

RCASmith：AIエージェントの根本原因解析能力の評価

川古谷 裕平^{1,a)} 大月 勇人¹ 塩治 榮太朗¹ 岩村 誠¹

概要：ソフトウェアの脆弱性を検出するため、ファジングが広く使われている。ファジングの結果、大量のクラッシュを発見することがある。そのため、これらクラッシュの原因究明、根本原因解析（Root Cause Analysis, 以後 RCA），の自動化が望まれている。この RCA を自動化する既存研究の 1 つ、統計的デバッグ手法は、その精度・効率性に問題があり、大量のクラッシュ解析には向かない。そこで、新たな RCA アプローチとして、高いソースコード理解力を持つ AI エージェントに、デバッガによる動的解析能力を与えた上で、人間の RCA 思考ステップを模倣させた RCASmith を紹介する。RCASmith の評価のため、我々が独自に発見した脆弱性から構成されるデータセットを用いて実験を行った。結果、ソースコードを活用できる場合、高い RCA 能力が見られた。しかし、同時に見当違いの指摘をする場合も増加するなど課題も見られた。

キーワード：根本原因解析、AI エージェント、脆弱性、ファジング

RCASmith: An Empirical Study of RCA Capabilities of AI Agents

YUHEI KAWAKOYA^{1,a)} YUTO OTSUKI¹ EITARO SHIOJI¹ MAKOTO IWAMURA¹

Abstract: Fuzzing is widely used to detect software vulnerabilities. As fuzzing often uncovers a large number of crashes, automating root cause analysis (RCA) has become highly desirable. One existing approach is statistical debugging; however, its accuracy and efficiency are limited, making it unsuitable for analyzing large volumes of crashes. In this paper, we introduce RCASmith, an AI agent that combines advanced source-code understanding with debugging capabilities, designed to imitate the reasoning steps and expertise of human for RCA. To evaluate RCASmith, we conducted experiments using a dataset of undisclosed vulnerabilities that we independently discovered. The results show that, when source code information is available, RCASmith outperforms existing approaches. At the same time, the experiments also revealed challenges, such as a higher incidence of misidentifying root causes.

Keywords: Root Cause Analysis, AI Agent, Vulnerabilities, Fuzzing

1. はじめに

ソフトウェアテストの 1 つにファジングがある。ファジングは解析対象プログラムをクラッシュさせる入力を自動的に探索するため、大量のクラッシュを発見することがある [4]。クラッシュダンプ解析を行えば、クラッシュしたプログラム箇所を特定することができる。しかし、このプログラム箇所が、クラッシュの本当の問題箇所（根本原因）

だとは限らない。そのため、クラッシュの深刻度評価や修正パッチ作成の前に、その根本原因を特定する Root Cause Analysis (RCA) が必要になる。

RCA を自動化・省力化する既存研究の 1 つに統計的デバッグ手法がある。これは、クラッシュ時と非クラッシュ時の挙動の違いを利用して根本原因箇所を特定する手法である。この統計的デバッグ手法は、人が RCA を行う際の補助ツールとして設計されていた。そのため、ファジングで見つけたクラッシュの RCA を逐次的に行わせる利用形態を考えると以下の 3 点が問題となる。①：統計的デバッグ手法は、根本原因候補として複数箇所を出力し、一

¹ NTT 社会情報研究所
NTT Social Informatics Laboratories
a) yuh.kawakoya@ntt.com

つに決定しない。言い換えると、その RCA 精度が低い。
②：対応できる脆弱性に限りがあり、ファジングが対象とする脆弱性の種類と差異が生じる。例えば、型混同の脆弱性は統計的デバッグ手法で適切に検知するのが難しい。
③：実行時間が長く、十数時間要することもある。これらの問題点のため、統計的デバッグ手法をファジングのワークフローに組み込み、大量のクラッシュを逐次的に処理させる利用形態は難しいと考える。

本論文では、統計的デバッグ手法とは異なる新しいアプローチとして、高いソースコード理解力を持つ AI エージェントに、デバッガを使った動的解析能力を与えて、RCA を行わせる RCASmith を提案する。RCASmith は、与えられたバイナリ、クラッシュを引き起こす入力、ソースコードから、自律的に根本原因解析を行う。この際、人間が根本原因解析を行う際の着眼点や思考過程をプロンプトとして与え、それらを模倣することで効率的に RCA を行わせる点を特徴とする。

RCASmith の評価のため、我々が独自に発見し、その詳細が未公開の 7 つの脆弱性を含むデータセットを利用して実験を行った。ここでは、ソースコードやシンボル情報の有無が RCA の精度に与える影響の評価も行った。

結果、RCASmith は 7 件の脆弱性に対してそれぞれ複数回の根本原因解析を実施した結果、5 件の脆弱性については 1 回以上完璧に根本原因箇所を指摘し、適切な修正方法も示した。また、残りの 2 件も根本原因箇所に関連のある箇所を指摘するなど高い RCA 能力を示した。この際、ソースコードの存在は根本原因特定の能力を向上させる効果が見られたが、大きく外す可能性も増加させ、その変動を大きくする傾向が見られた。一方、シンボル情報は RCA 失敗の可能性を増加させる影響も見られた。

本論文の貢献は以下の 3 つである。

- ファジングワークフローの中で統計的デバッグ手法を利用する際の問題点を整理し、実験によって確かめた。
- 新たなアプローチとして、AI エージェントを利用し自律的に根本原因解析を行う RCASmith を提案した。
- 独自発見の脆弱性から構成されるデータセットを利用して評価を行い、ソースコードやシンボル情報の有無が与える影響を考察した。

2. 研究動機

ここでは、プログラムのクラッシュ箇所とその根本原因箇所が異なる脆弱性の例 (CVE-2024-45679) を説明する。さらに、既存研究である統計的デバッグ手法について述べ、その問題点を指摘する。

2.1 動機の例：CVE-2024-45679

CVE-2024-45679 は assimp-5.4.3 以前に存在したヒープバッファオーバーフローの脆弱性である。加工した入力ファ

イルを与えることで、ヒープ上のデータ構造を破壊し、任意コードの実行を可能にする。図 1 にプロセスがクラッシュした際のバックトレースの抜粋、クラッシュ箇所、この脆弱性の根本原因箇所を示す。

このバックトレースによれば、./code/Material/MaterialSystem.cpp で new の呼び出し中に、libc の malloc が呼ばれ、不正なヒープ構造の検知により、クラッシュが発生している。

この脆弱性の根本原因は Vertex 要素が複数回定義されることを想定していない点である。具体的には、mGeneratedMesh→mVertices が NULL の場合（最初の Vertex 要素の定義の際）、mGeneratedMesh→mVertices に mNumVertices 要素数分の領域を確保する。しかし、2 回目以降では mGeneratedMesh→mVertices のサイズの調整が行われない。つまり、2 回目以降で、1 回目に確保した領域よりも大きな領域（大きな mNumVertices 要素数分の領域）を要求し、そこに書き込みを行った場合、1 回目で確保したヒープ上の領域を超過した書き込み（ヒープバッファオーバーフロー）が発生する。これにより、ヒープ上のデータ構造を壊しクラッシュの原因を作ってしまう。このように、脆弱性によっては、クラッシュ箇所と脆弱性の根本原因箇所が異なる場合がある。このような脆弱性を修正するには、クラッシュを引き起こす根本原因箇所を特定することが肝心となる。

2.2 統計的デバッグ手法

根本原因解析の最も原始的な方法は、人間がデバッガやソースコードを基に行う方法である。しかし、これは十分なソースコードの理解が必要になり、時間がかかる。そのため、根本原因解析を自動化、省力化する手法が提案されてきた。ここでは、代表的な手法である統計的デバッグ手法について述べる。その他の手法は 5 章で説明する。

統計的デバッグ手法は、クラッシュを引き起こす入力と引き起こさない入力の双方を与えた時に観測される挙動の違いを利用して根本原因箇所を自動的に見つける手法である ([1] [9] [8])。具体的には、クラッシュを引き起こす入力を少しづつ変化させて解析対象に与え実行する。これをクラッシュが起らなくなるまで繰り返す。これにより、一定数のクラッシュを引き起こす入力と引き起こさない入力を収集する。次に、それら入力の実行時の挙動の違いを観測、比較し、クラッシュを引き起こす入力の実行時にのみ現れる挙動を見つける。この挙動の違いを生む原因箇所を根本原因箇所とする。

この手法は、クラッシュを起こすバイナリと入力があれば、プログラムの内容を詳しく理解する必要はない。また一連の処理は自動的に行われ、実装によってはシンボル情報やソースコードも必要ないといった特徴を持つため、その実用性が主張されている。

```

...
#6 in malloc_printerr (..., "malloc(): invalid size (unsorted)") at
#7 in _int_malloc (...) at ./malloc/malloc.c:4071
#8 in __GI___libc_malloc (...) at ./malloc/malloc.c:3321
#9 in operator new(unsigned long) () from /lib/x86_64-linux-gnu/libstdc++.so.6
#10 in aiMaterial::aiMaterial() () from ./code/Material/MaterialSystem.cpp:395
#11 in Assimp::PLYImporter::LoadMaterial
...

391 aiMaterial::aiMaterial() :
392     mProperties(nullptr),
393     mNumProperties(0),
394     mNumAllocated(DefaultNumAllocated) {
395 // Allocate 5 entries by default
396     mProperties =
397         new aiMaterialProperty *[DefaultNumAllocated];
398 }
```

クラッシュ時のスタックトレース

./code/Material/MaterialSystem.cpp

```

if (nullptr == mGeneratedMesh->mVertices) {
    mGeneratedMesh->mNumVertices = pcElement->NumOccur;
    mGeneratedMesh->mVertices = new aiVector3D[mGeneratedMesh->mNumVertices];
}
```

根本原因箇所

図 1: CVE-2024-45679 のクラッシュ箇所と根本原因箇所

Fig. 1 Crash location and root cause of CVE-2024-45679

2.3 問題分析

この統計的デバッグ手法をファジングワークフローに組み込んで利用する形態を想定すると、以下の 3 つが問題になると考へる。

- 複数候補の列挙
- 対応脆弱性の差異
- 実行時間

複数候補の列挙に関して、統計的デバッグ手法（例、BENZENE [8], AURORA [1]）は人間による解析を補助するツールとして設計されている。そのため、その出力は根本原因の候補となる箇所を複数列挙するものであり、一つの根本原因に絞り込まない。各根本原因候補にはスコアがあるので、トップスコアを選ぶことで 1 つを選択できるが、その精度は高くない。例えば、BENZENE の場合、トップスコアのものだけを根本原因とする方法で評価した場合、その精度は $24/60=40\%$ まで低下してしまう^{*1}。

対応脆弱性の差異に関して、統計的デバッグ手法が対象としている脆弱性と、ファジングが検出可能な脆弱性の種類に差異がある。例えば、AURORA や BENZENE の現在公開されている実装では、クラッシュ入力と非クラッシュ入力の挙動を比較する際、ポインタが指しているデータ構造の違いを意識しない。そのため、あるポインタが異なるデータ構造を指していることが原因でクラッシュを引き起こす場合（例えば、型混同脆弱性）、その根本原因を特定できない。

実行時間に関して、我々の実験（表 4）では 1 つのクラッシュの根本原因を解析するのに、BENZENE では十数分、AURORA の場合は数時間から十数時間を必要とした。設定により、実行時間を短縮することは可能であるが、実行時間を短くすると十分な量のクラッシュ・非クラッシュ

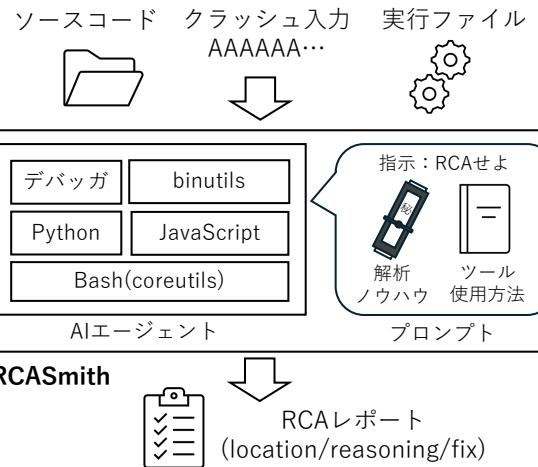


図 2: RCASmith 概要

Fig. 2 RCASmith Overview

入力を集めることができずに精度が（さらに）低下してしまう。

以上のように統計的デバッグ手法は 3 つの問題がある。それらは互いに影響しており、解決するのは容易ではない。そのため、統計デバッグ手法とは異なる新たなアプローチが望まれる。

3. RCASmith

ここでは統計的デバッグ手法とは異なるアプローチとして、AI エージェントを利用して自律的に根本原因解析を行わせる RCASmith を提案する。まず、RCASmith の概要について述べ、次に設計を説明する。続いて、RCASmith の実装と、AI エージェントに与えるプロンプトテンプレートの設計を述べる。

3.1 概要

図 2 に RCASmith の概要を示す。RCASmith は、Bash を通してデバッガや各種コマンドを自律的に利用できる AI

^{*1} BENZENE の論文での評価では精度は 93.3%と高いが、これは BENZENE が output する候補 50 位までの中に本当の根本原因が含まれる精度が 93.3%という意味であり、根本原因の特定精度が 93.3%という意味ではない

エージェントである。RCASmithへの入力は、クラッシュを引き起こす入力、解析対象のバイナリ、ソースコードの3点になる。また、出力はソースコード上の根本原因について分析したレポートになる。レポートには、主にクラッシュの状況分析、根本原因箇所とその説明、修正方法の提案の3つが含まれるようにプロンプトで指示している。

3.2 設計

RCASmithはAIエージェントが持つ高いソースコード理解力を活用した静的解析に加え、デバッガを利用した動的解析により、プログラムのクラッシュからその根本原因の特定を行うことを目的とする。

AIエージェントの基となる大規模言語モデル(LLM)は脆弱なコードの特徴も学習しているため、ソースコードを静的に解析させ、脆弱性検出に利用できる。しかし、LLM単体では、その脆弱性検出力は限定的であり、例えば、C/C++のポインタや構造体が関係する脆弱性は正確に検出できない可能性が報告されている[5]。また、LLMが扱えるソースコードの量(コンテキストウィンドウ)には制限があり、大量のコードを解析対象とする場合、検知性能が低下することもある。

そこで、RCASmithでは、デバッガを利用し、クラッシュに関する情報を集め、これらを活用することで、LLMの欠点を補完することを目指している。具体的には、デバッガの利用には以下の2つの意図がある。①：実行時のコンテキスト情報(レジスタ・メモリ)を取得し、ポインタや構造体の解析を補助し、複雑な脆弱性に対処させる。②：プログラム起動時からクラッシュまでの動作を観測させ、観測された関数群を解析対象として、見るべきコード量を絞り込む。

RCASmithでは、上述のデバッガの活用を促進するため、以下の2つの情報をプロンプトで与える。①：デバッガの使い方、その出力例、それらの解釈をFew-shotで与える。②：人間の解析者がRCAを行う際の手順や着眼点を与える、これを模倣させる。具体的には以下のような指示が含まれる。

- 最初に、与えられたクラッシュ入力で解析対象バイナリを実行し、クラッシュを再現する
- クラッシュ発生時のバックトレース情報を取得し、経路上の関数とそれらに関連する関数に着目する
- 着目した関数で利用されている変数を列挙し、それら変数を利用する他の関数にも着目する
- クラッシュ発生に寄与している変数と関係を持つ変数に着目する

また、デバッガ以外の各種コマンド(coreutils, binutils)の利用も許可している。例えば、objdumpでコードの逆アセンブルを取得したり、nmやreadelfでシンボル情報の取得を行うことができる。ソースコードへのアクセスは、AI

エージェントのファイルアクセス機能(読み込み、書き込み、検索)とシェルコマンドを通じたアクセスで行う。

3.3 実装

AIエージェントにはAnthropicのClaude Code v1.0.44(claude-sonnet-4-20250514)を利用した。RCASmithは、Ubuntu 24.04LTSをベースにしたDockerコンテナ上で動作する。このDockerイメージには、Claude Code, gdb, git, エディタ, JavaScript 実行環境, Python 実行環境をインストールして作成した。途中経過の出力や最終レポートの生成には日本語を利用させるように設定ファイルで指示を出した。

3.4 プロンプトテンプレート

RCASmithへは、下のプロンプトを与える。このプロンプトは、最初の命令文、解析対象情報、解析の目的、解析方法、ツールの利用方法、作業に関する指示の6つの部分から構成される。

プロンプトテンプレート

バイナリのクラッシュ解析を依頼します。以下の情報に基づいて、デバッガを最大限に活用し、根本原因を特定してください。

```
# 解析対象情報  
(入力ファイルやクラッシュ入力などの基本情報)  
# 解析の目的  
単なる表面上のクラッシュ情報に留まらず、クラッシュの本当の原因を特定してください。  
# 解析方法  
(解析者のノウハウやステップの説明)  
# ツールの利用方法  
(各種ツールの説明)  
# 作業に関する指示  
(出力ファイル形式、時間情報の取得、禁止事項、など)
```

最初の命令文と解析の目的は上のプロンプトの通りである。解析対象情報は、RCASmithへの入力と出力に関するファイルやディレクトリへのパスが含まれる。解析方法には、3.2節で説明したように、解析者が根本原因解析を行う際のポイントやステップに関する指示が書かれている。ツールの利用方法は、gdbコマンドとその結果例やその解釈がFew-shotとして書かれている。作業に関する指示には、出力ファイル形式の指示、実行時間の計測方法、禁止事項などが記載されている。

4. 実験

RCASmithの根本原因特定の能力を評価するため実験を行った。まず、検証項目(Research Question, RQ)を整理

する。次に、実験方法とデータセットについて述べる。その後、検証項目に回答する形で実験結果を述べる。

4.1 検証項目

本実験における検証項目として、以下の3つのResearch Question (RQ) を設定した。

RQ1 RCA能力とソースコード・シンボルの有無の影響はあるか？

RQ2 失敗時の傾向や特徴はあるか？

RQ3 既存研究に対する優位性はあるか？

RQ1に関して、RCASmithは、クラッシュするバイナリ、クラッシュを引き起こす入力ファイル、ソースコードの3つに基づいてそのクラッシュの根本原因を特定できるかを評価する。特に、ソースコードやシンボルの有無が根本原因解析の精度や説明の合理性に影響を与えるかを評価する。

RQ2に関して、実験の中で根本原因箇所の特定に失敗した場合、その失敗に関する脆弱性の特徴（脆弱性の種類、クラッシュと根本原因箇所の距離、クラッシュと根本原因の位置関係）を分析し、RCASmith (AIエージェント) が苦手とする属性や構造を調査する。

RQ3に関して、統計的デバッグ手法の既存研究 (BEN-ZENE [8], AURORA [1], VulnLoc [9]) の精度とパフォーマンスを RCASmith と比較する。

4.2 実験方法

実験では、RCASmithに対して入力を与え、クラッシュの根本原因解析を行わせ、レポートを出力させる。そのレポートを人間が評価する。なお、AIエージェントの確率的動作による変動を考慮し、同じ入力での試行を4回ずつ行う。つまり、合計で、脆弱性数 × 入力パターン数 × 4回の試行を行うことになる^{*2}。以下、入力パターン、評価方法、データセットについて説明する。

4.2.1 入力パターン

RQ1への回答するため、バイナリとソースコードの入力に関して、シンボル情報あり^{*3}・なしとソースコードあり・なしの合計4パターンを用意し結果を比較する。4パターンの表記、patXY のX部分はシンボルの有無(1:有、0:無)を意味し、Yの部分はソースコードの有無(1:有、0:無)を意味する。RQ2への回答として、各試行における失敗やスコアが低いケースへの属性情報の影響を統計的に分析する。またRQ3への回答として、pat11バイナリを利用し、既存研究ツールを動作させ、結果の比較を行う。なお、クラッシュ入力に関しては全パターンで共通である。

4.2.2 評価方法

RCASmithが出力するレポートの評価は、脆弱性分析の

^{*2} 本実験では、 $7 \times 4 \times 4 = 112$ 回の試行を行った

^{*3} -g2でコンパイルした

知識と経験のある人間（解析者）により行われ、0（最低点）から2（最高点）の3段階でスコアをつける。

各脆弱性の根本原因の正解データは、事前に解析者により解析が行われ、ソースコード上のファイル名と行番号で根本原因箇所が指定される。また、報告済み脆弱性に関しては、その修正パッチも公開済みのため、これらも各脆弱性の修正方法の正解データとして利用する。パッチが未公開のものは、解析者が根本原因を把握の上、作成したパッチを正解データとして利用する。

レポートの評価は、根本原因箇所の特定 (location), クラッシュの原因の説明 (reasoning), 修正方法の提案 (fix) の3つの観点で行われる。各項目、0（最低点）から2（最高点）で加点され、合計点を平均したものスコア ($= \frac{\text{location} + \text{reasoning} + \text{fix}}{3}$, s.t. location, reasoning, fix $\in \{0, 1, 2\}$) として利用する。

locationはRCASmithが特定したソースコードのファイル名と行数が正解データと合致するかで評価する。reasoningは、根本原因からクラッシュ発生に至るまでの経緯をRCASmithに自然言語で説明させ、その説明の合理性を解析者が評価する。fixはRCASmithによる修正方法の提案と正解データのパッチを比較し、妥当な修正であるか解析者が評価する。この際、必ずしも正解データのパッチと同一の修正である必要はなく、解析者が脆弱性を塞ぐのに妥当だと判断すれば加点される。

各レポートはまず解析者一名によりレポートの一次評価が行われ、もう一名によりその評価の確認を行う。これら二名による評価に差異が生じた場合は、合議により評価を決定する。

4.2.3 データセット

本実験では、脆弱性の詳細をLLMが学習しているケースを避けるため、その詳細情報がインターネット上に公開されていない独自発見のクラッシュと脆弱性から構成されるデータセットを評価に利用する（表1）。

このデータセットは5つの報告済みの脆弱性と2つの未報告の脆弱性から構成される。脆弱性の種類として、スタックバッファオーバーフロー (SBOF), ヒープバッファオーバーフロー (HBOF), 整数オーバーフロー (IOF), NULLポインタ参照 (Null-deref) が含まれる。距離はクラッシュ箇所と根本原因箇所の位置関係をコールグラフ上の最短パスのホップ数で表している。BT (BackTrace) は、クラッシュ箇所で取得したバックトレース情報の中に根本原因箇所を含む関数が現れるかを示している。

4.3 結果

ここでは、検証項目で定義した3つのRQへの回答という形で実験結果を示す。表2に各脆弱性4つのパターンに関して、4回の試行の平均スコアとカテゴリ別 (location, reasoning, fix) 平均スコアを示す。また、表3に4回の

表 1: データセットの概要
Table 1 Dataset overview

脆弱性	CVE	種類	距離	BT ^a
VUL-01	CVE-2024-40724	HBOF	9	-
VUL-02	CVE-2024-41881	SBOF	0	✓
VUL-03	CVE-2024-43700	SBOF	2	-
VUL-04	CVE-2024-45679	HBOF	14	-
VUL-05	CVE-2024-55577	SBOF	2	✓
VUL-06	- ^c	NULL-deref	0	✓ ^b
VUL-07	- ^c	IOF	2	-

^a ✓ バックトレース上に根本原因を含む関数が現れる ^b 最適化の影響で BT に現れる関数のソースコード行数が間違っている。^c 報告準備中

試行におけるスコアの最小値、最大値、標準偏差を示す。

実行時間を計測できた^{*4}80回の試行のうち、計測ミスによる外れ値を除外した後の平均実行時間は4分42秒（標準偏差：51秒、最大値：7分4秒、最小値：2分49秒）だった。

4.3.1 RQ1: RCA 能力とシンボル・ソースコードの影響

表2によると、最も平均スコアの高いpat01では、7件中5件でスコア1以上（根本原因箇所と関連のある箇所の指摘）だった。また表3によると、pat11は、7脆弱性のうち5件は、4回の試行中1回以上でスコア2.0（完璧に根本原因箇所を指摘）を取得しており高いRCA能力を示した。

ソースコード情報の影響に関して、pat00とpat01、pat10とpat11を比較する。どちらもソースコードにより、平均スコアが向上した。カテゴリ別で見ると、特にlocation（根本原因箇所の特定）スコアが向上していることがわかる。また、表3によると、ソースコードを与えた場合、最低値(min.)スコアが低下している。以上より、ソースコードを与えることで精度は向上するが、失敗の可能性も増加し、成否の幅が大きくなることを示している。

シンボル情報の影響に関して、pat00とpat10、pat01とpat11を比較すると、どちらも大きな性能向上は見られなかった。表2によると、シンボル情報はpat00、pat10でlocationスコアの向上が見られた以外はスコアの減少が見られた。これらの点に関しては6章で考察する。

4.3.2 RQ2: 失敗時の傾向

表2の結果では、VUL-05、VUL-03が他の脆弱性と比べ相対的に低いスコアとなった。また、VUL-05ではreasoningとfixの項目が、VUL-03ではlocationの項目が他よりも低いスコアになった。

脆弱性の属性（種類、距離、BT）に関して、これらの属性と平均スコア、カテゴリ別スコアの関係性をそれぞれ分散分析(ANOVA)、相関分析、t検定にて統計的に分析した。結果、距離と平均スコアに弱い負の相関関係が見られたが、VUL-05、VUL-03はどちらも距離=2で相対的に距離は短く、この負の相関だけからは低スコアの理由を説明

^{*4} AIエージェントにdateコマンドを実行させ計測させた

できない。

そこで、これら二つの脆弱性の根本原因箇所のソースコードを解析すると、どちらの根本原因もループ中に存在し、書き込み可能領域のサイズがループの回数に依存する箇所にバッファオーバフローが発生するものであった。^[6]によれば、LLMによるプログラム解析では動的な挙動の解析に限定的な能力しか持たない可能性が示唆されている。これは事前に学習したコードの大部分は静的なコードを利用しており、動的なコードの挙動についての学習が足りていない点が説明されている。そのため、このループにより動的に変化するバッファサイズといったプログラム構造がAIエージェントによる根本原因箇所の特定を難しくさせている可能性もあると考えている。

4.3.3 RQ3: 既存研究との比較

表4に既存研究であるBENZENE、AURORA、VulnLocを表1のデータセットで実験した結果を示す。また、RCASmithのpat11の平均スコアと最大スコア（括弧内）も示す。なお、コンパイルエラー等のため、既存研究ツールでは、7つすべての脆弱性を動かせなかっただため、ここでは既存研究3つすべて動作できた脆弱性の結果だけを示す。

結果として、RCASmithでは4件中3件の脆弱性で複数回(4回)の試行の中で完璧な根本原因箇所の指摘に1回以上成功している。一方、既存研究は根本原因箇所を順位1位で特定できたものはなかった。またVUL-03とVUL-06では根本原因箇所は候補にも入らなかった。さらに、既存研究では根本原因箇所の説明が述語(例、 $\min(\text{rax}) < 0xff$)なのに対して、RCASmithのレポートには、自然言語によるクラッシュ・根本原因箇所の説明と修正方法に関する提案も含まれている。

実行時間に関して、AURORAとVulnLocの実験では、クラッシュと非クラッシュの入力を一定数揃えるため、120分から720分の時間をかけた。また、BENZENEの実験では入力を揃える処理にデフォルトの設定である10分かかり、合計で十数分の時間がかかった。RCASmithの実行時間は、計測ができた80回の試行のうち、外れ値を除去した後の平均実行時間は4分42秒（標準偏差：51秒、最大値：7分4秒、最小値：2分49秒）であった。

以上の結果を踏まえた考察は6章で実施する。

5. 関連研究

ここでは、RCASmithの関連研究を述べる。なお、統計的デバッグ手法は2章で説明済みのため省略する。

POMP [10], POMP++ [7], RETracer [3], REPT [2]は、逆実行解析を利用して、クラッシュ箇所から実行フローを逆方向に遡って分析し、根本原因箇所を特定する手法である。RCASmithでもこれらの研究と同様に逆向きの解析とデータの依存関係の解析をプロンプトで指示している。

ARCUS [11]は、収集した実行トレースに対してシンボ

表 2: レポートのスコア詳細（全体/カテゴリ別）
Table 2 The detailed score (Overall / per category)

脆弱性	pat00	pat01	pat10	pat11	平均
VUL-01	0.42(0.00/0.25/1.00)	1.75 (2.00/1.50/1.75)	0.25(0.00/0.25/0.50)	1.00(0.75/0.75/1.50)	0.85(0.69/0.69/1.19)
VUL-02	0.92(0.00/2.00/0.75)	1.67 (2.00/2.00/1.00)	1.42(1.50/2.00/0.75)	1.67 (2.00/2.00/1.00)	1.41(1.38/2.00/0.88)
VUL-03	0.83(0.00/1.50/1.00)	1.42 (1.75/1.50/1.00)	0.42(0.00/0.25/1.00)	0.42(0.25/0.50/0.50)	0.77(0.50/0.94/0.88)
VUL-04	0.67(0.00/1.50/0.50)	1.42 (1.50/1.50/1.25)	0.42(0.00/0.75/0.50)	0.83(1.00/0.50/1.00)	0.83(0.62/1.06/0.81)
VUL-05	0.00(0.00/0.00/0.00)	0.50(0.75/0.25/0.50)	0.42(0.75/0.50/0.00)	0.83 (1.00/0.75/0.75)	0.44(0.62/0.38/0.31)
VUL-06	1.00(0.00/2.00/1.00)	1.83 (2.00/1.75/1.75)	1.00(0.00/2.00/1.00)	1.50(1.50/1.50/1.50)	1.33(0.88/1.81/1.31)
VUL-07	0.92(0.00/1.50/1.25)	0.83(0.50/1.25/0.75)	0.92(0.75/1.25/0.75)	1.67 (1.75/1.75/1.50)	1.08(0.75/1.44/1.06)
平均	0.68(0.00/1.25/0.79)	1.35 (1.50/1.39/1.14)	0.69(0.43/1.00/0.64)	1.14(1.18/1.14/1.11)	0.96(0.78/1.20/0.92)

4 回の試行の平均スコア (location/reasoning/fix の平均スコア)。

グレー背景太字 : 平均スコアの最高値。太字 : カテゴリ別スコアの最高値。

表 3: レポートのスコア詳細（最小、最大、標準偏差）

Table 3 The detailed score (Min, Max, Standard Deviation)

脆弱性	pat00			pat01			pat10			pat11			平均		
	min.	max.	std.	min.	max.	std.	min.	max.	std.	min.	max.	std.	min.	max.	std.
VUL-01	0.33	0.67	0.17	1.00	2.00	0.50	0.00	0.67	0.36	0.33	2.00	0.83	0.42	1.33	0.46
VUL-02	0.67	1.00	0.17	1.66	1.67	0.00	1.00	1.67	0.50	1.67	1.67	0.00	1.25	1.50	0.17
VUL-03	0.33	1.00	0.33	1.00	1.67	0.36	0.33	0.67	0.17	0.00	1.67	0.83	0.42	1.25	0.42
VUL-04	0.00	1.00	0.53	0.00	2.00	0.99	0.00	1.00	0.51	0.00	2.00	0.99	0.00	1.50	0.75
VUL-05	0.00	0.00	0.00	0.00	1.67	0.82	0.00	0.67	0.36	0.00	2.00	1.02	0.00	1.08	0.55
VUL-06	1.00	1.00	0.00	1.33	2.00	0.33	1.00	1.00	0.00	0.33	2.00	0.86	0.92	1.50	0.30
VUL-07	0.67	1.00	0.36	0.00	1.67	0.82	0.33	1.33	0.50	1.00	2.00	0.53	0.50	1.50	0.55
平均	0.43	0.81	0.22	0.71	1.80	0.55	0.38	1.00	0.34	0.48	1.90	0.72	0.50	1.38	0.46

グレー背景太字 : max. スコアの最高値。太字 : min. の最低値。

表 4: 既存ツールの実験
Table 4 Experiments with existing tools

脆弱性	[8] ^a	[1] ^b	[9] ^c	RCASmith
VUL-03	-	-	-	0.42(1.67)
VUL-04	#311	#393	#7	0.83(2.00)
VUL-06	-	-	-	1.50(2.00)
VUL-07	#2	-	#12	1.67(2.00)

#数字: 根本原因箇所の順位。^a BENZENE ^b AURORA ^c VulnLoc

リック実行を実行し、クラッシュの根本原因解析を行う手法である。ARCUS は特定の脆弱性タイプを検知するため事前に定義されたルールや分析戦略を必要とする。そのため、事前のルール定義が難しい脆弱性（例えば型混同）の場合、検知漏れが発生する可能性がある。

PATCHAGENT [12], CodeRover-S [13] は AI エージェントや LLM を利用してパッチ生成の自動化を目指したシステムである。このパッチ生成の 1 ステップとして、根本原因箇所の特定 (Fault Localization) が含まれている。人間のデバッグ動作を模倣するという方向性は PATCHAGENT と RCASmith は類似する。PATCHAGENT は静的なソースコード解析に頼っているが、RCASmith は適宜デバッガを利用させ、動的な解析も行う。また、CodeRover-S では、

クラッシュレポートと一緒にあらかじめ取得した動的解析の結果を使う方法を提案している。

6. 考察

ここでは、シンボル情報の悪影響、RCASmith のファジングワークフローへの適用性を考察する。

6.1 シンボル情報の悪影響

表 2 の実験結果によれば、pat11 よりも pat01 のほうが RCA 精度が高いという、予想外の結果になった。この原因として、バックトレース (BT) 情報の悪影響が考えられる。シンボル情報が利用可能な pat11 の場合、BT がほぼ正確に作成できる。そのため、RCASmith が、BT 上の関数に固執して解析を行う傾向が見られた。実際、BT 上に根本原因が現れる VUL-02 と VUL-05 は pat11 でトップスコアになっているが、それ以外は VUL-07 を除き、スコアが pat01 より低い。また、VUL-06 は BT に根本原因の関数が現れているが、最適化の影響で BT から辿れるソースコード箇所が間違っている。この間違った情報に影響を受け、pat10 では location を全て失敗する結果となった。今後、より実験データを増やして、この傾向を確かめると同時にプロンプト等を工夫し、解決方法を探っていきたい。

6.2 ファジングワークフローへの適用性

ここでは、2章で述べた統計的デバッグ手法の3つの問題点を4章の実験結果を基に考察し、RCASmithのファジングワークフローへの適用性を考察する。

RCASmithでは複数の根本原因候補を指摘することはなかったが、1回だけの実行では誤った指摘をする可能性がある。そのため、実際の運用では複数回実行し、正しい指摘のレポートを選び出す必要がある。RCASmithのレポートには、述語ではなく、根本原因からクラッシュへ至る経緯の説明とその修正方法が含まれ、検証に利用できる情報が多い。これらを活用したレポートの検証、評価方法は有効だと考える。

対応脆弱性に関して、RCASmithの実験では脆弱性の種類によるスコアへの影響は見られなかった。また、既存ツールが見逃した脆弱性の根本原因も検知できた。しかし、4章で言及したように、動的に発動条件が決定する脆弱性の場合、RCAの成功率が低下する可能性がある。RCASmithでは、このような場合、デバッガを利用して正確な脆弱性発動条件の特定を促したいが、ここでは思い通りに動作しなかった。この原因は追加調査を行なっていく。

実行時間に関して、RCASmithは統計的デバッグ手法の既存ツールよりも実行時間は短く、平均で5分程度であった。これは、統計的デバッグ手法で、最も実行時間が短いBENZENEの半分程度である。また、pat01などの入力パターンの場合、解析対象バイナリの再ビルトや特殊なバイナリ解析が必要ないため、RCASmithを適用できる可能性は、統計的デバッグ手法よりも高い。

以上より、追加の検討は必要ではあるが、RCASmithのファジングワークフローへの適用性は現実的だと考える。

7. まとめ

本論文では、従来の根本原因解析とは異なる新たなアプローチとして、AIエージェントにソースコードの静的な解析とデバッガによる動的な解析を行わせるRCASmithを提案した。また、そのRCA能力を、詳細が未公開な脆弱性データセットで評価した。結果、RCASmithは高いRCA能力を示したが、ソースコードとシンボル情報を与えることによる影響、LLMの特有の不安定さなどの課題も浮き彫りにした。今後は、これらの点を改良し、RCASmithをファジングワークフロー[12]に組み込み、大規模なデータセットを利用した評価や運用課題の解決を行っていく。

参考文献

- [1] Blazytko, T., Schlögel, M., Aschermann, C., Abbasi, A., Frank, J., Wörner, S. and Holz, T.: AURORA: statistical crash analysis for automated root cause explanation, *Proceedings of the 29th USENIX Conference on Security Symposium*, USA, USENIX Association (2020).
- [2] Cui, W., Ge, X., Kasikci, B., Niu, B., Sharma, U., Wang, R. and Yun, I.: REPT: Reverse Debugging of Failures in Deployed Software, *13th USENIX Symposium on Operating Systems Design and Implementation*, Carlsbad, CA, USENIX Association, pp. 17–32 (2018).
- [3] Cui, W., Peinado, M., Cha, S. K., Fratantonio, Y. and Kemerlis, V. P.: RETracer: triaging crashes by reverse execution from partial memory dumps, *Proceedings of the 38th International Conference on Software Engineering*, New York, NY, USA, Association for Computing Machinery, p. 820–831 (online), DOI: 10.1145/2884781.2884844 (2016).
- [4] Kawakoya, Y., Shioji, E. and Otsuki, Y.: PkgFuzz Project: A New Continuous Fuzzing Framework for Open Source Software, *CODE BLUE 2024*, Tokyo, Japan (2024).
- [5] Khare, A., Dutta, S., Li, Z., Solko-Breslin, A., Alur, R. and Naik, M.: Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities, *2025 IEEE Conference on Software Testing, Verification and Validation*, Los Alamitos, CA, USA, IEEE Computer Society, pp. 103–114 (online), DOI: 10.1109/ICST62969.2025.10988968 (2025).
- [6] Ma, W., Liu, S., Lin, Z., Wang, W., Hu, Q., Liu, Y., Zhang, C., Nie, L., Li, L. and Liu, Y.: LMs: Understanding Code Syntax and Semantics for Code Analysis (2024).
- [7] Mu, D., Du, Y., Xu, J., Xu, J., Xing, X., Mao, B. and Liu, P.: POMP++: Facilitating Postmortem Program Diagnosis with Value-Set Analysis, *IEEE Transactions on Software Engineering*, Vol. 47, No. 9, pp. 1929–1942 (online), DOI: 10.1109/TSE.2019.2939528 (2021).
- [8] Park, Y., Lee, H., Jung, J., Koo, H. and Kim, H. K.: Benzene: A Practical Root Cause Analysis System with an Under-Constrained State Mutation, *2024 IEEE Symposium on Security and Privacy*, pp. 1865–1883 (online), DOI: 10.1109/SP54263.2024.00074 (2024).
- [9] Shen, S., Kolluri, A., Dong, Z., Saxena, P. and Roychoudhury, A.: Localizing Vulnerabilities Statistically From One Exploit, *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, New York, NY, USA, Association for Computing Machinery, p. 537–549 (online), DOI: 10.1145/3433210.3437528 (2021).
- [10] Xu, J., Mu, D., Xing, X., Liu, P., Chen, P. and Mao, B.: Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts, *26th USENIX Security Symposium*, Vancouver, BC, USENIX Association, pp. 17–32 (2017).
- [11] Yagemann, C., Pruett, M., Chung, S. P., Bittick, K., Saltaformaggio, B. and Lee, W.: ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems, *30th USENIX Security Symposium*, USENIX Association, pp. 1989–2006 (2021).
- [12] Yu, Z., Guo, Z., Wu, Y., Yu, J., Xu, M., Mu, D., Chen, Y. and Xing, X.: PATCHAGENT: A Practical Program Repair Agent Mimicking Human Expertise, *Proceedings of the 34th USENIX Security Symposium*, Seattle, WA, USA, USENIX Association (2025).
- [13] Zhang, Y., Wang, J., Berzin, D., Mirchev, M., Liu, D., Arya, A., Chang, O. and Roychoudhury, A.: Fixing Security Vulnerabilities with AI in OSS-Fuzz (2024).