一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2019-62,CPSY2019-60,RECONF2019-52(2020-01)

データ生成プログラムを利用したデータ項目の型推定に基づく変異ベースファジング

樋口　瑛子[†]　石浦菜岐佐[†]　難波　学之[†]

† 関西学院大学 理工学部　〒669–1337 兵庫県三田市学園 2–1

あらまし　本稿では, ソフトウェアのセキュリティを対象としたファジングの効率を向上させるための情報を, データを生成するプログラムから抽出する手法を提案する．変異ベースのファジング手法は, 実装が容易で汎用性が高いが, 有効なテストデータの生成率が必ずしも高くないという課題がある．シードとなるデータの文法や構造情報を与えることによって, ファジングの効率を向上させる手法が提案されているが, この情報は人手で与える必要がある．本稿の手法は, シードとなるデータを生成するプログラムをデバッガの下で監視し, データを出力する関数の引数からデータ項目のサイズや個数を抽出する．ここから, データ項目の境界やデータ項目の型を推定し, 各データ項目に適した変異を行う．提案手法に基づくツールを Ruby 2.5.2 を用いて実装し, テストを行った結果, データ項目を考慮しない場合と比較してエラー検出率が向上した．

キーワード　ファジング, 変異ベース, データ生成プログラム

Mutation Fuzzing Based on Type Estimation of Data Items Utilizing Data Writer

Yoko HIGUCHI[†], Nagisa ISHIURA[†], and Noriyuki NAMBA[†]

† Kwansei Gakuin University, Gakuen 2–1, Sanda, Hyogo, 669–1337, Japan

Abstract　This article proposes a novel way of acquiring information, which is used for enhancing efficiency of fuzzing for software security, from data generation programs. Although mutation-based fuzzers are generally easier to implement and more applicable than grammar-based fuzzers, they face the challenge of low data quality. Better results may be attained by providing information on data formats or grammars, it requires quite a human effort. Our method runs an appropriate data generation program under a debugger to record the size and the numbers of the data items as well as their values at every call to data output subroutines. Then data items are effectively mutated based on the guess about the data format from the information. A prototype system based on our method has been implemented in Ruby 2.5.2, which demonstrated higher error detection ability than a rundom fuzzer.

Key words　Fuzzing, Mutation Fuzzing, Data writer

1. はじめに

ソフトウェアの脆弱性は社会的に深刻な問題をもたらしており, リリース前に十分なテストを行うことが重要な課題になっている．ファジングは, 自動的に生成した大量の正当/不当な入力データを用いて対象プログラムをテストする手法であり, 脆弱性検出の有効な一手法である [1]．ファジングには, データ形式の仕様書に基づいて一から入力データを生成する生成ベースの手法と, 既存のデータの一部を変異させることによって新たなデータを生成する変異ベースの手法がある．後者のほうが実装が容易で汎用性が高いが, 効率的なデータの生成が困難で, 検出能力が向上しないという課題がある．

変異ベースファジングの効率を向上させる手法としては, データ構造の情報を利用して, 正当な入力の生成確率を上げる手法 [6] や, 入力データの文法の情報に基づいて変異を行う手法 [4] がある．しかし, いずれもそれが既知でない場合, あるいはその記述作成に手間を要する場合には適用ができない．

そこで本稿では, ターゲットプログラムへの入力データを生成するプログラムを利用して, 入力データの構造情報を抽出する手法を提案する．そこで本稿では, 入力データの構造情報をターゲットプログラムへの入力データを生成するプログラムを利用して取得する手法を提案する．データ生成プログラム中のデータ出力関数を監視して得られるデータのサイズや個数からデータ項目の境界や型を推測し, この情報に基づいてデータ項目の変異を行う．

提案手法に基づくツールを Ruby 2.5.2 を用いて実装しテス

This article is a technical report without peer review, and its polished and/or extended version may be published elsewhere.
Copyright ©2020 by IEICE

トを行った結果, データ項目を考慮しない場合と比較してエラー検出率が向上した.

以下, 本稿では 2 章で ファジングついて述べ, 3 章で本手法に基づくシステムおよび, 推定方法について述べる. 4 章で実装および実験について述べた後, 5 章でまとめと今後の課題を述べる.

2. ファジング

2.1 ファジング

ファジングは, 自動生成した大量の入力データによりプログラムをテストする手法である. 入力は, ファイルで与えられる場合もあれば通信プロトコルを介して与えられる場合もある. また, データは HTML 等のテキストデータの場合もあれば, 画像ファイルの様にバイナリデータの場合もあるが, 本稿ではバイナリデータのファジングを対象とする. セキュリティのテストの場合には, バッファオーバーフロー攻撃や DoS 攻撃に利用可能なプログラムのクラッシュ, 無限ループ, メモリ異常が起きないかどうかをこれらの入力により調べる.

2.2 ファジングのアプローチ

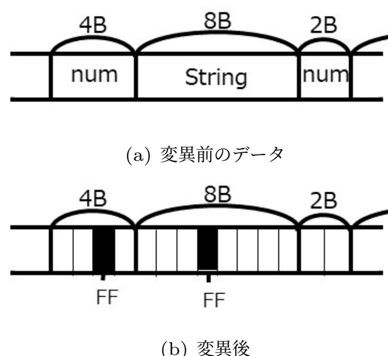
ファジングのデータ生成方法には, 生成ベース手法と変異ベース手法がある.

生成ベース手法は, データの仕様に基づいてデータを生成するものである. ターゲットプログラムの入力検査で棄却されにくく, コード網羅率の高いデータを生成できるという利点がある. しかし, 生成ツールの開発コストが大きく, また, データの仕様が非公開の場合には生成ツールの開発が難しい.

変異ベース手法は, 既存のデータを「シード」とし, その一部を書き換える (変異させる) ことにより, テストデータを生成する手法である. 基本的なファジング手法では, データ中のバイトをランダムに選択してそれをランダムなバイト, 最大値 (0xFF), あるいはビットを反転させたもの等に書き換える. 一般に実装が容易で, 汎用性が高いという長所がある. しかし, データがターゲットプログラムのエラー処理で棄却される確率が高く, 有効なデータが生成されにくいという欠点がある.

2.3 変異ベースファジングの課題とその解決のアプローチ

バイナリデータには, 例えば図 1(a) のように, サイズや型の異なるデータ項目がそのデータのフォーマットに従って並べられている. テスト効率の観点からは, 各データ項目は, そのデータ項目で表現可能な最小値や最大値等の境界値や, 問題を起こしやすい特定の値に変異させることが望ましい. しかし, データフォーマットの情報を持たない単純な変異ファジングでは, 図 1(b) の様にデータ項目の一部を書き換えるような変異しか行えない.



(a) 変異前のデータ

(b) 変異後

図 1: ランダム変異手法の流れ

このような変異ベースファジングの欠点を解決する方法がいくつか提案されている.

その一つは, 図 2 のようにターゲットプログラムのコードの情報を利用する方法である. 図中の Ⓜ は変異を表し, Target は テスト対象コードを表す. コード解析の結果に基づいて特定のコード部分に到達できるような入力データを生成する手法や, カバレッジを計測してそれを入力データの変異にフィードバックする方法等がある. American Fuzzy Lop [3] は, このこの手法を応用して, bash や OpenSSH といった様々なソフトウェアの不具合を検出している.

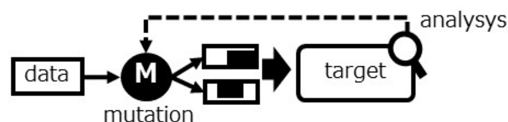


図 2: テスト対象コード解析による変異

また, 図 3 の様に, 入力データのフォーマットに関する情報を与えることによって, 変異ファジングの品質を向上させる手法も提案されている.

Pham [6] は, 入力データ中の「チャンク」を単位として変異を行う手法を提案している. チャンクは, ヘッダ, データ, チェックサムなどから構成される一まとまりのデータであり, バイナリデータにはチャンクの並びにより構成されるものがある. この手法では, フォーマット情報を用いてチャンクを認識し, チャンクの削除, 他のデータからのチャンクの挿入, チャンクの入れ替えなどを行って変異データを生成する. しかし, ユーザがチャンクのフォーマット情報を与える必要があり, 生成ベースの手法と同様の課題がある.

Wang [4] は, 文脈自由文法に基づいて入力データの集合 (コーパス) を学習し, テストに有効な文脈を反映したデータを生成する手法を提案している. 入力データが XML や JavaScript 等のテキストデータの場合には有効であるが, バイナリデータへの適用は難しい. また, この手法でもユーザが入力データの詳細な文法を与える必要がある.

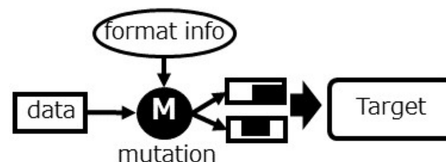


図 3: 文法情報を利用した変異

3. データ生成プログラムを利用したデータ項目の型推定

3.1 概　　要

本稿では, 変異ベースのファジングを効率的に行うために必要となる入力データの構造に関する情報を, データ生成プログラムを利用することによって取得する新たな手法を提案する.

提案手法の処理の流れを図 4 に示す. 図中の Writer はデータ生成プログラムを表す. データ生成プログラムとしては, 例えば入力データが JPEG ファイルの場合には, 別のフォーマットの画像ファイルを JPEG に変換して出力するプログラムや, 画像データを自動生成するプログラムなどが利用できる. 本手法では, Writer の出力をターゲットへの入力データとして利用

すると同時に, Writer の出力部を監視することによって, そのデータの構造に関する情報を取得する．この情報をデータの変異に反映させることにより, ファジングの効率向上を狙う．

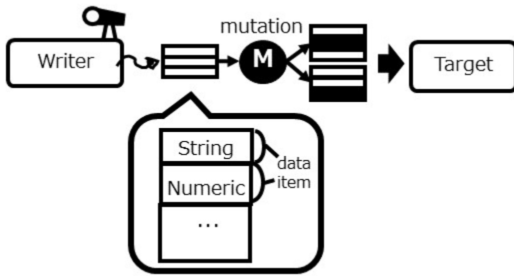


図 4: 本手法の流れ

3.2 データ出力関数とその呼び出しの監視

データ出力関数としては, 例えば Writer が C 言語で書かれている場合には fwrite 関数が利用できる．fwrite 関数は以下の 4 つの引数を持つ．

- buf : データの先頭ポインタ
- size : データのバイト数 (サイズ)
- count : データ数
- fp : ファイルポインタ

GDB 等のデバッガを用いて fwrite 関数の呼び出しを監視し, 引数の値を取得すれば, データ項目の境界を推定できる．また, データ項目のバイト数やデータそのものの情報から, データ項目の型推定が可能である．

3.2.1 データ項目の境界および型の推定

本稿の手法では, データ出力関数の引数から得られる, サイズ, データ数, データの値の 3 つに基づいてデータ項目の境界および型の推定を行う．

最も単純な方法は, 書き込みデータのサイズからデータ項目の型を決定する方法である．例えば, データのサイズが 4 バイトの場合, それは文字列よりも 4 バイトの数値データである可能性が高いと推定できる．

しかし, 書き込みデータのサイズが実際のデータ項目のサイズに等しいとは限らない．例えば, 2 バイトのデータ 8 個を出力する場合, それを 1 バイト × 16 個, あるいは 4 バイト × 4 個として出力するプログラムが存在する．また, ヘッダ情報のように, 数値データと文字列データが混在した 128 バイトのチャンク (データの塊) を 4 バイト × 32 個のデータとして出力することもある．

そこで本手法では, まずサイズ × 個数のデータの内容を検査して, それが数値データか文字列データか, あるいはその混在かを判断する．これに, サイズ情報を併用してデータ項目の境界を決定するという方針をとる．これによって必ずしも正確な推定が行えるわけではないが, ある程度は変異の有効性が向上すると考えられる．

データ項目の型および境界の推定の処理の流れを図 5 に示す．1 つの出力関数で出力されるデータごとに, その内容を検査して, (1) まず文字列と判断される部分を決定し, (2) 次に文字列と判断された以外のデータを数値とし, 出力関数のサイズからデータ項目の境界および各データのサイズを決定する．

（1） 文字列部分の推定

まず, サイズ × データ数のバイトデータを走査し, 「文字」と判定されるバイトが m 個以上連続する部分を文字列データと判定する．バイトデータが文字列かどうかは, ASCII コードに基づいて, それが ‘a’ や ‘7’ 等の可読文字, あるいは改行や NULL 文字等の特殊文字であれば文字, そうでなければ数値と推定する．m の値は 5 程度とする．

（2） 数値データの境界と型の推定

文字列と推定された部分以外は全て数値と推定し, データ項目の境界を決定する．まず, データ出力関数のサイズでデータを区切る．区切られたデータに文字列が含まれている場合, そのデータは全て 1 バイトデータとする．数値のみの場合, サイズが 2, 4, 8 バイトの場合はそのデータ項目はそのサイズの整数データとし, それ以外の場合は, 1 バイトの整数データの並びであるとする．

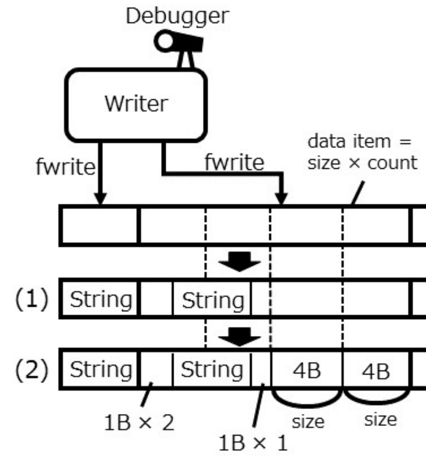


図 5: データ項目の型推定の流れ

3.3 型推定に基づく変異

データ項目の変異は前節で説明したデータのサイズと型に基づいて行う．変異のさせ方を表 1 にまとめる．

文字列データは, 特殊文字 (‘\n’, ‘\t’ 等) や記号 (‘{’, ‘@’ 等) に高確率で変異させる．それ以外の場合にはランダムな可読文字に変異させる．

数値データは, そのデータを符号付き整数または符号なし整数と見たときの最小値, 最大値に高確率で変異させる．また, 元の数値を ±10 変化させた数値やビットを反転させた値にも高確率で変異させる．それ以外の場合には, ランダムな数値に置き換える．

データの変異を行うかどうかはランダムに決定する．まず, 一つの出力関数で書き込まれるデータを変異させるかどうかを決定し, 変異させる場合にはその中のデータ項目それぞれについて変異させるかどうかを決定する．

各書き込み単位で変異させるかどうかは, あらかじめ定めた一定の確率で決定する．データ項目ごとの変異確率は, データ数が少ないほど高く設定する．これは, データ数が多い場合には画素データのように変化させてもプログラムには影響の少ない場合が多く, 逆に少ない場合にはヘッダやデータ数のように重要なデータ項目である場合が多いためである．

3.4 データ出力関数以外である場合の対策

着目している出力関数が出力するデータを変異させたものは, 必ずしもそのままファジングのデータとして用いることはできない．

理由の一つは, データ生成プログラムの出力は着目している出力関数だけで行われているとは限らないためである．C 言語の場合, fwrite 関数以外にも fput 等の様々な出力関数が存在し

表 1: 型に応じた変異

型	変異後の値
文字	特殊文字 (\n, \t 等), 記号 ({, @ 等), ランダムな文字
1B 整数	0xFF, 0x00, 0x7F, 0x80, 元数値 ±10, ビット反転, 乱数
2B 整数	0xFFFF, 0x0000, 0x7FFF, 0x8000, 元数値 ±10, ビット反転 乱数
4B 整数	0xFFFFFFFF, 0x0, 0x7FFFFFFF, 0x80000000, 元数値 ±10, ビット反転, 乱数
8B 整数	0xFFFFFFFFFFFFFFFF, 0x0, 0x7FFFFFFFFFFFFFFF, 0x8000000000000000, 元数値 ±10, ビット反転, 乱数

ており, データの生成はこれらの関数を併用して行われている場合がある. 最も低レベルの出力関数を監視することも考えられるが, その場合にはデータ項目の境界や型を推定するのに必要なデータのサイズや個数を取得できるとは限らない.

また, 着目している出力関数の全ての呼び出しがデータ生成に用いられているとは限らない. ログの出力やエラーメッセージの出力にも同じ関数が使われることがあるためである.

そこで本手法では, 変異させたデータをそのまま用いるのではなく, 図 6 のようにデータ生成プログラムを再度実行し, 注目している出力関数に変異したデータを出力させるようにする. これは, デバッガの元でデータ生成プログラムを実行して出力関数の呼び出しを補足し, 出力データを変異データに上書きすることにより実現できる. これによって, ファイルのエンディアン等の環境依存を考慮する必要もなくなる.

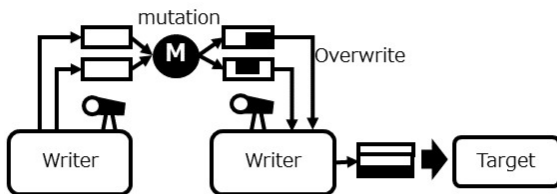


図 6: 変異したデータを反映させる流れ

4. 実験結果

本稿の手法に基づくシステムを Ruby2.5.2 で実装した. データ生成プログラムとしては, C 言語で書かれたデータフォーマット変換プログラムを用い, fwrite 関数の呼び出しを GDB で捕捉した.

実験の流れを図 7 に示す. (1) は比較用の単純なランダム変異手法, (2) は本手法である. (1) では, データ生成プログラムの出力データの中からランダムにバイトを選んで変異させたものをテスト対象プログラムへの入力とした. (2) では, fwrite 関数で取得した情報に基づいてデータを変異させ, 再度 Writer を実行し, それを fwrite 関数で書き出して得られる出力を Target に入力した.

(2) では, 文字が 5 個以上連続した場合にはそれを文字列と判定した. 変異確率は, 各 fwrite に対して 1/10 で変異させ, 各 fwrite のデータ項目については, データ数 c が 10 未満の場合には $1.2/c$, データ数が 10 以上の場合には $1.0/c$ とした. (1) の変異確率は, データ中で変異させるバイト数の割合が (2) と同じになるようにした. 変異させたデータを入力することによって Target が無限ループに陥ることがあるため, Target の実行は 10 秒で打ち切った.

実験に使用した Writer と Target, 及びデータのフォーマットを図 8 に示す. Writer には opusdec-1.1.2, Target には

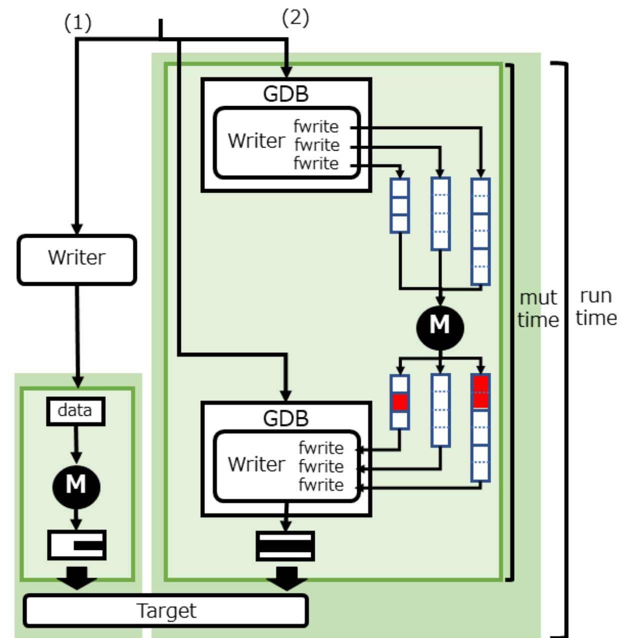


図 7: 比較実験の流れ

ffmpeg-3.2.3 および opusenc-1.1.2 を使用した. これらはいずれも音声データを他のファイルフォーマットに変換するプログラムである. opus 形式のファイルを入力として opusdec が出力した wav 形式のファイルを変異のシードとした. ffmpeg は mp3, mp4, mov, wma, flac, および ogg 形式のデータを出力させる 6 通りのモードで実行した. opusenc には opus 形式のデータを出力させた.

(1) と (2) の比較を ffmpeg の mp3 出力処理に対して行った. 結果を表 2 に示す. 10 個のシードデータからそれぞれ 1,000 個の変異データを生成し, 10,000 のデータによりテストを行った. (1) (2) いずれのプログラムでもデータ中でバイトが変異している割合は 1.35% であった. “run time”, “mut time” はそれぞれ実行全体に要した時間, ファジングデータの生成に要した時間である (図 7 参照). 本手法がファジングデータの生成に要した時間はランダム手法の約 2.35 倍であるが, これは Writer をデバッガの下で 2 回実行しているためと考えられる. Target の実行まで含めた実行時間は 2 倍未満である. “通過率” は, テストデータが ffmpeg のエラー処理で棄却されなかった率であり, 本手法の方が有効なデータを生成できていることがわかる. “#error” は 検出したエラーの数を表し, ランダム変異手法では 0 件であったのに対し, 本手法では 89 件のエラーを検出できた. 89 件のエラーは全て ffmpeg のタイムアウトであった.

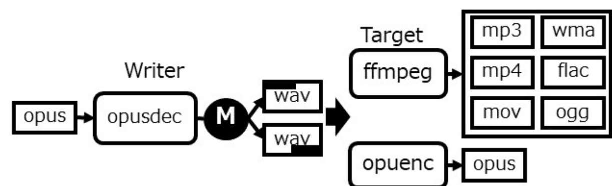


図 8: 実行の流れ

ffmpeg のその他 6 つの出力形式への変換処理, および opusenc に対するテストを行った結果を表 3 に示す. いずれの実験においても 1 つのシードデータから, 1,000 件の変異データを生成してテストを行った. ffmpeg の mp3 出力処理系

表 2: 実験結果

手法	#seed × #test	run time (mut time) [h]	通過率	#error
(1) ランダム	10 × 1,000	2.802 (1.229)	75.2%	0
(2) 本手法		4.943 (2.877)	88.8%	89

Writer:opusdec-1.1.2, Target:ffmpeg-3.2.3 (Intel Core i7-6850K@3.60GHz×6)

で 16 件, mov および mp4 出力処理系において 19 件のエラーを検出した.

これら 54 件のエラーは全てタイムアウトであった. ffmpeg の実行を解析したところ, いずれも同じ箇所で無限ループに陥っており, これは DoS 攻撃に対する脆弱性と考えられる.

表 3: 実験結果

Target	入力形式	出力形式	通過率	#test	#error
ffmpeg	wav	mp3	88.8%	1,000	16
		mov	81.9%		19
		mp4	83.7%		19
		wma	82.8%		0
		flac	57.5%		0
		ogg	82.8%		0
opusenc	wav	opus	57.1%		0

これらの実験結果は, ターゲットプログラムの内部構造やカバレッジ, および入力データの集合から推測できるデータフォーマット以外に, データ生成プログラムの動作情報がファジングの質向上に利用可能であることを示している. 従来法との組み合わせによりファジングの効率を向上できる可能性がある.

本稿では fwrite 関数の引数の情報のみを利用しているが, デバッガからは各 fwrite 関数がデータ生成プログラムのどの部分から呼ばれているかの情報も取得できる. これを用いれば, fwrite 関数の呼び出し毎の局所的なデータフォーマット情報だけでなく, 文法構造のような大域的な構造を抽出できる可能性がある.

一方で, ファジングデータの質を上げるためには, 多様なシードデータの生成も重要であり, このためにデータ生成プログラムをカバレッジ駆動で動作させることも一案として考えられる.

4. むすび

本稿では, 変異ベースファジングの効率向上のために, データ生成プログラムを利用する手法を提案した. データ出力関数の情報から, 推定したデータ項目の境界および型に基づいた変異により, ファジングの効率向上を実現した.

今後の課題として, データ生成プログラムをカバレッジ駆動で動作させる等によって, ファジングデータの質をさらに向上させることなどが挙げられる.

謝　　辞

本研究を行うにあたり, 御助言や御協力を頂いた, 関西学院大学理工学部石浦研究室の諸氏に感謝致します.

文　　献

[1] Michael Sutton, Adam Greene, Pedram Amini 著, ドキュメントシステム＋伊藤裕之 訳： ファジング ブルートフォースによる脆弱性発見手法, 毎日コミュニケーションズ (June 2008).

[2] honggfuzz (online), `https://github.com/google/honggfuzz` (accessed 2019-11-14).

[3] American Fuzzy Lop (online), `http://lcamtuf.coredump.cx/afl/` (accessed 2019-11-14).

[4] Junjie Wang, Bihuan Chen, Lei Wei and Yang Liu: "Superion: Grammar-Aware Greybox Fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2019)*, pp.724 735 (May. 2019).

[5] Tai Yue, Yong Tang, Bo Yu, Pengfei Wang, and Enze Wang: "LearnAFL: Greybox Fuzzing with Knowledge Enhancement," in *IEEE Access vol. 7* (Aug. 2019).

[6] Van-Thuan Pham, Marcel Bohme, Andrew E. Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury : "Smart Greybox Fuzzing," in *IEEE Transactions on Software Engineering (Early Access)*, pp. 1 17 (Sept. 2019).