# Multi-target Coverage-based Greybox Fuzzing 手法の提案

市川 正美[1,a)]　須崎 有康[1,b)]

**概要**：近年，Fuzzing はアプリケーションソフトウェアにとどまらず，Linux カーネルや firmware を含むシステムソフトウェアにも適用され，脆弱性発見のための有力な手法として広く活用されている．中でも，実行時のコードカバレッジ情報を活用する Coverage-based Greybox Fuzzing が主流である．従来の Fuzzing 手法では，主に単一のソフトウェアを対象とし，他のソフトウェアとの協調動作には着目してこなかった．しかし，現代のシステムソフトウェアは，RISC-V における OpenSBI や ARM における OP-TEE のような firmware と OS が特定のインターフェースを通じて協調的に動作する構成が一般的である．本研究では，OS と firmware による協調動作を行うアーキテクチャを対象とした Fuzzing 手法について検討する．特に，協調する各ソフトウェアのコードカバレッジをフィードバックとして活用することで，従来の単一対象型 Fuzzing よりも深くシステムの探索を行う Fuzzing 手法を提案する．そして，本提案手法を実装した Multi-target Coverage-based Greybox Fuzzer *MTCFuzz* を開発し，その有効性を評価した．

**キーワード**：Corverage-based Greybox Fuzzing，System-level Fuzzing，Firmware Security

# A Proposal for Multi-target Coverage-based Greybox Fuzzing

Masami Ichikawa[1,a)]　Kuniyasu Suzaki[1,b)]

***Abstract:*** In recent years, fuzzing has been applied not only to application software but also to system software, including the Linux kernel and firmware, and has been widely used as a powerful method for vulnerability detection. Coverage-based Greybox fuzzing, which utilizes runtime code coverage information, is the most popular. Traditional fuzzing methods primarily target a single piece of software and do not focus on its cooperative behavior with other software. However, modern system software typically consists of firmware and an operating system cooperating through a specific interface, such as OpenSBI for RISC-V or OP-TEE for ARM. This study investigates fuzzing methods for architectures in which the operating system kernel and firmware cooperate. In particular, we propose a fuzzing method that utilizes the code coverage of each cooperating piece of software as feedback to explore the system more deeply than conventional single-target fuzzing. We then developed a multi-target coverage-based Greybox Fuzzer called *MTCFuzz* that implements this method and evaluated its effectiveness.

***Keywords:*** Corverage-based Greybox Fuzzing, System-level Fuzzing, Firmware Security

## 1. はじめに

近年，Fuzzing はアプリケーションソフトウェアのみならず，Linux カーネルや firmware など低レイヤー分野でも広く活用されている．特に，実行時のコードカバレッジをフィードバックとして利用する Coverage-based Greybox Fuzzing は効率的に探索を行えることから主流となっている．オープンソースの開発では OFF-FUZZ プロジェクト[*1] により，2025 年 3 月時点では 1,000 件以上のプロジェ

[1] 情報セキュリティ大学院大学
Graduate School of Information Security, Institute of Information Security
[a)] mgs243501@iisec.ac.jp
[b)] suzaki@iisec.ac.jp

[*1] https://github.com/google/oss-fuzz/tree/master/

クトに対して継続的に Fuzzing が実行されている．Linux カーネルにおいては syzkaller による Fuzzing が継続的に行われている [*2] ．Fuzzing はソフトウェアテストと同様に，Whitebox，Blackbox，Greybox の 3 種類の分類方法がある．近年ではプログラム実行時のコードカバレッジ情報を活用する Greybox Fuzzing 手法が広く用いられている．主な Greybox Fuzzer として，アプリケーションソフトウェアを主な対象とする AFL++ [1]，LibFuzzer[2] では様々なアプリケーションやライブラリを対象に Fuzzing を行うことができる．オペレーティングシステム (OS) を対象とした Fuzzer では syzkaller[3] が広く使われており，Linux カーネルだけでなく FreeBSD や NetBSD などの BSD 系 OS の他，Android や Windows などの Fuzzing をサポートしている．

Fuzzer には特定のソフトウェアを Fuzzing 対象としたツールだけでなく，様々なソフトウェアを対象として Fuzzing を行うことができるものがある．しかし，Fuzzing 実行時の Fuzzing 対象となるソフトウェアは 1 つであり，Fuzzing 対象ソフトウェアと協調して動作するソフトウェア，データベースソフトウェアや firmware などのコードカバレッジ情報は利用されない．しかし，現代のシステムソフトウェアは，複数のソフトウェアが特定のインターフェースを通じて協調的に動作する構成が一般的である．例えば，RISC-V アーキテクチャにおける Linux と OpenSBI の協調 [4]，ARM アーキテクチャにおける TrustZone[5] における Linux が動作する Normal World と Trusted OS が動作する Secure World などが挙げられる．これらのアーキテクチャでは OS と firmware はそれぞれ独立したプロジェクトにより開発され，仕様によって定められたインターフェースを介してコミュニケーションが行われる．そのため，本来はエラーとなるべきデータが OS から送られたが，firmware 側の入力チェック処理が間違っているため処理が通常通りに実行されて，OS に制御が戻る場合がある．OpenSBI でもこのようなバグは修正されている [6]．このようなバグにて，firmware がクラッシュする場合は firmware を対象とした Fuzzing でバグを検出することができる．しかし，firmware 側でクラッシュが発生せずに OS 側で状態の不整合が原因でクラッシュが発生するような場合は firmware を対象とした Fuzzing ではバグを検出しにくい可能性がある．

そこで我々は，協調動作する複数のソフトウェアを同時に対象とし，それぞれのコードカバレッジをフィードバックとして活用することで，従来よりも深くシステムを探索できる Fuzzing 手法を提案することを目的とする．本稿では，複数のソフトウェアを対象とした Fuzzing 手法と提案手法によりコードカバレッジを向上した評価結果を示す．

我々の貢献は以下の通りである．

- Fuzzing 対象ソフトウェアのソースコードやバイナリを変更せずにコードカバレッジ計測可能とした
- QEMU を Fuzzing 実行環境とすることで，実行中の命令をメモリ上でトレースし，コードカバレッジ計測を実現した
- 複数アーキテクチャ (ARM・RISC-V) にまたがる実証評価による汎用性を確認した
- 協調動作するソフトウェアのコードカバレッジをフィードバックに用いることで探索性能を向上できることを示した
- 提案手法を実装した Multi-target Coverage-based Greybox Fuzzer (*MTCFuzz*) を開発した

*MTCFuzz* は GitHub[*3] にて OSS として公開した．

## 2. 背景

### 2.1 Greybox Fuzzing

Greybox Fuzzing では Whitebox Fuzzing と Blackbox Fuzzing の中間に位置する手法である．Whitebox Fuzzing では Fuzzing 対象ソフトウェアのソースコードの解析を行い，そのプログラムのコードフローやデータ構造なども含めた知識を利用して Fuzzing を行うが，Blackbox Fuzzing は Fuzzing 対象ソフトウェアの内部構造には着目せずに Fuzzing を進める．Greybox Fuzzing では，Whitebox Fuzzing のようにソースコードの解析までは行わないが，コードカバレッジの計測など軽量な解析を行う．Zalewski による AFL[7] にて採用されたソースコードのコンパイル時にコードカバレッジの計測コードを埋め込み，Fuzzing 実行時にコードカバレッジを収集してテストケース生成に役立てる Coverage-based Greybox Fuzzing 手法 (CGF) は [1], [2] に代表されるように Greybox Fuzzer で広く採用されている ．

基本的な Fuzzing の処理の流れは Algorithm 1 に示すとおりである．ここでは，初期シード $S$ から順に入力を生成し，プログラムを実行してクラッシュの有無を記録するという単純な反復処理を行う．これに対し，Greybox Fuzzing では，より効率的に新しい挙動を探索するための工夫が加えられている．その代表的な実装例を Algorithm 2 に示す．以下では，Algorithm 1 と Algorithm 2 の主な相違点を説明する．

Algorithm 2 のアルゴリズムでは初期シード $S$ を受け取り，1 行目でプログラムがクラッシュした場合にクラッシュしたシードを保存する集合 $S_\dagger$ を空で初期化する．2 行目から Fuzzing ループを実行する．3 行目の ChooseNext は初期シードからミューテーション対象のデータを選択する．4 行目の MutateInput にて選択したシードにミュー

projects

[*2] https://syzkaller.appspot.com/upstream

[*3] https://github.com/masami256/MTCFuzz

**Algorithm 1** Basic Fuzzing Algorithm

**Input:** Seed input $S$

```
1: S† ← ∅
2: repeat
3:     s ← ChooseNext(S)
4:     s′ ← MutateInput(s)
5:     t ← Execute(s′)
6:     if t is crashed then
7:         S† ← S† + s′
8:     end if
9: until Timeout reaches or all the seeds are checked
```

**Output:** Crash inputs $S_\dagger$

**Algorithm 2** Greybox Fuzzing Algorithm

**Input:** Seed input $S$

```
1: S† ← ∅
2: repeat
3:     s ← ChooseNext(S)
4:     p ← AssignEnergy(s)
5:     for i from 1 to p do
6:         s′ ← MutateInput(s)
7:         t ← Execute(s′)
8:         if t is crashed then
9:             S† ← S† + s′
10:        else if IsInteresting(t) then
11:            S ← S + s′
12:        end if
13:    end for
14: until Timeout reaches or all the seeds are checked
```

**Output:** Crash inputs $S_\dagger$

テーションを施し，5 行目の Execute でプログラムを実行する．もし，ミューテーションしたシード $S'$ でプログラムがクラッシュした場合は $S_\dagger$ にそのシードを追加する．本手順を設定した時間実行するかシードを使い切るまで実行する．

一方 Greybox Fuzzing のアルゴリズムは Algorithm 2 のようになる．Algorithm 1 との違いは 2 点あり，1 点目は 4 行目の AssignEnergy にて ChooseNext で選択したシードに対するテスト回数を計算し，5 行目の for ループにて $p$ 回のテストを行う点である．Algorithm 2 ではシードはランダムに選ぶだけであるが，Greybox Fuzzing ではシードによって重み付けを行い，その重みに応じたテスト回数を割り当てることで，新しいコードカバレッジの検出やシグナルの検出などが見られたシードなど Fuzzing 実行時に有効と判断されたシードを重点的にテストする．2 点目は，10 行目の IsInteresting であり，新しいコードカバレッジの検出やシグナルの受信などがあるかを確認し，興味深い挙動が見つかった場合は，テストケース $s'$ をシードに追加し，そのテストケースをシードとして Fuzzing を行えるようにする．複数のソフトウェアを対象として Fuzzing を行う場合，複数のソフトウェアから計測したコードカバレッジを IsInteresting にて解析する．

### 2.2 コードカバレッジ

CGF ではコードカバレッジを Fuzzing 実行時のフィードバックとして受け取り，IsInteresting にてテストケース $s'$ で新しいコードカバレッジに到達したかを確認する．そのため，コードカバレッジを計測するための手法は Fuzzing にとって重要となる．コードカバレッジを計測する方法としてはコンパイラによるコードカバレッジ計測機能を利用する方法 [8], [9], コンパイラの機能を拡張し，独自のコードカバレッジ計測コードを埋め込む手法 [1], [7]，ソフトウェアが自身の機能としてコードカバレッジ計測コードを埋め込む手法 [10]，QEMU などの仮想環境を利用し，コードカバレッジを計測する方法 [11]，ハードウェアのトレース機能を利用してコードカバレッジを計測する手法 [12] など様々な手法がある．これらの手法にはそれぞれ利点があるが，Linux カーネルと firmware のように独立したプロジェクトによって開発されたソフトウェアを同時に Fuzzing を行う場合には，コードカバレッジを計測する方法はソフトウェアによらず統一されている必要がある．コードカバレッジの粒度が異なると複数ターゲット間でのフィードバック統合が困難になる．

## 3. 関連研究

### 3.1 Titan

Huang らによる Titan[13] では，Multi-target Directed Greybox Fuzzing と称する多数のターゲットを一度に効率的に Fuzzing するための手法を提案している．Titan が扱うターゲットは複数のソフトウェアではなく，ソフトウェアに含まれる関数やパスである．Huang らはターゲット間の関連性を 1) 重なり (overlap)，2) 矛盾 (conflict)，3) 独立 (independent) の 3 種類に分類し，ターゲットをテストするのに適したシードの選択を行う．

### 3.2 MultiFuzz

MultiFuzz[14] は Chesser らによって提案された firmware の Fuzzing 手法で, firmware 複数の周辺機器 (UART, GPIO など) に対して Memory Mapped IO(MMIO) アクセスが行われる．このため，実行パスによっては異なるタイミング，順序で複数の周辺機器にアクセスを行う．この場合，通常の Fuzzing 手法ではシードのミューテーションが他の周辺機器の動作に影響を与える，必要なタイミングで必要なデータが共有されないといったことが起こりうる．このような問題の解決のため，単一の入力ではなく複数の入力を周辺機器の単位に分割して管理を行い，複数機器間で適切なデータが設定された状態にして Fuzzing を行う．

### 3.3 LibAFL QEMU

Fioraldi らによる LibAFL QEMU[15] は，LibAFL[16] を拡張し，そのモジュール性と QEMU を組み合わせた

Fuzzing フレームワークである．このフレームワークは，通常はアプリケーションとして動作する QEMU をライブラリとして利用可能にし，外部コードから自在に制御できるようにしている．さらに，QEMU のスナップショット復元処理を高速化し，スナップショットを用いた Fuzzing への最適化を実現している．また，コードカバレッジは QEMU のトレース機能を用いて計測する．

### 3.4 関連研究のまとめ

Titan[13] は，複数ターゲットを効率的に探索する手法を提案しているが，対象は単一ソフトウェア内の関数やパスに限られる．MultiFuzz[14] は，複数の周辺機器を持つ単一 firmware に対し，機器ごとに入力を分割管理することで依存関係を考慮した Fuzzing を可能にしているが，複数ソフトウェア間の協調動作は対象としていない．LibAFL QEMU[15] は，QEMU を Fuzzing の基盤環境として採用しており，複数の CPU アーキテクチャに対応し，トレース計測も可能である．しかし，その設計思想は基本的に単一ソフトウェアの効率的な Fuzzing である．

既存研究には，特定の構成を対象に複数ソフトウェアを考慮する手法や，汎用的な Fuzzing フレームワークを実装して複数ソフトウェアに適用可能なものも存在する．しかし，それらは Fuzzing の前提として複数ソフトウェアを対象としているわけではない．そこで我々は，複数のソフトウェアで構成された 1 つのシステムを Fuzzing 対象とし，その協調動作部分も考慮することで，より効果的な Fuzzing が可能であると考えた．

我々は，以下のリサーチクエスチョンに焦点を当て，提案手法の評価実験を実施した．

- **RQ1**：複数ソフトウェアのコードカバレッジを活用することで，単一ソフトウェアよりも高い探索性能が得られるか？
- **RQ2**：ARM や RISC-V など複数アーキテクチャに対応したコードカバレッジ計測手法は，マルチターゲット Fuzzing に有効か？

## 4. 提案手法: Multi-target Coverage-based Greybox Fuzzing

本章では，我々の提案手法である Multi-target Coverage-based Greybox Fuzzing について述べる．本提案手法では，複数のソフトウェアのコードカバレッジを計測し，それらを統一的に扱うことで従来手法よりも効率的な探索を可能にする．

我々は本提案手法を実装した Multi-target Coverage-based Greybox Fuzzer(*MTCFuzz*) を開発した．

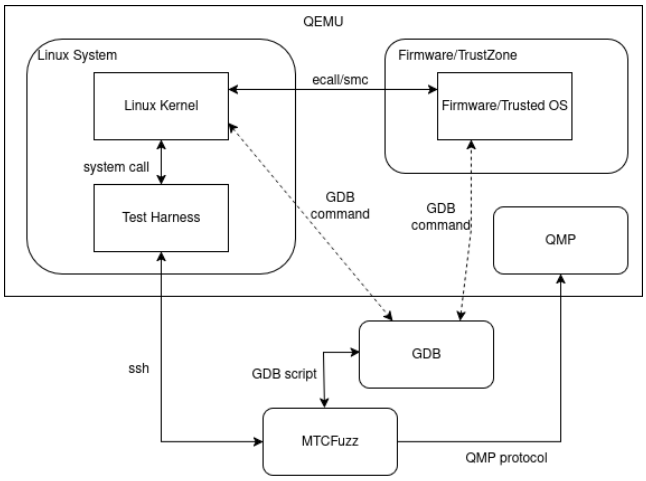


**図 1** MTCFuzz の構成図．Fuzzer がテストケースを生成し，テストハーネスにテストケースを渡す，もしくは GDB スクリプトを利用して GDB よりレジスタのデータを書き換え Fuzzing を実行する．Fuzzer は QMP プロトコルを利用してスナップショットの保存・復元，コードカバレッジの計測を行う．

### 4.1 MTCFuzz の設計

我々の Fuzzer の設計を 1 に示す．Fuzzing 対象のシステムは QEMU 上で実行する．Fuzzing フレームワークが生成したテストケースをテストハーネスに送りテストハーネスがそのテストケースを利用する方法と，GDB スクリプトを用いてテストケースをレジスタやメモリに書き込む方法をサポートしている．

### 4.2 QEMU によるコードカバレッジの計測

QEMU[17] はオープンソースのエミュレータであり，x86-64，ARM，RISC-V など様々な CPU アーキテクチャをサポートしている．QEMU は，エミュレーションの開始から終了までに実行された CPU の命令を出力する機能があるが，Fuzzing においてはコードカバレッジを計測する期間はテストケースを実行中のみにするのが望ましい．このため，我々は QEMU に備わる JSON ベースのプロトコルである QMP プロトコル [18] にトレースの ON/OFF を制御する API を実装し，任意のタイミングでコードカバレッジの計測を行うことを可能とした．QEMU はゲスト環境で実行される機械語命令を QEMU のアーキテクチャ非依存な中間命令列である TCG (Tiny Code Generator) に変換し，TCG をホスト向けの機械語命令に変換することでゲスト環境での機械語命令をホストで実行する．提案手法ではコードカバレッジの計測をゲストの機械語命令を TCG に変換するタイミングで行っている．これにより，QEMU がサポートするすべての CPU アーキテクチャでコードカバレッジを計測できることを可能とした．MTCFuzz では QEMU に対して 200 行以下のパッチを作成し，コードカバレッジ計測機能を実現した．パッチの内容は，コードカバレッジを記録する関数の追加と関数を呼び出す処理の追

加である．

提案手法の利点は次の通りである．

- Fuzzing 対象ソフトウェアの種類を問わず QEMU によりコードカバレッジを計測できる．QEMU がトレースを行うため，トレースの粒度は統一されたものとなる．
- Fuzzing 対象システムの CPU アーキテクチャに依らずコードカバレッジを計測できる．
- コードカバレッジを計測するために Fuzzing 対象のソフトウェアのソースコードやバイナリに対する変更が不要．例えば Linux カーネルの Fuzzing であれば，KCOV を有効化するということが必要になる場合があるが，提案手法では，KCOV が有効になっていない Linux ディストリビューションのカーネルをそのまま利用することができる．
- ソースコードがないプロプライエタリのソフトウェアであってもコードカバレッジを計測できる．ソースコードが無い場合，Fuzzing 後のレポート作成時にトレースログからどのコードが何回実行されたかという情報を得ることはできないが，CGF の実行としては問題がない．
- 我々が作成したパッチでは既存のコードに対する変更は 2 行であり，将来 QEMU のバージョンアップを行う場合もパッチの適用を行いやすい

### 4.3 SUT の状態管理

Fuzzing では OS や firmware の起動，テストハーネスの実行，コードカバレッジの収集などの処理を行う．このうち，OS の起動は時間がかかる処理である．また，OS の起動後に Fuzzing を行える状態にセットアップする必要がある．そのため，OS の起動とセットアップにかかる時間を削減することは Fuzzing の実行効率にとって重要である．従って，セットアップ処理の時間を削減するにはシステム全体の状態を保存し，状態を復元できることが望ましい．また，過去のテストによりシステムの状態が変わった場合に，継続して Fuzzing を行うと，過去のテスト結果の影響によりシステムがクラッシュする可能性がある．このようなバグはシステムの状態に依存するため，再現性に乏しい．このようなクラッシュを回避するにはテストケースの実行はシステムの状態が汚染されていない状況で行うのが望ましい．提案手法では，QEMU のスナップショット機能を用い，セットアップが完了した段階でシステム全体のスナップショットを作成し，テストケースの実行後にスナップショットを復元することで OS や firmware などの状態を初期状態に戻すことで，SUT 全体の再起動にかかる時間の短縮と過去の状態に依存したシステムのクラッシュを避けることを可能とした．

## 5. 評価実験

評価は，Intel(R) Core(TM) i7-14700(メモリ 64GB) を搭載した物理ホスト上で，Fedora 42 を実行する環境において行った．また，評価環境には Docker を用い，そのコンテナ上で Ubuntu 24.04 を動作させた．

### 5.1 RQ1: コードカバレッジの活用による探索性能の向上

本節では，評価対象に Linux v6.16-rc1，OpenSBI v1.6 を用いた．また，Fuzzing は QEMU は 1 インスタンスのみで実行を行った．*MTCFuzz* は，Linux カーネルのコードカバレッジのみを利用するモード，Linux カーネルと OpenSBI のコードカバレッジの両方をフィードバックとして利用するモードを選択できるようにし，フィードバックの扱い方以外は同一の条件で実験を行った．本稿では，Linux カーネルのコードカバレッジのみをフィードバックとして利用した場合を *single*，Linux カーネルと OpenSBI 両方のコードカバレッジを含めた場合を *multi* と呼称する．

Linux 側では，カーネルに含まれる `sbi_ecall` を呼び出して `ecall` 命令を実行するカーネルドライバを実装した．このドライバに対しては，`ioctl` を用いてユーザ空間から `ecall` 命令を実行できるようにした．

#### 5.1.1 実験概要

**コード探索**：`lib/sbi/sbi_ecall_base.c` の `sbi_ecall_base_handler` に含まれる `switch` 文を対象として，*single* および *multi* の各モードにおける `case` のカバー率を測定する．当該関数には `switch` 文が 1 つあり，以下の 7 つの `case` が含まれる．

- `SBI_EXT_BASE_GET_SPEC_VERSION`
- `SBI_EXT_BASE_GET_IMP_ID`
- `SBI_EXT_BASE_GET_IMP_VERSION`
- `SBI_EXT_BASE_GET_MVENDORID`
- `SBI_EXT_BASE_GET_MARCHID`
- `SBI_EXT_BASE_GET_MIMPID`
- `SBI_EXT_BASE_PROBE_EXT`

**評価指標**：評価においては，*single* および *multi* の双方に初期シード値として 0 を設定し，1 回あたり 5 分間の実行を 1 セットとして 50 回の Fuzzing を実施した．評価指標としては，(i) すべての `case` をカバーした回数，ならびに (ii) 各試行におけるカバーされた `case` 数の平均値を用いた．

#### 5.1.2 OpenSBI base ecall のカバレッジ探索結果

**コード探索**：本評価の結果を以下に示す．(i) については Figure 2 に示す通り，すべての `case` を網羅した回数は *single* では 22 回，*multi* では 35 回であった．(ii) については，Fuzzing 1 セットあたりにカバーされた `case` 数の平

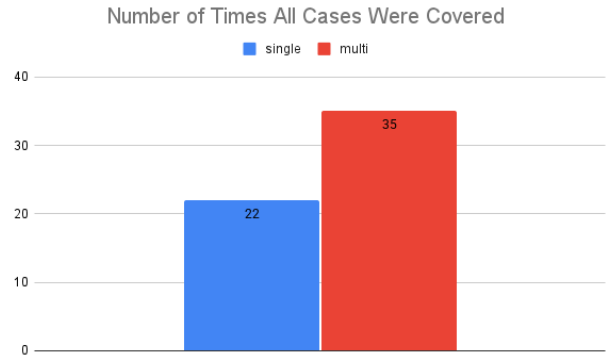


**図 2**　すべての case をカバーした回数

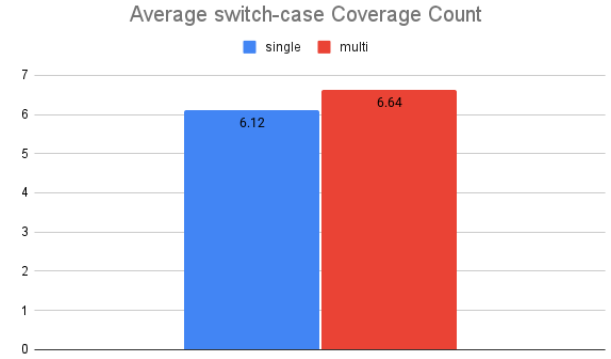


**図 3**　case の平均カバー数

均値を Figure 3 に示す.

1 セット辺りのテスト実行回数は Fuzzer のセットアップ時間も含めて，*multi* が 221 件，*single* が 222 件であったため，1 テストケースの実行にかかる時間は約 0.73 秒であった.

### 5.2　RQ2: アーキテクチャの違いに対する適用性評価

本節では，RQ2 に対する評価として ARM v8 TrustZone と RISC-V OpenSBI の 2 種類のアーキテクチャに対する適用性を評価する.

#### 5.2.1　ARMv8 TrustZone: 手法の適用

**適用環境**：OP-TEE のマニフェストリポジトリ[*4](実験時点での revision f92aacd1b103af137438f2e434303ffa5dff3d09) を用い，このマニフェストリポジトリに含まれる `qemu_v8.xml` を用いて Armv8-A 向けの Linux カーネル，ルートファイルシステム，Trusted OS，Trusted 一式を OP-TEE のマニュアルに沿ってビルドし，本提案手法が適用できることを評価した．主なソフトウェアのバージョンは，Linux カーネル 6.14.0，optee_os 4.7.0-rc1，trusted-firmware-a v2.13-rc0 である．ARMv8 アーキテクチャにおける TrustZone の構成を 4 に示す．本適用事例の実験において Fuzzing の実行環境設定のため，Buildroot の rootfs

[*4] `https://github.com/OP-TEE/manifest.git`

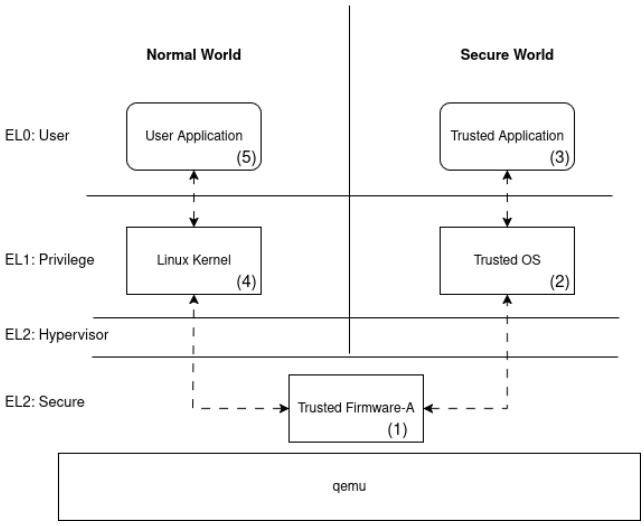


**図 4**　QEMU(arm64) 環境におけるシステム構成．Normal World の Linux と Secure World の Trusted Firmware-A および optee_os が特定のインターフェースを通じて協調動作する.

に対して dropbear パッケージの追加，Linux カーネルの起動時のコマンドラインにて kaslr の無効化，u-boot の起動時のディレイ時間を 0 に変更は行ったが，Linux カーネルや optee_os などのソースコードに対する変更は行っていない.

#### 5.2.2　ARMv8 TrustZone: 手法の適用結果

本適用事例の構成において，Linux のユーザーランドアプリケーションから optee_os を対象とした Fuzzing を行い計測した QEMU のトレースログに記録されたアドレスを表 1 に示す．4 内にて括弧で示した数字と 1 のソフトウェア列内にて括弧で示した数字は同じソフトウェアを表す．表 1 に示したアドレス範囲より，Fuzzing の対象外とした Trusted Application 以外のソフトウェアの実行が記録されていることが確認できる.

#### 5.2.3　RISC-V OpenSBI: 手法の適用

**適用環境**：Linux カーネル 6.16-rc1，OpenSBI v1.6 を対象として Fuzzing を行った．実験においてはコードカバレッジの計測範囲を絞り，Linux カーネルは OpenSBI 関連のコードのみ，OpenSBI はすべてのコードを対象とした．本実験においても Linux や OpenSBI のソースコードに対する変更は行っていない．また，コードカバレッジの計測範囲を Linux カーネルは `0xffffffff8001a512 - 0xffffffff8001b732` として `ecall` に関連する処理，OpenSBI は `0x80000000 - 0x80050000` として OpenSBI 全体を指定した.

#### 5.2.4　RISC-V OpenSBI: 手法の適用結果

本適用事例の構成において，Linux のユーザーランドアプリケーションから OpenSBI を対象とした Fuzzing を行い計測した QEMU のトレースログに記録されたアドレスを表 2 に示す．5 内にて括弧で示した数字と 1 のソフト

**表 1** ARMv8 TrustZone Fuzzing のトレースに現れたアドレス範囲に基づくソフトウェア分類

| アドレス範囲 | ソフトウェア |
|---|---|
| 0xe090000 – 0xe0a00c4 | Trusted Firmware-A (1) |
| 0xe100000 – 0xe19f4c8 | optee_os (2) |
| 0x40000400 – 0x402ab510 | Linux: FDT / Kernel Image / initrd |
| 0xaaaaab0e55d4 – 0xaaaae95ff1ac | Linux: ユーザー空間アプリケーション (4) |
| 0xffff814d3ac4 – 0xffff80008110f928 | Linux: カーネル空間 (5) |

**表 2** RISC-V OpenSBI Fuzzing のトレースに現れたアドレス範囲に基づくソフトウェア分類

| アドレス範囲 | ソフトウェア |
|---|---|
| 0x800004f8 – 0x8000a73e | OpenSBI (1) |
| 0xffffffff8001b2fc – 0xffffffff8001b340 | Linux: カーネル空間 (2) |

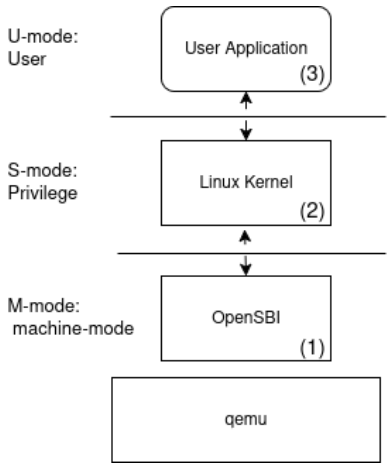


**図 5** QEMU(risc-v) 環境において Linux カーネルが Supervisor モードで動作し，OpenSBI が Machine モードで動作する構成．両者は Supervisor Binary Interface(SBI) を通じて連携する．

ウェア列内にて括弧で示した数字は同じソフトウェアを表す．表 2 に示したアドレス範囲より，Linux カーネルと OpenSBI それぞれ指定したアドレス範囲のトレースが計測できていることが確認できる．

## 6. 考察

### 6.1 RQ1: 複数ソフトウェアのコードカバレッジを活用することで，単一ソフトウェアよりも高い探索性能が得られるか？

5.1.2 より，評価指標 (i)，(ii) いずれも *multi* が *single* よりも良い結果となった．Linux カーネルの `sbi_ecall` は条件分岐の無い単純な関数であり，一度この関数を実行すると，その後にどのようなテストケースを利用しても `sbi_ecall` のコードカバレッジは向上しない．しかし，OpenSBI の `sbi_ecall_base_handler` には `switch` 文による分岐が存在するため，コードカバレッジをフィードバックとして活用し，生成されたテストケースの有効性を評価しながら Fuzzing を継続することが重要である．*single* では OpenSBI のコードカバレッジを利用しなかったため，コードカバレッジのフィードバックを活用しきれなかったが，*multi* では OpenSBI のコードカバレッジもフィードバックとして利用できたため，*single* よりも良質なテストケースを生成できたと考えられる．本実験の対象となる関数はサイズも小さく分岐も少ないため，Fuzzing 実行時間を長くすれば *single* と *multi* どちらも case のカバー率を上げることはできると考えられるが，提案手法では短時間の実行でコードカバレッジをより網羅することができると言える．

### 6.2 RQ2：ARM や RISC-V など複数アーキテクチャに対応したコードカバレッジ計測手法は，マルチターゲット Fuzzing に有効か？

5.2 にて ARMv8 TrustZone 環境と RISC-V 環境の 2 つの異なるアーキテクチャに対して提案手法の Fuzzing を行い，Fuzzing 対象ソフトウェアのコードを変更することなく，複数のソフトウェアのコードカバレッジ計測を行うことができた．提案手法では QEMU がターゲット向けの機械語から TCG に命令列への変換を行うタイミングでトレースを行うため，CPU アーキテクチャに依存せずにトレースを記録することを可能にすることができた．

### 6.3 今後の展望

我々の提案手法は QEMU 上での Fuzzing を前提としている．そのため，以下の制約が存在する．

- 1)QEMU が対応していない CPU アーキテクチャやハードウェア構成への適用が難しい
- 2)Fuzzing 対象のアーキテクチャと Fuzzer を実行するホスト環境の CPU アーキテクチャが違う場合，Intel®Virtualization technology (VT-x) などのハードウェアによる仮想化支援機能を利用できないため，エミュレーションによるオーバーヘッドが生じる
- 3) トレースには Fuzzing 対象機能とは無関係な実行ログも含まれるため，適切なフィルタリングが必要である

1 については QEMU に対してコードを実装することで解決できる可能性があるが，2 についてはエミュレーターの利用に伴う制限のため，解決し難い．そのため，Fuzzer の並列実行など Fuzzing の実行方法を工夫することによる効率の改善が考えられる．

3 の QEMU によるトレースでは，QEMU 内で動作するソフトウェアのトレースをすべて記録することができるが，記録されるトレースには Fuzzing 対象機能とは無関係な機能，例えば，OS のスケジューラーやメモリ管理機能などフィードバックとしては不要なログ，なども含まれる．そのため，アドレス範囲を指定しないと Fuzzing 対象と関係ない部分でコードカバレッジに変化があるため，関係ないアドレスはフィルタリングする必要がある．フィルタリングを適切に行わないと，IsInteresting での判定に影響を及ぼす．我々の Fuzzer ではアドレスをフィルタリングすることは可能だが，アドレス範囲はユーザーが調べて記述する必要があり，設定が煩雑である．そのため，機械的にフィルタリングを行える仕組みを導入することでフィルタリングの設定効率を改善できると考える．

今後は，これらの課題を解決することで，より多様なアーキテクチャやシステム構成への適用，そして効率的なマルチターゲット Fuzzing の実現を目指す．

## 7. まとめ

本研究では，OS と firmware のように独立して開発され協調動作する複数ソフトウェアを同時に対象とする Multi-target Coverage-based Greybox Fuzzing 手法を提案した．本提案手法を実装した Fuzzer *MTCFuzz* を開発し，提案手法は単一ソフトウェアを対象とした Greybox Fuzzing に比べて探索性能が向上することを確認した．

## 8. 参考文献

### 参考文献


[1] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.

[2] libFuzzer: a library for coverage-guided fuzz testing. LLVM 22.0.0git documentation — llvm.org. `https://llvm.org/docs/LibFuzzer.html`. [Accessed 25-07-2025].

[3] Google. GitHub - google/syzkaller: syzkaller is an unsupervised coverage-guided kernel fuzzer — github.com. `https://github.com/google/syzkaller/tree/master`. [Accessed 25-07-2025].

[4] RISC-V Platform Runtime Services Task Group. Risc-v supervisor binary interface specification: Version 2.0. Technical report, RISC-V International, January 2024. Ratified specification. No changes are allowed.

[5] Documentation Arm Developer — developer.arm.com. `https://developer.arm.com/documentation/100690/0201/Arm-TrustZone-technology?lang=en`. [Accessed 25-07-2025].

[6] Clément Léger. lib: sbi: fwft: check feature value to be exactly 1 or 0 · riscv-software-src/opensbi@e8717d1 — github.com. `https://github.com/riscv-software-src/opensbi/commit/e8717d126401435896b0e96c18e187f0b2431d5e`, 2024. [Accessed 26-07-2025].

[7] Michał Zalewski. American Fuzzy Lop - Whitepaper. `https://lcamtuf.coredump.cx/afl/technical_details.txt`, 2016. [Accessed 25-07-2025].

[8] Inc. Free Software Foundation. Gcov (Using the GNU Compiler Collection (GCC)) — gcc.gnu.org. `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`. [Accessed 28-07-2025].

[9] The Clang Team. Source-based Code Coverage &x2014; Clang 22.0.0git documentation — clang.llvm.org. `https://clang.llvm.org/docs/SourceBasedCodeCoverage.html`, 2007-2025. [Accessed 28-07-2025].

[10] ©The kernel development community. KCOV: code coverage for fuzzing – The Linux Kernel documentation — docs.kernel.org. `https://docs.kernel.org/dev-tools/kcov.html`. [Accessed 28-07-2025].

[11] AFLplusplus/qemu_mode/README.md at stable · AFLplusplus/AFLplusplus — github.com. `https://github.com/AFLplusplus/AFLplusplus/blob/stable/qemu_mode/README.md`. [Accessed 28-07-2025].

[12] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. {kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels. In *26th USENIX security symposium (USENIX Security 17)*, pp. 167–182, 2017.

[13] Heqing Huang, Peisen Yao, Hung-Chun Chiu, Yiyuan Guo, and Charles Zhang. Titan: efficient multi-target directed greybox fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 1849–1864. IEEE, 2024.

[14] Michael Chesser, Surya Nepal, and Damith C Ranasinghe. {MultiFuzz}: A {Multi-Stream} fuzzer for testing monolithic firmware. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 5359–5376, 2024.

[15] Romain Malmain, Andrea Fioraldi, and Aurélien Francillon. Libafl qemu: A library for fuzzing-oriented emulation. In *BAR 2024, Workshop on Binary Analysis Research, colocated with NDSS 2024*, 2024.

[16] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1051–1065, 2022.

[17] QEMU — qemu.org. `https://www.qemu.org/`. [Accessed 30-07-2025].

[18] Documentation/QMP - QEMU — wiki.qemu.org. `https://wiki.qemu.org/Documentation/QMP`. [Accessed 30-07-2025].