Vol.2023-OS-158 No.8
2023/2/21

Linux Kernel におけるトリビアルバグの解析

鈴木慶汰[1]　河野健二[1]

概要：Linux はよく開発されたコードベースであるにもかかわらず、驚くほど簡単なバグに悩まされている．我々の調査によると、Linux v5.11 からサンプリングされたバグ修正パッチの約 60 % が「トリビアルなバグ」であることがわかった．これは、単一のソースファイルを検査することで、関数間のポインタ解析のような複雑な解析を行わずに静的に検出できることを意味する．この事実は、開発者が日々の開発においては静的解析器を使用していないことを示唆している．これは日々の開発で使用するにあたって 2 つのハードルがあるからと考えられる: (1) 長い解析時間と (2) 過剰な量の警告の数である．
本論文では、日常的な開発における静的解析器に着目し、この 2 つのハードルを克服するトリビアルバグに特化した解析器の必要性を論じる．トリビアルバグに特化した解析器は、複雑な解析を行わず教科書レベルの解析のみを行うため、解析時間が短い．また、生成される警告は開発者が修正したソースファイルのみのものであるため、警告の生成数は少ない．これらの解析器は既存のコンパイラに統合できるため，通常のビルドプロセスに変更を加える必要なく Linux に適応できる．単純な解析しか行わないにもかかわらず、我々の試作したトリビアルバグ特化の解析器を組み込んだコンパイラでは、Linux v5.15 で 45 個のバグ（内 13 個が開発者によって承認済み）が発見された．開発の初期段階でこれらのバグを除去することで、後の統合フェーズでのデバッグやテストが加速されると考えられる．

1. Introduction

オペレーティングシステムとして広く普及している Linux は、最も洗練されたシステムソフトウェアの 1 つであり、多くの経験豊富な開発者が日々コードベースに貢献している．2021 年には 86,023 件のコミットが行われ (1 日平均 200 件以上)、4,500 人以上の開発者が 3000 万行を超えるコードベースに貢献している (v5.11 現在)．しかしながら，Linux カーネルには多くのバグが存在していることが報告されている [9], [10], [11], [12], [13], [14], [17], [18], [23], [24], [26], [27], [29], [32], [33], [37], [38], [41], [42], [43], [45], [46], [50], [51].

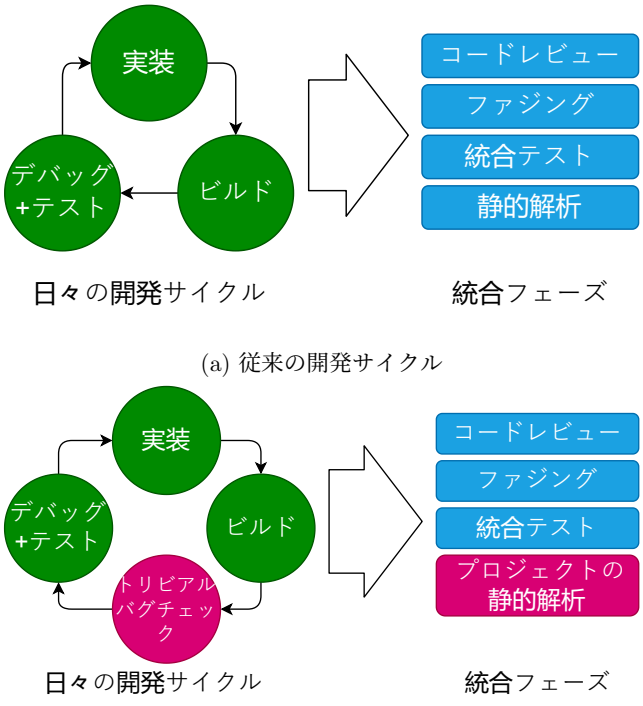
直近の Linux で発見されたバグの傾向を把握するため，v5.9 (2020 年 9 月リリース) から v5.11 (2021 年 1 月リリース) の間に適応された *double free*, *out-of-bounds*, や *integer overflow* といったキーワードを含むバグ修正パッチをサンプルし、調査を行った．

我々の調査によると，　サンプルされたバグのうち約 60% (117 パッチのうち 68 件) がトリビアルバグ であることがわかった. これらのバグは，静的解析において複雑な解析を伴わないバグであり，以下の条件を全て満たす:

- **コンパイル単位を超える解析を必要としない** 1 つのコンパイル単位の解析のみで発見できる．つまり，バグに関わる全ての関数が 1 つのコンパイル単位に存在していた．サンプルされたバグのうち 77 件がこれの特徴があった．
- **フィールドオフセットが静的に決まる**フィールドオフセットの計算を動的の行う必要がない．つまり，バグに関わるフィールドオフセット計算は構造体のフィールドかハードコードされた配列の添字のみである．コンパイル単位内のバグのうち 70 件のバグがこの特徴があった．
- **簡単なエイリアス解析のみで発見可能**エイリアス情報は必要ない，もしくは関数内でかつ path-insensitive なエイリアス情報のみで発見可能である．コンパイル単位内のバグのうち 76 件のバグがこの特徴があり，さらに内 61 件はエイリアス情報を必要としなかった．
- **関数の間接呼び出しがない**関数の関節呼び出しがバグのデータフローに存在しない．コンパイル単位のバグのうち 69 件がこれにあたった．

これらの トリビアルバグ の存在は，Linux の開発者は静的解析器を日常的な開発では使用していないことが示唆されている．これは，2 つのハードルが存在するからと考えられる: (1) 長い解析時間と (2) 過剰な量の警告の数である．静的解析器は関数間のエイリアス解析や path-sensitive な

[1] 慶應義塾大学
Keio Uniersity



(a) 従来の開発サイクル

(b) 提案の開発サイクル

図 1: トリビアルバグに特化した解析器ありとなしの開発サイクル

データフロー解析などの複雑な解析をコードベース全体に行うため，解析に長い時間がかかる．例えば， PATA [31] は Linux 全体を解析するのに 33 時間以上かかる．日々の開発は， ソースコードの変更とビルドを何度も繰り返し行うため，長い解析時間は許されない．

もう一つのハードルとして開発者にとって過剰な量の警告数が挙げられる．開発者はこれらの警告を全て精査して自分に関係のあるモジュールについての警告を確認する必要がある．警告の数が多いため，開発者はそれぞれの警告に十分な時間を費やせず， トリビアルバグ を見逃す，もしくは確認そのものをしない可能性がある．Clang Static Analyzer [6] のようなツールは 132,196 件の警告を出す．

本稿では，日常的な開発における， トリビアルバグ に特化した静的解析器に着目する．トリビアルバグに特化することで、前述した 2 つのハードルを克服することができる．第一に、トリビアルバグに特化したチェッカは、短時間で解析を終えることができる．これは、トリビアルバグを見つけるのに、関数間のポインタ解析のような複雑な解析を必要としないためである．第二に、トリビアルバグの解析はコンパイル単位で行われ、開発者が修正したソースファイルに対してのみ警告を発生させるため、警告の数を抑えることができる．また、これにより既存のコンパイラと統合することが可能であり，開発者はビルド設定をそのまま利用することができる．

トリビアルバグに特化した解析器は、これまでの複雑な分析を行うツールを補完する．ソフトウェア開発は、日常的な開発フェーズと統合フェーズの 2 つで構成されている．図 **??**は、典型的な開発の流れを示している．統合フェーズは、リリース候補の時など、たまに発生し，ファジング、統合テスト、コードレビューなどを用いて開発したコードをコードベースの残りの部分と統合する．トリビアルバグに特化した解析器は、図 **??**に示すように、日常的な開発を想定して設計されている．トリビアルバグを開発初期に除去することで、のちのデバッグやテストを加速させることを想定している．開発者は、膨大な数の警告の中から FiT のバグを探し出す必要がないため、統合フェーズでのデバッグやテストの迅速化に貢献する．

とり日あるバグに特化した解析器の有用性を示すために、LLVM を用いてタイプステート解析 [20], [22], [25], [31], [40], [44], [48] を行うプロトタイプを実装した．教科書レベルの簡単な解析にもかかわらず、我々のプロトタイプは Linux 5.15(allyesconfig 付き) で　45 件の新しいバグを発見した．内 13 件はすでに開発者によって確認されている．また解析時間は、50%のソースファイルで 0.20 秒以下、90%のソースファイルで 0.98 秒以下を実現し，99.9%のソースコードで 0–2 件の警告のみ出した．これらの機能は、日々の開発において静的バグチェッカーの利用を促し、統合フェーズを加速させるものと考える．

本論文は以下のように構成されている．2 章では、トリビアルバグ を定義することにより、我々の研究の動機付けを行う．3 章では、本論文の目標を説明する．4　から 5 章では，設計と実装を示す．6 章では、我々のアプローチの評価を行う．7 は関連研究の紹介を行い，8 で本論文の結論を述べる．

2. Motivation

2.1 Linux におけるトリビアルバグ

Linux の最近のバグ動向を把握するために、バージョン 5.9(2020 年 9 月リリース) からバージョン 5.11 (2021 年 2 月リリース) までのバグ修正パッチをサンプリングして、Linux のバグを調査した．*double free*, *out-of-bounds*, *integer overflow* などの単語を含む以下の 6 種類のバグに関するパッチを 117 件サンプルした: Use Before Initialization (UBI), Double Free (DF), Out of Bounds Access (OoB), Integer Overflow (INT), Null Pointer Dereference (NULL), Reference Counting Error (REF), and Permission Check (PC).

それぞれのパッチに対して静的解析の特性を用いて分類をした．具体的に以下の特性で分類をする:

(1) 解析に必要なスコープ．3 種類のスコープを対象とする．関数, 単一関数の解析で発見可能, ユニット, 単一コンパイルユニットの解析で発見可能, プロジェクト, プロジェクト全体の解析を行うことで発見可能.

特徴			パッチ数
サンプルしたパッチ			117
ターゲットのスコープ (関数 + プロジェクト)			77 (25 + 52)
詳細 (77 patches)	フィールド オフセット	なし * コンパイル時 * 実行時	36 34 7
	エイリアス	なし * 関数内 * 関数間	61 15 1
	コントロール フロー	直接 * 間接	69 8
トリビアルバグ			68

表 1: 調査結果．トリビアルバグは ∗ の特徴全てを満たすもの．

(2) フィールドオフセット計算のレベル.2 種類のオフセット計算について対象とする．コンパイル時, コンパイル時に静的にオフセット計算が可能, 実行時, 実行時に動的に計算する．オフセット計算を行う必要がない場合は無しと表記する．

(3) 必要なエイリアス解析のレベル.3 種類のレベルについて対象とする．無し エイリアス解析を必要としない, 関数内 関数内の解析で発見可能, 関数間 関数間の解析で発見可能．

(4) ダイレクトコントロールフローの解析のみで発見可能か. これは，関数の間接呼び出しなしで構成されているかを意味する．

表 1 にこれらの特性の調査結果を示す．117 件のパッチのうち，77 件が単一コンパイルユニットの解析で発見可能であった．内 25 件は関数内の解析で発見可能であった．これらのパッチに対して, 上記の特徴に基づいて詳細の調査を行った．

フィールドオフセット計算　コンパイルユニット内で発見可能な 77 件の内 7 件のみが動的な計算が必要であった．残り 70 件は構造体関連かハードコードされた配列の要素へのアクセスであった．

エイリアス解析　1 件のパッチのみ関数間のエイリアス解析が必要であった．残りのパッチは関数内の解析 (15 件) もしくはそもそもエイリアス解析が必要なかった (61 件)．

コントロールフロー　8 件のパッチのみ関数の間接呼び出しが関わっていた．残りの 69 件は全てダイレクトコントロールフローの解析で発見可能であった．

我々の調査の結果,117 件のパッチのうち，　６８件が単一のコンパイルユニットの解析で path-insensitive で，フィールド情報を考慮した関数間のデータフロー解析で発見可能であった．本論文ではこれらのバグを**トリビアルバグ**と表す．

2.1.1 Motivating Example

図 2 は実際に発見されたトリビアルバグの例である．

drivers/usb/host/oxu210hp-hcd.c

```
1  struct ehci_qh *oxu_qh_alloc(struct oxu_hcd *oxu) {
2    spin_lock(&oxu->mem_lock);
3
4    for (i = 0; i < QHEAD_NUM; i++)
5      if (!oxu->qh_used[i]) break;
6
7    if (i < QHEAD_NUM) {
8      ...;
9      qh->dummy = ehci_qtd_alloc(oxu);
10     if (qh->dummy == NULL)
11       goto unlock;
12     oxu->qh_used[i] = 1;
13   }
14 unlock:
15   spin_unlock(&oxu->mem_lock);
16   return qh;
17 }
18
19 struct ehci_qtd *ehci_qtd_alloc(struct oxu_hcd *oxu) {
20   spin_lock(&oxu->mem_lock); // Double Lock!!
21   ...;
22   spin_unlock(&oxu->mem_lock);
23 }
```

図 2: ドライバで発見されたダブルロックの例

手法	ツール	合計
静的解析	コンパイラ	6
	Clang Static Analyzer	1
	Coverity	1
動的解析	Syzkaller	11
	Abaci fuzz	1
未記載		48
合計		68

表 2: トリビアルバグを発見するのに使用された手法

この例はダブルロックを引き起こす例である．2 行目で spinlock `oxu->mem_lock` が獲得されている．その後に 9 行目で `ehci_qtd_alloc` が呼ばれ，同じロックが再度獲得される (20 行目). これによってダブルロックが発生する．この例は全ての関数が 1 ソースファイル内に収まっていてかつダイレクトコントロールフローで繋がっていて (9 行目), フィールド計算は全て構造体のメンバアクセス (2 行目，20 行目) である. また，エイリアス情報は必要ない．そのためトリビアルバグに特化した解析器の特性を満たす．

2.2 既存のツールとトリビアルバグ

既存のツールを用いてトリビアルバグを発見するのは容易である．しかし，我々の調査の結果が示唆するように多くのトリビアルバグが Linux に残っている．

実際にどれくらい既存のツールが使用されているかを理解するため, サンプルしたパッチの調査をさらに行った．既存のツールを使用して発見されたバグはその表記がパッチに行われる．例えば, Coccinelle [35] によって生成されたパッチは “Generated by” タグが存在する．これらのクレジットを使用して，使用された解析器を 3 つの種類に分類する: 静的解析 (e.g. コンパイラ警告, 静的解析器), 動的解析 (e.g. syzkaller [4], サニタイザ), 未表記.

表 2 に結果を示す．68 件のトリビアルバグのうち, 8 件

が静的解析を用いて発見された (コンパイラ警告で 6 件, Clang Static Analyzer [6] と Coverity [2] で 1 件ずつ). 12 件のバグは動的解析で発見された (11 件が syzkaller, 1 件が AbaciFuzz [47]). 残りの 48 件は未表記であった．これらの事実は解析器を十分に使用されていないことが示唆されている．これには 2 つの理由が考えられる:

長い解析時間. 既存のツールの多くはさまざまなバグを発見するため長い解析時間を有する．開発者は日々に開発においてこれらのツールが終わるのを待つことを避ける．PATA [31] は Linux の解析に 33 時間以上かかる．

過剰な量の警告 既存のツールは過剰な量の警告を生成する．これらの多くはプロジェクト全体に対してソースファイルをまたぐ解析を行う．そのため，バグの要因が開発者の編集したソースファイル外にあっても警告を生成する．また，開発者は誤検知を洗い出すためにこれらの警告を全て精査する必要がある．警告の数が多いため，開発者はこれらを精査する時間を十分に割くことができず，多くの警告が無視されている．

表 3 は既存のツールで `allyesconfig` の Linux(PATA は v5.6, その他は v5.15) を解析した際に有する解析時間と警告の数である．使用したツールは Clang Static Analyzer (CSA, v10.0.1) [6], Cppcheck (v1.90) [19], Coccinelle (v1.1.1) [35], Saber (v2.1) [49], そして PATA [31] である.CSA, CppCheck, そして Saber はデフォルトの設定で実行した．Coccinelle は Linux で提供されている NullPointer dereference のセマンティックパッチを使用した．PATA は一般向けに公開されていないため，著者らの論文で示されたデータを使用した [31]. 表 5 の環境で実行した.

PATA と Saber は関数間の path-sensitive なデータフロー解析や関数間のエイリアス解析などの複雑な解析を行う．解析時間については PATA は 33 時間以上の解析時間を必要とした．また，Saber はメモリ不足で解析が終了しなかった．警告の数については PATA は 627 件の警告を誤検知率 28%で出した．これらを全て精査するのに Ph.D の学生が 12 時間以上の時間をかける必要があった [31].

CSA も複雑な解析を行うツールである (path-sensitive なデータフロー解析). CSA は単一コンパイルユニットの解析を行うが,90%のソースファイルで 10.75 秒の解析時間を必要とする．また，5,207 ファイルに対して 10 件以上の警告を出力する.

PATA, Saber, CSA とは異なり CppCheck と Coccinelle は比較的簡単な解析を一つのコンパイルユニットに対して行う．CppCheck は path-insensitive なデータフロー解析をエイリアス情報なしで行う．そのため，90%のソースファイルで 0.32 秒の解析時間を有し,99.8% of the source files (25,345 files) で 5 件未満の警告数を出す．Coccinelle は元々はソースファイルを開発者指定のルールで変形させるツールであるが，特定のコードパターンを満たすかを検証することも可能である．しかし，関数間のデータフロー解析などは行わない [35], [39]．そのため,90%のソースファイルで 0.81 秒の解析時間がかかり,99.9% のソースファイル (25,473 files) で警告を出さなかった. 残念ながら，単純すぎる解析手法ではトリビアルバグを見逃してしまう可能性がある．例えば図 2 のようなバグは関数間のデータフロー解析が必要なため発見が難しい.

3. 目的

本稿の目的はトリビアルバグに特化した解析器の有用性を示すことである．これらにより開発者はのちの統合フェーズでトリビアルバグを気にする必要がなくなり，その後のテスト等を加速させることができる．トリビアルバグに特化した解析器は２つのメリットがある: 短い解析時間と少ない数の警告数である.

短い解析時間. トリビアルバグは教科書レベルの簡単な解析で発見可能である．これらに特化した解析器は時間の有する複雑な解析は行う必要がない．そのため，トリビアルバグに特化した解析器は解析時間を抑えることができる.

少ない警告数. トリビアルバグに特化した解析器は単一のコンパイルユニットの解析で発見できるため，出力する警告の数を開発者が変更したソースファイルの物のみに制限できる．これにより警告の数を抑えることができ，警告の原因は変更されたソースファイルのものになるため，開発者に警告を確認することを促進させる.

これらの特徴により，トリビアルバグに特化した解析器は日々の開発サイクルに対応ができる．トリビアルバグを開発の早い段階に除去することで統合フェーズでは開発者はより複雑なバグに集中できる．また，トリビアルバグに特化した解析器は既存のコンパイラなどに統合可能である．これによりトリビアルバグに特化した解析器はこれまでのビルドシステムに変更を加えることなく使用可能である.

トリビアルバグに特化した解析器は PATA [31] や CSA [6] などの既存のツールを補完する役割を持つ．トリビアルバグに特化した解析器がこれらのバグを発見していたら, その後のツールで出される警告の数が減り，開発者が一度に確認する必要のある警告の数も減る．これにより，より複雑なバグに集中することができる.

4. トリビアルバグの解析のためのフレームワーク

トリビアルバグに特化した解析器の有用性を示すため，簡単なトリビアルバグに特化した解析器のためのフレームワークの設計と実装を行う．プロジェクト固有なバグに対応するため，このフレームワークは開発者が発見するバグを Typestate 解析を用いて指定できるようにカスタマイズ可能にしている．我々のフレームワークはこれらのバグに

Tools		CSA	CppCheck	Coccinelle	Saber	PATA (元論文より引用 [31])	提案手法
ソースファイルあたりの解析時間 (秒)	合計	27 時間 1 分	2 時間 32 分	4 時間 20 分	メモリ不足	33 時間 1 分	2 時間 33 分
	50%tile	3.33	0.03	0.48	-	N/A	0.20
	90%tile	10.75	0.32	0.81	-	N/A	0.98
	99%tile	32.85	3.00	2.46	-	N/A	3.87
ソースファイルあたりの警告数 (ソースファイルの数)	合計	132,196	1,528	31	-	627	237
	0 件	317	24,479	25,473	-	N/A	20,473
	1 ~ 4 件	5,042	866	11	-	N/A	158
	5~ 件	12,454	53	2	-	N/A	3

表 3: 各ツールのソースファイルあたりの解析時間と警告数

ついてコンパイラプラグインを生成する．

本論文の目的はトリビアルバグに特化した解析器の日々の開発においての有用性示すことである．そのため，使用する解析の新規性を謳うものではなく，それぞれの解析は教科書レベルの解析にとどめている．

4.1 概要

我々の解析は typestate property 解析をもとにしており [31], [40], それぞれのバグを typestate property として有限状態遷移として定義する．本解析では状態遷移の受理状態をバグとする．

typestate の定義を用いてコントロールフロー解析を行い状態遷移を満たすものを探索する．本解析では path-insensitive でフィールド情報を考慮したな関数間の解析を行う．関数間の解析を効率的に行うため，関数の返り値を考慮した bottom-up summary-based 解析を行う．また，関数内の path-insensitive なエイリアス解析も行う．

4.2 Typestate Property の指定

本フレームワークははバグを typestate property で表す．これは各変数に対しての状態遷移である．遷移時のルールは 3 つのコンポーネントで構成されている: フック命令，オペランドの条件，そして対象としたオペランドである．フック命令は, LLVM IR [30] の形で表され, 状態遷移を引き起こす命令である．オペランド条件は, は状態遷移を引き起こすためのオペランドの条件である．これらの条件を満たしたときにのみ状態遷移が発生する．対象とするオペランドはフック命令に付随しているオペランドである．

4.3 CFG-based Typestate Analysis

定義された typestate property を用いて, path-insensitive でフィールド情報を考慮したな関数間の解析を行う．本解析では各ベーシックブロックを解析し，それぞれの変数がバグの状態を満たすかを確認する．

各ベーシックブロックについて，本解析ははじめにプレデセッサにおける各変数の状態を伝播する．もし全てのプレデセッサが同じ状態を持っていたら，その状態を伝播する．そうでなければ，伝播する状態を判定する必要がある．

伝播する状態を特定するため，本解析では各状態の優先度を指定し優先度の高いものをサクセッサに伝播する．デフォルトでは，バグの状態からの距離で優先度を決定する．距離が遠いものほど優先度を高く設定する．これはなるべく誤検知を減らすためである．優先度は開発者によってカスタマイズ可能である．状態を伝播したのちに，本解析ではベーシックブロックに含まれる各命令を確認し，状態遷移を引き起こすかを確認する．状態遷移の条件を満たしたら対応する変数の状態を更新する．

4.4 関数の要約情報を用いた関数間解析

本解析は各変数の状態を関数間の解析を用いて収集する．これらを効率的に集めるために，関数の要約を作成し，bottom-up な関数間解析を行う．これは，各関数を解析する際に，それぞれの引数と返り値の状態を集め，それらの最終的な状態を要約する．その関数への呼び出しが発生した場合はこれらの要約された情報を使用し，再解析をする手間を省く．

それぞれの引数の入力状態は実際の caller が決まるまでわからないため，本解析では引数は全ての状態を取りうると仮定し，各状態を起点とした遷移を記録する．これらをもとに対応する状態を caller へコピーする．もし関数がコンパイルユニット外に定義されていたら状態は不明とし，変数の状態を追うことをやめる．

4.4.1 返り値を考慮した状態の判定

関数はしばしば複数の return 文をもち，それぞれの場所ごとに異なる状態を持つ．また，caller はある地点で callee が終了することを期待することがある．例えば，caller のエラーハンドラは callee がエラーの状態で終了することを期待している．これらを無視することは，誤検知を発生させる要因になりかねない．

これらを考慮するため，本解析では各関数の要約を作る際，各返り値を考慮して作成する．本解析は整数の返り値は各関数の終了状態を示すことが多くあることを利用する．In Linux coding convention, a Linux では負の数をエラーのコードとし，正の数をサクセスコードとしている [34].

フックの名前	詳細
即時	状態遷移後即時 (デフォルト)
変数終わり	変数のライフタイム終了後
ブロック終わり	ベーシックブロックの解析終了後
関数終わり	関数の解析終了後
モジュール終わり	解析終了後

表 4: バグチェックのフック

本解析は各関数についてこれらの終了コードを収集する．特に変数の返り値はサクセスコードを表すことが往往にしてあるため，定数の返り値を収集する．もし変数の返り値が定数を含んでいたら　(use-def chain で探索), それぞれの定数が代入されたベーシックブロックの状態を集め，それぞれを返り値の状態とする．複数のベーシックブロックで同じ返り値が返されていたら，それらをひとまとめにする．

各関数呼び出しについて，本解析では caller が特定の終了コードを期待しているかどうかを確認する．これは条件分岐等で使用されているかどうかで判定する．もし使用されていたら，該当する状態をコピーする．複数の該当する終了コードが存在した場合はそれらをまとめて返す．もし返り値が確認されていなかったらサクセスコードが期待されていると考える．

4.5 警告出力のタイミング

収集した状態について，本解析はバグの状態を満たす変数について警告を出す．これらはデフォルトでは各状態遷移発生後に確認をするが，他のタイミングでも状態を確認することが考えられる．これらをサポートするために本解析では表 4 に示すタイミングでの確認も行えるようにする: 各ベーシックブロック (ブロック終わり), 関数 (関数終わり), 変数のライフタイム (変数終わり) の終わり, そして，解析終了時 (モジュール終わり) である．

5. 実装

我々のフレームワークは LLVM コンパイラフレームワーク [30] と Clang コンパイラを用いて実装した [1]. 4.1 章で述べたように、本フレームワークは各バグの型状態定義を受け取る. 状態と遷移規則を定義するために、C++インターフェースが提供される. これらから Clang プラグインを生成する. 開発者は、チェッカを有効にした Clang コンパイラでソースファイルをビルドすることで、チェッカを実行することができる。

フレームワークを用いて以下の 6 つのバグ検出器を実装した: Double Free (DF), Double Lock / Unlock (DL / DUL), Memory Leak (ML), Use After Free (UAF) and Ref Count Error (RC). すべてのチェッカーは 50 行以下の C++コードで実装されている．Coccinelle と CSA はそれぞれ 134 と 3,428 の LoC を必要とする．

OS	Ubuntu 20.04
CPU	16 Core Intel Xeon CPU E5-2620
RAM	96 GB (limited to 32 GB)
LLVM	10.0.1
Target Kernel	v5.15
Config	`allyesconfig`

表 5: 実験環境

バグの種類	警告数	正検知数
ダブルフリー	62	17
ダブルロック	15	1
ダブルアンロック	41	1
メモリリーク	21	12
ユーズアフタフリー	71	13
リファレンスカウンタ	21	1
合計	231	45

表 6: Number of generated warnings

6. 評価

実装した解析器を用いて，Linux Kernel v5.15 の解析を行った．　はじめに解析器で実際にバグを発見できるかを確認した．また，これらの解析時にかかった解析時間とそれぞれのソースファイルあたりの警告の数を測定した．表 5 が本解析の実験環境である．各開発者の日々の開発環境を模倣するため，使用できるメモリの数を 32GB と制限をしている．また，kernel の configuration として `allyesconfig` を用いて，Clang でコンパイルできないものを無効にしている. 我々の環境では 20,634 ファイルコンパイル可能であった. 各コンパイルはデバッグオプション (`-g`) を用いて行った.

6.1 解析結果

6.1.1 Number of bugs found

トリビアルバグに特化した解析器が実際にバグを発見できることを示すため，はじめに Linux のビルドを解析器を有効にして行った．表 6 は各バグごとの警告の数を示している．合計で 231 件の警告を出力した. 手動でそれぞれ確認したところ，45 件の警告が実際にバグであることが分かった．正検知率は 19.5%である．発見されたバグの中で double free が一番多く発見された (17 件). その後，use after free(13 件) と memory leak(12 件) が続いた. 他のバグは 1 件ずつ発見された．また 16 件のバグについてパッチを作成し，開発者に報告したところ 13 件が確認され，すでに修正されている．残りの 3 件は開発者の返答を待っている．

percentile	解析時間 (秒)
50%tile	0.20
90%tile	0.98
99%tile	3.87

表 7: ソースファイルあたりの解析時間

警告数	ソースファイルの数
1 件	123
2 件	23
3 件	9
4+件	6

表 8: ソースファイルあたりの警告数

6.1.2 ソースファイルあたりの測定結果

開発者の待ち時間を確認するため各ソースコードあたりに必要な解析時間を測定した．図 7 は解析時間のパーセンタイル値である．本解析器は 50%のソースファイルで 0.20 秒,90%のソースファイルで 0.98 秒の解析時間を有した．99%tile 値は 3.87 秒と,CSA に比べて 8.5 倍短く想定が終わっていた (表 3).

また，開発者が確認する必要のある警告を確認するため，各ソースファイルあたりに生成される警告の数も測定した．図 8 はこれらのヒストグラムである．99.9%のソースファイルは最大で 3 件の警告を出力した．3 つのソースファイルについては 5 件以上の警告を出力したが，これは同じ関数ないのバグ候補が他の関数に伝播したことによるものであり，確認するのは簡単であった．

6.2 誤検知の理由

主な誤検知の理由として挙げられるのは本解析で採用している簡単な解析である．4 章で示した通り, 本解析では教科書レベルの解析のみ行う．例えば，path-sensitive な解析などは行わない．これらは非常に効率的に解析が可能ではあるが，path の関係性等を考慮しないため，不正確な結果を導くことが往往にしてある．関数の要約情報を用いた解析や関数内のエイリアス情報なども誤検知に起因していると考えられる．また，それぞれのバグの解析器の状態遷移のルールなどもこれらの誤検知の一因として考えられる．

7. 関連研究

Linux におけるバグの調査　これまで様々な Linux のバグの調査に関する研究が行われてきた [16], [36]. Chou らはバージョン 1.0 から 2.4.1 のバグについての調査を静的解析を用いて行った [16]. Palix らも同様により新しいバージョン 2.6.0 から 2.6.33 [36] について調査を行った．これらでどの程度 Linux が変化したを確認するため，Chou らと同じ手法を用いて調査を行い，これまでと異なり driver 以外のコンポーネントにもバグが含まれることが分かった．

静的解析を用いた手法　これまで様々な Linux におけるバグを静的解析の手法で発見する研究が行われてきた [2], [3], [6], [7], [8], [9], [10], [11], [12], [13], [14], [17], [19], [23], [24], [29], [32], [33], [37], [41], [42], [43], [45], [46], [50], [51]. Linux は数千万行以上のコードで構成されているため，これらの膨大な行数にスケール可能でかつ高い精度を持つ手法を考える必要がある．これらの研究は Linux のセマンティックに依存した特定のバグパターンを対象にすることで Linux に適応できるようにしている．DCUAF [9] は use-after-free を,DSAC [13] はブロックしてはいけないコンテキストでの sleep を対象としている．

Coccinelle [35] は Linux で広く使用されているツールである．これは開発者の指定した “セマンティックパッチ” を用いて既存のコードを新しい API などに対応するために変形することが主な目的である．これらを応用し，特定のバグになりうるコードパターンなどを発見することができる．しかし，関数間のデータフロー解析等は行わない [35], [39].

PATA [31] はオペレーティングシステム向けのバグ解析フレームワークである．これらはスケール可能な手法を用いてかつ path-sensitive なエイリアス解析を用いているが，30 時間で解析を終了できる．

一般向けに様々なバグ検知ツールが提供されている．Saber [49] は様々なプロジェクトで発生し得る memory leaks を発見する．しかし，使用している手法が Linux 規模の大規模なコードを想定していないためスケールしない．Clang Static Analyzer [6] や CppCheck [19] は長い解析時間や多い警告の数が発生する．

動的解析を用いた手法　動的解析を用いた手法も存在する [4], [5], [15], [18], [21], [26], [27], [28], [38]. これらはバグを実行時の動作をモニタすることで発見することを試みる．Kmemleak [28] や KASAN [5] などのサニタイザは Linux に組み込まれている．これらはメモリが有効かどうかを shadow memory を用いて管理して不正なメモリアクセスを発見する．

Fuzzers は Linux において広く使用されている動的解析手法である．Syzkaller [4] は coverage-guided fuzzing を行い，自動でシステムコールを発行してなるべく多くのコードパスを走らせるようにする．Fuzzer は複雑なバグを低い誤検知率で発見できる一方，解析に非常に多くの時間がかかる上に質の高いテストケースをデザインする必要がある．

8. 結論

本稿では単一コンパイルユニットの簡単な解析で発見可能なバグであるトリビアルバグがまだ Linux に多く残っていることを示し，トリビアルバグに特化した解析器の有用性を示した．これらの存在は主に 2 つの理由でバグ解析器

は日々の開発で使用されていないことが示唆されている: 長い解析時間と数多くの警告の数である．トリビアルバグに特化した解析器はこれらを克服することができる: 時間のかかる複雑な解析を行わないことと開発者に関係のある警告のみ出すからである．トリビアルバグに特化した解析器の有用性を示すため，これらのためのフレームワークの設計と実装を行なった．教科書レベルの簡単な解析のみ行わないにも関わらず，我々の解析器は Linux 5.15 で 45 件のバグを発見する (13 件が開発者によって確認ずみ) ことができ，90%のソースファイルで 0.98 秒の解析時間がかかり，99.9%のソースファイルで最大で 2 件の警告を出した．

謝辞 本研究は、JST 次世代研究者挑戦的研究プログラム JPMJSP2123 の支援を受けたものである．

参考文献

[1] : Clang Compiler, `http://clang.llvm.org/`.
[2] : Coverity, `https://scan.coverity.com`.
[3] : Linux Driver Verification, `http://linuxtesting.org/ldv`.
[4] : Syzkaller: an unsupervised, coverage-guided kernel fuzzer, `https://github.com/google/syzkaller`.
[5] : The Kernel Address Sanitizer, `https://www.kernel.org/doc/html/latest/dev-tools/kasan.html`.
[6] : Clang Static Analyzer, `https://clang-analyzer.llvm.org/` (2022).
[7] : Facebook Infer: a tool to detect bugs in Java and C/C++/Objective-C code., `https://fbinfer.com/` (2022).
[8] : Smatch: a static bug-finding tool for C., `http://smatch.sourceforge.net/` (2022).
[9] Bai, J.-J., Lawall, J., Chen, Q.-L. and Hu, S.-M.: Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers, *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 255–268 (2019).
[10] Bai, J.-J., Lawall, J. and Hu, S.-M.: Effective Detection of Sleep-in-atomic-context Bugs in the Linux Kernel, *ACM Trans. Comput. Syst.*, Vol. 36, No. 4, pp. 1–30 (2020).
[11] Bai, J.-J., Lawall, J. and Hu, S.-M.: Effective Detection of Sleep-in-Atomic-Context Bugs in the Linux Kernel, *ACM Trans. Comput. Syst.*, Vol. 36, No. 4 (online), DOI: 10.1145/3381990 (2020).
[12] Bai, J.-J., Li, T. and Hu, S.-M.: {DLOS}: Effective Static Detection of Deadlocks in {OS} Kernels, *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 367–382 (2022).
[13] Bai, J.-J., Wang, Y.-P., Lawall, J. and Hu, S.-M.: DSAC: Effective Static Analysis of Sleep-in-Atomic-Context Bugs in Kernel Modules, *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 587–600 (2018).
[14] Ball, T., Bounimova, E., Kumar, R. and Levin, V.: SLAM2: Static driver verification with under 4% false alarms, *Formal Methods in Computer Aided Design*, ieeexplore.ieee.org, pp. 35–42 (2010).
[15] Cadar, C., Dunbar, D. and Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, USA, USENIX Association, p. 209–224 (2008).
[16] Chou, A., Yang, J., Chelf, B., Hallem, S. and Engler, D.: An Empirical Study of Operating Systems Errors, *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, New York, NY, USA, Association for Computing Machinery, p. 73–88 (online), DOI: 10.1145/502034.502042 (2001).
[17] Cong, K., Xie, F. and Lei, L.: Symbolic Execution of Virtual Devices, *Proceedings of the 2013 13th International Conference on Quality Software*, QSIC '13, USA, IEEE Computer Society, pp. 1–10 (2013).
[18] Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Kruegel, C. and Vigna, G.: DIFUZE: Interface Aware Fuzzing for Kernel Drivers, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, New York, NY, USA, Association for Computing Machinery, pp. 2123–2138 (2017).
[19] CPP Check: cloc, `https://cppcheck.sourceforge.io/` (2022).
[20] Das, M., Lerner, S. and Seigle, M.: ESP: path-sensitive program verification in polynomial time, *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation*, PLDI '02, New York, NY, USA, Association for Computing Machinery, pp. 57–68 (2002).
[21] Deligiannis, P., Donaldson, A. F. and Rakamaric, Z.: Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers (T), *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 166–177 (2015).
[22] Dhurjati, D., Das, M. and Yang, Y.: Path-Sensitive Dataflow Analysis with Iterative Refinement, *Static Analysis*, Springer Berlin Heidelberg, pp. 425–442 (2006).
[23] Emamdoost, N., wu, q., lu, k. and McCamant, S.: Detecting Kernel Memory Leaks in Specialized Modules with Ownership Reasoning, *The Network and Distributed System Security Symposium (NDSS) 2021*, (online), DOI: 10.14722/ndss.2021.24416 (2021).
[24] Fan, G., Wu, R., Shi, Q., Xiao, X., Zhou, J. and Zhang, C.: SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code, *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 72–82 (2019).
[25] Fink, S. J., Yahav, E., Dor, N., Ramalingam, G. and Geay, E.: Effective typestate verification in the presence of aliasing, *ACM Trans. Softw. Eng. Methodol.*, Vol. 17, No. 2, pp. 1–34 (2008).
[26] Hu, Y., Wang, W., Hunger, C., Wood, R., Khurshid, S. and Tiwari, M.: ACHyb: a hybrid analysis approach to detect kernel access control vulnerabilities, *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, New York, NY, USA, Association for Computing Machinery, pp. 316–327 (2021).
[27] Jiang, Z.-M., Bai, J.-J., Lawall, J. and Hu, S.-M.: Fuzzing Error Handling Code in Device Drivers Based on Software Fault Injection, *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 128–138 (2019).
[28] Kernel.org: Kernel Memory Leak Detctor(Kmemleak), `https://www.kernel.org/doc/html/v4.17/`

dev-tools/kmemleak.html (2019).

[29] Kuznetsov, V., Chipounov, V. and Candea, G.: Testing closed-source binary device drivers with DDT, *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, USA, USENIX Association, p. 12 (2010).

[30] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California (2004).

[31] Li, T., Bai, J.-J., Sui, Y. and Hu, S.-M.: Path-Sensitive and Alias-Aware Typestate Analysis for Detecting OS Bugs, *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, New York, NY, USA, Association for Computing Machinery, p. 859–872 (online), DOI: 10.1145/3503222.3507770 (2022).

[32] Lu, K., Pakki, A. and Wu, Q.: Detecting Missing-Check Bugs via Semantic-and Context-Aware Criticalness and Constraints Inferences, *USENIX Security Symposium (USENIX Security 19)* (2019).

[33] Mao, J., Chen, Y., Xiao, Q. and Shi, Y.: RID: Finding Reference Count Bugs with Inconsistent Path Pair Checking, *SIGARCH Comput. Archit. News*, Vol. 44, No. 2, pp. 531–544 (2016).

[34] Marinescu, P. D. and Candea, G.: Efficient Testing of Recovery Code Using Fault Injection, *ACM Trans. Comput. Syst.*, Vol. 29, No. 4 (online), DOI: 10.1145/2063509.2063511 (2011).

[35] Padioleau, Y., Lawall, J., Hansen, R. R. and Muller, G.: Documenting and automating collateral evolutions in linux device drivers, *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)* (2008).

[36] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J. and Muller, G.: Faults in Linux: Ten Years Later, *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, New York, NY, USA, ACM, pp. 305–318 (2011).

[37] Saha, S., Lozi, J., Thomas, G., Lawall, J. L. and Muller, G.: Hector: Detecting Resource-Release Omission Faults in error-handling code for systems software, *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12 (2013).

[38] Schumilo, S., Aschermann, C., Gawlik, R. and others: kAFL:Hardware-Assisted Feedback Fuzzing for OS Kernels, *26th USENIX Security* (2017).

[39] Serrano, L., Nguyen, V.-A., Thung, F., Jiang, L., Lo, D., Lawall, J. and Muller, G.: SPINFER: Inferring Semantic Patches for the Linux Kernel, *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, pp. 235–248 (online), available from ⟨https://www.usenix.org/conference/atc20/presentation/serrano⟩ (2020).

[40] Strom, R. E. and Yemini, S.: Typestate: A programming language concept for enhancing software reliability, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, pp. 157–171 (online), DOI: 10.1109/TSE.1986.6312929 (1986).

[41] Suzuki, K., Kubota, T. and Kono, K.: Detecting Struct Member-Related Memory Leaks Using Error Code Analysis in Linux Kernel, *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 329–335 (2020).

[42] Talebi, S. M. S., Yao, Z., Sani, A. A., Qian, Z. and others: Undo workarounds for kernel bugs, *30th USENIX Security* (2021).

[43] V Shakti D Shekar, B. M. and Varshapriya, M.: Device Driver Fault Simulation Using KEDR, *International Journal of Advanced Research in Computer Engineering and Technology*, p. 580–584 (2012).

[44] Wang, H., Xie, X., Li, Y., Wen, C., Li, Y., Liu, Y., Qin, S., Chen, H. and Sui, Y.: Typestate-guided fuzzer for discovering use-after-free vulnerabilities, *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, New York, NY, USA, Association for Computing Machinery, pp. 999–1010 (2020).

[45] Wang, X., Chen, H., Jia, Z., Zeldovich, N. and Kaashoek, M. F.: Improving Integer Security for Systems with KINT, *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, Hollywood, CA, USENIX Association, pp. 163–177 (online), available from ⟨https://www.usenix.org/conference/osdi12/technical-sessions/presentation/wang⟩ (2012).

[46] Wu, Q., Pakki, A., Emamdoost, N., Mc Camant, S. and Lu, K.: Understanding and detecting disordered error handling with precise function pairing, *30th USENIX Security Symposium* (2021).

[47] X., W.: io_uring: always let io_iopoll_complete() complete polled io, `https://github.com/torvalds/linux/commit/dad1b1242fd5717af18ae4ac9d12b9f65849e13a5` (2022).

[48] Xiao, X., Balakrishnan, G., Ivančić, F., Maeda, N., Gupta, A. and Chhetri, D.: ARC++: effective typestate and lifetime dependency analysis, pp. 116–126 (2014).

[49] Xie, Y. and Aiken, A.: Context- and Path-Sensitive Memory Leak Detection, *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, New York, NY, USA, Association for Computing Machinery, p. 115–125 (online), DOI: 10.1145/1081706.1081728 (2005).

[50] Zhai, Y., Hao, Y., Zhang, H., Wang, D., Song, C., Qian, Z., Lesani, M., Krishnamurthy, S. V. and Yu, P.: UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel, *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, New York, NY, USA, Association for Computing Machinery, pp. 221–232 (2020).

[51] Zhang, T., Shen, W., Lee, D., Jung, C., Azab, A. M. and Wang, R.: PeX: a permission check analysis framework for Linux kernel, *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1205–1220 (2019).