

Fuzz4B: ファジングツール AFL の利用支援ツール

宮木 龍^{1,a)} 吉田 則裕¹ 都築 夏樹¹ 山本 椋太¹ 高田 広章¹

概要: 近年、自動化が可能なソフトウェアテストの 1 つとしてファジングが注目されており、ファジングを自動で実施するファジングツールが数多く開発されている。AFL は代表的なファジングツールの 1 つであり、数多くの脆弱性を検出した運用実績を持つ。これまで、AFL を拡張して数多くのファジングツールが開発されてきた。しかし、ファジングについての知見がない者には、AFL をデバッグに活用することが難しいという問題がある。本研究では、AFL の利用を支援するツール Fuzz4B を提案する。Fuzz4B は AFL のフロントエンドとして機能するだけでなく、AFL が検出した不具合を GDB によって再現する機能、AFL が出力したテストケースをデルタデバッグングによって削減する機能を提供する。Fuzz4B をオープンソースのライブラリである librope に適用し、その有効性を確認した。

Fuzz4B: Supporting Tool for the Use of a Fuzzing Tool AFL

RYU MIYAKI^{1,a)} NORIHIRO YOSHIDA¹ NATSUKI TSUZUKI¹ RYOTA YAMAMOTO¹ HIROAKI TAKADA¹

1. はじめに

ソフトウェア開発において、ソフトウェアが要求を満たしていることを確認するため、ソフトウェアテストが行われる。近年、ソフトウェアの開発規模が飛躍的に増大し、ソフトウェアテストの重要性が一層高まっている [1]。ソフトウェア開発にて、ソフトウェアテストは開発工程における工数のうち約 40 % を占めるようになってきている [2]。しかし、ソフトウェアテストの大部分は手作業で行われ、テスト工程で全ての不具合を抽出しきれなかったり、大きなコストがかかったりするという問題がある [3]。

このような問題を軽減するため、ソフトウェアテストの自動化に対する需要が高まっている。自動化が可能なソフトウェアテストの 1 つとして、ファジングが注目されている。ファジングは、ファズと呼ばれるデータを生成してテスト対象のソフトウェアに入力し、挙動を監視するという流れを繰り返すことで、不具合を自動で検出するテスト手法である。ファジングを自動で行うツールをファジングツールと呼ぶ。ファジングによって不具合の低減やテスト

にかかる労力の削減といった効果が得られる [4]。しかし、ファジングは、以下に示す要因により、ファジングツールの利用経験がないと導入の敷居が高いという問題がある。

- 実際にファジングツールを利用してファジングを行う際には、必要な準備が多い。
- ファジングによって不具合を検知できたとしても、使用したファジングツールについての知識がないと、得られた結果を不具合の修正作業に活かすことが難しい。

本研究では、ファジングツールの導入の敷居が高いという問題を軽減することを目的とする。目的を達成するため、ファジングツールの利用経験がない開発者でもファジングを利用できるよう、ファジングツールの利用を支援するツール **Fuzz4B** を提案する。**Fuzz4B** で対象とするファジングツールは、数多くの脆弱性を検出した運用実績を持つ **AFL** とした。**Fuzz4B** に求められる要件を決定するため、**AFL** を用いてソフトウェアの不具合を検出し、それを修正するまでの流れにおいて、**AFL** の利用経験がないユーザにとってどのようなことが問題となるかを検討した。そして、それを基に **Fuzz4B** に求められる要件を決定し、**Fuzz4B** の開発を行った。**Fuzz4B** はオープンソースソフトウェアであり、<https://github.com/Ryu-Miyaki/Fuzz4B> から利用可能である。

¹ 名古屋大学
Nagoya University, Nagoya, Aichi 464-8601, Japan
^{a)} miyaki@ertl.jp

本論文は、本章を含めて全5章で構成される。2章では、**AFL**の特徴と実際に利用する際の流れ、**AFL**の持つ問題点について述べる。3章では、2章で挙げた**AFL**の問題点を軽減するために必要な要件と、その要件に対応する提案ツール**Fuzz4B**について述べる。4章では、**Fuzz4B**の利用例を紹介するとともに、**Fuzz4B**が要件に対応していることを確認する。5章では、本研究のまとめと、今後の課題について述べる。

2. ファジングツール AFL

AFL^{*1}は数多くの脆弱性を発見した運用実績があるオープンソースのファジングツールである。本章では、**AFL**の特徴と**AFL**を実際に利用する際の流れ、**AFL**の持つ問題点について述べる。

2.1 AFLの特徴

Fuzz4Bで対象とするファジングツール**AFL**は、変異ベースのグレーボックスファザーである[5]。変異ベースとは、ランダムに生成されたデータ、またはユーザが与えたデータを変化させることによってファズを生成することを表す[6]。グレーボックスファザーとは、オーバヘッドが無視できる程度の軽量な計測によって、ファズを入力してテスト対象プログラムの実行ファイルを実行した際の実行経路を調べ、その情報を次回以降のファズ生成に活用するファジングツールである[7]。

AFLは、テスト対象プログラムの実行ファイルと、**AFL**が生成するファズの元となるデータ（以下、このデータを初期入力とする）を入力として受け取る。実行ファイルは、**AFL**に付属されているまたはというプログラムによって、テスト対象プログラムをコンパイルすることでユーザが生成する必要がある。テスト対象プログラムがCプログラムの場合は、C++プログラムの場合はを使用する。これらを用いてコンパイルすることによって、**AFL**が実行ファイルにファズを入力して実行する際に辿った実行経路情報を収集することが可能になる。

実行ファイルと初期入力を与えた後の**AFL**の動作を以下に示す。

- 1 初期入力をキューと呼ばれる領域に保存する。
- 2 キューからデータを1つ選択し、選択したデータのビットを反転する、定数を加減算するなどの操作（以降、これを変異とする）を加え、新たなファズを生成する。
- 3 生成したファズを実行ファイルに入力して実行する。この時に辿った実行経路情報を収集する。
- 4 実行ファイルの終了ステータスから不具合が発生したかを判定する。不具合が発生した場合、その時入力し

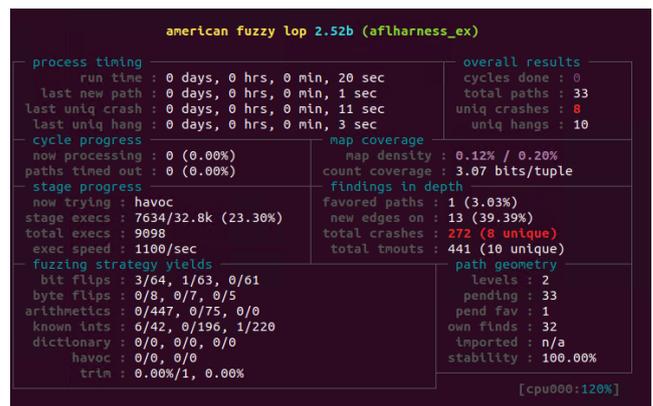


図1 AFL実行中の画面

たファズを出力先に保存する。

- 5 不具合が発生せず、新たな実行経路情報を得られた場合、その時入力したファズをキューに保存する。
- 6 ユーザが**AFL**を停止するまで2から5を繰り返す。

AFLが実行中に表示する画面を図1に示す。図1には**AFL**を起動してから経過した時間や検出した不具合の数などが表示されている。

ユーザは任意のタイミングで**AFL**を停止させ、ファジングを終了できる。**AFL**は、起動時にユーザによって指定された出力先に、表1に示すものを出力する。

以下、表1のディレクトリに保存されているファズのファイル名の命名規則について説明する。説明のため、crashesディレクトリに保存されているファズのファイル名の一例を示す。

id:000000,sig:06,src:000000,op:flip1,pos:5

idは各ディレクトリ内のファズ毎に固有の6桁の番号、sigはそのファズを実行ファイルに入力して実行した際に**AFL**が受け取った終了ステータス、srcはそのファズの変異元となったqueueディレクトリ内のファズのid、opは適用した変異の内容を、それぞれ表している。sigは06となっており、これはabort命令によってプログラムが中断されたことを意味するSIGABRT^{*2}を終了ステータスとして受け取ったことを表している。opはflip1,pos:5となっており、これは変異として先頭を0バイト目としたときの5バイト目の先頭1ビットを反転したことを表している。

queue、hangsディレクトリに保存されているファズのファイル名は、上記の例のうち、sigを除いたものとなっている。ただし、queueディレクトリ中のユーザが与えた初期入力のファイル名は、「id+元のファイル名」となっている。

2.2 AFL利用の流れ

実際に**AFL**を利用してテスト対象プログラムの不具合

*1 <http://lcamtuf.coredump.cx/afl/>

*2 <http://man7.org/linux/man-pages/man7/signal.7.html>

表 1 AFL が出力するディレクトリの構成

名前	ファイル or ディレクトリ	説明
crashes	ディレクトリ	不具合を起こすファズが保存されている
hangs	ディレクトリ	タイムアウトを起こすファズが保存されている
queue	ディレクトリ	キュー内に含まれるファズが保存されている
plot_data	ファイル	図 1 中の情報が時系列順に記録されている
fuzzer_stats	ファイル	AFL 起動時の設定情報などが記録されている
fuzz_bitmap	ファイル	実行経路情報を記録するために AFL が使用する

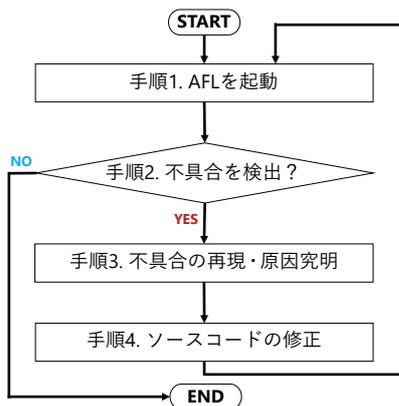


図 2 AFL によってテスト対象プログラムの不具合を修正する手順

を検出し、それを修正するまでの手順を図 2 に示す。図 2 に示した手順の詳細を以下に述べる。

手順 1 AFL を起動するために、まずテスト対象プログラムを afl-gcc、または afl-g++ によってコンパイルし、実行ファイルを作成する。afl-gcc、afl-g++ の利用方法は gcc、g++ と同じである。次に、ディレクトリを作成し、そこに AFL が最初に生成するファズの元となるデータを記述したファイルを 1 つ以上保存して初期入力を用意する。実行ファイルの作成、初期入力の用意が完了したら、それらを AFL に入力し、AFL を起動する。afl-fuzz -i [input] -o [output] [program] というコマンドを実行することで AFL を起動することができる。このコマンド中の [input] は初期入力のパス、[output] は AFL の出力が保存されるパス、[program] は実行ファイルのパスを表す。

手順 2 「Ctrl-C」を入力するなどの方法で AFL のプロセスを停止させ、任意のタイミングでファジングを終了する。ファジング終了時に不具合が検出されていない場合は、ファジングによるテストを終了する。不具合が検出された場合は手順 3 に進む。

手順 3 不具合が検出された場合、まず表 1 に示した AFL の出力中の crashes ディレクトリを確認し、不具合を起こすファズを特定する。次に、GDB などのデバッガを用いて、不具合を起こすファズを実行ファイルに入力して実行し、不具合を再現して、不具合が発生し

た箇所を特定する。GDB を使用する場合は、GDB を起動後、run < [fuzz] というコマンドによって実行ファイルに不具合を起こすファズを入力して実行し、不具合を再現する。このコマンド中の [fuzz] は不具合を起こすファズのパスを表す。不具合を再現したら、backtrace というコマンドによって、実行トレースを取得し、不具合が発生した箇所を特定する。不具合の再現、発生箇所の特定が完了したら、GDB などのデバッガを使用する、表 1 に示した AFL の出力を利用するといった方法で、不具合の原因を究明する作業（以降、この作業をデバッグとする）を行う。

手順 4 不具合の原因を究明できたら、テスト対象プログラムを修正する。以降、手順 2 で不具合を検出しなくなるまで、手順 1 から手順 4 を繰り返す。

2.3 AFL の問題点

ソフトウェア開発プロセスのテスト工程にファジングを導入するため、2.2 節で述べた実際に AFL を利用する際の流れを AFL の利用経験がないユーザーが行う場合、どのようなことが問題となるかを検討した。検討した結果、以下の 3 つの問題点があることが分かった。

問題点 1 AFL を使ってファジングを開始するには、必要な準備が多い。

問題点 2 ファジングによって不具合を検出した後、それを再現する方法が分からない。

問題点 3 ファジングによって得られた結果を、どのようにデバッグに活かせば良いか分からない。

問題点 1 について説明する。AFL を使ってファジングを開始するためには、2.2 節の手順 1 で述べたように、実行ファイルの作成、初期入力の用意、AFL を起動するコマンドの実行の 3 つの準備が存在する。AFL の利用経験があるユーザーはこれらの必要な準備を認識し、AFL を起動してファジングを開始することができる。しかし、AFL の利用経験がないユーザーは AFL を起動するだけでも、AFL について学習する必要が生じ、コストがかかる。

問題点 2 について説明する。2.2 節の手順 3 で述べた通り、不具合が検出された場合は、不具合を起こすファズを特定し、それをテスト対象プログラムの実行ファイルに入力して不具合を再現する必要がある。AFL の利用経験が

あるユーザは、**AFL** の出力中の `crashes` ディレクトリに保存されている不具合を起こすファズをテスト対象プログラムの実行ファイルに入力し、不具合を再現することができる。一方で、**AFL** の利用経験がないユーザは、`crashes` ディレクトリに不具合を起こすファズが保存されているということを認識しておらず、不具合を再現することができない。

問題点 3 について説明する。2.2 節の手順 **3** で述べた通り、不具合が再現し、発生した箇所を特定できたら、デバッグを行う。**AFL** の利用経験があるユーザは、不具合を起こすファズだけでなく、それ以外のファズを活用してデバッグが可能である。例えば、不具合を起こすファズとその変異元となったファズをそれぞれテスト対象プログラムの実行ファイルに入力すれば、不具合が起こる場合と起こらない場合とで実行結果を比較することができる。**AFL** が出力するファズのファイル名は、2.1 節で述べた規則に則っている。**AFL** の利用経験があるユーザは、ファズのファイル名から、不具合を起こすファズの変異元となったファズを特定することができる。このように、**AFL** の利用経験があるユーザは、ファジングによって得られた結果をデバッグに活かすことができる。一方で、**AFL** の利用経験がないユーザは、**AFL** がどこに何を出力しているのかが分からず、**AFL** の出力をデバッグに活かすことができない。

3. 提案ツール: Fuzz4B

本章では、2 章で述べた **AFL** の利用についての問題点を軽減し、**AFL** の利用を支援するツールに求められる要件について述べる。そして、決定した要件に対して、本研究で提案するツール **Fuzz4B** においてどのように対応したかについて述べる。**Fuzz4B** で対象とするユーザは、基本的な情報工学教育を受けている開発者とする。具体的にどのような能力や経験を持っているかを、以下のように定める。

- C 言語または C++ 言語を用いてソフトウェアを開発した経験がある。
- ソフトウェア工学に関する基本的な知識を有している。
- Linux のシェルを扱った経験がある。
- デバッガの 1 種である GDB の操作ができる。
- **AFL** の利用経験はない。

3.1 求められる要件

2 章で挙げた **AFL** の問題点を軽減するため、本研究で提案するツール **Fuzz4B** に求められる要件を以下の 3 つとした。

要件 1 ツールの指示どおりに入力を与えるだけでファジングを開始できる。

要件 2 ファジングで検出した不具合をユーザが簡単に再現できる。

要件 3 ファジングによって得られた結果を活かしたデバッグを支援する。

要件 1 について説明する。問題点 **1** に対処するべく、**AFL** を起動するために必要な準備のうち、テスト対象プログラムを `afl-gcc`、または `afl-g++` によってコンパイルして作成した実行ファイルと、初期入力については、**Fuzz4B** からユーザに入力を指示する。**AFL** を起動するコマンドの実行は **Fuzz4B** が受け持つ。このようにすれば、ユーザは **AFL** について学習しなくても、**AFL** を起動してファジングを開始することができる。

要件 2 について説明する。問題点 **2** に対処するべく、**Fuzz4B** から不具合を起こすファズをテスト対象プログラムの実行ファイルに入力して実行できるようにする。このようにすれば、**AFL** の出力のディレクトリ構成について詳細に学習しなくても、ユーザは簡単に不具合を再現することができる。

要件 3 について説明する。問題点 **3** に対処するべく、ファジングによって得られた結果を活かしたデバッグ方法を **Fuzz4B** からユーザに提示する。このようにすれば、ユーザがファジングによって不具合を検出した後に何をすれば良いか分からなくなることを防ぐことができる。

3.2 Fuzz4B の機能と要件への対応

Fuzz4B で提供する主な機能と、3.1 節で決定した各要件に対して、**Fuzz4B** でどのように対応したかを述べる。**Fuzz4B** では、主に以下の 3 つの機能を提供する。

機能 1 **AFL** 起動に必要な情報の入力をユーザに指示し、**AFL** を起動する機能

機能 2 **AFL** が出力した任意のファズをテスト対象プログラムの実行ファイルに入力した状態で GDB を起動する機能

機能 3 **AFL** が出力した不具合を起こすファズをデルタデバッグングによって最小化する機能

機能 1 について説明する。ユーザが **Fuzz4B** に **AFL** の起動を求めた場合、**Fuzz4B** は **AFL** 用にコンパイルした実行ファイルを作成したかを問い合わせる。ユーザがまだ **AFL** 用にコンパイルした実行ファイルを作成していない場合、**Fuzz4B** はユーザに **AFL** を起動するためには `afl-gcc`、または `afl-g++` によってテスト対象プログラムをコンパイルする必要があることと、それを実行するためのコマンドを提示する。ユーザが既に **AFL** 用にコンパイルした実行ファイルを作成している場合、**Fuzz4B** はユーザに作成した実行ファイルと、初期入力が入力されたディレクトリの入力を求める。ユーザがそれらを入力すると、**Fuzz4B** は **AFL** を起動してファジングを開始する。また、**AFL** 起動時にエラーが発生した場合は、**Fuzz4B** はその内容と対処法の例をユーザに伝える。以上より、ユーザは **Fuzz4B** の指示どおりに入力を与えるだけでファジングを開始する

ことができる。これによって、要件 1「ツールの指示どおりに入力を与えるだけでファジングを開始できる」に対応した。

機能 2 について説明する。GDB^{*3}とは、UNIX システムをはじめとした多くのシステムで動作するデバグである。GDB を使用すれば、プログラムの実行を任意の箇所まで止めることや、実行中の変数の値を確認することができる。Fuzz4B では、AFL が出力した任意のファズをテスト対象プログラムの実行ファイルに入力した状態で GDB を起動する機能を提供する。この機能によって、不具合を起こすファズをテスト対象プログラムの実行ファイルに入力した状態で GDB を起動し、起動した GDB を操作すれば、不具合を再現することができる。これによって、要件 2「ファジングで検出した不具合をユーザが簡単に再現できる」に対応した。また、この機能によって、不具合を起こすファズと、その変異元となったファズをテスト対象プログラムの実行ファイルに入力した状態でそれぞれ GDB を起動すれば、不具合が起こる場合と起こらない場合とで実行結果を比較することができる。このように、Fuzz4B は、不具合が起こる場合と起こらない場合とで実行結果を比較する作業を支援する。

機能 3 について説明する。AFL は不具合を起こすファズを出力するが、そのサイズが大きい場合、そのファズのうち不具合が起こる直接の原因となった箇所が分からないという問題がある。この問題を軽減するための手段として、Fuzz4B ではデルタデバッキング [8] を使用した。デルタデバッキングは、プログラムに不具合を起こすデータのサイズを、入力すると不具合を起こすという条件を保ったまま、最小化するアルゴリズムである [8]。ここでの最小化とは、デルタデバッキングによって得られたデータの全てが不具合に関連していることとする [8]。Fuzz4B は、不具合を起こすファズをデルタデバッキングによって最小化する機能を提供する。これによって、不具合を起こすファズのうち、不具合の原因となる箇所を特定する作業を支援する。

以上、要件 3「ファジングによって得られた結果を活かしたデバグを支援する」については、以下の 2 点の作業を支援することで対応した。

- 不具合が起こる場合と起こらない場合とで実行結果を比較する作業
- 不具合を起こすファズのうち、不具合の原因となる箇所を特定する作業

4. Fuzz4B の利用例

本章では、3 章で述べた Fuzz4B の利用例^{*4}を紹介する。

^{*3} <https://www.gnu.org/software/gdb/>

^{*4} ここで述べる利用例で用いた計算機は Intel Core i7-8665U、メモリ 8GB を搭載している。Windows 10 Pro 上で VMware

また、Fuzz4B が 3 章で述べた要件に対応できていること、それによって 2 章で述べた問題点が軽減されていることを確認する。

4.1 テスト対象プログラム

利用例を紹介するにあたって、テスト対象とするプログラムについて述べる。対象プログラムは、オープンソースの C ライブラリである librope^{*5} のスナップショット^{*6} を使用する。librope は、UTF-8 文字列を取り扱うためのライブラリであり、ソースコードの行数は 794 行である。ここで述べる利用例では、librope の rope.insert 関数に対してファジングを行う。rope.insert 関数に対してファジングを行うため、librope リポジトリの afl/aflharness.c を基としたテストハーネスを使用する。このテストハーネスは 2 行からなる文字列の組を標準入力から受け取る。各組の 1 行目は、rope.insert 関数によって文字列を挿入する位置を、2 行目は rope.insert 関数によって挿入する文字列となる。このテストハーネスは入力された組の 2 行目に、UTF-8 文字コードで表現できない文字が現れると不具合が起こる。この不具合は、librope の開発者が AFL を用いて発見したものである [9]。

4.2 Fuzz4B の利用方法

4.1 節で述べた対象プログラムに対して、Fuzz4B を用いてファジングを実行し、Fuzz4B の利用方法について述べる。また、Fuzz4B が 3 章で述べた 3 つの要件に対応できていること、それによって 2 章で述べた問題点が軽減されていることを確認する。

Fuzz4B は、Fuzz4B のソースコードが保存されたディレクトリをカレントディレクトリとして、python3 ExecTool.py というコマンドを端末に入力することで起動することができる。Fuzz4B 起動時の画面を図 3 に示す。画面上部にはメニューバーがあり、「ファジング」、「表示」、「デバグ」の 3 つのメニューが含まれている。画面左にはそれぞれ「クラッシュを起こすファズ」、「元となったファズ」、「最小化した結果」というラベルが付けられた 3 つのテキストボックスが配置されている。画面右にはテーブルが配置されている。

図 3 中のメニューバーから「ファジング」メニューを選択すると、「ファジングの開始」、「ファジングの停止」という 2 つのサブメニューが表示される。「ファジングの開始」を選択すると、対象プログラムを AFL 用にコンパイ

Workstation 15 Player によって Ubuntu 18.04 LTS の仮想環境を作成し、メモリ 4GB、CPU4 コアを割り当てた。また、Python のバージョンは 3.6.8、wxPython のバージョンは 4.0.7.post2、AFL のバージョンは 2.52b であった。

^{*5} <https://github.com/josephg/librope/>

^{*6} <https://github.com/josephg/librope/commit/44c6a0fc875bf636756116edfde68003b29f9a38>

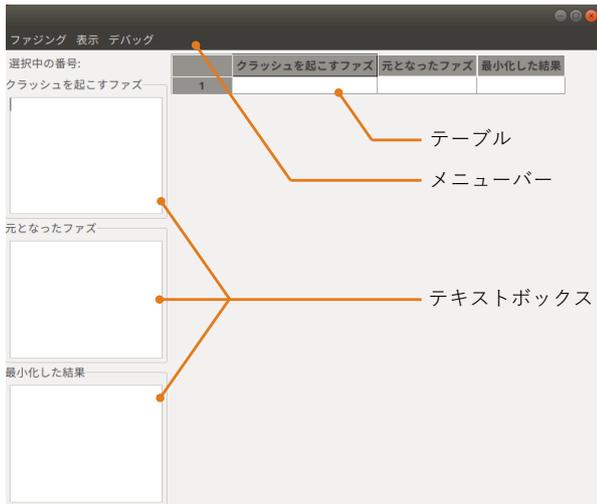


図 3 Fuzz4B 起動時の画面と各オブジェクトの説明

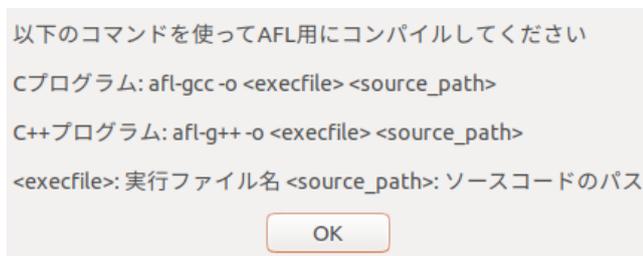


図 4 AFL 用にコンパイルするためのコマンドをユーザに通知するダイアログ

ルしたかを問われ、「はい」か「いいえ」の選択を求められる。「いいえ」を選択すると、図 4 に示すダイアログが表示され、対象プログラムを **AFL** 用にコンパイルするためのコマンドが提示される。これによって、ユーザは、対象プログラムを **AFL** 用にコンパイルする必要があることと、それを実行するために必要なコマンドを認識することができる。「はい」を選択すると、図 5 上部に示すダイアログが表示され、対象プログラムを **AFL** 用にコンパイルして生成した実行ファイルの入力を求められる。実行ファイルを入力すると、図 5 下部に示すダイアログが表示され、初期入力を保存したディレクトリの入力を求められる。これによって、ユーザは、対象プログラムに対する入力の例を記載した初期入力が必要であることを認識することができる。本節の利用例では、「0\nomg hi\n\n」という文字列が記載されたファイルを in というディレクトリに保存し、それを初期入力とした。初期入力を保存したディレクトリを入力すると、ユーザにファジングを開始することを確認した後、ファジングが開始される。

以上より、**AFL** を使ったことがないユーザでも、**AFL** を起動するために必要な情報を認識し、**AFL** を起動してファジングを開始することができる。よって、**Fuzz4B** は要件 1 「ツールの指示通りに入力を与えるだけでファジングを開始できる」を満たしていることを確認した。これ



図 5 実行ファイルの入力を指示するダイアログ (上) と初期入力が保存されたディレクトリの入力を指示するダイアログ (下)

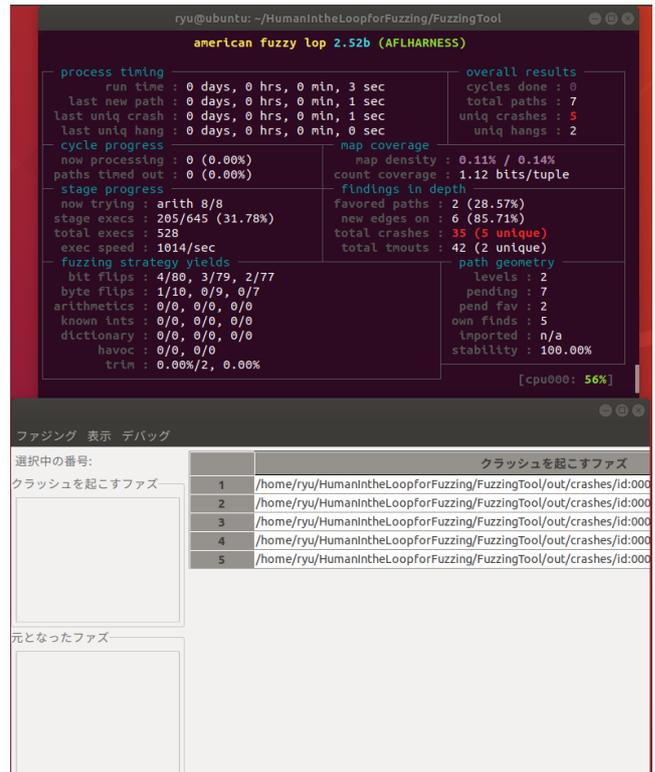


図 6 ファジング実行中の画面 (図上部は **AFL** が実行中に表示する画面、図下部は **Fuzz4B** が表示する画面)

によって、問題点 1 「**AFL** を使ってファジングを開始するには、必要な準備が多い」を軽減することができた。

ファジング実行中の画面を図 6 に示す。ファジング実行中は、図 6 に示すように、**Fuzz4B** を起動した端末に **AFL** が実行中に表示する画面を出力する。不具合を検出した場合は、不具合を起こすファズの保存先と、その変異元となったファズの保存先をテーブルに出力する。テーブルの行番号をクリックすると、その行が示す不具合を起こすファズ、その変異元のファズの内容をテキストボックスに表示する。また、この時、図 3 中のメニューバーから、「デバッグ」メニューに含まれる「ファズの最小化」、「ファズを入力して実行」の 2 つのサブメニューを選択できるようになる。

「デバッグ」メニューから、「ファズを入力して実行」を選択すると、図 7 に示すダイアログが表示される。図 7 のラジオボタンから入力するファズを選択し、図 7 の GDB

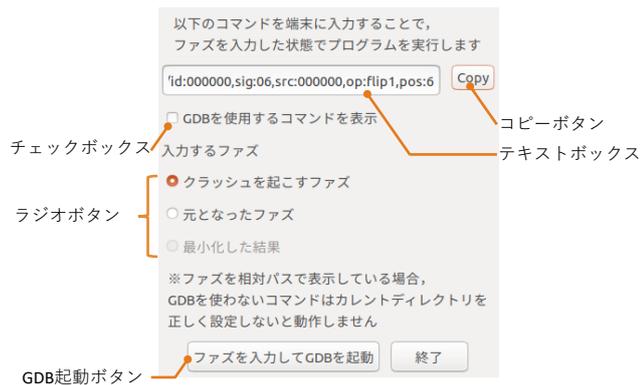


図 7 任意のファズを入力した状態で GDB を起動するダイアログ

起動ボタンをクリックすると、ラジオボタンで選択したファズを対象プログラムに入力した状態で GDB を起動することができる。図 7 の「クラッシュを起こすファズ」と表示されたラジオボタンを選択して図 7 の GDB 起動ボタンをクリックすると、Fuzz4B が表示する画面を表示したまま、新たな端末を起動し、対象プログラムに不具合を起こすファズを入力した状態で GDB を起動する。その後、起動した GDB を操作すれば、GDB 上で不具合を再現することができる。

以上より、AFL の利用経験がないユーザでも、不具合を再現することができる。よって、Fuzz4B は要件 2 「ファジングで検出した不具合をユーザが簡単に再現できる」を満たしていることを確認した。これによって、問題点 2 「ファジングによって不具合を検出した後、それを再現する方法が分からない」を軽減することができた。

また、図 7 の「元となったファズ」と表示されたラジオボタンを選択して図 7 の GDB 起動ボタンをクリックすると、新たな端末を起動し、対象プログラムに不具合を起こすファズの元となったファズを入力した状態で GDB を起動する。この機能によって、図 8 のように、Fuzz4B から端末を 2 つ起動し、不具合が発生する場合と発生しない場合とで実行結果を比較することができる。よって、Fuzz4B は、不具合が起こる場合と起こらない場合とで実行結果を比較する作業を支援していることを確認した。

「デバッグ」メニューから、「ファズの最小化」を選択すると、図 9 のように、「クラッシュを起こすファズ」のラベルが付けられたテキストボックスに表示されている不具合を起こすファズをデルタデバッグングによって最小化し、その結果を「最小化した結果」のラベルが付けられたテキストボックスに表示する。また、Fuzz4B は、新たにファイルを作成し、デルタデバッグングによって最小化した結果を、作成したファイルに出力する。作成したファイルの保存先の情報を、図 3 のテーブルに追加する。図 9 の

「最小化した結果」のラベルが付けられたテキストボックスには、「\n?\n」(‘?’は UTF-8 で表現できない文字)という文字列が表示される。よって、不具合を起こすファズのうち、不具合の原因となる箇所を特定する作業を支援していることを確認した。

以上より、以下の 2 点の作業を支援していることを確認した。

- 不具合が起こる場合と起こらない場合とで実行結果を比較する作業
- 不具合を起こすファズのうち、不具合の原因となる箇所を特定する作業

したがって、Fuzz4B は要件 3 「ファジングによって得られた結果を活かしたデバッグを支援する」を満たしていることを確認した。これによって、問題点 3 「ファジングによって得られた結果を、どのようにデバッグに活かせば良いか分からない」を軽減することができた。

5. まとめ

本研究では、ファジングはファジングツールの利用経験がないと導入の敷居が高いという問題を軽減するため、ファジングツールの利用経験がない開発者でもファジングを利用できるように、ファジングツールの利用を支援するツール Fuzz4B を提案した。本研究で対象とするファジングツールは、数多くの脆弱性を検出した運用実績を持つファジングツール AFL とした。ツールを提案するため、まず、AFL の利用経験がないユーザが AFL を利用する上で問題となる点を検討した。検討した問題点を軽減するため、Fuzz4B に求められる要件を決定した。決定した要件に基づいて、Fuzz4B の提案、開発を行った。最後に、不具合を含むプログラムに対して Fuzz4B を用いてファジングを実施し、Fuzz4B が決定した要件を満たしていることを確認した。

今後の課題として、以下を検討している。

- ファジングの進捗はコードカバレッジなどによって測ることができる [10] が、Fuzz4B の対象ユーザが、AFL がユーザに提供する情報からファジングの進捗を測り、ファジングを終了するタイミングを判断することは困難であると考えられる。よって、ファジングの進捗を測り、ファジングを終了するタイミングの判断を支援することは今後の課題である。
- Fuzz4B から起動した GDB でステップ実行を行うと、AFL がファジング中に実行経路情報を収集するためにテスト対象プログラムの実行ファイルに挿入した命令が現れる。Fuzz4B の対象ユーザが、これらの命令について認識しているとは考えにくく、これはユーザを混乱させる原因となると考える。この問題についての対策を考えることは今後の課題である。
- 本研究では、Fuzz4B が対象ユーザの AFL 利用を支

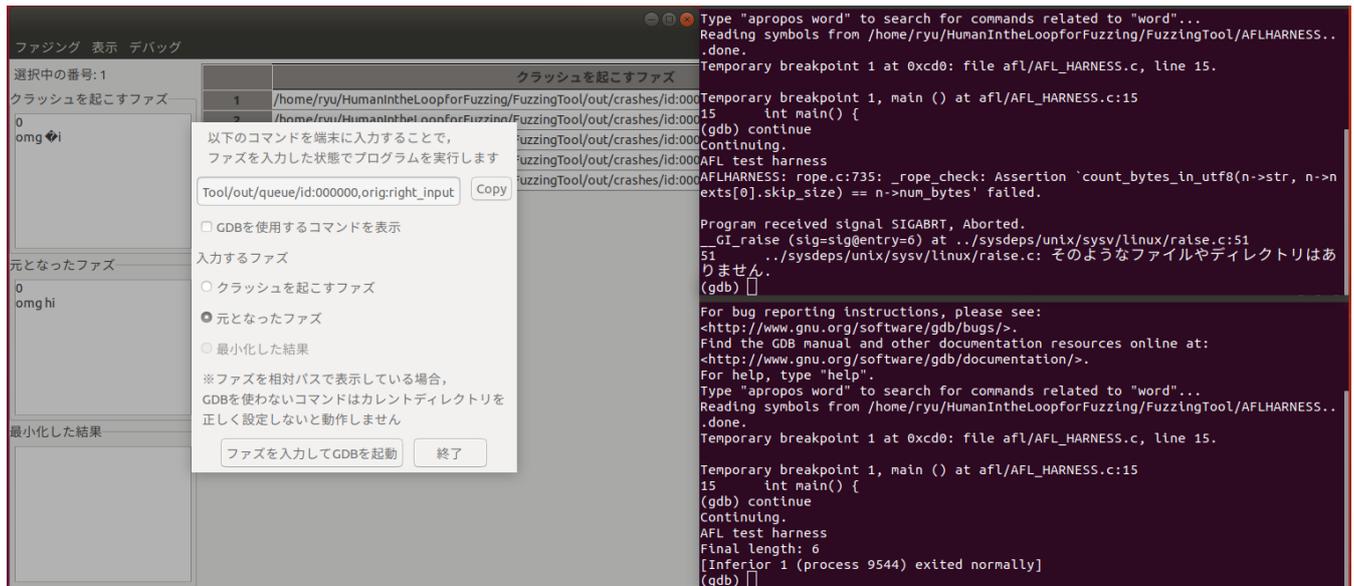


図 8 不具合発生時・不発時での実行結果の比較 (図右上の端末は不具合発生時, 図右下の端末は不具合不発時の実行結果を表す)

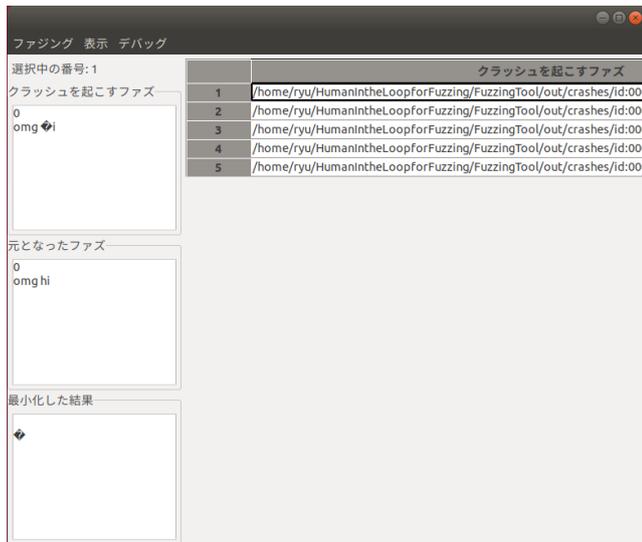


図 9 デルタデバッグによって最小化した結果の表示

援していることを示す定量的な評価を行うことができていない。Fuzz4B の定量的な評価方法の検討, およびその実施は今後の課題である。

参考文献

- [1] 中野隆司, 田中裕大, ダンティホンイエン: ソフトウェアのテスト工数・期間を削減するためのシステムテスト自動化技術, 東芝レビュー, Vol. 73, No. 3, pp. 40-44 (2018).
- [2] 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, 独立行政法人情報処理推進機構 (2018).
- [3] 丹野治門, 張 曉晶, 田端啓一, 生沼守英, 村主一仁: ソフトウェアの品質確保と開発コスト削減を目指したテスト自動化技術, NTT 技術ジャーナル, Vol. 25, No. 10, pp. 19-22 (2013).
- [4] 独立行政法人情報処理推進機構セキュリティセンター: ファジング活用の手引き, 独立行政法人情報処理推進機

- 構 (2013).
- [5] Nagy, S. and Hicks, M.: Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing, *Proc. of S&P 2019*, pp. 787-802 (2019).
- [6] Peng, H., Shoshitaishvili, Y. and Payer, M.: T-Fuzz: fuzzing by program transformation, *Proc. of S&P 2018*, pp. 697-710 (2018).
- [7] Böhme, M., Pham, V.-T., Nguyen, M.-D. and Roychoudhury, A.: Directed greybox fuzzing, *Proc. of CCS 2017*, pp. 2329-2344 (2017).
- [8] Zeller, A. and Hildebrandt, R.: Simplifying and isolating failure-inducing input, *IEEE Transactions on Software Engineering*, Vol. 28, No. 2, pp. 183-200 (2002).
- [9] Gentle, J.: Bug hunting with American Fuzzy Lop, Seph (online), available from <https://josephg.com/blog/bug-hunting-with-american-fuzzy-lop/> (accessed 2020-04-17).
- [10] Zeller, A., Gopinath, R., Böhme, M., Fraser, G. and Holler, C.: When To Stop Fuzzing, *The Fuzzing Book* (online), available from <https://www.fuzzingbook.org/html/WhenToStopFuzzing.html> (accessed 2020-02-05).