Vol.2019-SE-201 No.8
2019/3/7

競技プログラミングの提出コードを用いたファジングツールの定量的評価

都築 夏樹[1,a)]　吉田 則裕[1]　戸田 航史[2]　山本 椋太[1]　高田 広章[1]

概要：ソフトウェアのテスト手法の 1 つとしてファジングがある．この手法は，テストケースを自動生成してテストを行う．現在，ファジングツールの開発が盛んに行われており，ツール同士の比較評価が行われている．しかし，テスト対象のメトリクス値に着目した評価が行われておらず，提案されたツールがどのようなソフトウェアに対して有効かわかっていない．そのため，本研究ではメトリクス値に着目したファジングツールの評価を目的として調査を行った．具体的には，AFL と AFLFast の 2 つのファジングツールを Codeforces が企画した競技プログラミングへ提出されたソースファイルに適用した．そして，ツールが検出したクラッシュ数，パス数とソースファイルのメトリクス値を用いて線形回帰分析を行った．本研究の結果，先行研究の比較結果と異なり，ツールが検出したクラッシュ数，パス数の観点でツール間に有意差がないことがわかった．また，メソッドや関数から計算した複雑度の平均値が低いほど検出したクラッシュ数が多くなるといったファジング結果とメトリクス値の相関がわかった．

Quantitative Evaluation of Fuzzing Tools Using Programming Competition Archives

Natsuki Tsuzuki[1,a)]　Norihiro Yoshida[1]　Koji Toda[2]　Ryota Yamamoto[1]　Hiroaki Takada[1]

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に従って動作するかを確認するため，ソフトウェアテストが行われている. ソフトウェア規模の増大に伴い，開発工数におけるソフトウェアテストの割合は増加しており, 一般的なソフトウェア開発において 3 割から 5 割を占めるといわれている [1]．しかし，コストをかけてソフトウェアテストを行っているにも関わらず，作成したテストケースでは検出が困難な不具合を生じる可能性がある．

このような背景の中で，従来の人手によるソフトウェアテストだけでなく，自動的なソフトウェアテストが注目されている．ファジングは，ソフトウェアテストを自動的に行う手法の 1 つである．この手法では，ソフトウェアがクラッシュを引き起こす可能性のあるテストケースを自動生成する．そして，生成したテストケースを用いてテストを行った際の応答を取得する．この作業を繰り返して，自動的にソフトウェアテストを行う．現在，ファジングに関する研究が盛んに行われており，この手法を実装したファジングツールとして AFL[2] や AFLFast[3] などが開発されている．ファジングツールには，人手で検出できなかった不具合の検出実績があり [2][3]，このような不具合を検出することが期待されている．

それぞれのツールの評価は，提案したツールを既存のツールと比較することによって行われている．しかし，比較評価はファジングツールが検出したクラッシュ数やパス数などのツールの実行結果に着目して行われており [3][4]，テスト対象のメトリクス値の変化がファジングツールの実行結果に及ぼす影響の評価が行われていない．そのため，あるソフトウェアに対してどのファジングツールを使えばよいかわからないという問題がある．本研究では，メトリ

[1] 名古屋大学
Nagoya University, Nagoya, Aichi 464–8603, Japan
[2] 福岡工業大学
Fukuoka Institute of Technology, Fukuoka, Fukuoka 811–0295, Japan
[a)] tuzuki@ertl.jp

クス値がファジングツールの実行結果に及ぼす影響を明らかにすることを目的として，メトリクス値に着目したファジングツールの定量的評価を行う．

本研究では，テスト対象のデータセットとして Codeforces[*1]が企画した競技プログラミングへ提出されたソースファイル（以下，Codeforces 提出ソースファイルと呼ぶ）を利用し，評価用のファジングツールとして AFL と AFLFast を利用した．データセットにファジングツールを適用した後，ソフトウェアのメトリクス値を従属変数とし，ファジングツールが検出したクラッシュ数，パス数を目的変数として線形回帰分析を行った．その後，どのソフトウェアのメトリクス値がファジング結果に影響するかを調査した．

本研究の結果，ファジングツールが検出したクラッシュ数，パス数の観点でメトリクス値を利用して評価した場合，AFL と AFLFast の間に差はないことがわかった．先行研究の Böhme らの研究 [3] によると，GNU binutils[*2]に対してファジングを適用すると AFLFast は AFL よりも検出できるクラッシュ数が多いなど，本来は AFLFast のほうが優れている．しかし，今回利用したデータセットに対してファジングを適用したところ異なる結果となった．また，提出したソースファイル中の関数やメソッド単位で複雑度を計算し，その平均の複雑度が低いほど検出したクラッシュ数が多くなるといったファジング結果とメトリクス値の相関がわかった．

2. 関連研究

2.1 ファジング

ファジングは，テストケースの生成とテスト対象へのテストケースの適用を繰り返し行うことでテストを自動化する．ファジングを大別するとホワイトボックスファジング [5]，ブラックボックスファジング [6]，グレーボックスファジング [3] の 3 種類に分類される．

ホワイトボックスファジングは，テスト対象のソースファイルから取得した情報を利用してテストケースを生成する．情報を得るほど不具合の検出に有効なテストケースの生成が可能となるが，テストケースの生成に要する時間が増大する．ブラックボックスファジングは，テスト対象の内容にかかわらず，無作為に大量のテストケースを作成しテストを行う．テストケースの生成が短時間で済むため，テストケースの生成に要する時間が少ないという利点があるが，網羅的なテストケースの生成が難しい．グレーボックスファジングは前述した 2 つのファジングの中間に位置し，実行したテスト対象の基本ブロックの遷移といった実行中の情報を利用してテストケースを生成する．

[*1] https://codeforces.com/
[*2] https://www.gnu.org/software/binutils/

2.2 ファジングツール

2.2.1 American Fuzzy Lop（AFL）

AFL[2] は，グレイボックスファジングを実装したファジングツールである．AFL の動作を説明するために，テストケースに着目した AFL の動作フローを図 1 に示す．以下，AFL の動作について説明する．

まず，AFL はユーザが用意した初期テストケースをキューに保存する．この処理はツールの実行開始時のみ行われる．次に，キューからテストケースを 1 つ選択し，変異戦略に基づいて新たなテストケースを生成する．テストケースをテスト対象に適用してテストするとともに，基本ブロックの遷移などの実行中の情報を取得する．テストケースは，標準入力もしくはファイルとして与えられる．テスト対象がこれらの入力形式に対応していない場合，テストハーネスを作成する必要がある．テストハーネスとは，ドライバなどのテスト環境を提供するためのソフトウェアである．

テストケースの適用後，取得した情報をもとにテストケースの価値を判断する．テストケースの価値は，テストケースをテスト対象に適用した場合の実行時間やテストケースのサイズなどから判断される．テストケースの価値がツールの基準を満たす場合はキューに保存し，新たなテストケースの生成に利用する．ツールの基準を満たさない場合は破棄する．

上記の AFL の動作フローからわかるように，初期テストケースや変異戦略はファジングが検出したクラッシュ数やパス数といったファジング結果に影響を与える．そのため，これらを改善することでファジング結果の向上が期待できる．このことから，AFL をもとに変異戦略を変更したファジングツール [3][7] や，記号実行を用いて初期テストケースの質を高めたファジングツール [8] などの開発が行われている．

2.2.2 AFLFast

AFLFast[3] は，AFL をもとにその変異戦略を変更したファジングツールである．グレイボックスファジングには，生成されたテストケースの多くが特定のパスを通過する場合があり，テスト効率の点で問題がある．この問題が発生する例として，libpng のファジングが考えられる．この場合，有効な画像ファイルから新たなテストケースを生成すると，90%が不正な画像を拒否するパスを通過する．そして，画像ファイル以外から新たなテストケースを生成すると，99.999%が不正な画像を拒否するパスを通過する [3].

AFLFast は，この問題を解決するために変異戦略の変更を行った．具体的には，生成されたテストケースの少数が通過するパスを価値の高いパスと考え，これらのパスを通過するテストケースがキューから選択される優先度とそのテストケースから新たにテストケースを生成する数を増大した．

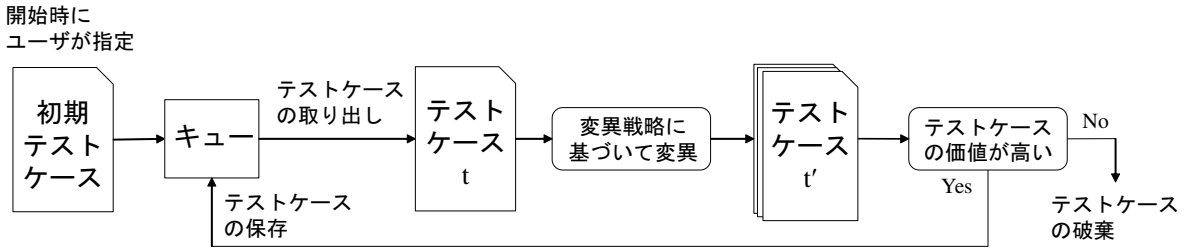

図 1 テストケースに着目した AFL の動作フロー
参考文献 [3] の 4 ページの図 2 をもとに著者が手を加えて作成した

上記の結果，GNU binutils に含まれる 6 種類のソースファイルに対して，ファジングを適用したところ 3 種類において検出したクラッシュ数で AFL を上回る結果を示し，残りの 3 種類において AFL と同等の結果を示した [3].

2.3 問題提起

現在，ファジングツールの評価では検出したクラッシュ数やパス数といったファジング結果を尺度としてツール同士を比較している．例えば，AFL と AFLFast の比較 [3] において検出したクラッシュ数や共通脆弱性識別子 (CVE)[*3]に登録されている脆弱性を検出するまでの時間を評価している．

しかし，これらの評価ではデータセットに含まれるソースファイルのファイル数が少ないことが多く，統計的な評価があまり行われていない．そのため，ファジング結果が一般的なものか判断できないという問題がある．また，データセット内のソースファイルについて評価が行われておらず，どのような特徴を持つソースファイルに対してどのファジングツールが有効か判断できないという問題がある．

3. 利用するデータセット

本研究では，堤がまとめた 2016 年 5 月 19 日から 2016 年 11 月 15 日の 6 ヶ月間に Codeforces が企画したプログラミングコンテストに提出された 1,644,636 ファイルのソースファイル [9] の中で，ファジングツールの評価に適すると考えられる 111,477 ファイルをデータセットとして用いる．以下，Codeforces について説明する．

Codeforces は，競技プログラミングを企画するサイトの 1 種である．競技プログラミングでは，競技者は定められた時間で与えられた問題を解く．このときの解答数や正否，解答までにかかった時間などをもとに計算した得点を他の競技者と競う．Codeforces では，提出されたソースファイルの問題に対する成否判定にオンラインジャッジシステムを利用している．オンラインジャッジシステムは，提出されたソースファイルにあらかじめ準備されたテストケースを適用する．テストケースは標準入力として入力される．テストケースを適用した後，テスト対象がテストケースに対応した応答や実行時間などの制約を満たしていることを確認し，テストケースの通過の成否を判断する．

上記から，Codeforces 提出ソースファイルへファジングツールを適用するに当たって，利点となる特徴は以下の 4 つである．

(1) コンパイルや実行が容易にできる．
(2) 提出されたソースファイルが豊富にある．
(3) 入力形式が統一されている．
(4) テストケースが用意されている．

（1）の利点は，テスト対象を用意しやすいことである．提出されたソースファイルは，オンラインジャッジシステムにてコンパイルされ実行される．そのため，オンラインジャッジシステムと同様の手順を踏めばテスト対象を作成することができる．このことから，ソースファイルに依存関係があり，コンパイルに多くの手順が必要なものと比較して優位である．

（2）の利点は，統計的な評価がしやすいことである．比較評価では多くのソースファイルを用いてテストを行い，統計的に評価を行うことが必要である．このことから，データ数が多いことは優位である．

（3）の利点は，テスト環境を整えやすいことである．入力形式が統一されている場合，ソースファイルごとにテストハーネスを作成する必要がなく，共通の形式で対応することができる．特に，AFL や AFLFast においては標準入力を入力形式としているため新たにテストハーネスを作成する必要がない．このことから，テスト環境を整えやすく，データセットのソースファイルのファイル数を増加させやすいという点で優位である．

（4）の利点は，初期テストケースの基準が定められることである．ファジング結果は初期テストケースによる影響を受ける．そのため，初期テストケースの質を共通にして評価を行う必要がある．しかし，テストケースの質を定量化することは難しく，ツールごとに質をそろえたテストケースを用意することは困難である．その中で，問題のために用意されたテストケースを利用することで，初期テストケースに一定の基準を設けて評価を行うことができる．

本研究では，不完全なソースファイルの除外やファジン

[*3] http://cve.mitre.org/

グツールの評価に適さないソースファイルの除外を目的として，次の 3 つの基準を定めて利用するソースファイルの絞り込みを行った．

(1) オンラインジャッジを通過している．
(2) C/C++言語を利用している．
(3) ソースファイルに含まれるメソッドや関数から複雑度をそれぞれ計算し，その中で最大の複雑度が 10 を上回る．

不完全なソースファイルとは，提出されたもののコンパイルができないソースファイルや問題に対応したテストケースに適切な応答ができないソースファイルを指す．評価に適さないソースファイルは，ファジングツールに対応外のプログラミング言語を使用しているソースファイルや複雑度が低いソースファイルである．ソースファイルの複雑度がソフトウェアの構造上望ましいとされる複雑度である 10 以下 [10] の場合，ソースファイルのテストが人手でしやすいとされる．そのため，人手で検出しづらい不具合を検出するというファジングの目的に反し，評価に適さない．

ソースファイルの絞り込みを行った結果，利用するソースファイルのファイル数は，1,644,636 ファイルから 111,477 ファイルに減少した．

4. 調査

AFL と AFLFast の比較 [3] では，1 つのデータセットに 6 時間の打ち切り時間でファジングを適用した結果を示している．そのため，ファジングの打ち切り時間やデータセットの変更がファジングに与える影響がわかっていない．ファジング時間を変更した場合や異なるデータセットである Codeforces 提出ソースファイルへファジングツールを適用した場合，ファジング結果が変化する可能性がある．そのため，調査が必要である．また，比較においてはデータセットの内容ついて評価していないため，どのようなソースファイルに対してどのようなファジングツールが有効か明らかになっていない．そのため，データセットの内容とファジングツールの実行結果の関係を調査する必要がある．

以上から，次の RQ を設定して調査を行った．

RQ1 ファジングの打ち切り時間はファジング結果に影響を与えるか？

RQ2 検出したクラッシュ数やパス数はツール間で異なるか？

RQ3 ファジングが有効なソースファイルが持つ特徴は何か？

調査に用いたコンピュータの環境を表 2 に示す．調査には，3 章で示したように堤がまとめた Codeforces 提出ソースファイル [9] に絞り込みを行ったものをデータセットとして利用した．

調査において，メトリクス値の収集にはメトリクス値収集ツールの SourceMonitor[*4] を利用した．また，自作スクリプトを用いたメトリクス値の収集も行った．SourceMonitor および自作スクリプトから取得したメトリクス値を表 1 に示す．SourceMonitor から取得可能なメトリクス値の詳細はツールのヘルプから確認できる．表 1 での予約語の分類を付録 A.1 から A.3 に示す．

調査で行われた有意差の検定では，Wilcoxon の順位和検定を用いる．検定では，統計計算向けのプログラミング言語である R[*5] の wilcox.exact 関数を利用した．

4.1 RQ1: ファジングの打ち切り時間はファジング結果に影響を与えるか？

前述したように，AFL と AFLFast の比較 [3] ではファジングを 6 時間で打ち切り評価を行っている．そして，それよりも短縮した打ち切り時間での評価を行っていない．そのため，本調査では打ち切り時間を短縮した影響を調査した．調査の手順は以下のとおりである．

まず，データセットから無作為に 10 ファイルを抽出する．データセットにヘッダファイルなどは含まれていないため，抽出したファイルはすべて標準入力に対応する．そして，AFL と AFLFast をそれぞれ適用する．このとき，ファジングの打ち切り時間は 15 分と 6 時間の 2 種類で行う．最後に，異なる打ち切り時間から得られたファジング結果に有意差が存在するか R にて検定する．

手順に従いファジングを適用した．そして，AFL，AFLFast が検出したクラッシュ数，パス数について，ファジングの打ち切り時間の変化による差を検定した. その結果，AFL では検出したクラッシュ数について $p = 0.422$，パス数について $p = 0.516$ で，有意水準 5%で差が認められなかった．また．AFLFast では検出したクラッシュ数について $p = 0.340$，パス数について $p = 0.342$ で，有意水準 5%で差が認められなかった．この結果から，AFL，AFLFast のいずれも検出したクラッシュ数，パス数において，ファジングの打ち切り時間が 15 分と 6 時間で有意差がないことがわかった．

以上のことから，Codeforces 提出ソースファイルを利用してファジングを行った場合，打ち切り時間を 6 時間から 15 分に短縮する影響が小さいことがわかった．

4.2 RQ2: 検出したクラッシュ数やパス数はツール間で異なるか？

GNU binutils に含まれるソースファイルにファジングを適用した場合，AFL と AFLFast に有意差があるといわれている [3]．しかし，Codeforces 提出ソースファイルをファジングの対象とすることを考えると，データセットが異なるため同様に有意差が存在するかは判断できない．

*4 http://www.campwoodsw.com/sourcemonitor.html
*5 https://www.r-project.org/

表 1　SourceMonitor および自作スクリプトから取得したメトリクス値

利用したツール	メトリクス値の種類	説明
SourceMonitor	Lines	ソースファイルの行数
	Statements	空行を除いたソースファイルの行数
	% Branch	ソースファイルの行における分岐命令の割合
	Class Defs	クラスの定義数
	Methods/Class	クラスあたりのメソッドの数
	Functions	関数の数
	Avg Statements/Method	全メソッドの空行を除いた行数の総和をメソッド数で除算した値
	Max Complexity	メソッドや関数ごとに計算した複雑度の最大値
	Avg Complexity	メソッドや関数ごとに計算した複雑度の平均値
	Max depth	行ごとのネスト値の最大値
	Avg depth	各行のネスト値の総和を行数で除算した値
自作スクリプト	ELOC	空行とコメント行を除いたソースファイルの行数
	Reserved word branch	分岐に関係する予約語
	Reserved word declaration	変数宣言に関係する予約語
	Reserved word other	上記 2 種に属さない予約語
	Function call	ソースファイル内で定義された関数が呼び出された回数

表 2　調査環境

種類	内容
OS	Ubuntu 16.04.4 LTS
CPU	i7-6700 @ 3.40 GHz
メモリ	8GB
AFL のバージョン	2.52b
AFLFast のバージョン	AFL のバージョン 2.51b を拡張

よって，本調査ではテスト対象を Codeforces 提出ソースファイルに変更した場合でも同様に有意差が存在するか調査した．調査の手順は以下のとおりである．

まず，データセットから無作為に 100 ファイルを抽出する．そして，AFL と AFLFast をそれぞれ適用する．このとき，ファジングの打ち切り時間は 15 分で行う．最後に，AFL と AFLFast から得られたファジング結果に有意差が存在するか R にて検定を行う．

AFL，AFLFast をそれぞれ適用した後，検出したクラッシュ数，パス数についてツール間の差の検定を行った．その結果，検出したクラッシュ数について $p = 0.848$，検出したパス数について $p = 0.824$ で，有意水準 5%でツール間に差は認められなかった．

以上のことから，Codeforces 提出ソースファイルを利用して AFL，AFLFast を適用した結果，検出したクラッシュ数，パス数に有意差が存在せず，ツール間の差は見られないことがわかった．

4.3 RQ3: ファジングが有効なソースファイルが持つ特徴は何か？

本調査では，2.3 節で提起した，どのような特徴をもつソースファイルに対して，どのようなファジングツールが有効かということを明らかにするため，ソースファイルのメトリクス値とファジングツールが検出したクラッシュ数，パス数の関係を調査した．調査の手順は以下のとおりである．

表 3　検出したクラッシュ数，パス数に対する線形回帰分析の結果（抜粋）

	検出したクラッシュ数		検出したパス数	
	AFL	AFLFast	AFL	AFLFast
Lines	-1.172	-1.157	-0.971	-0.944
Class Defs	2.074	2.120		
Methods/Class	-2.072	-2.117		
% Branches	-1.498	-1.399	-1.013	-1.379
Functions			2.552	2.389
Max Complexity	2.389	2.250		0.454
Avg Complexity	-1.653	-1.546		
Max Depth		1.273		
Avg Depth	2.487	2.380	2.328	3.638
Function call	-0.814	-0.753	-1.376	-1.358

まず，データセットから無作為に 100 ファイルを抽出する．次に，AFL と AFLFast をそれぞれ適用する．このとき，ファジングの打ち切り時間は 15 分で行う．そして，SourceMonitor や自作スクリプトを用いてメトリクス値を収集する．最後に，AFL と AFLFast から得られたファジング結果とメトリクス値の関係を線形回帰分析によって調査する．分析では線形モデル $y = a_0 + a_1x_1 + a_2x_2 + \cdots + a_nx_n$ を用いる．y を説明したい目的変数，x を説明変数とすると，ある x の係数 a は x がどれだけ y を説明することに寄与しているかを表す．線形回帰分析には，R パッケージの lm 関数と step 関数を組み合わせて利用する．メトリクス値は，常用対数で対数化して利用する．

AFL，AFLFast をそれぞれ適用してメトリクス値を収集した．その後，線形回帰分析を行い，説明変数の係数をまとめたところ表 3 となった．線形回帰分析は検出したク

ラッシュ数，パス数を目的変数，収集した全メトリクス値を説明変数として行った．表 3 で除外されているメトリクス値が存在する原因は，step 関数が適当なモデルを探索するなかで該当の項を除外したためである．

表 3 から，検出したクラッシュ数においてツール共通で Max Complexity に正の相関があり，Avg Complexity に負の相関があることがあることがわかった．また，Class Defs に正の相関や Methods/Class に負の相関が存在することがわかった．検出したクラッシュ数，パス数共通では，Lines，Branches および Function call に負の相関，Avg Depth に正の相関があった．ただし，Lines，Branches および Function call の相関は他のメトリクス値と比較して小さかった．ツール間の比較を行うと，検出したパス数において AFL のみ Max Depth に正の相関が，AFLFFast のみ Max Complexity に正の相関があった．値は他のメトリクス値と比較して小さかった．

5. 考察

調査結果から，ファジングツールの評価方法と利用方法について考察する．まず，評価方法について考察する．ファジングの打ち切り時間について考えると，Codeforces 提出ソースファイルに対してファジングを適用する場合，RQ1 の結果から，ファジングの打ち切り時間が 15 分と 6 時間で有意差がないことがわかった．そのため，Codeforces 提出ソースファイルを利用すれば，評価におけるファジングの打ち切り時間を短縮し，一定の時間で多くのソースファイルをテストできる可能性があることがわかった．

データセットについて考えると，AFL と AFLFast の比較 [3] では AFLFast は AFL よりも優位とされていたが，RQ2 の結果から，異なるデータセットを利用した場合，ツール間に有意差がないことがわかった．

AFL と AFLFast のファジング結果に差があることが知られているソースファイルに近いメトリクス値を持つソースファイルを利用して追加調査を行ったが，ツール間に有意差は認められなかった．追加調査には，AFL と AFLFast の比較 [3] でファジング結果に差があることが知られている GNU binutils に含まれる cxxfilt を利用した．利用したソースファイルに近いメトリクス値を持つソースファイルをユークリッド距離を用いて 100 ファイル抽出し，AFL および AFLFast を適用した．そして，そのファジング結果に対して検定を行ったところ検出したクラッシュ数について $p = 0.993$，検出したパス数について $p = 0.966$ で，有意水準 5%でツール間に差は認められなかった．

以上の結果は，データセットによって，結論が変わる可能性があることを意味する．そのため，ファジングツールの比較評価では多様なデータセットを用いて評価を行う必要があることがわかった．

次に，ファジングツールの利用方法について考察する．ファジング結果とテスト対象のメトリクス値の関係は RQ3 で示されている．この調査結果を利用してテスト対象のファジング結果を向上させる方法として，RQ3 で示された関係をもとにテスト対象を加工することが考えられる．今回の場合では，検出したクラッシュ数について平均の複雑度に負の相関があるため，複雑度を低下させるようにテスト対象を加工することで，検出したクラッシュ数が増加する可能性がある．また，検出したクラッシュ数について最大の複雑度に正の相関があるため，ソースファイル内に特にテストしたい関数などがあれば，その部分の複雑度を高くすることによってファジングが重点的に行われるようになると考えられる．

本調査では，先行研究の比較評価 [3] と異なり，初期テストケースを空データではなく競技プログラミングが提供するテストケースとした．そのため，初期テストケースの変更が調査結果に影響を与えた可能性がある．利用する初期テストケースを変更したのは，競技プログラミングの制約条件を満たすテストケースを重点的にファジングするためである．競技プログラミングの問題には，与えられる数値は a 以上 b 以下といったテストケースの制約条件が存在する．空のテストケースからファジングを開始した場合，制御コードといった制約条件を満たさないものが重点的に探索される．これを避けるために，利用する初期テストケースを変更した．

6. おわりに

本研究では，ファジングツールの評価において統計的な評価があまり行われていないという問題と，テスト対象に対する評価が行われていないという問題を提起した．そして，Codeforces 提出ソースファイルに対してファジングツールの適用とメトリクス値を用いた分析を行った．

その結果，Codeforces 提出ソースファイルにファジングを適用する場合，ファジング時間を 6 時間から 15 分に短縮できる可能性があることがわかった．そして，データセットによってツールの比較評価が異なる結果を導く可能性があることを示した．さらに，テスト対象の平均の複雑度を低下させるようにテスト対象のソースファイルを加工するなど，RQ3 で示された相関にしたがってソースファイルを加工することにより，ファジング結果が向上する可能性があることがわかった．

今後の課題として，比較するツールを増加させることと RQ3 で示されたファジング結果とメトリクス値の相関の調査が挙げられる．今回は，AFL とそれを拡張した AFLFast の比較的似たファジングツールを使用したが，新たなファジングツールを比較対象に加えて多様なファジングツールの評価が必要であると考えている．また，RQ3 で示された相関関係に従ってソースファイルを加工し，ファジング結果の変化を調査することを考えている．

謝辞 本研究にあたり，豊田工業高等専門学校情報工学科の藤原賢二助教に分析結果の考え方など多くの助言を賜りました．深く感謝いたします．

参考文献

[1] 上原忠弘. シンボリック実行を活用した網羅的テストケース生成 (特集研究開発最前線)–(ICT を支える先端技術デバイス, ものづくり技術). *Fujitsu*, Vol. 66, No. 5, pp. 34–40, 2015.
[2] Michal Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl`, (2019 年 2 月 3 日閲覧).
[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.
[4] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proc. of CCS*, pp. 2123–2138. ACM, 2018.
[5] Godefroid, Patrice and Levin, Michael Y and Molnar, David. Automated whitebox fuzz testing. In *Proc. of NDSS*, Vol. 8, pp. 151–166, 2008.
[6] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, Vol. 33, No. 12, pp. 32–44, 1990.
[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proc. of CCS*, pp. 2329–2344. ACM, 2017.
[8] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *ICSE Companion*, pp. 61–64. ACM, 2018.
[9] 堤祥吾. プログラミングコンテスト初級者・上級者間におけるソースコード特徴量の比較. 大阪大学大学院情報科学研究科修士論文, 2018. http://sel.ist.osaka-u.ac.jp/lab-db/Mthesis/contents.ja/137.html.
[10] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. *Structured testing: A testing methodology using the cyclomatic complexity metric*, Vol. 500. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1996.

付　　録

A.1 Reserved word branch に含まれる予約語

if，else，or，while，break，continue，goto，switch，case，default，return，do，catch．

A.2 Reserved word declaration に含まれる予約語

bool，char，class，const，const_cast，double，dynamic_cast，enum，explicit，extern，float，friend，int，long，namespace，new，operator，private，protected，public，reinterpret_cast，short，signed，static，static_cast，struct，template，typedef，typename，unsigned，void．

A.3 Reserved word other に含まれる予約語

and，and_eq，asm，auto，bitand，bitor，compl，delete，export，FALSE，inline，mutable，not，not_eq，or，or_eq，register，sizeof，this，throw，TRUE，try，typeid，union，using，virtual，volatile，wchar_t，xor，xor_eq．