/bugs/?57377⟩ (accessed 2020-09-11).
[16] lwIP Project: lwIP – A Lightweight TCP/IP stack – Bugs: bug #51447, Sequence number comparisons invoke implementation-defined behavior [Savannah] (online), available from ⟨https://savannah.nongnu.org/bugs/?51447⟩ (accessed 2020-09-11).
[17] lwIP Project: lwIP – A Lightweight TCP/IP stack – Bugs: bug #57375, Assertion "mss_local is too small" failed [Savannah] (online), available from ⟨https://savannah.nongnu.org/bugs/?57375⟩ (accessed 2020-09-11).
[18] lwip-devel mailing list: lwip-devel – Reporting crashes found by running a fuzzing campaign (online), available from ⟨http://lwip.100.n7.nabble.com/Reporting-crashes-found-by-running-a-fuzzing-campaign-td35038.html⟩ (accessed 2020-04-02).
[19] Böhme, M., Pham, V.-T. and Roychoudhury, A.: Coverage-based Greybox Fuzzing as Markov Chain, *Proc. 2016 ACM SIGSAC Conference on Computer and Communications Security* (*CCS'16*), pp.1032–1043, ACM Press (2016).
[20] Böhme, M., Pham, V.-T., Nguyen, M.-D. and Roychoudhury, A.: Directed Greybox Fuzzing, *Proc. 2017 ACM SIGSAC Conference on Computer and Communications Security* (*CCS'17*), pp.2329–2344, ACM Press (2017).
[21] Lemieux, C. and Sen, K.: FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage, *Proc. 33rd ACM/IEEE International Conference on Automated Software Engineering* (*ASE 2018*), pp.475–485, ACM Press (2018).
[22] Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z. and Chen, Z.: CollAFL: Path Sensitive Fuzzing, *2018 IEEE Symposium on Security and Privacy* (*SP*), pp.679–696, IEEE (2018).
[23] Nagy, S. and Hicks, M.: Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing, *2019 IEEE Symposium on Security and Privacy* (*SP*), pp.787–802, IEEE (2019).
[24] Chen, P. and Chen, H.: Angora: Efficient Fuzzing by Principled Search, *2018 IEEE Symposium on Security and Privacy* (*SP*), pp.711–725, IEEE (2018).
[25] Yue, T., Tang, Y., Yu, B., Wang, P. and Wang, E.: LearnAFL: Greybox Fuzzing With Knowledge Enhancement, *IEEE Access*, Vol.7, pp.117029–117043 (2019).
[26] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C. and Vigna, G.: Driller: Augmenting Fuzzing Through Selective Symbolic Execution, *Proc. 2016 Network and Distributed System Security Symposium*, Internet Society (2016).
[27] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C. and Bos, H.: VUzzer: Application-aware Evolutionary Fuzzing, *Proc. 2017 Network and Distributed System Security Symposium*, Internet Society (2017).
[28] Wang, M., Liang, J., Chen, Y., Jiang, Y., Jiao, X., Liu, H., Zhao, X. and Sun, J.: SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing, *Proc. 40th International Conference on Software Engineering Companion Proceeedings* (*ICSE'18*), pp.61–64, ACM Press (2018).
[29] Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R. and Holz, T.: REDQUEEN: Fuzzing with Input-to-State Correspondence, *Proc. 2019 Network and Distributed System Security Symposium*, Internet Society (2019).
[30] Yun, I., Lee, S., Xu, M., Jang, Y. and Kim, T.: QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing, *27th USENIX Security Symposium* (*USENIX Security 18*), pp.745–761, USENIX Association (2018).
[31] Han, H. and Cha, S.K.: IMF: Inferred Model-based Fuzzer, *Proc. 2017 ACM SIGSAC Conference on Computer and Communications Security* (*CCS'17*), pp.2345–2358, ACM Press (2017).
[32] Jang, J. and Kim, H.K.: FuzzBuilder: Automated building greybox fuzzing environment for C/C++ library, *Proc. 35th Annual Computer Security Applications Conference* (*ACSAC'19*), pp.627–637, ACM Press (2019).

伊藤 弘将 （学生会員）

2020 年名古屋大学大学院情報学研究科博士前期課程修了．現在，同博士後期課程在学中．組込みソフトウェアのテスト，セキュリティ等の研究に従事．

松原 豊 （正会員）

2006 年名古屋大学大学院情報科学研究科博士前期課程修了．同大学院情報科学研究科附属組込みシステム研究センター研究員，助教等を経て，2018 年より名古屋大学大学院情報学研究科准教授．2015 年 4 月～9 月米国ワシントン大学およびカリフォルニア大学サンディエゴ校客員研究員．組込みシステム向けのリアルタイム OS，リアルタイムスケジューリング理論，システム安全，セキュリティ等の研究に従事．博士（情報科学）．IEEE，電子情報通信学会，自動車技術会各会員．

高田 広章 （正会員）

1988 年東京大学大学院理学系研究科情報科学専攻修士課程修了．同専攻助手，豊橋技術科学大学情報工学系助教授等を経て，2003 年より名古屋大学大学院情報科学研究科情報システム学専攻教授．2014 年より名古屋大学未来社会創造機構教授．同大学大学院情報学研究科教授・附属組込みシステム研究センター長を兼務．リアルタイム OS，リアルタイムスケジューリング理論，組込みシステム開発技術等の研究に従事．オープンソースのリアルタイム OS 等を開発する TOPPERS プロジェクトを主宰．博士（理学）．IEEE，ACM，電子情報通信学会，日本ソフトウェア科学会，自動車技術会各会員．本会フェロー．