



VERIFYING SYSTEMS RULES USING RULE-DIRECTED SYMBOLIC EXECUTION

HEMING CUI ET AL.
ASPLOS'13

0. ABSTRACTION

- 背景とモチベ:
 - ルール検証のための既存アプローチ
 - Static Analysis : 誤検出が多い
 - Symbolic Execution: 見逃しが起きやすい、スケールしない
 - 直感とアイデア:
 - 既存のSymbolic Executionは冗長な検査が多いからスケールしない
 - 冗長: ルール検証とは無関係なパス探索など
 - → 提案手法: **WOODPECKER** (キツツキ)
 - 特徴: ユーザ指定のイベントにフォーカスしたパス探索の最適化アルゴリズム
 - 特徴: 任意のイベント、探索ヒューリスティクスに対応するインターフェース
- 評価:
 - 計136個のシステムプログラムを検査 (vs KLEE, 各検体1時間の検査)
 - 検証率 WP: 28.7%, KLEE: 8.5%
 - 113個のルール違反 (内10個は重大なデータ消失エラー) を検出

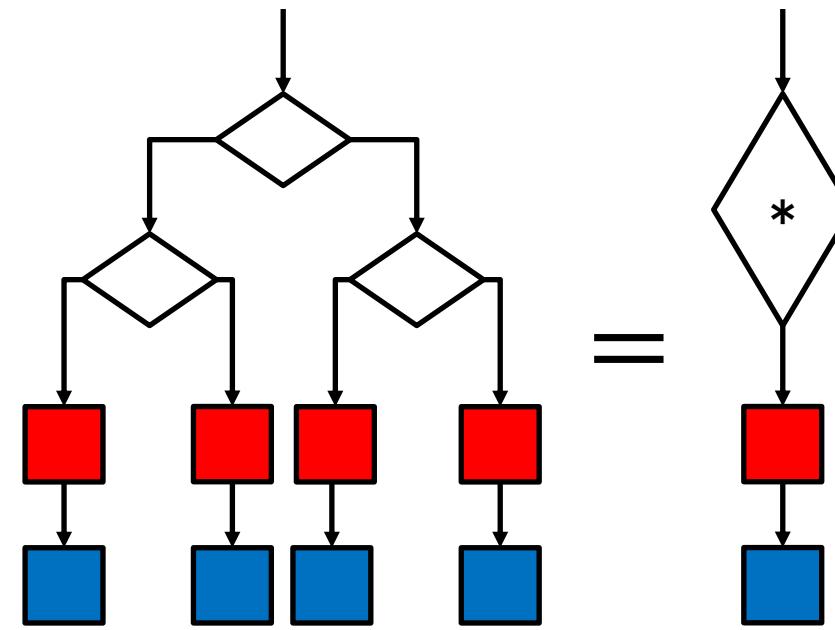
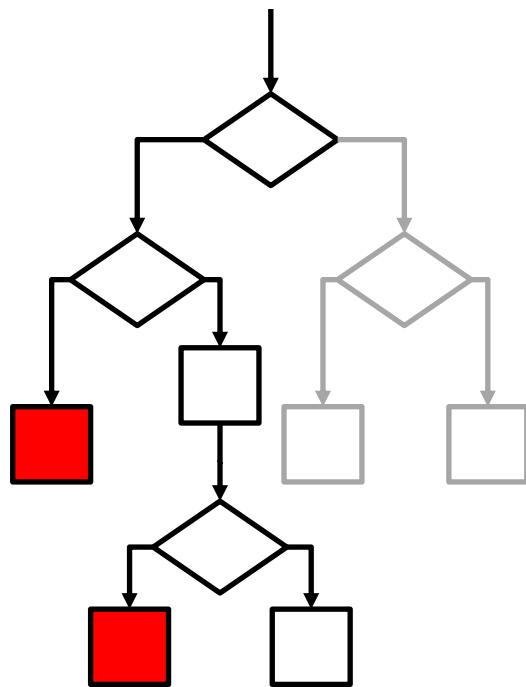
1. INTRODUCTION

- ルール: コードが従うべき規則
 - assert, alloc-free, open-close, open-r/w など
 - ルール違反は重大なバグにつながる可能性
 - ルール検証: プログラム内に特定のルール違反の有無を確認すること
- ルール検証のアプローチ①: Static Analysis
 - 利点: 効率が高い
 - 欠点: 実行時の動作を想定しきれない → 誤検出となる可能性
- ルール検証のアプローチ②: Symbolic Execution
 - 利点: 実際に実行すること(パス検証)で誤検出を訂正可能
 - 欠点: パス爆発によって全探索が困難
 - → 魅力的な技術ではある

1. INTRODUCTION

- 著者の直感:

- ルールに関わるパスはパス全体の一部である
 - 小泉: 他のヒューリスティクスと発想は同じ
- 検証の視点: 同一のイベントシーケンスに至るパスは等価
 - 小泉: マージに近いが手法が異なる



1. INTRODUCTION

- 本論文の目標
 - ①既存のSymbolic Executionの利点を踏襲している
 - 誤検出の抑制、パスの再現性など
 - ②ユーザ定義の任意のルールを検出可能である
 - ③任意のパス探索戦略に対応できる
- →これを達成する提案手法: WOODPECKER (以下, WP)

- WPの課題:
 - 与えられたルールから、どのようにパスの冗長性を判断するか
 - 異なるルールに対して、どのように対応するか
 - 既存の探索戦略と如何にして組み合わせるか
- →例題(2章)を挟んで、4章以降で詳しく解説

2. EXAMPLE

- 例: catのコア部分 (右コード)

- 既存手法
- 1.8が真 → 1.21が真
の関係性を見極めるのが困難
- Static Analysis
 - 1.8が真 ∧ 1.21が偽 or
1.8が偽 ∧ 1.21が真
を報告して誤検出する可能性
- Symbolic Execution (KLEE[12])
 - 1.14 ~ 1.20のループ
→パス爆発の加速
→冗長な検査が増える
- WP:
イベントシーケンス 1.8, 9, 21, 22, 23 を識別しつつ (vs Static)
ルールと無関係なパス探索を間引く (vs KLEE)

```

1 : int main(int argc, char **argv) {
2 :   FILE *input_desc;
3 :   int argind = 1;
4 :   const char *infile = "-";
5 :   do { // iterate over input files and print one by one
6 :     if(argind < argc)
7 :       infile = argv[argind];
8 :     if(strcmp(infile, "-") == 0) // input is a file
9 :       input_desc = fopen(infile, "r");
10:    else // input is stdin
11:      input_desc = stdin;
12:    if(!input_desc) continue;
13:    int c;
14:    while((c = fgetc(input_desc)) != EOF) {
15:      if(c < 32 && c != '\n') { // non-printable char
16:        putchar('^');
17:        putchar(c + 64);
18:      } else // printable char
19:        putchar(c);
20:    }
21:    if(infile[0] != '-' || infile[1] != 0)
22:      fclose(input_desc); // input is a file
23:  } while (++argind < argc);
24:  return 0;
25: }
```

2. EXAMPLE

- WP:

イベントシーケンス 1.8, 9, 21, 22, 23 を識別しつつ、冗長なパス探索を間引く

- 例: スライスを求めて、

冗長なパス候補を削ろう

- 仮定: 右図の順でパス実行が終了

- イベント: open-close

- スライスに含める条件:

- イベント or スライスに導く条件 or
スライスに関係するデータフロー

- 手順: パスを逆順に見ていく

```

3:     argind = 1;
4:     infile = "-";
6: true  argind < argc
7:     infile = argv[argind];
8: true  strcmp(infile, "-")
9:     input_desc = fopen(infile);
12:    if(input_desc) continue;
14: true (c=fgets(input_desc)) != EOF
15: false c<32 && c != '\n'
19:    putchar(c);
14: false (c=fgets(input_desc)) != EOF
21: true infile[0]!='-' || infile[1] != 0
22:    fclose(input_desc);
23: false ++argind < argc
24:    return 0;

```

2. EXAMPLE

- WP:

イベントシーケンス 1.8, 9, 21, 22, 23 を識別しつつ、冗長なパス探索を間引く

- 例: スライスを求めて、

冗長なパス候補を削ろう

- 仮定: 右図の順でパス実行が終了

- イベント: open-close

- スライスに含める条件:

- イベント or スライスに導く条件 or
スライスに関係するデータフロー

- 手順: パスを逆順に見ていく

- 1.24:

イベント未登場なのでスルー

```

3:     argind = 1;
4:     infile = "-";
6: true  argind < argc
7:     infile = argv[argind];
8: true  strcmp(infile, "-")
9:     input_desc = fopen(infile);
12:    if(input_desc) continue;
14: true (c=fgets(input_desc)) != EOF
15: false c<32 && c != '\n'
19:     putchar(c);
14: false (c=fgets(input_desc)) != EOF
21: true infile[0]!='-' || infile[1] != 0
22:     fclose(input_desc);
23: false ++argind < argc
24:     return 0;

```



2. EXAMPLE

- WP:
イベントシーケンス 1.8, 9, 21, 22, 23を識別しつつ、冗長なパス探索を間引く
- 例: スライスを求めて、冗長なパス候補を削ろう
 - 仮定: 右図の順でパス実行が終了
 - イベント: open-close
 - スライスに含める条件:
 - イベント or スライスに導く条件 or スライスに関係するデータフロー
- 手順: パスを逆順に見ていく
 - 1.23:
イベントに導く条件になりえる
→スライスに含める(後述)



```

3:     argind = 1;
4:     infile = "-";
6: true  argind < argc
7:     infile = argv[argind];
8: true  strcmp(infile, "-")
9:     input_desc = fopen(infile);
12:    if(input_desc) continue;
14: true (c=fgets(input_desc)) != EOF
15: false c<32 && c != '\n'
19:    putchar(c);
14: false (c=fgets(input_desc)) != EOF
21: true infile[0]!='-' || infile[1] != 0
22:    fclose(input_desc);
23: false ++argind < argc
24:    return 0;

```

2. EXAMPLE

- WP:

イベントシーケンス 1.8, 9, 21, 22, 23 を識別しつつ、冗長なパス探索を間引く

- 例: スライスを求めて、

冗長なパス候補を削ろう

- 仮定: 右図の順でパス実行が終了

- イベント: open-close

- スライスに含める条件:

- イベント or スライスに導く条件 or
スライスに関係するデータフロー

- 手順: パスを逆順に見ていく

- 1.22:

イベントなのでスライスに追加

```

3:     argind = 1;
4:     infile = "-";
6: true  argind < argc
7:     infile = argv[argind];
8: true  strcmp(infile, "-")
9:     input_desc = fopen(infile);
12:    if(input_desc) continue;
14: true (c=fgets(input_desc)) != EOF
15: false c<32 && c != '\n'
19:     putchar(c);
14: false (c=fgets(input_desc)) != EOF
21: true infile[0]!='-' || infile[1] != 0
22:     fclose(input_desc);
23: false ++argind < argc
24:     return 0;

```



2. EXAMPLE

- WP:

イベントシーケンス 1.8, 9, 21, 22, 23 を識別しつつ、冗長なパス探索を間引く

- 例: スライスを求めて、

冗長なパス候補を削ろう

- 仮定: 右図の順でパス実行が終了

- イベント: open-close

- スライスに含める条件:

- イベント or スライスに導く条件 or
スライスに関係するデータフロー

- 手順: パスを逆順に見ていく

- 1.21:

1.22へ導くための条件

スライスに追加

```

3:     argind = 1;
4:     infile = "-";
6: true  argind < argc
7:     infile = argv[argind];
8: true  strcmp(infile, "-")
9:     input_desc = fopen(infile);
12:    if(input_desc) continue;
14: true (c=fgets(input_desc)) != EOF
15: false c<32 && c != '\n'
19:     putchar(c);
14: false (c=fgets(input_desc)) != EOF
21: true infile[0]!='-' || infile[1] != 0
22:     fclose(input_desc);
23: false ++argind < argc
24:     return 0;

```



2. EXAMPLE

- WP:

イベントシーケンス 1.8, 9, 21, 22, 23 を識別しつつ、冗長なパス探索を間引く

- 例: スライスを求めて、

冗長なパス候補を削ろう

- 仮定: 右図の順でパス実行が終了

- イベント: open-close

- スライスに含める条件:

- イベント or スライスに導く条件 or
スライスに関係するデータフロー

- 手順: パスを逆順に見ていく

- 1.14-20

1.21の条件に影響しないので
スライスに含めない

```

3:     argind = 1;
4:     infile = "-";
6: true  argind < argc
7:     infile = argv[argind];
8: true  strcmp(infile, "-")
9:     input_desc = fopen(infile);
12:    if(input_desc) continue;
14: true (c=fgets(input_desc)) != EOF
15: false c<32 && c != '\n'
19:    putchar(c);
14: false (c=fgets(input_desc)) != EOF
21: true infile[0]!='-' || infile[1] != 0
22:    fclose(input_desc);
23: false ++argind < argc
24:    return 0;

```

2. EXAMPLE

- WP:
イベントシーケンス 1.8, 9, 21, 22, 23 を識別しつつ、冗長なパス探索を間引く
- 例: スライスを求めて、冗長なパス候補を削ろう
 - 仮定: 右図の順でパス実行が終了
 - イベント: open-close
 - スライスに含める条件:
 - イベント or スライスに導く条件 or
スライスに関係するデータフロー
- 手順: パスを逆順に見ていく
 - 1.12
ここはfalseでなければ
1.21に到達しないので
スライスに含めるがfalse固定

```

3:     argind = 1;
4:     infile = "-";
6: true  argind < argc
7:     infile = argv[argind];
8: true  strcmp(infile, "-")
9:     input_desc = fopen(infile);
12: false if(input_desc) continue;
14: true (c=fgets(input_desc)) != EOF
15: false c<32 && c != '\n'
19:     putchar(c);
14: false (c=fgets(input_desc)) != EOF
21: true infile[0]!='-' || infile[1] != 0
22:     fclose(input_desc);
23: false ++argind < argc
24:     return 0;

```

2. EXAMPLE

- WP:

イベントシーケンス 1.8, 9, 21, 22, 23 を識別しつつ、冗長なパス探索を間引く

- 例: スライスを求めて、

冗長なパス候補を削ろう

- 仮定: 右図の順でパス実行が終了

- イベント: open-close

- スライスに含める条件:

- イベント or スライスに導く条件 or
スライスに関係するデータフロー



- 手順: パスを逆順に見ていく

- 1.9

イベントなのでスライスに追加

```

3:     argind = 1;
4:     infile = "-";
6: true  argind < argc
7:     infile = argv[argind];
8: true  strcmp(infile, "-")
9:     input_desc = fopen(infile);
12: false if(input_desc) continue;
14: true  (c=fgets(input_desc)) != EOF
15: false c<32 && c != '\n'
19:     putchar(c);
14: false (c=fgets(input_desc)) != EOF
21: true  infile[0]!='-' || infile[1] != 0
22:     fclose(input_desc);
23: false ++argind < argc
24:     return 0;

```

2. EXAMPLE

- WP:
イベントシーケンス 1.8, 9, 21, 22, 23を識別しつつ、冗長なパス探索を間引く
- 例: スライスを求めて、冗長なパス候補を削ろう
 - 仮定: 右図の順でパス実行が終了
 - イベント: open-close
 - スライスに含める条件:
 - イベント or スライスに導く条件 or
スライスに関係するデータフロー
- 手順: パスを逆順に見ていく
 - 1.8
 - 1.9へ導くための条件なので
スライスに追加
同時に1.8 → 1.21なので1.21を
trueに固定



```

3:     argind = 1;
4:     infile = "-";
6: true  argind < argc
7:     infile = argv[argind];
8: true  strcmp(infile, "-")
9:     input_desc = fopen(infile);
12: false if(input_desc) continue;
14: true (c=fgets(input_desc)) != EOF
15: false c<32 && c != '\n'
19:     putchar(c);
14: false (c=fgets(input_desc)) != EOF
21: true infile[0]!='-' || infile[1] != 0
22:     fclose(input_desc);
23: false ++argind < argc
24:     return 0;

```

2. EXAMPLE

- WP:

イベントシーケンス 1.8, 9, 21, 22, 23 を識別しつつ、冗長なパス探索を間引く

- 例: スライスを求めて、

冗長なパス候補を削ろう

- 仮定: 右図の順でパス実行が終了



- イベント: open-close

- スライスに含める条件:

- イベント or スライスに導く条件 or
スライスに関係するデータフロー

- 手順: パスを逆順に見ていく

- 1.7

スライスに影響する代入なので
スライスに追加

```

3:     argind = 1;
4:     infile = "-";
6: true  argind < argc
7:     infile = argv[argind];
8: true  strcmp(infile, "-")
9:     input_desc = fopen(infile);
12: false if(input_desc) continue;
14: true  (c=fgets(input_desc)) != EOF
15: false c<32 && c != '\n'
19:     putchar(c);
14: false (c=fgets(input_desc)) != EOF
21: true  infile[0]!='-' || infile[1] != 0
22:     fclose(input_desc);
23: false ++argind < argc
24:     return 0;

```

2. EXAMPLE

- WP:

イベントシーケンス 1.8, 9, 21, 22, 23 を識別しつつ、冗長なパス探索を間引く

- 例: スライスを求めて、

冗長なパス候補を削ろう

- 仮定: 右図の順でパス実行が終了

- イベント: open-close

- スライスに含める条件:

- イベント or スライスに導く条件 or
スライスに関係するデータフロー

- 手順: パスを逆順に見ていく

- 1.6

- 1.7 に導くための条件なので
スライスに追加



```

3:     argind = 1;
4:     infile = "-";
6: true  argind < argc
7:     infile = argv[argind];
8: true  strcmp(infile, "-")
9:     input_desc = fopen(infile);
12: false if(input_desc) continue;
14: true  (c=fgets(input_desc)) != EOF
15: false c<32 && c != '\n'
19:     putchar(c);
14: false (c=fgets(input_desc)) != EOF
21: true  infile[0]!='-' || infile[1] != 0
22:     fclose(input_desc);
23: false ++argind < argc
24:     return 0;

```

2. EXAMPLE

- WP:

イベントシーケンス 1.8, 9, 21, 22, 23 を識別しつつ、冗長なパス探索を間引く

- 例: スライスを求めて、

冗長なパス候補を削ろう

- 仮定: 右図の順でパス実行が終了

- イベント: open-close

- スライスに含める条件:

- イベント or スライスに導く条件 or
スライスに関係するデータフロー

- 手順: パスを逆順に見ていく

- 1.3, 4

1.6などと関係する代入なので
スライスに追加



```

3:     argind = 1;
4:     infile = "-";
6: true  argind < argc
7:     infile = argv[argind];
8: true  strcmp(infile, "-")
9:     input_desc = fopen(infile);
12: false if(input_desc) continue;
14: true  (c=fgets(input_desc)) != EOF
15: false c<32 && c != '\n'
19:     putchar(c);
14: false (c=fgets(input_desc)) != EOF
21: true  infile[0]!='-' || infile[1] != 0
22:     fclose(input_desc);
23: false ++argind < argc
24:     return 0;

```

2. EXAMPLE

- WP:

イベントシーケンス 1.8, 9, 21, 22, 23 を識別しつつ、冗長なパス探索を間引く

- 例: スライスを求めて、

冗長なパス候補を削ろう

- 仮定: 右図の順でパス実行が終了

- イベント: open-close

- スライスに含める条件:

- イベント or スライスに導く条件 or
スライスに関係するデータフロー

- スライスの完成

- このパスから得られる探索候補
→ 1.6, 1.8, 1.23

- 1.14, 1.15などは探索候補から除外

```

3:     argind = 1;
4:     infile = "-";
6: true  argind < argc
7:     infile = argv[argind];
8: true  strcmp(infile, "-")
9:     input_desc = fopen(infile);
12: false if(input_desc) continue;
14: true  (c=fgets(input_desc)) != EOF
15: false c<32 && c != '\n'
19:     putchar(c);
14: false (c=fgets(input_desc)) != EOF
21: true  infile[0]!='-' || infile[1] != 0
22:     fclose(input_desc);
23: false ++argind < argc
24:     return 0;

```

3. WOODPECKER OVERVIEW

- WPのアーキテクチャ概要

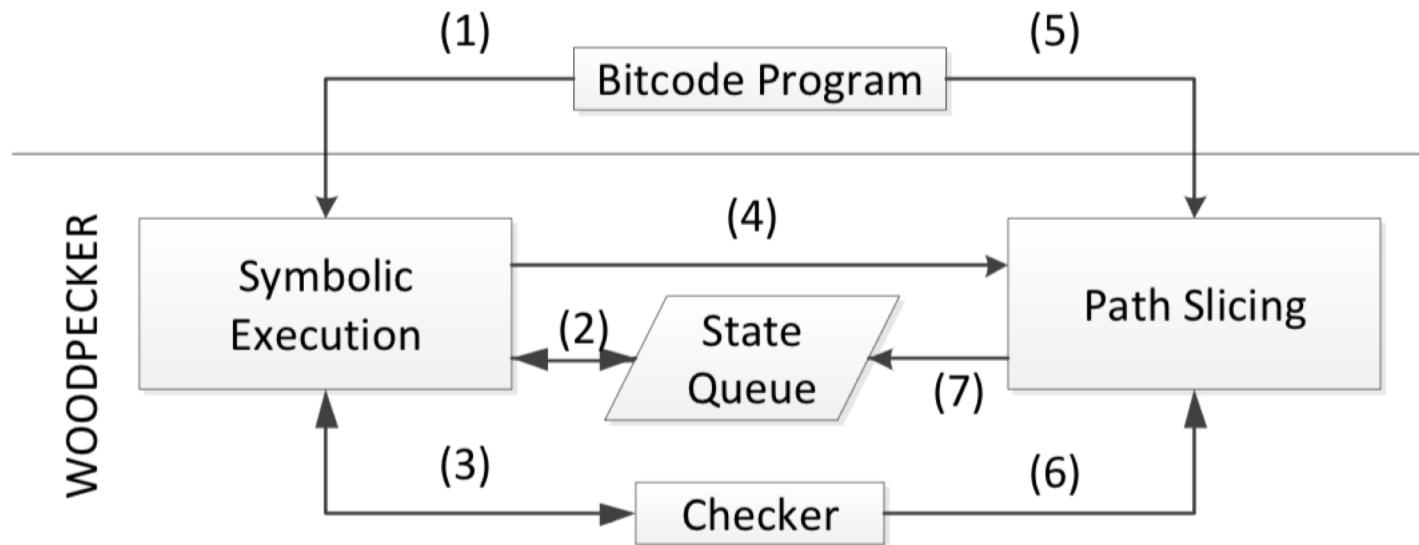


Figure 3: WOODPECKER *architecture*.

4. CHECKER INTERFACE

- WPの目標②: 任意の規則について対応できる
 - 規則を記述できるインターフェースを用意
→ ユーザはチェックカーメソッドを記述することで任意の規則を表現
→ WPはメソッドの返り値から規則を判別

```
class Checker {  
    Checker *Clone();  
    bool OnExecution(  
        const ExecutionState *s,  
        const KInstruction *ki,  
        Bitmask& evmask );  
    bool MayBeEvent(  
        const Instruction *i );  
}
```

4. CHECKER INTERFACE

- WPの目標②: 任意の規則について対応できる

```
class Checker {
    Checker *Clone();
    bool OnExecution(
        const ExecutionState *s,
        const KInstruction *ki,
        Bitmask& evmask );
    bool MayBeEvent(
        const Instruction *i );
}
```

- **Clone()**: チェッカの複製 = 実行エンジンの状態と対応させる
- **OnExecution()**: 返り値: ルール違反が起きた場合にfalse
 - 副作用①: 状態sの更新 (イベント情報の付与)
 - 副作用②: 実行した命令(ki)がイベントか否かをマスキング(evmask)
- **MayBeEvent()**:
 - **返り値:** パス内分岐から到達可能な
静的なパス外命令(i)がイベントになりえるか
 - 2. Exampleで1.23がスライスに含まれた要因

5. SEARCH ALGORITHM

- WPの目標③: 任意のパス探索戦略に対応できる
 - 任意のパス探索戦略と組み合わせられる冗長性除去アルゴリズム
 - →パス探索戦略をキュー(state queue q)で表現
 - q : 次の探索候補のパスの集合
 - $q.add(<s, i>)$: 発見された新規状態を q へ追加
 - $<s, i>$: エンジンの状態 s と、対応する命令 i のペア
 - $q.remove()$: q の内から、次に探索するパスを抽出
 - $q.remove()$ で任意のパス探索戦略を表現
 - 例: DFSなら末尾の状態をremove
 - 冗長性除去アルゴリズム
 - →渡されたチェッカーからイベントのスライスを計算
 - →スライスに入っていない状態 $<s, i>$ をキュー q から除去
≒WPのパス探索戦略との積集合にする（ちょっと違う）

[1] Howard: a Dynamic Excavator for Reverse Engineering Data Structures [NDSS'11]

[2] DDE: Dynamic data structure excavation [APSys'10]

6. CHECKER

- WPは5つの組み込みチェックを有する
 - assertion checker
 - 規則: assert_failed()を踏まない
 - memory leak checker
 - 規則①: すべてのヒープオブジェクトが解放される（プログラム終了時）
 - 規則②: double-freeが発生しない
 - open-close checker
 - 規則①: openされたファイルがすべてcloseされる（プログラム終了時）
 - 規則②: closeされたファイルが再度closeされない
 - file-access checker
 - 規則①: r/wはopenされたファイルに対してのみ行われる
 - 規則②: エラーが生じているファイルに対してr/wが行われない
 - data loss checker
 - 規則①: ファイル名変更前にflush, syncが実行されている
 - 規則②: ファイル名変更前に変更後のリンクが解除されない
 - 規則③: FILEオブジェクトと関連付けられているfdをclose()で閉じない

6. CHECKER

- ユーザは独自のチェックを定義可能
 - in section 4 (Checker Interface)

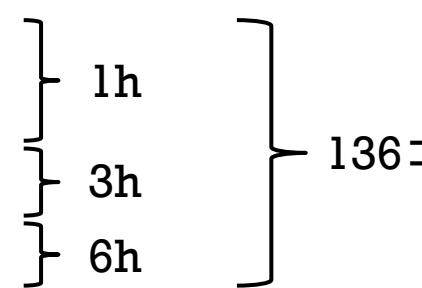
- ユーザのチェック定義を手助けするWPの機能
 - 間接呼び出しを解決するためのエイリアス解析
 - ハッシングによる同一エラー報告のフィルタリング
 - 記号値の具体化（興味ある入力のみに絞ることが可能）

7. IMPLEMENTATION

- LLVMとKLEE実行システム上にWPを実装
 - KLEEの修正 :2,968行
 - Path Slicing :5,771行
 - 組込チェック:1,589行
- 約10,000行
- KLEEの修正内容
 - ファイルシステムのモデリング改善
 - mmapのモデリングの追加
- Path Slicingの実装内容
 - 関数サマリの作成
 - エイリアス解析 (**bdbbdb [58]**) のチューニング
 - 関数のクローン数に上限を設けるなど
 - 努力: キャッシング

[58] bdbbdb Project. <http://bdbbdb.sourceforge.net>

8. EVALUATION

- 検体
 - GNU CoreUtils-8.12[20] から96コ
 - shadow-4.1.5[54] から30コ
 - git-1.7.9.4[30] から7コ
 - tar-1.26[56], sed-4.2.1[31], CVS-1.11.23[24]
 - コンパイル: LLVM gcc-2.7, -O2/O3
 - 環境:
 - 物理メモリ: 64GB
 - OS: Linux-2.6.38
 - CPU: Intel Xeon X5660 (6コア)
 - 比較対象のKLEEとWPのKLEE部分の設定は同一
- 

[20] <http://www.gnu.org/software/coreutils>[30] <http://git-scm.com/>[31] <http://www.gnu.org/software/sed>[54] <http://pkg-shadow.alioth.debian.org/>[56] <http://www.gnu.org/software/tar/>[24] <http://www.cvshome.org>

8. EVALUATION

8.1 VERIFICATION RESULTS

- ルール検証効率の評価 (vs KLEE)

- 136コの検体×組込チェックの内、有効な387ペアを検査

Checkers	Programs Checked	Programs Verified		Relevant Paths Verified		Redundant Paths	
		WOODPECKER	KLEE	WOODPECKER	KLEE	WOODPECKER	KLEE
Assertion	57	13	3	195,268	45,763	69,795	195,178
Memory leak	103	32	7	1,024,676	176,475	451,836	1,657,721
Open-close	72	19	4	528,676	82,883	203,407	512,439
File access	120	40	12	1,694,393	377,181	496,651	2,141,111
Data loss	35	7	7	132,136	89,779	22,996	117,225
Total	387	111	33	3,575,149	772,081	1,244,685	4,623,674

- 結果:

- 全ペアの内、WPは28.7%(111ペア)で検証を完了
KLEEは8.5%(33ペア)
-
-
-
-
-
-
-

8. EVALUATION

8.1 VERIFICATION RESULTS

- ルール検証効率の評価 (vs KLEE)

- 136コの検体×組込チェックの内、有効な387ペアを検査

Checkers	Programs Checked	Programs Verified		Relevant Paths Verified		Redundant Paths	
		WOODPECKER	KLEE	WOODPECKER	KLEE	WOODPECKER	KLEE
Assertion	57	13	3	195,268	45,763	69,795	195,178
Memory leak	103	32	7	1,024,676	176,475	451,836	1,657,721
Open-close	72	19	4	528,676	82,883	203,407	512,439
File access	120	40	12	1,694,393	377,181	496,651	2,141,111
Data loss	35	7	7	132,136	89,779	22,996	117,225
Total	387	111	33	3,575,149	772,081	1,244,685	4,623,674

- 結果:

- 全ペアの内、WPは28.7%(111ペア)で検証を完了
KLEEは8.5%(33ペア)
- WPが未完了ペアの内、探索されたパス数の比較
 - ルールとの関連パス数: WPはKLEEの4.6倍関連パスを探索
 - …

x4.6

8. EVALUATION

8.1 VERIFICATION RESULTS

- ルール検証効率の評価 (vs KLEE)

- 136コの検体×組込チェックの内、有効な387ペアを検査

Checkers	Programs Checked	Programs Verified		Relevant Paths Verified		Redundant Paths	
		WOODPECKER	KLEE	WOODPECKER	KLEE	WOODPECKER	KLEE
Assertion	57	13	3	195,268	45,763	69,795	195,178
Memory leak	103	32	7	1,024,676	176,475	451,836	1,657,721
Open-close	72	19	4	528,676	82,883	203,407	512,439
File access	120	40	12	1,694,393	377,181	496,651	2,141,111
Data loss	35	7	7	132,136	89,779	22,996	117,225
Total	387	111	33	3,575,149	772,081	1,244,685	4,623,674



x3.7

- 結果:

- 全ペアの内、WPは28.7%(111ペア)で検証を完了
KLEEは8.5%(33ペア)
- WPが未完了ペアの内、探索されたパス数の比較
 - ルールとの関連パス数: WPはKLEEの4.6倍関連パスを探索
 - 冗長なパス数 : **KLEEはWPの3.7倍冗長なパスを探索**
→検査時間を延ばすとKLEEはより冗長なパスを探索

8. EVALUATION

8.1 VERIFICATION RESULTS

- ルール検証効率の評価 (vs KLEE)

- 136コの検体×組込チェックの内、有効な387ペアを検査

Checkers	Programs Checked	Programs Verified		Relevant Paths Verified		Redundant Paths	
		WOODPECKER	KLEE	WOODPECKER	KLEE	WOODPECKER	KLEE
Assertion	57	13	3	195,268	45,763	69,795	195,178
Memory leak	103	32	7	1,024,676	176,475	451,836	1,657,721
Open-close	72	19	4	528,676	82,883	203,407	512,439
File access	120	40	12	1,694,393	377,181	496,651	2,141,111
Data loss	35	7	7	132,136	89,779	22,996	117,225
Total	387	111	33	3,575,149	772,081	1,244,685	4,623,674

- 結果:

- 全ペアの内、WPは28.7%(111ペア)で検証を完了
KLEEは8.5%(33ペア)
- WPが未完了ペアの内、探索されたパス数の比較
 - ルールとの関連パス数: WPはKLEEの4.6倍関連パスを探索
 - 冗長なパス数 : KLEEはWPの3.7倍冗長なパスを探索
→検査時間を延ばすとKLEEはより冗長なパスを探索
- 定義: 探索効率 = $\frac{\# \text{ of relevant paths}}{\# \text{ of explored paths}}$ (ペア単位)
 - WP: 平均: 64.9% KLEE: 29.2%

8. EVALUATION

8.2 RULE VIOLATIONS DETECTED

- 検出されたルール違反の詳細

- → 「大半は例外処理→プログラム終了」による leak/open-close 違反
 - 小泉: つまりそんなに脅威ではない

Programs	mem leak	open-close	data loss
coreutils	40	13	0
shadow	11	5	1
tar	4	0	0
sed	3	0	0
CVS	3	1	2
git	19	4	7
Total	80	23	10

- data loss違反は重大なバグにつながる可能性大

- gitで7コ, CVSで2コ, shadowで1コ
- git, CVS:
 - 特定の処理(such as “git add”)内のrenameでfsyncが未使用
 - 特に”git add”的違反は5つの関数呼び出しをまたぐので静的解析では困難
- shadow:
 - /etc/default/useraddの更新でrename中にクラッシュするとリンクが消失

8. EVALUATION

8.3 OVERHEAD OF PRUNING

■ 分岐の間引のオーバーヘッドの調査

- Pruning : 間引 (state queueからの削除) そのもののオバヘ
- Alias : 間引くために必要なスライスの計算のオバヘ
 - 実質的にエイリアス解析の時間
- 解析全体を100%とした場合の割合

Programs	Pruning (%)	Alias (%)
coreutils	0.82	0.49
shadow	1.98	0.11
tar	4.59	5.87
sed	0.69	0.50
CVS	4.36	4.64
git	8.99	11.71

■ 結果:

- 単純なプログラムほどオーバーヘッドは減少
gitなどの複雑なプログラムではやや増加
 - → 総じてオバヘ自体は少なめ

CONCLUSION

- まとめ
- **提案手法: WOODPECKER**
 - ルール違反検証を主眼とした記号実行器
 - 任意のルールチェックを記述可能なインターフェースを実装
 - 任意のパス探索戦略をキューで表現可能
 - &スライスを利用してルールと無関係なパスを間引
- 実験結果
 - 既存手法(KLEE)と比較してより効率的にルール検証が可能
 - gitなどから重大なデータロス違反の検出に成功
 - 追加オーバーヘッドは比較的軽量

感想

- 特定のイベントに注目するというアイデアは良
 - ただし、イベントに初回に到達するまでの方法は書かれていない
= 誘導するには最低一回イベントに自力で到達する必要アリ
 - 多分スライスの静的チェックである程度カバーしている
 - 初期入力が重要になってくる？
- memory leakのチェックはあまり参考にはならず
 - プログラム終了時に解放しないケースなんてごまんとある