ファジングにおける根本原因解析のための字句化・局所化されたデータフローグラフの生成

青木 克憲[1,a)]　品川 高廣[1]

概要：ファジングは脆弱性検出のために広く使われている手法であり，入力をランダムに変えながら実行を繰り返すことにより，クラッシュを引き起こす入力を自動的に発見することができる．しかし，発見されたクラッシュが真に脆弱性によるものであるかどうかを識別するためには，人手による根本原因解析（Root Cause Analysis）が必要であり，非常に手間と時間がかかる作業が必要になる．従来の研究では，制御フローやデータフローを抽出して根本原因解析の支援情報を生成するものや，多数のクラッシュから根本原因の絞り込みを試みるものなどがあるが，支援情報が人間にとって分かりにくかったり，多数のクラッシュ入力の生成が必要であったりした．本研究では，人間にとって分かりやすい解析支援情報を１つのクラッシュ入力のみから生成するために，字句化及び局所化されたデータフローグラフを生成する手法を提案する．ソースコード計装により変数名や型名など字句レベルの情報を付加したデータフローを実行時に記録し、人間がデータフローとソースコードの関係を理解しやすいようにする．また、クラッシュの前後のシードからデータフローの差分をとることで、単一のクラッシュ情報のみから根本原因に関連した部分に局所化したデータフローグラフを生成する．ファジング評価用データセットである Magma に含まれる 2 つの既知の脆弱性に対して提案手法を適用した結果，データフロー上から根本原因に至ることが容易になったことを確認した．

キーワード：ファジング, 根本原因解析，データフロー局所化，ソースコード計装

Generating Lexicalized and Localized Data Flow Graphs for Root Cause Analysis in Fuzzing

Katsunori Aoki[1,a)]　Takahiro Shinagawa[1]

Abstract: Fuzzing is a widely used method for vulnerability detection that can automatically find inputs that cause crashes by repeating execution with randomly changing inputs. However, to identify whether a discovered crash is truly caused by a vulnerability or not, a manual Root Cause Analysis is required, which is a very time-consuming and labor-intensive process. Previous studies have attempted to generate support information for root cause analysis by extracting control flow and data flow, or to narrow down the root cause from a large number of crashes, but the support information was difficult for humans to understand or required the generation of a large number of crash inputs. In this study, we propose a method for generating lexicalized and localized data flow graphs to generate human-intuitive analysis support information from only one crash input. The data flow with lexical-level information such as variable names and type names is recorded at runtime by source code instrumentation, making it easier for humans to understand the relationship between the data flow and the source code. By differencing data flows from the seeds before and after a crash, we generate a data flow graph that is localized to the root-cause related parts from only a single crash information. We applied the proposed method to two known vulnerabilities in Magma, a fuzzing evaluation dataset, and observed that it is easier to get to the root cause from the data flow.

Keywords: Fuzzing, Root Cause Analysis, Data Flow Localization, Source Code Instrumentation

1. イントロダクション

ファジングはソフトウェアの脆弱性を自動的に検出する手法として広く使われている．ファジングでは、ランダムに変異させて大量に生成した入力を検査対象のソフトウェアに与え，検査対象がセキュリティ問題を起こすかどうかを観測する．ファジングの有効性は既に実証されており、例えばファジング基盤である ClusterFuzz は、2022 年 5 月現在で Google（Chrome など）において 25,000 以上のバグを発見しているほか，550 以上のオープンソースプロジェクトにおいて 36,000 以上のバグを発見している [4].

しかし、ファジングにおいてクラッシュを引き起こす入力を発見したとしても、それが真に脆弱性によるものであるかどうかを識別することは容易ではない．これは，クラッシュを引き起こした場所と，真にそれを引き起こす原因となった根本原因（Root Cause）の場所が離れている場合が多く，通常は解析者が人手による根本原因解析（Root Cause Analysis）によって場所と理由を特定する必要があり、非常に時間と手間がかかるためである．

従来の研究では，実行時に制御フローやデータフローを記録して，人手による根本原因解析を支援するための情報を生成する手法がある [8, 9, 11–13]．しかし，これらの手法はバイナリプログラムを対象としており，生成される情報は命令やメモリアドレス，レジスタなどの低レイヤ情報や，それ対応するファイル名や行番号 [8, 11]，基本ブロック [13] などに限定されており，解析者にとっても解釈が難しい．また，ファジングによって生成されるクラッシュ入力を活用して根本原因を絞り込む手法も提案されている [8, 13]．しかし，これらの手法では原因箇所を絞り込むために多数のクラッシュ入力が必要であり，必ずしもすべての場合において容易に適用できるとは限らない．

本論文では，解析者にとって分かりやすい根本原因解析支援情報をファジングで発見された一つのクラッシュだけから生成するために，字句化及び局所化されたデータフローグラフを生成する手法を提案する．解析者に有用な情報を提供するために，ソースコード計装（Source Code Instrumentation）により変数の名前や型，定数，関数といったソースコードの字句レベルの情報を付加したデータフローを実行時に記録する．これにより，クラッシュ時のデータフローを解析者が理解しやすい形でトレースできるようにする．また，データフローの中でクラッシュに関係した部分だけに絞り込むために，クラッシュ前後のシードからデータフローグラフの差分を取ることにより，クラッシュに局所化したデータフローグラフを生成する．これにより，解析者にとって不要な情報を排除して，根本原因にたどり着くことを容易にする．

[1] 東京大学 (The University of Tokyo)
[a)] aoki@os.ecc.u-tokyo.ac.jp

提案手法を AFL [1] と clang をベースに実装し，ファジング評価用のデータセットである Magma [10] に含まれる既知の脆弱性に対して適用して，その有効性を検証した．その結果, libpng 及び libtiff の 2 件の脆弱性に対して，データフローから根本原因にたどることが容易になったことを確認した．

本論文の貢献は以下の通りである．

- ファジングにおける根本原因解析を支援するために，ソースコード計装による字句化及びクラッシュ前後のシードツリーを用いた局所化により，解析者が分かりやすいデータフローグラフを生成する手法を提案した．
- 実際のオープンソースソフトウェア（OSS）の脆弱性 2 件を対象に本手法を適用したケーススタディにより，提案したデータフローグラフの有効性を確認した．

2. 全体像

図 1 に提案手法の全体像を示す．提案手法は 3 つのフェーズに分かれている．フェーズ 0 はファジング実施である．まず，ファジングを開始する前に，ソースコードをコンパイルすると同時に，ファジングに必要な機能をバイナリに導入（計装）する．この時，AddressSanitizer [7] が導入されていることを仮定する．AddressSanitizer はメモリエラー検出器で，バグ発生時にどこでどんなエラーが発生したのかを記録することができる．またファジング実行中に，変異で使用される入力はシードと呼ばれ，AFL や libfuzzer [5] では，シードがどのシードから生成されたのかや，クラッシュを引き起こしたシードがどのシードから生成されたのかが記録されている．以下では，これらの記録をシードツリーと呼ぶ．

フェーズ 1 はデータフロー追跡である．まず，(1-1) ソースコード計装において，ソースコードの字句レベルの情報を付加した（字句化された）データフローのトレースを記録する機能を対象のプログラムに追加する．このソースコード計装を施してコンパイルして得たバイナリを tracee と呼ぶ．次に，(1-2) データフロー追跡において，tracee に入力を与えてデータフローのトレースを取得する．この時，トレースを実施する入力は，非クラッシュ入力とクラッシュ入力の 2 つであり，非クラッシュ入力はシードツリー上でクラッシュ入力と親子関係にあるものから選択する．最後に，(1-3) データフローグラフ構築において，(1-2) で得たデータフローをもとにデータフローグラフを構築する．

フェーズ 2 はデータフローグラフ局所化である．まず，(2-1) クラッシュ関連変数候補抽出において，クラッシュの種類とクラッシュ箇所を確認し，クラッシュに関係する変数の候補を抽出する．次に，(2-2) クラッシュ関連データフローグラフ生成において，(1-3) で構築したデータフローグラフから，クラッシュした箇所から辿れる範囲のグラフだけを抽出する．この抽出したグラフをクラッシュ関

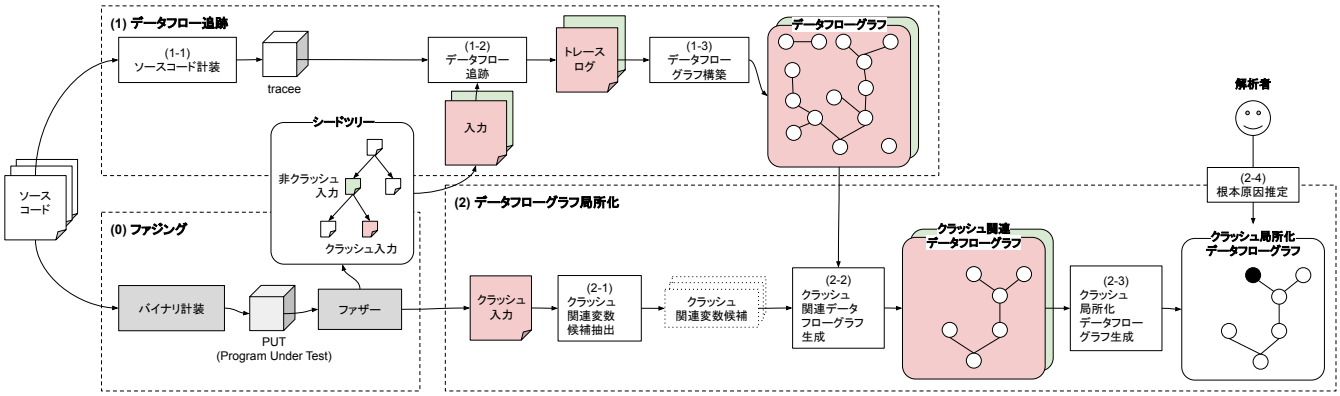

図 1: 提案手法の全体像

連データフローグラフと呼ぶ．最後に，(2-3) クラッシュ局所化データフローグラフ生成において，クラッシュ前後のクラッシュ関連データフローグラフと非クラッシュ入力から生成したクラッシュ関連データフローグラフとの差分グラフである**クラッシュ局所化データフローグラフ**を生成する．最後に，(2-4) 根本原因推定において，解析者がクラッシュ局所化データフローグラフを参照して，根本原因を推定する．

本手法は，クラッシュ前後でファジングの変異に由来するデータフローの小さな違いがあり，それが脆弱性の根本原因に気づくきっかけになる，という予想に基づき，その違いを分かりやすく可視化することを目指したものである．

2.1 既知脆弱性によるデモ

以下の説明では，実際の脆弱性 CVE-2013-6954 [] を例として扱うため，本節ではこの脆弱性の概要を説明する．CVE-2013-6954 は，libpng のヌルポインタ参照のバグである．図 2 に，脆弱性に関係あるコードの抜粋を示す．図 2(a) の 2 行目に定義されている `png_set_PLTE` 関数は，9 行目でヒープ確保したメモリを `png_ptr->palette` に割り当てる処理を含む．もし，関数の引数である `num_palette` が 0 であるなど，異常があるとヒープ確保せずに同関数からリターンする点を注目されたい．図 2(b) に，クラッシュ箇所である，`png_do_expand_palette` 関数の抜粋を示す．ヌルポインタ参照は，同関数の引数である `palette` を参照している 9 行目で発生する．この原因は，`palette` が前述の `png_set_PLTE` 関数において，ヒープメモリを確保せずにリターンした場合においても，変数 `palette` を無条件に参照しているためである．

```
1 void PNGAPI
2 png_set_PLTE(png_structrp png_ptr, png_inforp
      info_ptr, png_const_colorp palette, int
      num_palette) {
3   ... snipped ...
4   if ((num_palette > 0 && palette == NULL) || (
      num_palette == 0)) {
5     png_chunk_report(png_ptr, "Invalid palette",
      PNG_CHUNK_ERROR);
6     return; // メモリ確保せずにリターン
7   }
8   png_free_data(png_ptr,info_ptr,PNG_FREE_PLTE,0);
9   png_ptr->palette = png_voidcast(png_colorp,
      png_calloc(png_ptr, PNG_MAX_PALETTE_LENGTH * (
      sizeof (png_color)))); // メモリ確保
```

(a) `png_set_PLTE` 関数内

```
1 static void
2 png_do_expand_palette(png_row_infop row_info,
      png_bytep row, png_const_colorp palette,
      png_const_bytep trans_alpha, int num_trans) {
3   png_uint_32 row_width=row_info->width;
4   ... snipped ...
5   sp = row + (size_t)row_width - 1;
6   dp = row + (size_t)(row_width * 3) - 1;
7   i = 0;
8   for (; i < row_width; i++) {
9     *dp-- = palette[*sp].blue; // クラッシュ箇所
10    *dp-- = palette[*sp].green;
11    *dp-- = palette[*sp].red;
12    sp--;
13  }
```

(b) `png_do_expand_palette` 関数内

図 2: CVE-2013-6954 に関係あるコード抜粋

3. 提案手法

提案手法は，フェーズ 0 のファジング実施，フェーズ 1 のデータフロー追跡，フェーズ 2 のデータフロー局所化の 3 つのフェーズに分かれている．本章では，フェーズ 1 とフェーズ 2 の詳細を説明する．

3.1 フェーズ 1: データフロー追跡

3.1.1 (1-1) ソースコード計装

まず，トレース機能を計装するためのソースコード書き換え方法について述べる．図 3 に，ソースコードで右辺値として読み込まれた変数 `x` に対する計装例を示す．当該変数は，clang の抽象構文木（AST）で `declRefExpr()` で表現される．計装の目的は，トレース対象の字句がいつ読み書きされたのかを記録することである．計装後の `x` がマク

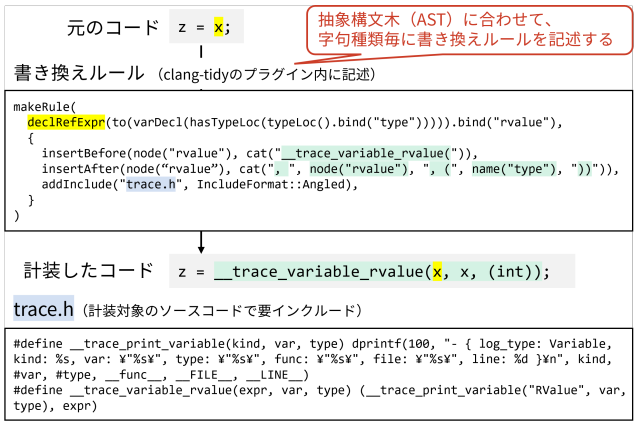

図 3: ソフトウェア計装の例

ロ__trace_variable_rvalue に囲まれた点に注目されたい．insertBefore と insertAfter の行が x の前後に当該マクロを挿入する．当該マクロの定義は同図の trace.h で示した通りで，オリジナルの処理を維持したまま，読み書きされた字句の名前，型，定義箇所の記録をおこなう．同じ要領でトレース対象の字句種類に合わせて定義を拡充すれば，字句情報を保持したトレースが実現できる．表 1 に計装するマクロの一覧を示す．次に，書き換えツール（詳細は 4 章参照）によりソースコードを書き換える．最後に，書き換え後のコードをコンパイルして tracee を得る．

3.1.2 (1-2) データフロー追跡

ファジングで得た入力で tracee を実行し、データフローのトレースを取得する．

入力は 2 節で述べた通り，シードツリーで親子関係にある非クラッシュ入力とクラッシュ入力を選択する．AFL のシードツリーは，入力ファイルの名前の id と src フィールドに記録されている．例えば，id:001239,src:001096+000171,op:splice,rep:16,+covと id:000066,sig:11,src:001239+000892,op:splice,rep:32の 2 入力だと，id:000066の入力は id:001239の入力から生成されたという親子関係が分かる．

3.1.3 (1-3) データフローグラフ構築

(1-2) データフロー追跡で得たトレースからデータフローグラフを構築する．このとき，グラフのノードにソースコードの字句を記載する．本論文のグラフでは，のノードラベルは「変数名：型 参照元の関数（ノード ID）」の形式で表示する．グラフの辺は図 4 に示す 5 種類がある．

データフローグラフ構築アルゴリズムを表 1 に示す．(1-1) ソースコード計装で計装したマクロは，トレースログを記録する（ログの形式は図 3 の trace.h の抜粋を参照）．データフローグラフ構築アルゴリズムは各ログを時系列順で受け取り，ログ種別に対応する同表のアルゴリズムを実行する．辺 equal を生やす際の等価性判定は field_equality を見ればわかるように，フィールドと型だけを根拠にした簡易的な判定を併用しており今後の課題である（困難な点と対策は 6 節で述べる）．

```
int g(int x) {
  return x;
}
```

名前	意味	例示コード 関数f()内	グラフでの表現 ノードのラベルは変数名：関数
bind	読んだ変数の値がどの変数に与えられたのかを表す	c = a + b;	a: f —bind→ c: f ←bind— b: f
func-call	関数呼出元と先で、関数引数を対応付ける	r = g(a);	a: f —func-call→ x: g
return	関数の戻り値で参照された変数と、戻り値を受け取る変数を対応付ける	r = g(a);	x: g —return→ r: f
member	構造体や配列で、参照したフィールド・添字とその親の変数を表す	s.a	s: f —member→ s.a: f
equal	読まれた変数が，どのノードと等しいのかを表す	b = a; c = b;	b: f —equal→ a: f

図 4: データフローグラフの辺の定義

```
1 ERROR: AddressSanitizer: SEGV on unknown address 0
      x00000000009b
2 The signal is caused by a READ memory access.
3   #0 in png_do_expand_palette pngrtran.c:4386:27
4   #1 in png_do_read_transformations pngrtran.c
      :4810:10
5   #2 in png_read_row pngread.c:578:7
6   #3 in LLVMFuzzerTestOneInput contrib/oss-fuzz/
      libpng_read_fuzzer.cc:209:7
```

図 5: CVE-2013-6954 再現時の AddressSanitizer 出力抜粋

3.2 フェーズ 2: データフロー局所化

3.2.1 (2-1) クラッシュ関連変数候補抽出

クラッシュ関連変数候補抽出では，まずクラッシュ箇所を把握する．クラッシュ入力を tracee で実行し，AddressSanitizer がメモリエラーを検出したときの出力から，クラッシュ箇所のファイル名と行番号を特定する．図 5 は，CVE-2013-6954 が再現したときの AddressSanitizer の出力の抜粋である．#0 の行を見れば，クラッシュ箇所は 2.1 節で述べた通り，palette を参照するコードとわかる．

次に，クラッシュに関わった変数の候補を把握する．クラッシュ箇所のコードに出現する変数を選べばよく，CVE-2013-6954 の例では，*sp, palette[*sp].blue, dp の 3 つが候補である．もし，対応した行に登場する変数で，トーレスに記録されていない変数はクラッシュ時に参照されていないので除外する．今回の例では dp が除外する．

3.2.2 (2-2) クラッシュ関連データフローグラフの生成

(1-3) で構築したデータフローグラフから，(2-1) で選択した変数に対応するノード（以下，起点ノード）を選び，その祖先のノードだけでサブグラフを構成してクラッシュ関連データフローグラフを生成する．ノードのラベルには，変数名と参照元の関数名が記されており，データフローグラフを検索して対応するノードを得られる．

3.2.3 (2-3) クラッシュ局所化データフローグラフ生成

まず，クラッシュ前後のクラッシュ関連データフローグラフの差分を取り，クラッシュ後に出現または消失したノードをクラッシュ関連データフローグラフ上で色付けして，クラッシュ局所化データフローグラフを作成する．図

（注）マクロでvarやrecordの名前・型・参照元の関数を得られるとする．var.field_nameは参照された構造体のフィールド名，var.typeは型を意味する．
contextはタプル(def, use)をアイテムとするLIFOキューである．context.get()はキューの末尾アイテムへの参照を渡す．context.up()はdefとuseが空であるタプル({}, {})をキューに追加する．context.down()は，キューの末尾を取り出して得たuseを，次の末尾のuseに追加する．
defとuseはノードの集合であり，context.get()で得たとする．
equalityはノードの等価性を記憶するためのUnion-Findである．euality.union(x, y)やy = equality.find(x)は，xとyが等しいことを意味する．
field_equalityは連想配列である．
paramsはuseをアイテムとするFIFOキューである．

トレース対象のASTノード	マクロ名と引数（マクロ引数は簡略）	計装例	ログ種別	データフローグラフ構築アルゴリズム（注）
右辺値の変数参照	__trace_variable_rvalue(var)	a = __trace_variable_rvalue(x);	RValue	(1) varのノードがdefにあれば，それをuseに追加する．無ければ．新規ノードとしてdefとuseに追加する． (2) (1)のノードをxとする．y = equality.find(x) がx と異なるならば，ノードx, yを辺equalで結ぶ．
左辺値の変数参照	__trace_variable_lvalue(var)	__trace_variable_lvalue(a) = x;	LValue	(1) varの新規ノードをdefに追加する． (2) useにある各ノードと(1)のノードそれぞれを辺bindで結ぶ． (3) useの配列長が1ならば，useにあるノードxと(1)のノードyを辺equalで結び，equality.unite(x,y) を呼ぶ．
変数宣言	__trace_variable_declaration(var)	int x; __trace_variable_declaration(x);	Declaration	（LValueと同じ）
右辺値の構造体または配列の参照	__trace_member_rvalue(var, record)	b = __trace_member_rvalue(y[0], y);	RMemberValue	(1) recordのノードがdefにあればそれを使う．無ければ，新規ノードとしてdefに追加する． (2) varのノードがdefにあれば，それをuseに追加する．無ければ，新規ノードとしてdefとuseに追加する． (3) (1)のノードと(2)のノードを辺memberで結ぶ． (4) (2)のノードをxとする．y = equality.find(x) がx と異なるならば，ノードx, yを辺equalで結ぶ．同じならば，field_equality[(x.field_name, x.type)] とxを辺equalで結ぶ（field_equalityにキーが存在しなければ中止）．
左辺値の構造体または配列の参照	__trace_member_lvalue(var, record)	__trace_member_lvalue(y[0], y) = 0;	LMemberValue	(1) recordのノードがdefにあればそれを使う．無ければ，新規ノードとしてdefに追加する． (2) varの新規ノードをdefに追加する． (3) (1)のノードと(2)のノードを辺memberで結ぶ． (4) useにある各ノードと(2)のノードそれぞれを辺bindで結ぶ． (5) useの配列長が1ならば，useにあるノードxと，(2)のノードyを辺equalで結び，equality.unite(x, y) を呼ぶ． (6) field_equality[(y.field_name, y.type)] にｙを入れる．
関数呼び出し元	__trace_function_call(expr)	__trace_function_call(f(x));	Call	context.up() を呼ぶ．
呼び出し元が与えた関数引数	__trace_function_call_param(expr)	f(__trace_function_call_param(x));	CallParam	(1) paramsの末尾にuseを加える． (2) useを空にする．
関数呼び出し先	__trace_function_call_enter()	f (int x) { __trace_function_call_enter(); { return x; } }	CallEnter	(1) paramsにuseを末尾に加える． (2) context.up() を呼ぶ．
関数引数の定義	__trace_function_param_decl(var)	f (int x) { __trace_function_param_decl(x, (int)); { return x; } }	ParamDecl	(1) varの新規ノードをdefに追加する． (2) paramsの先頭アイテムを取り出し，アイテムにある各ノードと(1)のノードそれぞれを辺bindで結ぶ．
関数の戻り値	__trace_function_return(expr)	f (int x) { return __trace_function_return(x); }	Return	useを空にする．
__trace_function_returnマクロ組み込み（exprの評価後，CallExitログをトレースに記録）			CallExit	context.down() を呼ぶ．
__trace_function_callマクロ組み込み（exprの評価後，CallEndログがトレースに記録）			CallEnd	context.down()を呼ぶ．

表 1: 計装するマクロの一覧とデータフローグラフの構築アルゴリズム

6 は CVE-2013-6954 におけるクラッシュ局所化データフローグラフである．赤色の点線の太枠が付いたノードは，クラッシュ入力のクラッシュ関連データフローグラフでは見られなかったノードを表す．

3.2.4 (2-4) 根本原因推定

最後に，解析者は根本原因の推定のため，クラッシュ局所化データフローグラフを読み取る．図 6 が示す差分は「`png_ptr->palette` へのメモリ確保がクラッシュ入力で欠落した」（同図右上のノード `ret :(png_voidp) <png_calloc> (1601)`に注目されたい）なので, Root Cause は「`png_ptr->palette` へのメモリ確保が欠落した」と推定する．なお，紙面の都合で左側の `pal_ptr->blue :(png_byte) <png_handle_PLTE> (1502)`の祖先をカットした．プログラムの挙動を考えれば，`palette[*sp].blue` で最初に参照されるのは `palette` なので，グラフの右側がより根本原因に近いと考えるのは自然である．

4. 実装

提案手法の有効性の評価のために，プロトタイプを実装した．本節では実装に関する重要な事項を説明する．

clang-tidy. ソースコードの書き換えは clang-tidy[3] をベースとし，書き換えルールはそのプラグインとして実装した．clang-tidy は登録されたルールに従ってソースコードを書き換えるツールで，処理は， ルールと照合し，適用箇所を決定する処理，と 決定内容をソースコードに反映する処理の 2 段階に分かれている．後者の処理は clang-tidy とは独立したツール clang-apply-replacements によって実行されるが，本提案手法で都合の悪い挙動があったため，我々が再実装した．本家との差異は，同じ位置に別々の書き換えが要求されたとき，本家は最初に照合したルールだけを適用するが，本研究では照合したすべてのルールを適用する．このとき，ルールの適用順（言い換えると，マクロの展開順序）は重要である．例えば，関数のリターン `return x;` を計装するとき，マクロの展開順序は，(1) リターンの開始，(2) リターンで参照された変数 `x` の読み込み，の順でなければならない．本実装では，書き換えに適用順を設定して解決した．

`compile_commands.json`. 本提案手法でソースコード計装する前に `compile_commands.json` が必要である．clang-tidy は，各ソースコードがどのようなコマンドでコンパイルされたのかを記録した当該ファイルを必要とするからだ．当該ファイルはビルド時に，CMake では`-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` オプションを追加，Make では bear[2] を使用すれば生成できる．

5. 評価実験

提案手法の有効性を確かめるため，実際の脆弱性を題材に，プロトタイプによる解析結果と，我々の人手による根本原因解析結果を比較して，どの程度根本原因に近づけたのかを評価する．

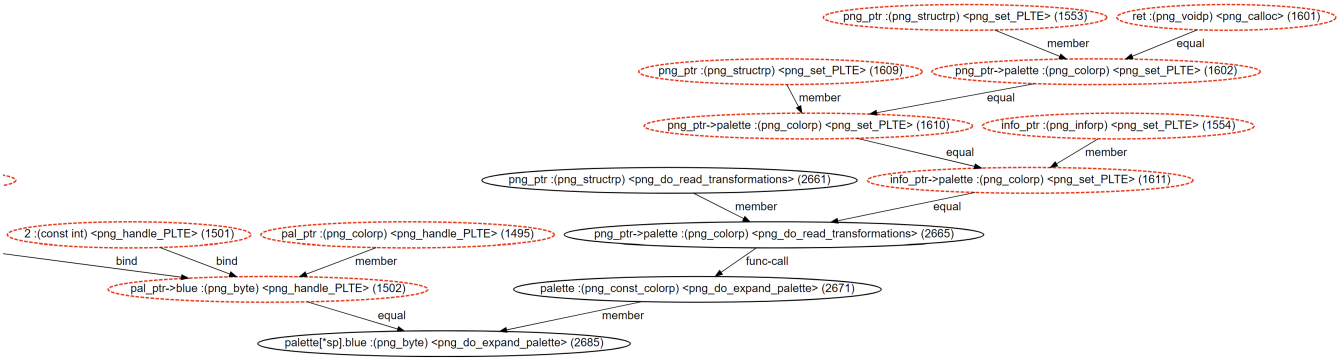

図 6: CVE-2013-6954 のクラッシュ局所化データフローグラフ

5.1 実験条件

実際の脆弱性を再現するため，ファジング評価用データセットである Magma[10] v 1.2 を採用した．Magma は既存 OSS の C/C++ソースコードで既知脆弱性を再現できる仕組みを持っているためである．

本実験でケーススタディする脆弱性は以下の 2 つである．#付きの数字は実験のケース番号，カッコ内の数字は Magma での脆弱性識別 ID である．

- #1 libpng: CVE-2013-6954 (PNG007)
- #2 libtiff: CVE-2016-10269 (TIF005)

これらの脆弱性を選んだ基準は 2 つある．1 つ目は，Magma で取り扱いのあるソフトウェアライブラリのソースコードの大部分を計装できることである．ソースコードの抽象構文木に合わせて計装する都合上，未対応のコードが残るライブラリがあった（詳細は 6 節で述べる）．libpng と libtiff は大部分のコードを計装できたため実験で扱うことができた．2 つ目は，脆弱性がデータフロー依存であり，クラッシュ箇所から根本原因までデータフローが辿れることである．なぜなら，本提案手法はクラッシュ箇所から辿れるデータフローに依拠しているためである．我々は事前に Magma で取り扱いのある脆弱性の根本原因解析を人力で実施した上で，2 つ目の基準を満たす脆弱性を選んだ．

本実験では，各脆弱性の修正パッチが根本原因を示すと仮定する．根本原因解析は本質的に難しく，2.1 節で取り上げた CVE-2013-6954 を例に取ると，未初期化のまま関数 `png_set_PLTE` が呼ばれた原因こそ根本原因と考えることもできるが，提案手法で推定したクラッシュ原因が根本原因とどの程度近いのかを客観的に評価するために，上記の仮定をした．

5.2 実験手順

実験は 2 節で説明した手順に沿って行う．本節では，実験中に我々の判断が伴った作業について詳細を述べる．

ファジング． Magma はファジング結果をアーティファクトとして公開している [6]．本実験では，ファーティファクトに含まれるクラッシュ入力および非クラッシュ入力を無改変で利用した．

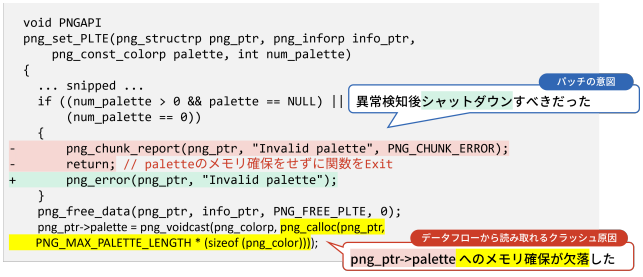

図 7: #1 libpng: CVE-2013-6954 の根本原因解析結果

クラッシュ前後の入力． 両ケースとも，入力を我々が人手で選んだ．Magma のアーティファクトでは，クラッシュ入力のファイル名に再現した脆弱性の ID が付記されている．その情報頼りに，各ケースにつき 1 つのクラッシュ入力を選び，シードツリーで親子関係あるクラッシュ入力を選ぶ．#1 は，選んだクラッシュ入力の祖先（父，祖父，曽祖父，…）のクラッシュ入力が，クラッシュとは関係ないエラー（libpng でエラーとして適切に処理される）が発生したため，結局そのようなエラーが発生しない祖先の始祖にあたる入力を非クラッシュ入力として選んだ．#2 は，選んだクラッシュ入力の父にあたる入力を選んだ．

ソースコード計装． `compile_commands.json` は，Magma がライブラリ毎に用意しているビルドスクリプトを改変して，生成可能とした．

5.3 実験結果

表 2 は，ケース#1, #2 の各起点ノードに対応するクラッシュ局所化データフローグラフから読み取れる差分と，事前に我々が行った根本原因解析結果と突合して，根本原因であるかを判定した結果をまとめたものである．図 7,8 は，根本原因があると認めた差分について，それぞれの差分と脆弱性の修正パッチの意図をソースコード上にマップしたものである．両ケースおいて，差分とパッチ箇所の間にはデータフローは無く，因果関係はあるので，差分内容はデータフロー上から根本原因に至ったと判断した．

ケース	クラッシュ関連変数候補	クラッシュ局所化データフローグラフが示す差分	RC?
#1	`*sp`	`png_ptr->flags` に対する 定数 0x2000U や 0x0002U を伴う演算がされなくなった	No
	`palette[*sp].blue`	`png_ptr->palette` へのメモリ確保が欠落した	Yes
#2	`n`	（差分なし）	No
	`tif->tif_dir.td_colormap`	（差分なし）	No
	m	(1) クラッシュ入力だけが呼んだ関数`LogLuvClose` で，`td_bitspersample` フィールドが 16 に上書きされた	Yes
		(2) クラッシュ入力だけ，関数`_TIFFVSetField` での`td_bitspersample` フィールドへの書き込みが無くなった	No

表 2: 実験結果．「RC?」の列は，その差分が根本原因か否かを意味する．

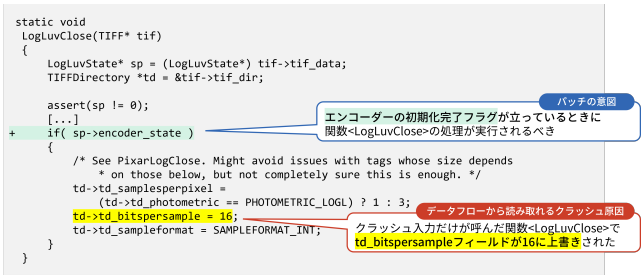

図 8: #2 libtiff: CVE-2016-10269 の根本原因解析結果

6. 今後の課題

現在の提案手法およびプロトタイプでは，評価実験のケースを増やすことが難しい．要因は以下の 3 つある．

根本原因とクラッシュ箇所の間にコントロールフローが介在するケース．この場合，提案手法がデータフローに依拠するため，クラッシュ関連データフローグラフが根本原因に行き着かない．例えば，CVE-2015-8472（Magma の PNG003）は入力由来の値によって for ループが実行され，範囲外参照が発生する脆弱性である（図 9 に該当コードを示す）．我々は提案手法にシンボリック実行を取り入れることで，コントロールフロー依存の脆弱性もデータフロー依存の脆弱性も扱えるようになると考える．シンボリック実行とは，プログラムの入力（ファイル，実行開始時のレジスタやメモリ）をシンボル化（数式でいう変数に相当）してプログラムを実行する動的解析手法である．解析結果として，ある地点が実行されるための制約や，特定のメモリアクセスがされるための制約が得られるため，その制約を解くことで脆弱性の発生有無を判定することができる．ARCUS [12] が先行事例にあたる．ARCUS はクラッシュが検出された状況で，クラッシュの状況（メモリダンプとコントロールフロー）からシンボリック実行を用いた根本原因解析を実施，および修正パッチの提案をした．

実装不足により実験対象のソフトウェアを拡充不能．本実装の不十分な点の一つとして，マクロを展開してから計装する処理がある．例えば，openssl は構造体の参照でマクロを多用しており，データフローが正しく取れないため実験対象から除外した．clang-tidy は，マクロの展開や，マクロ呼び出し箇所の書き換えができない．対策としては，プリプロセッサが出力したマクロ展開済みのコードに対して計装する手段がある．

```
1  void /* PRIVATE */
2  png_handle_PLTE(png_structrp png_ptr, png_inforp
       info_ptr, png_uint_32 length)
3  {
4     png_color palette[PNG_MAX_PALETTE_LENGTH];
5     ... snipped ...
6     /* The cast is safe because 'length' is less
        than 3*PNG_MAX_PALETTE_LENGTH */
7     num = (int)length / 3; // lengthが入力由来
8     ... snipped ...
9     for (i = 0; i < num; i++)
10    {
11       png_byte buf[3];
12       png_crc_read(png_ptr, buf, 3);
13       palette[i].red = buf[0];  // OOB Write
14       palette[i].green = buf[1];
15       palette[i].blue = buf[2];
16    }
```

図 9: CVE-2015-8472 のクラッシュ箇所

変数間の等価性判定が不完全．提案手法の計装は以下の 2 つ問題がある．1 つ目は，実行されるすべてのコードが計装されないためデータフローが必ず途切れる．よって，関知しない等価関係が発生する．2 つ目は，構造体の構造を関知しないため，等価と判定できないケースが存在する．例えば，ある構造体の参照が `s.a.b` とできるとき，`a = s.a; a.b` のように変数を挟んだ場合，`s.a.b` と `a.b` を等価とみなせない）．対策としては，テイント解析の要領で，タグを用いて変数の等価関係を tracee 実行時に追跡する手段が考えられる．テイント解析とは，プログラムの実行で値の伝播を追跡する動的解析手法である．

7. 関連研究

Fault localization は，プログラムへ正常入力（テストが成功する入力）と異常入力（テストが失敗する入力）を与

え，両者の挙動の差異を統計的に評価し，原因とする差異箇所の候補群を提示する手法である．近年，ファジングを前提とした fault localization が提案されている [8, 13]．AURORA [8] は，コントロールフローとデータフローに着目した．クラッシュ入力をもとに，ファジングを用いて正常・異常入力を大量に生成し，実行トレースから生成した各 predicate（プログラムのある地点においてクラッシュを決定づける因子；例えば，レジスタの値が一定値より大きいか）が独立にクラッシュに寄与すると仮定した fault localization をした．しかし，クラッシュの原因と推定した 50 個の predicate に false positive が多く混じることが課題である．DeFault [13] は，クラッシュの原因の候補（基本ブロック）の間の依存関係に着目した．評価実験では，AURORA よりも false positive が低い結果となった．この論文は，データフローに依存した脆弱性では DeFault の効果が薄いため，そのような脆弱性は他の手段 [14] の採用を提案している．AURORA と DeFault はどちらも，クラッシュ入力を得た後に，正常・異常入力を生成するためにファジングする点が共通している．

根本原因解析を目的とした点と，正常・異常入力を必要とする点で，本研究は fault localization に分類されると考える．本研究が先行研究と異なる点は，人間にとって分かりやすいソースコードの字句情報の粒度で根本原因解析する点と，ファジングでクラッシュを見つけた時に得たシードツリーを活用する点である．

8. 結論

本論文は，ファジングにおける根本原因解析を支援するために，ソースコード計装による字句化及びクラッシュ前後のシードツリーを用いた局所化により，解析者が分かりやすいデータフローグラフを生成する手法を提案した．字句情報はソースコード計装によって付加した．局所化のために，クラッシュの前後のシードで得たデータフローの差分を利用した．評価実験では，Magma に含まれる 2 つの既知の脆弱性に対して，データフロー上から根本原因に至ることが容易になったことを確認した．

参考文献

[1] american fuzzy lop. `https://lcamtuf.coredump.cx/afl`. [Online; accessed 20. Aug. 2022].

[2] Bear. `https://github.com/rizsotto/Bear`. [Online; accessed 20. Aug. 2022].

[3] Clang-Tidy — Extra Clang Tools 16.0.0git documentation. `https://clang.llvm.org/extra/clang-tidy`. [Online; accessed 20. Aug. 2022].

[4] ClusterFuzz. `https://google.github.io/clusterfuzz`. [Online; accessed 16. Aug. 2022].

[5] libFuzzer – a library for coverage-guided fuzz testing. `https://www.llvm.org/docs/LibFuzzer.html`. [Online; accessed 20. Aug. 2022].

[6] Magma Artifacts. `https://osf.io/resj8`. [Online; accessed 20. Aug. 2022].

[7] sanitizers. `https://github.com/google/sanitizers/wiki/AddressSanitizer`. [Online; accessed 21. Aug. 2022].

[8] T. Blazytko, M. Schlögel, C. Aschermann, A. Abbasi, J. Frank, S. Wörner, and T. Holz. AURORA: Statistical crash analysis for automated root cause explanation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 235–252. USENIX Association, Aug. 2020.

[9] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. RETracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM - Association for Computing Machinery, May 2016.

[10] A. Hazimeh, A. Herrera, and M. Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), Dec. 2020.

[11] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure Sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 344–360, New York, NY, USA, 2015. Association for Computing Machinery.

[12] C. Yagemann, M. Pruett, S. P. Chung, K. Bittick, B. Saltaformaggio, and W. Lee. ARCUS: Symbolic root cause analysis of exploits in production systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1989–2006. USENIX Association, Aug. 2021.

[13] X. Zhang, J. Chen, C. Feng, R. Li, W. Diao, K. Zhang, J. Lei, and C. Tang. DeFault: Mutual information-based crash triage for massive crashes. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 635–646, New York, NY, USA, 2022. Association for Computing Machinery.

[14] X. Zhang, N. Gupta, and R. Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Softw. Engg.*, 12(2):143–160, apr 2007.