Vol.2018-CSEC-82 No.31
Vol.2018-SPT-29 No.31
2018/7/25

KLEE と Libfuzzer を組み合わせたハイブリッド型 KleeFuzzer に関する提案

三嶋秀宗[†1]　中沢実[†1]　西川幸延[†1]

概要：現在，ソフトウェア開発は大規模であるため，リリース前にすべてのバグを手動で検出をすることが困難である．そこで，自動でバグを検出する手法として，fuzzing，symbolic execution が用いられている．しかし，fuzzing は，良い結果を得るために洗練された初期値が必要である．一方，symbolic execution では，ソフトウェアを検査するのに余計な時間がかかる問題やメモリと CPU のリソースを余分に消費してしまう問題がある．そこで，fuzzing 側で必要になる初期値に symbolic execution から生成される結果を用いることが考えられた．しかし，連携のシステムとして用いられている fuzzing のソフトウェアである American Fuzzy Lop (AFL) では，実行ファイル形式ではない単一のライブラリファイルに対して fuzzing を行うことが困難である．そこで，libfuzzer を AFL の代わりに用いることで実行ファイル形式ではない単一のライブラリファイルに対しても fuzzing を行うことが容易にできるようになる．本研究では，fuzzing のソフトウェアである libfuzzer と symbolic execution のソフトウェアである KLEE を組み合わせた新たなハイブリッドシステム KleeFuzzer を提案する．

1. はじめに

現在，ソフトウェア開発は大規模であるため，手動でバグを検出することは困難である．そのためリリース後に，ユーザーから報告されたバグ，開発者自身が見つけたバグをもとに，デバッグを行っている．しかし，ソフトウェアを使用するユーザーの中には，バグを利用してサイバー攻撃として利用する悪意を持ったユーザーも存在する．

従って開発者としては，悪意を持ったユーザーよりも早くバグを見つけて修正をしなければならない．しかし，ソフトウェア開発が大規模になっていることを考えるとバグを検出することは困難である．また，バグを検出するための，ソフトウェア開発の保守コストが多くかかる．

そこで，ソフトウェアに対して手動でバグを検出するのではなく，自動でソフトウェアのバグを検出することが考えられた．自動でバグを検出する手法として，ファジング（fuzzing），シンボリック実行（symbolic execution）が用いられている．

Fuzzing とは, 遺伝的アルゴリズムや Deep Neural Network（DNN）を用いてソフトウェアの入力となる値（入力値）を変異させ，ソフトウェアの出力がエラーとなる結果を見つける手法である．symbolic execution とは，ソフトウェアの変数を symbolic execution 側で用意した記号的な値（symbolic value）に置き換え，ソフトウェアの条件文を数学的に解き, 条件文の先にあるバグを検出する手法である．

しかし，fuzzing と symbolic execution の手法を用いてもソフトウェアの全てのバグを検出することができない．Fuzzing は入力値をもとにしたコーパスが必要になってくるため，より良い結果を得るためには，優れたコーパスを生成しなければならない．また，ソフトウェアの条件文の中に 4 バイト以上の長さの変数や定数の比較があった場合 fuzzing では条件文が真となる条件または偽となる条件を見つけることができない場合がある．一方，symbolic execution では，ソフトウェアの条件文が symbolic execution で解けない場合やソフトウェアに条件文が多い場合、または，ソフトウェアにループ処理がある場合に爆発的に冗長な命令が増えるため，バグを検出するのに余計な時間がかかる問題やメモリと CPU のリソースを余分に消費してしまう問題がある．

そこで，Fuzzing と symbolic execution の問題を解決するために連携させる方法が考えられた．連携させる方法は、fuzzing が不得意な部分を symbolic execution が処理を行い，symbolic execution が不得意な部分は fuzzing が処理を行うものである．これにより，今までの fuzzing と symbolic execution のそれぞれの実行結果よりも、良い結果が得られた[1,2].

しかし，fuzzing のソフトウェア（fuzzer）として使われている American Fuzzy Lop（AFL）は，実行ファイル形式ではない単一のライブラリファイルに対して fuzzing を行うことが困難である．そこで，libfuzzer を AFL の代わりに用いることで実行ファイル形式ではない単一のライブラリファイルに対しても fuzzing を行うことが容易にできるようになる[3,4].

本研究では，fuzzer である libfuzzer と symbolic execution のソフトウェア（symbolic execution engine）である KLEE を組み合わせた新たなハイブリッドシステム KleeFuzzer を提案する[5].

†1　金沢工業大学
Kanazawa Institute of Technology

Vol.2018-CSEC-82 No.31
Vol.2018-SPT-29 No.31
2018/7/25

2. 関連技術

2.1 Fuzzing の種類

Fuzzing には，ホワイトボックスファジング（white-box fuzzing），ブラックボックスファジング（black-box fuzzing），グレイボックスファジング（grey-box fuzzing）の 3 種類の手法がある[6].

White-box fuzzing は、ソフトウェアの中身を知っている前提で入力値を手動または自動で用意して，自動で入力値をソフトウェアに与える方法である．ソフトウェアを予め知っているため，優れた入力値を生成してテストを行うことができる．

しかし，ソフトウェアが大規模であった場合それを理解することが困難である．また，ソフトウェアが複雑であれば，条件文を網羅するテストケースの数も爆発的に増え，多くのテストケースが必要になるため，優れた入力値を多く生成することが困難である．

Black-box fuzzing は，ソフトウェアの中身を知らずに入力値を手動または自動で用意して，自動で入力値をソフトウェアに与える方法である．ソフトウェアの中身を知らずに入力値を用意しているため，どのような値を入れてもよい．そのため，入力値を自動で生成することができ，シンプルかつ素早くテストを行うことができる．

しかし，ソフトウェアの中身を知らないため，使用する入力値がソフトウェアにとってテストする価値が全くないものがテストケースの大部分を占める問題がある．例えば、条件文が真となる条件が文字列の hello としたとき，black-box fuzzing のソフトウェア（black-box fuzzer）は自動で入力値を生成し，自動でソフトウェアに入力を行う．つまり，black-box fuzzing がアルファベットの小文字を a から z まで順番に入力値を生成し，自動でソフトウェアに対して入力することを考えると、hello という文字列になるまでの試行回数は単純計算で26^5回になる．

Grey-box fuzzing は，white-box fuzzing と black-box fuzzing のハイブリッド手法である．Grey-box fuzzing は 2 つの方法がある．1 つ目は，ソフトウェアの中身を知らず入力値を手動または自動で用意し，自動で入力値をソフトウェアに与える方法である．2 つ目は，ソフトウェアの中身を知って，入力値を手動または自動で用意し，自動で入力値をソフトウェアに与える方法である．

Grey-box fuzzing の技術が使われているソフトウェア（grey-box fuzzer）によって入力された値が，閾値に達する出力があった場合，その入力した値をコーパスのシード値として，遺伝的アルゴリズムや DNN といったアルゴリズムを用いて，入力となる値を変える戦略に沿って変異させる．

grey-box fuzzing は，無作為に入力値を入力する black-box fuzzing よりも良い結果が得られる．また，grey-box fuzzing はソフトウェアの中身を知って，優れたコーパスを用意したときのほうがソフトウェアの条件分岐を網羅することができる．さらに，grey-box fuzzing は入力値を次々と変異させていくため，white-box fuzzing のように条件分岐を網羅するために，数多くの優れた入力値を用意する必要がない．

AFL と libfuzzer は grey-box fuzzer に該当する．本稿において，fuzzing は grey-box fuzzing のことを指す．

2.1.1 Fuzzing の課題

Fuzzing は，ソフトウェアの条件分岐を網羅することが重要である．ソフトウェアの条件分岐を網羅する範囲のことをカバー範囲という．例えば，すべての条件分岐を網羅することを 100%のカバー範囲にするという．つまり，カバー範囲を 100%達成できるように，優れたコーパスが必要になってくる．そのため，テストを行うソフトウェアのことをよく知り，入力値を用意することが重要である．

しかし，fuzzing は条件分岐のとき，単純な変数の定数比較であったとしても，条件文を解釈せずに入力となる値を自動で与えるため，条件文が真または偽の条件を見つけることが困難である．その例を図 1 に示す．

```
int main{
   int x;
   read(0, &x, sizeof(x));
   if(x == 0x12345678);
     coredump_error();
}
```

図 1　Fuzzing で解くことが難しい例

図 1 では，read 関数で変数 x の値を読み取り，条件文で変数 x を 0x12345678 と比較した後，真であれば coredump_error 関数が実行できる例である．

しかし，この条件文を fuzzing で解くことは困難である．整数型である変数 x を 4 バイトであると仮定すると単純計算で2^{32}回試行しなければ解くことができない．つまり，図 1 のような例を fuzzing するとき，新しい条件分岐先である真の条件を網羅することができず，カバー範囲を広げることができないため，新しい条件分岐先のバグを見逃してしまう場合がある．

2.2 Symbolic execution

Symbolic execution とはすべての変数を記号的な値（symbolic value）に置き換えて条件文を数学的に解いていく．また，条件文の条件式を symbolic value に置き換える仕組みを制約（constraint）という．制約は条件文があるたびに更新されていき，そこまでの条件文までにたどり着くための経路の制約のことをパス制約（path constraint）という．path constraint の更新例を図 2 に示す．

Vol.2018-CSEC-82 No.31
Vol.2018-SPT-29 No.31
2018/7/25

```
int main{
   int x,y;
   read(0, &x, sizeof(x));
   read(0, &y, sizeof(y));
   if(x > 10){
      if(x > y + 1){
         …
```

図 2　path constraint の更新例

Symbolic execution では変数をすべて symbolic value に置き換えるため変数 x は x0，変数 y は y0 に置き換わる．この時の path constraint を pct とおく．

最初の条件文である$x > 10$のとき，path constraint は更新され，$pct = x0 > 10$になる．次の条件文の$x > y + 1$とき，path constraint は更新され，$pct = (x0 > 10) \cap (x0 > y0 + 1)$になる．

path constraint が解けることを充足可能であるという．また，path constraint が解けないことを充足不可能であるという．この path constraint を解く仕組みをソルバー（solver）という．Solver を使って path constraint が充足可能であれば，真，偽の両方を探索していく．このとき、symbolic execution のソフトウェアでは真と偽の状態（states）として記録し，真と偽の states を同時に探索する．一方，例えば，solver を使って真の条件で path constraint が充足不可能である場合，その path constraint では真の条件でそれ以上探さないように states に記録する．

しかし，symbolic execution はソフトウェアのループ文の解釈が上手くいかず，path constraint が無限に増大してしまう問題がある．その例を図 3 に示す．

```
int add(int x){
   int sum = 0;
   int i = 0;

   for(; i < x; i++)
      sum += i;

   return sum;
}
```

図 3　path constraint の無限増大例

図 3 では，ユーザー関数である add 関数の第 1 引数に整数型のユーザー引数 x があり，その値に応じて for 文でループを行う．しかし，図 3 の例で symbolic execution を行うとすべての変数を symbolic value に置き換えるため，for 文を何回繰り返せばよいのか不明である．そのため，solver で for 文の解釈を行うと無限に path constraint の制約が増えてしまう問題がある．

この問題を解決するために，Dynamic Symbolic Execution(DSE)という手法が考えられた．

2.3 Dynamic Symbolic Execution

DSE とは，symbolic value と具体値（concreate value）のハイブリットな仕組みである．以下に DSE の基本的な動作を示す[7].

1. symbolic value にする入力となる変数を指定する．
2. symbolic value に指定した入力となる変数に，初期値となる concreate value を与える．
3. ソフトウェアを実行する．
4. ソフトウェア実行後に得られた path constraint から，次に探索する経路を決定する．
5. Solver を使って最後の経路で path constraint の真理値を逆にし，入力となる concreate value の特定を行う．
6. もし，solver を使って解けない場合は，別の経路で path constraint の真理値を逆にし，入力となる concreate value の特定を行う．
7. 4 に戻る．

図 4 を用いて，DSE が解く様子を以下に示す．

```
1  add(int z){
2     int sum = 0;
3     int i = 0;
4
5     for(; i < z; i++)
6        sum += i;
7
8     return sum;
9  }
10
11 int main(){
12    int x;
13    int y;
14
15    read(0, &x, sizeof(x));
16    read(0, &y, sizeof(y));
17
18    if(x > 10){
19       if(x > y + 1){
20          add(x);
21       }
22    }
23    return 0;
24 }
```

図 4　DSE で解くソースコードの例

1. 整数型の変数 x と y を symbolic value に指定し、変数 x を x0，y を y0 という名前にする．
2. $x0 = 0$と$y0 = 0$を初期値として与えてソフトウェアを実行する。
3. $x0 = 0$であるため，18 行目の条件が満たせず偽になる．path constraint は，$pct = x0 > 10$を得る．次の経路を探すために solver で path constraint を解き，$x0 = 11$で真になる解を得る，このとき次の経路は，18 行目が真になる経路に決定する．
4. $x0 = 11$と$y0 = 0$を初期値として，ソフトウェアを実

Vol.2018-CSEC-82 No.31
Vol.2018-SPT-29 No.31
2018/7/25

行する．このとき 19 行目の条件は真になるため，add 関数が実行される．このとき，add 関数のユーザー引数は$x0 = 11$であるため，ループ文を済ませて終了する．path constraint は，$pct = 11 > y0 + 1$を得る．

5. 次の経路を探すために 19 行目の条件が偽になる値を solver で探す．この時$x0 = 11$，$y0 = 11$を得る．
6. $x0 = 11$，$y0 = 11$を与え，19 行目の条件が偽になり，すべての条件文を網羅したため終了する．

DSE は symbolic execution とは違い，2 つの利点がある．

1. 入力となる変数を限定している．
   symbolic value に置き換える変数を指定しているため，図 3 のような問題　が起きにくい点である．
2. ソフトウェアを実行する
   concreate value を得るために, 実際にソフトウェアを実行させて，path constraint を solver で解いているため，図 3 のような問題が起きにくい点である．

2.3.1 Dynamic Symbolic Execution の課題

DSE は symbolic execution とは違い，symbolic value に置き換える入力値を限定し，プログラムを実行することによって，図 3 のような例の問題を解決できる．しかし，DSE には以下の 3 つの課題がある．

1. DSE は実行速度が遅い．
   実行速度が遅い原因が 2 つある。1 つ目は，DSE のソフトウェア(symbolic execution engine)が実行する命令コードを解釈する速度が遅いことである．2 つ目は，solver が path constraint を解く速度が遅いことである．特に，solver が path constraint を解くところは，NP 完全問題を含んでいるため，解くのに膨大な時間がかかることによる充足不可能がある．そのため，symbolic execution engine が NP 完全問題を見極めて，早い段階で充足不可能であることを決定しなければならない．
2. States 爆発問題がある．
   States 爆発問題とは，条件文で真または偽の条件で実行可能な states を記録していくときに，実行可能な states が指数関数的に増え，余計にメモリや CPU リソースを消費してしまう問題である．States 爆発問題を起こす要因は，ソフトウェアにループ文や条件文が多い場合または，再帰的な処理があった場合に states の数が爆発的に増える．
3. 経路爆発問題がある．
   経路爆発問題とは，条件分岐の時に，真と偽の両方が実行可能であればそれぞれの経路を実行してそれを states として保持する．つまり，ソフトウェアに条件文が多いと組み合わせ次第で指数関数的に経路の数が増えていく．そのため，どの経路選択して実行すればよいのかという問題がある．

2.4 fuzzing と Dynamic Symbolic Execution の連携

fuzzing と DSE の課題となっている部分をお互いにカバーするために fuzzer と symbolic execution engine の連携を行うことが考えられた．文献[1]の Driller では，AFL と angr と連携を行っている[8]．図 1 で示したような fuzzer で解くことが難しい問題は symbolic execution engine に任せ，2.3.1 節で示したような symbolic execution engine で解くことが難しいようなことは fuzzer に任せるようなことを行っている．しかし，文献[1]では，3 つの課題がある．1 つ目は，fuzzer と symbolic execution engine の連携の通知がうまくいいかないことである．つまり，fuzzing の課題となっている部分，DSE の課題となっている実行する命令コードを見極めて適切に通知して，それぞれに任せるように処理するようにしなければならない．2 つ目は，新たなカバー範囲を見つけたときにお互いに通知して適切に処理をしなければならないことである．つまり, fuzzer と symbolic execution engine は，全く同じ構造の states を共有データとして保持しなければならない．3 つ目は，fuzzer と symbolic execution engine でも対処できないような states がある場合，適切に処理をしなければならないことである．例えば，ハッシュアルゴリズムのような複雑な計算があった場合の例外処理としてそこは処理しないようにするといったことを連携するソフトウェアで考えなければならない．

文献[2]の KleeFL では，AFL の課題となっていたコーパスの提供を KLEE で生成することによって，解決することができる．しかし，KleeFL には 4 つの課題がある．1 つ目は KLEE を使って path を見つけているため、KLEE と同じカバー範囲である．2 つ目は，KLEE が生成したシード値を与えるため，初動のオーバーヘッドが大きいことである．3 つ目はシード値を 1 度しか与えていないため、KLEE で新たにシード値を得ても AFL に与える方法がないことである．4 つ目はハッシュを生成するような計算アルゴリズムを解かないようする計算制限がないことである．

3. KleeFuzzer の提案

KleeFuzzer は，libfuzzer と KLEE を連携させたシステムである．これまでの fuzzer と symbolic execution engine の連携とは違い，実行ファイル形式ではない単一のライブラリに対しても fuzzing を容易に行うことができる点である．これまでの連携では fuzzer として AFL が使われていたため，実行ファイル形式ではない単一のライブラリに対しては fuzzing を行うことが困難であった．

そこで，fuzzer として AFL の代わりに libfuzzer を用いることによって，実行ファイル形式ではない単一のライブラリに対しても fuzzing を行えることを考えた．

Vol.2018-CSEC-82 No.31
Vol.2018-SPT-29 No.31
2018/7/25

3.1 連携するソフトウェアについて

3.1.1 Libfuzzer

Libfuzzer とは Low Level Virtual Machine（LLVM）の compile-rt プロジェクトのライブラリである[9]．LLVM とはあらゆるプログラミング言語で利用できるコンパイラである．Libfuzzer は同じ LLVM のプロジェクトの clang という C 言語, C++言語と Objective-C のコンパイラであるオプションとして提供されている．

Libfuzzer は同じ fuzzer である AFL とよく似ている．Libfuzzer は以下のような目標で実装されている．

1. Fuzzing をするときに同じプロセスで異なる入力しても何回も実行できる設計である。
2. どんな種類の入力があっても耐えられるような設計である．
3. どんな入力に対しても exit はしない，
4. 入力が何バイト行われるかわからない非決定論は可能な限り行わない．
5. とにかく速く実行し，メモリや CPU リソースを余計に消費しないような設計である．
6. できるだけ、グローバルデータとしてある共有リソースに対して変更を行わない．
7. データに影響を与えない最少単位までトリミングを行い効率よく fuzzing を行う．

Libfuzzer は対象となるファイルに対してのコーパスに依存している．このコーパスは，対象となるファイルに対しての有効な入力と無効な入力の様々な種類のシードがあることが理想である．そのため，対象のファイルが複雑な仕組みであり，コーパスが優れていない場合は効率が低下する．

また，libfuzzer は入力となる値（入力値）を様々な戦略をランダムに選んで変異させていく．以下に入力を変異させる戦略（変異戦略）を示す．

- ユーザーが提供した変異戦略を使って入力値を変異させる．
- ユーザーが提供したシードを使用して入力値を変異させる．
- 入力値をバイト単位でランダムに選んでシャッフルする．
- 入力値をバイト単位でランダムに選んで消去する．
- 入力値をバイト単位でランダムに選んでランダムな場所に挿入する．
- 入力値をバイト単位でランダムに選んで同じ内容を数回複製して，ランダムに挿入する．
- 入力値をバイト単位でランダムに選んで 1 ビット反転する．
- 入力値をバイト単位でランダムに選んで 1 バイト反転する．
- 入力値をバイト単位でランダムに選んで内容をコピーしてランダムな場所に挿入する．
- ユーザーが提供した辞書からデータを選んで入力値に加える．
- 自動で作成された辞書からデータを選んで入力値に加える．
- 入力値から ASCII 整数を見つけてそれを別の ASCII 整数に置き換える．
- Libfuzzer 側で用意した 1，2，4，8 バイト整数のいずれかをランダムに選んで入力値に挿入する．
- コーパスとして用いられる値をいくつかランダムに選んで入力値に挿入する．

Libfuzzer は並行処理を行うことができる．通常の状態で使用可能な CPU コアの半分を使用して並行処理を行う．例えば，CPU コアが 12 個使用可能であるならば，6 コアを使用して，並行処理を行う．

また，libfuzzer は，履歴機能をもっている．履歴機能というのは，fuzzing を途中でやめても記録している内容を使用して途中から再実行できる機能である．

3.1.2 KLEE

KLEE[5]とは，Cristian Cadar 氏らによって開発された conclic executor である．KLEE は，KLEE は低レイヤレベル（C 言語レベル）のプログラミングを行っているため，エラー出力はどんなレベルでも拾える．KLEE は，LLVM を使用している．また，KLEE はあらゆる CPU に対応するために，LLVM Intermediate Representation（LLVM IR）といわれる LLVM 専用のアセンブリ言語を使っている．この LLVM IR を使って，レジスタ，スタック，ヒープを木構造で states として保持している．

KLEE の solver は，STP，Z3，metaSMT である．以下に KLEE の基本的な動きを示す[10,11,12].

1. KLEE は条件分岐の時，solver に問い合わせて，実行可能である経路が真または偽の条件に基づいて実行するコードの命令ポインタを変更する．
2. もし両方の条件分岐が実行可能であった場合は states をコピーして，真と偽の経路の実行するコードの命令ポインタと path constraint を適切に更新する．
3. KLEE 側で危険な操作と定めている命令が行われるときにエラーを引き起こすような入力は KLEE のほうで自動的にブランチを生成する．例えば，除算命令の場合は 0 で除算が行われるようなものを KLEE 側で条件を生成し，エラーをわざと検出させる．この時，エラーで終了はせずに，実行は継続し，path constraint

に 0 で除算しない条件を入れる．それ以外のエラーの場合はエラーとなるような concreate value を生成して states を終了させる．

4. load 命令, store 命令の時は同じように危険な操作として処理を行うが，少し特殊なことをする．まず，メモリアドレスのアクセス範囲が有効であるかチェックを行う．しかし，独自のメモリマッピングのメモリアクセス範囲のチェックを行っている．STP の場合は全ての load 命令，store 命令をカバーすることができなかったため, STP 配列でメモリにマッピングするような形式にしている．STP 配列というのは，配列レベルのメモリ空間マッピングのことである．STP は，メモリアドレスのアクセス範囲のチェックと STP 配列でメモリにマッピングをする以外の操作は基本無視をしている．
5. ポインタが N 個のオブジェクトを参照していれば，STP配列レベルのメモリ参照まで落とし込むためにN回 states を複製する．オブジェクト指向型の言語のような大きなポインタ参照の場合は参照回数が増える．1 回参照のポインタや配列の場合にのみ最適化される．

KLLE の利点は 2 つある．1 つ目は，KLEE の states はオブジェクトレベルで扱っているため, 省メモリである．2 つ目は，heap 構造を複数の states が共有することができる．そのため，クローンする時の時間がオブジェクトにかかわらず一定である．この 2 つの利点のおかげで，同時に探索できる states の数が増え，操作できるすべての states を動かすようなキャッシュの実装とヒューリスティックの実装が単純化できている．

KLEE は，ヒューリスティックスケジューリングとして主に 2 つを用意している．以下にその 2 つとその共通点を示す．

1. Random Path Selection
   プログラムの経路を 2 分木で記録し，すべての実行可能な states を維持する．States は優先的に選択される条件が 2 つある．1 つ目は，優先度の高い states(favors states)を実行することである．favors states とは，記号値の制約が少なく，カバーされていないコードに到達する場合がある states である．2 つ目は，ループによって states 爆発的問題がある states ではないことである．つまり, States 爆発問題がある states は優先度が低くなることである．
2. Coverage Optimized Search
   新しいコード範囲をカバーするような states を優先的に選択する．優先的に選択される states の重みづけは様々用意されている．例えば，カバーされていない命令までの最小距離，states のコールスタック，および states が最近新しいコードをカバーしたかどうかを考慮する．
3. 共通点
   2 つの戦略に共通していることが 2 つある．1 つは，高いカバー範囲を実現するためには時間がかかるが，邪魔になるような states を排除できるようにするために各戦略はラウンドロビン方式をとっている．2 つ目は，時間がかかるような命令を持つ states が実行時間を支配しないようにタイムスライス方式を採用している．

KLEE は path constraint の最適化を行っている．path constraint の最適化とは, solver に path constraint を投げる前に symbolic execution engine で最適化させ、solver が解く時間を短縮させることである．以下に path constraint の最適化の手法を示す．

1. path constraint の書き換え（Expression Rewriting）
   最も基本的な path constraint の最適化は, 簡単な算術最適化（$x + 0 = x$），強度低下（$x * 2n = x << n$），線形最適化（$2 * x - x = x$）といったコンパイラが使う手法を反映させている．
2. path constraint の単純化（Constraint Set Simplification）
   プログラムの自然な構造で考えると同じ変数の場合，制約が特定的な傾向になりやすい．例えば，$x < 10$という path constraint があったとき，$x = 5$が追加されると前の path constraint である$x < 10$が, $x = 5$で真になるという最適化された path constraint になる．
3. path constraint の明示化（Implied Value Concretization）
   $x + 1 = 10$という path constraint が経路に加えられた場合，$x = 9$という定数化にし，最適化する．
4. path constraint の独立性（Constraint Independence）
   path constraint は参照するメモリは別であるそのため，とある path constraint のセット（制約集合）があった場合 1 つ 1 つのグループに分割する（制約部分集合）ことも可能である．つまり余計な制約をチェックしなくて済むことができる．例えば，制約集合$\{i < j, j < 20, k > 0\}$があった場合，まず，$i = 20$という問い合わせを solver 投げれば，余計な問い合わせを solver に投げなくて済む．
5. 反例制約のキャッシュ（Counter example Cache）
   path constraint が反例を持たない場合に使用する特別な flag と path constraint の反例が反例制約のキャッシュというところに保存されている．path constraint の反例とは，path constraint の真理表を全て逆にしたものである．これらを使用して path constraint を解く．これにより，solver に対する問い合わせの path constraint の部分集合と path constraint の上位集合を効率よく検索でき，solver に対しての冗長な問い合わせを削除できる．

Vol.2018-CSEC-82 No.31
Vol.2018-SPT-29 No.31
2018/7/25

path constraint の独立性と反例制約式のキャッシュは KLEE が用意しているオプションである. 文献[5]によると，2 つともオプションとして使用した場合，2 つともオプションを使用しなかった場合，1 つずつオプション使用した場合の 4 つと比べて，2 つともオプションを使用した場合が solver の解く時間が減り，実行速度が良くなることが分かっている[5].

3.2 KleeFuzzer の構成

KleeFuzzer のコンパイルから実行するまでの動作を以下に示す.

1. コンパイル
   python で klf-clang というコンパイル用スクリプトを用意して KLEE 側のオプションではklee =という形で提供を行う．また，libfuzzer 側のオプションではfuzzer =という形で提供する. clang のオプションはすべて提供する.
2. 実行
   python で kleefuzzer という実行用スクリプトを用意して，KLEE 側のオプションでは klee=という形で提供を行う．また，libfuzzer 側のオプションでは fuzzer=という形で提供する．KLEE と libfuzzer のオプションはすべて提供する．また，KLEE 側のディレクトリ内に生成されたシードを libfuzzer に与えて実行するオプションを提供する.

3.3 KleeFuzzer の課題

KleeFuzzer では libfuzzer と KLEE で連携をするにあたって，2 つ課題がある．以下にその課題を示す.

1. Libfuzzer が KLEE のライブラリを一部認識しない.
   KLEE が libfuzzer の未処理の外部ライブラリ、組み込み関数をサポートしていないためである．解決策は，2 つある．1 つ目は libfuzzer に対して KLEE のライブラリを認識するようなサポートを追加する。2 つ目は，KLEE に対して libfuzzer を認識するサポートを追加することである.
2. KLEE は main 関数がないと実行できない.
   KLEE は実行ファイル形式となっているソースコードを対象としているため，main 関数がないと図 4 で示すようなエラーを出力する.

```
kit@ubuntu:~/work/KleeFuzzer$ klee -debug-z3-dump
els -use-construct-hash-z3 -use-cex-cache -debug-
s test_fuzzer.bc
KLEE: ERROR: 'main' function not found in module.
```

図 4　KLEE の main 関数がないことによるエラー

図 4 で示したように，KLEE を実行させるには実行ファイル形式である main 関数があるソースコードが必要である．しかし，libfuzzer の実行ファイル形式ではない単一のライブラリで fuzzing を行うことができる利点がなくなり，既存の連携として使われている AFL と同じになってしまう問題がある.

そこで，KLEE に部分的に実行させるような仕組みを作る必要がある．つまり，KLEE を libfuzzer のようにライブラリ化させればよい．しかし，そのためには KLEE のコアとなる部分のソースコードを理解していかなければならない.

また，そのほかの解決策として，別の DSE と連携することである．例えば，angr，Triton といった symbolic execution engine を使って連携させる方法がある[8,13].

4. おわりに

新たな fuzzer と symbolic execution engine の組み合わせの KleeFuzzer を提案した．しかし，fuzzer と symbolic execution engine 連携を行うにはそれぞれのソフトウェアの親和性を考えなければならない.

また，fuzzer と symbolic execution engine のそれぞれを最適化することも考えなければならない．これをすることによってカバー範囲の拡大，経路を奥深くまで探索を行うことができれば，連携を行ったときに今までより良い結果になる．今後の課題として，KLEE のライブラリ化とそのほかの symbolic execution engine の連携を検討していきたい.

参考文献

[1] “Driller: augmenting AFL with symbolic execution!”. https://github.com/shellphish/driller(参照 2018-06-06)
[2] “Seeding fuzzers with symbolic execution”. https://github.com/julieeen/kleefl(参照 2018-06-06)
[3] M. Zalewski,” american fuzzy lop (2.52b)”. http://lcamtuf.coredump.cx/afl/,(参照 2018-06-06)
[4] “LibFuzzer – a library for coverage-guided fuzz testing.”.http://releases.llvm.org/docs/LibFuzzer.html(参照 2018-06-06)
[5] Cristian Cadar, Daniel Dunbar, Dawson Engler .KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)
[6] “Black box, grey box, white box testing: what differences? ”. https://www.nbs-system.com/en/blog/black-box-grey-box-white-box-testing-what-differences/(参照 2018-06-06)
[7] “はじめてのコンクリックテスト”. http://jasst.jp/symposium/jasst15tokai/pdf/S4-1.pdf(参照 2018-06-07)
[8] “angr”. https://github.com/angr.(参照 2018-06-15)
[9] “The LLVM Compiler Infrastructure”.https://llvm.org/(参照 2018-06-07)
[10]“Simple Theorem Prover, an efficient SMT solver for bitvectors”.

https://stp.github.io/(参照 2018-06-11)
[11] “The Z3 Theorem Prover”. https://github.com/z3prover/z3.(参照 2018-06-11)
[12] “meta SMT”. https://github.com/agra-uni-bremen/metaSMT(参照 2018-06-11)
[13] “Triton - A DBA Framework”. https://triton.quarkslab.com/(参照 2018-06-15)