Vol.2014-OS-130 No.10
2014/7/28

Linux カーネルにおけるバグの実態調査

吉村　剛[1,a)]　河野　健二[1,b)]

概要：Linux カーネルにおいてバグ対策は不可欠である．しかし，コードの大規模化に伴い，バグ対策のために必要となるシステム全体の深い知識や誤りやすいパターンを把握することは難しい．特に誤りやすいパターンを把握するためには過去に大量に蓄積されたバグ報告やパッチの変更履歴を把握しなければならない．本研究は linux のパッチ 37 万件に対してパッチの説明文を自然言語処理し，トップダウンクラスタリングを用いることで全体の中でより高い頻度で発生した話題を抽出し，パッチ集合を 66 クラスタに分割してバグの実態調査に利用する．調査の有用性を示すため，割り込みに関するクラスタを調査してバグパターンを定義し，コード解析による検査を linux 3.15 において行って 2 件のバグを発見した．

1. はじめに

Linux カーネルは様々なデバイス上で動作するオペレーティングシステムとして社会的に広く使われるようになっており，高い信頼性が要求される場面が増えている．一方で信頼性を低下させる要因となるバグは様々な調査研究 [1] [2] [3] で指摘されておりバグの対策が不可欠となる．

開発段階におけるバグ対策手法のひとつはコード解析により発生しやすい誤りのパターンを検査することである．Windows のデバイスドライバ開発においては Static Driver Verifier [4] を用いることでドライバのカーネル API の使用方法の誤りを検査することができる．このようなバグを人為的に発見するためには， OS カーネルに対する深く正確な知識と，発生しやすい誤りを知るための多くの OS レベル開発をした経験が必要となる．また，誤りやすいパターンは実装に依存するため，バグ対策のためには特定のソフトウェア実装に特化した知識や経験が必要になることがバグ対策をさらに難しくしている．それに対し，一度コード解析の検査コードを高度な知識や経験を持つ開発者が実装すれば，その後は知識や経験がない開発者でもコードの誤りを自動的に検査することができる．

しかし，コード量が大規模になった linux の場合，どのようなバグを検査項目とするべきかを明確にすることは難しい．仕様は明確に定められていない部分が多いため，システム全体の動きを正確に理解することが難しいこと，大量に蓄積された誤りの報告を全てを把握することが難しいことが原因となる．特に誤りやすいパターンを知るためには世界中のユーザからの報告や，数千人規模の開発者により提供されるパッチの変更履歴をできる限り多く把握しなければならない．

Linux におけるバグの実態調査は様々な分類指標を元に行われている [5] [6] [3]．しかし，既存の調査の多くは特定のバグの種類の実態を明らかにすることを目的としているため，バグ対策で利用する場合カバーできるバグの種類が限られてしまう．

本研究では自然言語で記述されたパッチの説明文を解析することで， linux で過去報告されたバグの実態を示す．またパッチの集合をトップダウンクラスタリング [7] することで，全体の中でも特に発生しやすかったバグを優先的に抽出する．そして得られた結果を利用して，バグの検査を実装し，検査を行って実際に過去に発生しやすかったバグを発見することで方法の有効性を示す．この一連の流れを通して，既存の静的解析の手法の前提である，知識や経験を持つ開発者による検査の実装がソフトウェアが複雑化・大規模化した現在でも適用可能であることを示す．

本研究の方法のポイントは，大量に蓄積された過去のバグ報告から機械学習や情報検索の典型的な手法を利用して発生頻度の高い誤りを抽出することで，より効果的な検査を可能にすることである．自然言語の解析には並列分散処理環境で動作する機械学習ライブラリである Apache Mahout で実装された Latent Dirichlet Allocation (LDA) [8] を利用し，linux 2.6.12 から 3.12 のパッチ 37 万件に書かれた文書が共通して持つ話題を抽出する．その結果を元に，最終的に文書の内容が類似したパッチが集まった 66 のクラスタに分ける．

ケーススタディとして，割り込みに関する内容を持つク

[1] 慶應義塾大学
[a)] yos@sslab.ics.keio.ac.jp
[b)] kono@ics.keio.ac.jp

ラスタに注目し，さらに ”free” という単語が強く表れた内容を持つパッチ 331 件をマニュアルで分析する．その結果，非決定的な障害を引き起こす恐れのある割り込みリクエスト (IRQ) の登録・解除の API 使用誤り 9 パターンが得られた．パターンのうち多くの誤りについては linux のソースのコメント内でも言及されているものの，明確に記載されていないパターンも観察されている．最後にこのバグパターンと同様のバグパターンがないか linux 3.15 において検査し，実際に 2 件のバグを発見した．

2 節では自然言語処理及びクラスタリング手法を述べる．3 節で linux の 37 万件のパッチを利用したバグの実態調査の結果を示す．4 節はその結果を受けて，実際にバグの検査を実装した事例を紹介し，5 節でその結果を述べる．6 節は過去多数行われてきたバグの実態調査やバグの検査手法と比較し，本研究の立ち位置を示す．最後に 7 節で本論文をまとめ，今後の展望を示す．

2. 方法

本節では，始めに調査の対象となる linux のパッチの特徴を示す．そしてその特徴をどのように解析・クラスタリングし，最後にパッチの集合として得られたクラスタからどのようにバグの実態を抽出するかを示す．

2.1 対象とするパッチ

より実態を反映した調査をするためには対象となるパッチの性質が重要となる．本研究は linux のバージョン管理システムである git の 1 コミットを 1 つのパッチと見なし，メインラインツリーの最初 (バージョン 2.6.12-rc2) から，2013 年 10 月 (バージョン 3.12-rc5) までのマージコミットを除いた 370,403 件を対象とする．安定バージョンのツリーは多くがメインラインツリーからバックポートされたコミットが占め，重複した内容になるため対象外とする．

パッチは大きく分けてコードの変更部分となる C 言語部と，そのコードの変更がなぜ必要か，どのように変更したのかを説明した自然言語部に分けられる．C 言語部は実際に過去発生した問題をどのように修正するかを知ることはできるものの，過去発生した問題が何かを直接的に示さない．一方で自然言語部は実際にどのような実行パスを経て問題が起きたかを明確に書かれることは多くないものの，過去に実際に発生した問題の大まかな記述が含まれる．本研究では自然言語部を利用してパッチをクラスタリングし，得られたクラスタからバグのパターンを分析するときに C 言語部を利用する．

linux のパッチの自然言語部は英語で書かれるため，本研究は英語に対象を絞って自然言語処理をする．英語を解析対象とする場合の問題，linux のパッチ特有の問題などについては次の自然言語処理の方法の説明と合わせて述べる．自然言語処理の精度を高めるためには解析の目的や対象とする文書集合に強く依存したヒューリスティックを多用する必要がある．

2.2 自然言語処理

本研究はパッチの説明文を単語の集合 (bag of words) とみなし，単語の出現パターンからパッチの特徴量を算出する．単語の区切りはスペースとするものの，自然言語の性質上そこで得られた単語の集合をそのまま解析するとノイズが多く解析の精度が悪くなる恐れがある．また Latent Dirichlet Allocation (LDA) を利用し，多数出現する類語により発生する問題を回避する．

最初に解析におけるノイズを減らすため，解析に不必要な単語を除去する．英語の動詞活用や，複数形を単数形に置換する（ステミング）．次に得られた単語で高頻度で出現するのに重要な意味を持たない単語 (‘is’, ‘a’, ‘that’ などのストップワード) を除外する．他にも，得られた単語の中でも 10 進数数値や 16 進数値 (‘8’, ‘0xff’, ’1e’ など) のみから成る単語や，“Signed-off-by:” から始まるパッチの署名に関する段落は本研究の目的にとって有用な情報はほとんど得られないため除外した．

次にパッチからより有用な情報を取り出すため，単語の除去だけでなく単語の分割も行う．Linux のパッチの自然言語部はソースコードで出現する関数名を使用して説明することが多い．そして linux の関数命名規約は直感的で第三者が読んでもその関数が何をする関数なのかがはっきりわかることが多い．例えば ‘fat_alloc_inode()’ や ‘ext3_alloc_inode()’ などのようにそれぞれ FAT や EXT3 の inode の割当てに関する関数であることがわかる．分割する上では ‘_’ をスペースに変換すると，どのサブシステムでどの機能に対して何をしたのかを含めた解析が可能になる．この分割をしない場合，‘fat_alloc_inode()’ という 1 つの単語としてみなされるため，解析すると出現頻度が極めて低い重要でない単語として扱われてしまう一方，分割をすれば一般的な割り当てに関する問題や，特定のサブシステムの機能に関する問題などを取り出せる可能性がある．この他にも記号は単語の区切りであるスペースに変換することで，linux のパッチにおいてより重要になりやすい単語を取り出すようにしている．

最後に，単語としては異なるものの同義となる語が多数出現する問題に対処する必要がある．これまで述べた方法で得られた単語をそのまま文書の類似度の比較に使うと，類語が全て異なる意味の単語として扱われてしまう．例えば linux のパッチにおいて ‘crash’ と，‘failure’ という単語はほとんど同じ意味で使われ，‘leak’ と ‘crash’ は異なる意味で使われるものの，これら 1 単語の違いが全て同じ重み付けとして特徴量に反映されてしまう．しかし，‘crash’ と ‘failure’ が同じで ‘leak’ とは異なるといった類語の情報

$$p(D|\alpha,\beta) = \prod_{d=1}^{M} \int p(\theta_d|\alpha)(\prod_{n=1}^{N_d} \sum_{z_{dn}} p(z_{dn}|\theta_d)p(w_{dn}|z_{dn},\beta))d\theta_d \quad (1)$$

$$p(\theta,z|w,\alpha,\beta) = \frac{p(w|\alpha,\beta)}{p(\theta,z,w|\alpha,\beta)}, \quad p(w|\alpha,\beta) = \int p(\theta|\alpha)(\prod_{n=1}^{N} \sum_{z_n} p(z_n|\theta)p(w_n|z_n,\beta)d\theta \quad (2)$$

図 1 LDA の文書モデルと推論

を全て人間が与えることは不可能である．

その問題に対し，LDA [8] を用いると文書に含まれるトピックを元に文書の類似度を比較することが可能になる．トピックは文書集合に出現する全ての単語の確率分布で表され，文書全体集合 D を図 1 (1) の式をモデルとする．LDA は文書の背景には複数のトピックが混在していると仮定し，その混合率に従って文書に含まれる単語の背景トピックが生成され，その単語の背景トピックに従って単語が生成されるとモデル化している．トピックの混在率 θ はディリクレ分布（パラメータ α），文書内の N 個の単語 w_n に対して，トピックの生成確率 z_n はトピックの混在率をパラメターとする多項分布，単語 w_n の生成確率 $(p(w_n|z_n,\beta))$ はパラメータ β の多項分布に従う z_n に対する条件付き確率を仮定する．なおトピック数は最初に与える必要があるため，本研究では 500 トピックを与える．

そして (1) からトピックの確率分布を推論する (図 1 の (2))．このままでは計算できないため，実際には近似を用いて計算し，最終的に最適なパラメータ α, β を求め，最適なトピックの単語による生起確率分布を得る．具体的な計算方法は割愛する．その過程で文書のトピックの生起確率分布が得られるため，その結果を本研究では文書の類似度の計算に用いる．

Linux のパッチ文書に対して LDA を適用する場合，文書の構造が考慮されない点が精度を低下させる恐れがある．自然言語部にはコードの変更の説明以外にも，その説明をより詳細にするためのエラーログ，障害を再現するコード，ユーザが障害報告した URL へのリファレンスなど様々な定型文が存在する．このような文書はバグの実態を示すことはあまりなく，似たような文書になりやすいため特徴量として強く出るノイズとなる恐れがある．しかし機械的にフィルタリングできるほど固まった定型文は定められていな．また一方でデッドロックの警告などのようなバグとして特徴付けるべきパターンも考えられる．

上記の問題を解決する簡単な方法として文書を段落で分割して段落を 1 つの文書と見なして LDA を適用し，次節で説明するクラスタリングの段階でパッチが複数のクラスタに属するようにする．段落で分割する理由はエラーログなどのノイズとなる文書は基本的に段落で区切られているという我々の観察結果をもとにしている．結果の節で示すように，段落の分割は bugzilla のリファレンスのクラスタを他のクラスタと独立して取り出すことを可能とし，一方で定型文になりやすいデッドロックの警告を集めることで並行処理に関するバグのクラスタを得ることを可能にしている．段落に分割すると最終的には約 100 万件の文書を LDA で処理することになる．LDA の最適なパラメータ計算のための反復は 1000 回，その他のスムーシングパラメタは Mahout 推奨の値を利用する．

2.3 トップダウンクラスタリング

次に，全体から俯瞰して特徴量がより類似したパッチの集合を決定する必要がある．LDA も文書をトピックに所属するかどうかを確率 (0 から 1 の間の値）で算出するため，ソフトクラスタリングする手法とみなすことも可能であるものの，その値から直接パッチ同士が類似しているかどうかを判断するのは簡単ではない．例えばパッチ A がトピック 1 とトピック 2 に対して 0.5 ずつの確率で属する場合と，パッチ B がトピック 1 とトピック 2 に対して順に 0.6, 0.4 の確率で属する場合，パッチ A と B が類似しているかどうかは文書集合全体の傾向を考慮しなければ判別できない．

本研究は分割型階層的クラスタリング（トップダウンクラスタリング） [7] でトピックに属する確率分布が類似した文書が集まったクラスタを計算する．凝縮型の階層的クラスタリング（ボトムアップクラスタリング）のほうが使われるケースが多いものの，本研究は厳密にクラスタを決定する必要性はないため，より高速で並列計算のしやすいアルゴリズムを選択した．

トップダウンクラスタリングのサブアルゴリズムとして 2-means を利用する．最初は全てのパッチを 1 つのクラスタとみなし，2-means で 2 分割し，分割して得られたクラスタをさらに 2 分割し，それをクラスタに属するパッチが 5000 件以下になるまで再帰的に分割する．実際の入力は段落であるため，段落が属するパッチも段落と同じクラスタに属すると考えて終了判定する．例えばあるパッチが段落を 3 つ持ち，クラスタが順にクラスタ A，クラスタ B，クラスタ A, に属する場合，そのパッチはクラスタ A, B に属すると考える．クラスタリングで利用する特徴量は文書のトピックの 500 次元の確率ベクトルとなり，類似度はユークリッド距離で計算する．

3. 結果：バグの実態

本節では自然言語処理およびクラスタリングを用いて得

られたバグの実態を示す．また，特定のクラスタについて詳細に調査した結果も示す．

3.1 結果概要

37 万件のパッチを階層的クラスタリングした結果，クラスタは最初の 37 万件の全てのパッチを含むクラスタから 1 件のパッチを含むクラスタまで様々なクラスタが得られる．その中から，最終的に 5,000 から 10,000 件のパッチを含む代表クラスタを抽出する．5,000 件よりも所属パッチが少ない比較的少ないクラスタは相対的に影響が小さいため本研究では無視する．また再帰的にクラスタリングするため，クラスタの中でも親子関係にあるクラスタが両方とも 5,000 から 10,000 のパッチを含む場合がある．その場合は子クラスタを選び代表クラスタとする．

表 1 がクラスタリングで得られた代表クラスタ 66 件を示している．前節で説明したように，文書は潜在的トピックの確率分布に変換される．本研究では 500 トピックを与えて計算するため 500 次元の確率ベクトルになる．表の重心はクラスタに属する段落のベクトルの重心を取り，その中で最も確率の大きさが多い順に並べたものである．またトピックは文書の全体集合で出現する単語の確率で示されるため，それについても確率の多い順に並べて “(単語, 単語..)” で表している．単語はステミングされており，名詞の複数系を単数に合わせたり動詞を現在形に合わせるために単語の末尾の ‘s’ や ‘e’ が抜けていたり，‘y’ が ‘i’ に置換されている．

トピック及びトピックを表す単語の組はおおよそパッチの内容の要約となっている．そのため，重心トピックを表現する上位単語はそのまま linux のパッチの集合を特徴付けている単語となる．全体を俯瞰した結果を述べると，まずサブシステムやデバイスドライバ名を示すクラスタ (#32: arm, #38: drm, #45: usb など) が多く，開発が盛んに行われていてコミット数が多いコンポーネントがクラスタとして出現しやすいことがわかる．また，一般的に知られたバグに関連する単語のクラスタも一部見られている (#30: null, #60: memory, #63: lock)．カーネルの一般的な機能に関する単語のクラスタ (#16: irq, #22: page など) はその機能についてのミスに関する言及が多くみられる．その他にもパッチ特有の言い回しを表すクラスタ (#24: http, #27: thank, #64: comment など) が見られる．

これらのクラスタはあくまでパッチを自然言語部の記述を利用してクラスタリングしたもので，バグを正確に集めているわけではない．しかし，実際にクラスタに含まれるパッチを観察した限りある一定の割合でバグとそれ以外の機能追加などのパッチが含まれている．この点は後述のクラスタの精度の評価と実際に #16 を調査した結果の節で検討する．

次に，表 2 はクラスタに含まれるパッチとその特徴量として現れる上位 3 トピックの例である．ここでは #32: arm, #30: null #22: page に絞り，重心にユークリッド距離で最も近い文書をそれぞれ示したものである．前述のように，トピック及びトピックを表す単語の組はおおよそパッチの内容の要約となっていることがわかる．#32 は arm に特有の問題について言及された文書で，#30 は短いものの，Null チェックを追加するパッチであることが予想される．#22 は文書ではなくページの異常を報告するクラッシュログで，パッチ全体もこのログに関連する内容であることが予想される．ただし，あくまで重心である

表 2 クラスタごとの重心に近い文書

クラスタ #32: arm, commit 9cff337 3 段落目 トピック: (arm, mach), (watchdog, nmi), (specif, code, bank)
So far as I am aware this problem is ARM specific, because only ARM supports software change of the CPU (memory system) byte sex, however the partition table parsing is in generic MTD code. The patch below has been tested on NSLU2 (an IXP4XX based system) with a patch, 10-ixp4xx-copy-from.patch (submitted to linux-arm-kernel - it's ARM specific) required to make the maps/ixp4xx.c driver work with an LE kernel.
クラスタ #30: null, commit a61cc44 1 段落目 トピック: (null, derefer), (free, descriptor), (code, abl)
[CRYPTO] Add null short circuit to crypto_free_tfm
クラスタ #22: page, commit 3ac19f8 3 段落目 トピック: (page, insert), (client, layer), (cpu, hotplug)
init_memory_mapping: [mem 0x80000000-0x9fffffff] Built 2 zonelists in Node order, mobility grouping on. Policy zone: Normal BUG: Bad page state in process bash pfn:9b6dc page:ffffea0002200020 count:0 mapcount:0 mapping: (null) page flags: 0x2fdfdfdfd5df9fd (locked|referenced| Modules linked in: netconsole acpiphp pci_hotplug Pid: 988, comm: bash Not tainted 3.6.0-rc7-guest #12 Call Trace: [<ffffffff810e9b30>] ? bad_page+0xb0/0x100 [<ffffffff810ea4c3>] ? free_pages_prepare+0xb3/0x100 [<ffffffff810ea668>] ? free_hot_cold_page+0x48/0x1a0 [<ffffffff8112cc08>] ? online_pages_range+0x68/0xa0 [<ffffffff8112cba0>] ? __online_page_increment_counters [<ffffffff81045561>] ? walk_system_ram_range+0x101/0x110 [<ffffffff814c4f95>] ? online_pages+0x1a5/0x2b0 [<ffffffff8135663d>] ? __memory_block_change_state0 [<ffffffff81356756>] ? store_mem_state+0xb6/0xf0 [<ffffffff8119e482>] ? sysfs_write_file+0xd2/0x160 [<ffffffff8113769a>] ? vfs_write+0xaa/0x160 [<ffffffff81137977>] ? sys_write+0x47/0x90 [<ffffffff814e2f25>] ? async_page_fault+0x25/0x30 [<ffffffff814ea239>] ? system_call_fastpath+0x16/0x1b Disabling lock debugging due to kernel taintt (枠からはみ出す分は削除している)

表 1 得られた代表クラスタ
左から，クラスタ番号（括弧内は所属パッチ数），クラスタの所属パッチの重心を表すトピックの確率上位 3 トピックを表したもので，そのトピックを表す確率の高い単語上位最大 2 単語を示している．最上位の単語がトピックを表す確率が 0.9 を超えるものは 1 単語のみ表示している．

クラスタ	重心の上位 3 トピック	クラスタ	重心の上位 3 トピック
#01 (5177)	(build, defconfig), (powerpc, undefin), (updat)	#34 (7238)	(stage, comedi), (ieee80211, dead), (macro, inlin)
#02 (5557)	(net, linu), (ocf2, truncat), (convert, appropri)	#35 (5909)	(flow, ethtool), (tx, rx), (net, linu)
#03 (5513)	(perf, counter), (event, pars), (top, util)	#36 (5062)	(media, video), (v4l2, sensor), (info, displai)
#04 (5777)	(sh, migrat), (info, displai), (support)	#37 (7515)	(variabl, unus), (remov, useless), (remov, redund)
#05 (7018)	(dvb, v4l), (media, video), (v4l2, sensor)	#38 (5025)	(drm, radeon), (auto, engin), (i915, pipe)
#06 (5044)	(com, sign), (suggest, date), (thank, cc)	#39 (5179)	(cleanup, minor), (stage, comedi), (comment, typo)
#07 (5436)	(scsi, fc), (save, sa), (length, sg)	#40 (5167)	(ap, beacon), (frame, mac80211), (hw, ath9k)
#08 (5498)	(socket, y), (addr, ipv6), (net, linu)	#41 (6493)	(quirk, laptop), (report, hid), (input, touch)
#09 (5745)	(debug, messag), (level, format), (print, us)	#42 (5238)	(powerpc, undefin), (build, defconfig), (arch, blackfin)
#10 (7763)	(timer, idl), (watchdog, nmi), (cpu, cpumask)	#43 (6405)	(actual, yet), (system, sleep), (avoid, correctli)
#11 (5783)	(warn, spars), (warn, len), (found, checkpatch)	#44 (5002)	(ata, libata), (id, cd), (mode, network)
#12 (5804)	(nf, netfilt), (addr, ipv6), (ip, b)	#45 (5135)	(usb, gadget), (cpufreq, frequenc), (incorrect, sound)
#13 (5385)	(crypto, bss), (o, drive), (text, p)	#46 (8731)	(master, slave), (frame, mac80211), (nf, netfilt)
#14 (5005)	(tx, rx), (queue, blk), (packet, skb)	#47 (5348)	(pci, slot), (bu, driver), (cmd, pcie)
#15 (5790)	(soc, asoc), (detect, pin), (imx, fsl)	#48 (5187)	(test), (null, derefer), (compil, posit)
#16 (5334)	(irq), (interrupt, msi), (handler, c)	#49 (5211)	(fs, uml), (declar, static), (symbol, rout)
#17 (7843)	(pm, resum), (serial, consol), (entri, maintain)	#50 (5407)	(need, longer), (case, effect), (sinc, shouldn)
#18 (5499)	(valu, wrong), (loop, valu), (defin, magic)	#51 (6318)	(alloc), (memori, leak), (slab, kmalloc)
#19 (6065)	(chang), (patch), (patch)	#52 (5230)	(replac, simpl), (implement, similar), (method, us)
#20 (7814)	(pass, structur), (rang, signal), (name)	#53 (7934)	(modul, pnp), (delet, elimin), (info, displai)
#21 (5514)	(dma, channel), (map), (id, cd)	#54 (5222)	(select, kconfig), (gpio, restor), (config)
#22 (8078)	(page, insert), (map), (scan, direct)	#55 (5161)	(time, second), (first, multipl), (time, offset)
#23 (5133)	(bit), (mask, bit), (clear, thu)	#56 (7397)	(mai, due), (even, confus), (much, less)
#24 (7021)	(http, org), (bug, show), (id, cd)	#57 (6252)	(code), (clean, code), (code, duplic)
#25 (8243)	(write), (complet, abort), (caus, race)	#58 (5152)	(gcc, git), (like, low), (version, increment)
#26 (5195)	(perform, optim), (wai, want), (see, affect)	#59 (5186)	(s390, facil), (dirti, nr), (page, insert)
#27 (8921)	(thank, cc), (manag, appli), (miss, add)	#60 (5296)	(memori, leak), (cpu, hotplug), (node, numa)
#28 (8047)	(drop, sysf), (file), (header, file)	#61 (5314)	(limit, increas), (number, calcul), (byte, cycl)
#29 (7029)	(bio, segment), (mark, receiv), (read)	#62 (5733)	(block, transact), (queue, blk), (length, sg)
#30 (6955)	(null, derefer), (pointer, cast), (close, cap)	#63 (5697)	(lock, unlock), (lock, poll), (lock, protect)
#31 (5426)	(smp, kill), (issu, fix), (code)	#64 (5387)	(comment, typo), (document, txt), (cleanup, minor)
#32 (5331)	(arm, mach), (h, asm), (omap, omap2)	#65 (5005)	(x86, iommu), (pci, slot), (max, min)
#33 (6276)	(h, asm), (h, constant), (includ)	#66 (8212)	(updat), (scsi, fc), (document, txt)

ため必ずしもクラスタに属するパッチ全てが類似した内容になっているのではないことに注意する必要がある．実際 #30, #22 の重心から最も遠い文書のトピックは重心とは異なり，従って内容も null や page に関係のないパッチであった．この点に関しても次のクラスタリングの精度の節で検討する．

まとめ： 全体から見て linux は開発が盛んなデバイスドライバにおけるバグ (#32: arm, #38: drm, #45: usb など) が多く，また，一般的に知られたバグも一部見られる (#30: null, #60: memory, #63: lock)．そしてカーネルの一般的な機能に関するバグ (#16: irq, #22: page など) も多くみられる．

3.2 クラスタリングの精度

次にクラスタリングの精度を示すため，grep で既知のバグを調査する場合と比較する．null などのように一般に広く知られているバグの場合，“null” で検索し，パッチの対象を絞るほうが目的のバグを得る精度は高くなると予想され，その精度にどこまで近づけているかが問題となる．しかし精度を直接比較するためにはクラスタに所属するパッチと，grep で得られたパッチ集合全てをマニュアルで調べる必要があるものの，それは現実的でない．そこで近似として既存の Lu らのバグ調査結果を正解集合と仮定し，その一致率を調べることで間接的に精度を比較する．

Lu らは linux 2.6 のいくつかのファイルシステムを対象とした調査を行っており，その他として分類されたパッチを除いた対象パッチは我々の対象と 1947 件が重なっ

表 3　grep との精度比較 (memory bug)

クラスタ	TP	FP	FN	Precision	Recall	F 値
null	13	14	140	0.48	0.08	0.14
leak	26	7	127	0.79	0.17	0.28
corrupt	3	7	150	0.30	0.02	0.04
#60	27	24	126	0.53	0.18	0.26
#30	12	25	141	0.32	0.08	0.13
#24	12	63	141	0.16	0.08	0.11

ている．Lu らはいくつかの分類軸でパッチを分類をしているものの，比較をわかりやすくするためにバグの根本原因の指標 (semantic, concurrency, memory, error code, not-bug) で分類した結果と比較する．semantic はファイルシステムのアルゴリズムの問題や I/O 要求の順序などが含まれ，concurrency は atomicity violation, order violation, deadlock, missing/double/wrong lock が含まれる．memory は resource leak, null dereference, dangling pointer, uninitialized read, double free, buffer overrun が含まれる．error code はエラーチェック忘れやリターン値の誤りが含まれる．not-bug はパフォーマンス障害やセキュリティに関するパッチ，機能の追加やコードやドキュメントのメンテナンスが含まれる．

最初にメモリバグを抽出しようとする状況での検索精度を調査する．比較はあるクラスタについて， memory bug とタグ付けされたコミットを含むなら true positive (TP)，それ以外にタグ付けされたコミットを含む場合 false positive (FP) とする．また， memory bug とタグ付けされたのにクラスタにコミットが含まれない場合を false negative (FN) として Precision, Recall, F-measure を測定する．比較対象はステミング後のパッチに対する単語の一致検索で，“null” and “derefer”, “memori” and “leak”, “memori” and “corrupt” のキーワードで持つ 3 つのパッチ集合 (順に null, leak, corrupt と呼ぶ) と比較する．Lu らの調査対象になっている 1947 件のパッチ以外は正解も不正解にもせず，無視する．

比較結果は表 3 で，66 件のクラスタのうち F 値が上位 3 件のクラスタを抽出している．Lu らの結果で memory に属するパッチは全部で 153 件であるため，TP から 153 引いた値が FN となる．null は 2315 件，leak は 1927 件，corrupt は 678 件のパッチが得られている．#60 と比較して Recall が低く， Precision も考慮した結果である F 値は leak のみ上回っているものの，機械的にクラスタリングした結果を大きく上回っているわけではない．また，#60, #30 の重心のトピックが (memori, leak), (null, derefer) が含まれることから，トピックがそのままクラスタの特徴を反映していることもわかる．#24 のトピックは (http, org) が含まれていることから， Bugzilla などへのリンクが多く含まれていることが予想され，その結果 Lu らの memory bug との一致率が高くなったと考えられる．ただし，FP が TP を大幅に上回っていることからもわかるように，memory bug 以外のバグも多く含んでしまっている．そのため memory bug を持つパッチに類似した性質を持つパッチを調べようとする場合は #24 や #30 は避け， #60 を選ぶべきである．

表 4　grep との精度比較 (concurrency bug)

クラスタ	TP	FP	FN	Precision	Recall	F 値
race	75	28	180	0.73	0.29	0.42
#63	67	54	188	0.55	0.26	0.36
#62	58	358	197	0.14	0.23	0.17
#51	25	116	230	0.18	0.10	0.13

次はメモリバグと同様に，並行処理に関するバグを抽出しようとする状況での検索精度を調査する．Lu らの指標では concurrency bug として分類されている．“race” という単語でステミング後のパッチを検索して得られたパッチ集合と比較する．

比較結果は表 4 で，66 件のクラスタのうち F 値が上位 3 件のクラスタを抽出している．Lu らの結果で concurrency に属するパッチは全部で 255 件であるため，TP から 255 引いた値が FN となる．grep で得られたパッチは全部で 4702 件でそのうち TP は 75 件，クラスタで最も良い精度となった #63 は 67 件であった．一方，FP は grep が 28 件， #63 は 54 件と TP, FP ともに grep のほうが高精度となった．しかし，memory bug の結果と同様，精度が大きく上回る結果にはなっていない．従って，よく知られたバグを表現する単語の検索で得たパッチ集合を調べる方法とほとんど変わらない精度で，全体から見て発生しやすいバグが抽出されることがわかる．

まとめ：評価結果から，クラスタリングを利用すると過去に渡って相対的に高頻度で linux において話題になってきたバグをまとめて抽出することができることがわかる．また，トピックを構成する単語からクラスタに含まれるバグのおおよその概要を得ることができるため対象クラスタを絞り調査の効率化が図りやすいという特徴もある．そして相対的に単語の検索した場合とも精度は変わらずにバグに関連するパッチが得られる．とはいえ，絶対的な精度としてあまり高いとは言えない．

これらの特徴から，クラスタリングは過去発生しやすかったバグと類似したバグを探索するためのヒントとして利用しやすいことが予想される．そこで次に実際に #16 のクラスタに注目した調査をすることで，割り込みのような OS 特有の機能を利用した時に発生する問題についてさらに掘り下げた調査を行う．

3.3　クラスタの分析

本節では得られたクラスタのうち, #16: (irq), (interrupt, msi), (handler, c) を用いて割り込みに関するバグを調査する．Lu らの指標で上がった memory bug や concurrency

表 5 #16 のクラスタに含まれるトピック上位 10 件
トピックはこれまでと同様に確率
上位 2 単語で示し，“:” 以降の数値はトピックを持つパッチの数を示す．

(disabl, stat):558, (fix):479, (add):454, (chip, speed):442, (gpio, restor):399, (handl):398, (patch):381, (regul, suppli):359, (free, descriptor):331, (enabl, wakeup):329

表 6 #16 のバグ調査結果

バグの詳細	数
free_irq() with inconsistent dev_id	41
missing free_irq() (initialization error)	25
free_irq() with an invalid irq number	25
missing free_irq() (module unload)	13
double free_irq()	9
releasing other src before free_irq()	7
releasing pages with interrupt disabled	7
missing free_irq() before device suspend	6
freeing shraed irq with interrupt disabled	5
other	22
not bug	171

bug と異なり，割り込み処理に関するバグは OS やデバイスドライバのプログラミングでのみ発生する問題になる．37 万件のパッチクラスタリングで得られたひとつのクラスタの重心として出現する以上，linux において主要な誤りのひとつである可能性は高く，対策する価値があると考えられる．

どのようなバグが含まれるか不明な状況で 5000 件以上のクラスタをそのまま調査することは困難であるため，対象のパッチをさらに絞り込むためにトピックを利用する．表 5 は重心に出現したトピックを除いた，クラスタに含まれるパッチが持つ最も多いトピック 10 件になる．計測方法は #16 に所属するパッチがそれぞれもつ確率が最も高いトピック 10 件をパッチのトピックと考え，そのトピックの数を数えた結果になる．同一パッチに属する段落がある場合は，より重心に近い段落をそのパッチのトピックと考える．この中で，331 件と適度な数である (free, descriptor) を持つパッチを抽出し，調査を行う．ここでの調査はコードの変更や，リファレンスがあればそのリファレンスなどを含めて調査を行い，できるだけ正確にバグを検証する．また報告されたパッチは全て正しい修正であると仮定する．

調査結果は表 6 である．IRQ に関するカーネルコアの誤りも見られたもののデバイスドライバの問題と比べて少なかったため，ここではそれらは無視してバグでないとしている．多くは linux の割り込みハンドラ登録解除の API である free_irq() の呼び出し方法の誤りであり，特定のページ，特に DMA 用のページの解放処理で割り込み禁止にしてはならないという規約を無視しているものに限り割り込み状態の設定誤りであった．“irq”, “free” という単語を話題の中心に持つパッチであれば特に不自然な結果ではないことがわかる．

free_irq() は基本的に割り込みハンドラ登録のための request_threaded_irq() の呼び出しと一貫してペアで正しく使われる必要がある．最も多い誤りは共有割り込みの識別のために使われる最後の引数 dev_id が 2 つの API で一貫していないケースであった．この問題は多くは既存の静的解析ツールのひとつである coccinelle によって発見されたものであったことも報告数が多い要因となっている．

それ以外にも引数となる割り込み番号の誤りや，2 重解放，共有割り込みに関わらずデバイスに対する割り込み停止コマンドを送らずにハンドラを解放してハンドラが呼べずにクラッシュするケースが見られた．また単純な解放忘れなども見られる．共有割り込みの場合，デバイスサスペンド前にハンドラを解放しないとレジューム後，再初期化する前に他のデバイスが割り込みハンドラを呼んで問題が起きる場合がある．いずれもコードの上では memory leak のような誤り方である一方，症状としてはデバイスやカーネルのスケジューリングに依存した，非決定的なバグになる．

ここで観察されたバグは基本的に free_irq() で解放する資源を確保する request_threaded_irq() のコメントで記載されている注意事項が多い．しかし，2 番目に多いエラーパスでの解放忘れは割り込みに限る問題ではないため明記されていない．request_threaded_irq() は多くはドライバの初期化処理の最後に行われる場合が多いものの，そうならない例外的なケースで問題になる．その場合ドライバの持つその他の資源，たとえば DMA やメモリなどの資源確保で失敗した場合，ドライバの初期化に失敗したことをカーネルに通知する前に必ず free_irq() で登録したハンドラを削除しなければならない．そうしなければデバイスが割り込みを起こすと解放した資源へ割り込みハンドラ内でアクセスする恐れがある．しかし，このバグパターンはエラー処理の誤りでしかもデバイスが割り込みを起こすことが顕在化条件になるため極めて顕在化しにくい非決定的バグになる．

ここで見られる非決定的バグは一般に，データフローとコントロールフロー解析を基本とする検査で発見することは難しい．割り込みハンドラはドライバが活動している間ずっと解放されず，ドライバの活動が停止するときに初めて解放されるため，時間経過やユーザの入力（サスペンドや終了処理コマンドを送ること）依存になるためである．一方，SymDrive や S2E のようにシステム全体に渡りドライバの生存期間となるパス全てを検査すればパターンの検査は可能である．ただしバグの顕在化にデバイスからの割り込みを必要とするパターンの発見は難しいかもしれない．

一方，linux に特化したカーネル API の使用誤りと見なしてチェッカを実装すればバグの検査は簡単にできると考えられる．ここで得られたバグは顕在化の条件が非決定的

でもコードで見るとペアとなる API 関数の使用誤りであるため，SymDrive や S2E のような path-sensitive analysis でエラーパスを含め検査することができる．次節ではそのチェッカの設計と実装，そしてチェッカを利用して実際に類似したバグを linux 3.15 で発見したことをレポートする．

まとめ: #16: irq に属するパッチの中で，(free, descriptor) をトピックに持つパッチ 331 件を調査した．その結果，コード上ではカーネル API の使用誤りのパターンで，顕在化するときは発見の難しい非決定的なパターンが観察された．

4. バグの検査

本節では前節の結果を受けて，割り込みハンドラの登録・解除のカーネル API の使用誤りを検査するためのチェッカの設計・実装と検査した結果を示す．本研究はチェッカを clang の静的解析のプラグインとして実装する．また静的解析の制約を緩和するため，チェック対象となるコードを instrument してできるだけ前節で得られたバグを検査をする．検査対象はこの原稿の執筆時の最新バージョンの linux 3.15 とする．

4.1 clang static analyzer

clang static analyzer は path-sensitive 解析を行うチェッカの実装を可能にする．大雑把に言うとプログラムのコンパイルで作成された抽象構文木 (AST) などの出力結果を使って解析エンジンが動作する．解析エンジンは翻訳単位内のトップレベルの式（関数宣言など）から再帰的に AST を辿り，分岐があれば変数の制約を算出し，関数定義のある関数呼び出しが検知された場合はその関数定義へジャンプする．従って，解析の単位はコンパイラの翻訳単位となり，関数定義などが複数の .c ファイルにまたがる場合は戻り値はシンボル値と扱われる．また引数として渡されたポインタ先は渡した関数がどのようにポインタ先を書き換えるか不明なので不明な値として扱う．そのため結果的にデバイスの出力など，外部からの入力は全てシンボル値として扱われる．

チェッカは一般の式の前後，関数呼び出しの前後やポインタエスケープ，シンボルの消滅時など，様々なイベントハンドラを登録し，解析エンジンに登録したイベントや式が出現したときにチェッカが呼び出されるようにすることができる．またシンボリック実行時のプログラムの状態を明示的にフォークさせることができる．本研究の利用例では，request_threaded_irq の戻り値を 0 とそれ以外の値の場合でフォークし，正しくエラーハンドルしているかを検査する．ここでは関数呼び出し後のイベントハンドラを登録し，そこで状態を作り出してフォークをする．

現在 linux の clang/llvm 対応は公式にされ始めており，ほとんどのドライバをビルドできるようになっている．本研究でも後述のように PCI ドライバ 593 件を解析することができている．

表 7 検査項目

項目	割り込みコンテキストで free_irq 呼び出し
#A	irq 番号ゼロで request_irq 呼び出し
#B	割り込みハンドラを null 指定して request_irq 呼び出し
#C	flag = IRQF_SHARED & dev_id = null で request_irq 呼び出し
#D	使用済みの IRQ 番号に対して request_irq 呼び出し
#E	共有割り込みのとき使用済み dev_id を利用して request_irq 呼び出し
#F	(irq, dev_id) の組を間違えて free_irq 呼び出し
#G	解放済みの (irq, dev_id) に対して free_irq 呼び出し
#H	request_irq が失敗したパスで free_irq 呼び出し
#I	free_irq 忘れ

4.2 設計・実装

チェッカは request_threaded_irq または request_irq の呼び出し時に引数として渡された変数や値のシンボル値および具体値をプログラムの状態として保存する．そしてそれら関数の戻り値が成功値の 0 とエラー値のそれ以外の値でプログラムの状態をフォークし，割り込みハンドラの登録がうまくいかなかったパターンをハンドルしているか確認する．その後，free_irq の呼び出し時に保存した引数と一貫したシンボル値や具体値が渡されるかをチェックする．チェッカは基本的にこの動作を行うものの，バグのパターンによってはこれ以外にも工夫が必要となる．

表 7 は対象とする検査項目である．検査項目 #D, #E, #F, #G は free_irq の呼び出し時に保存した状態と比較することでチェックができる．#A は free_irq の呼び出し時にシンボリック実行のコンテキストから呼び出し元の関数の返値の型を調べ， irqreturn_t でないか検査する．#B, #C は request_irq の状態保存する時に一緒にチェックする．#H は登録が失敗したパスで free_irq が呼ばれたときにバグ報告をするように実装する．

しかし，#I はいつ free_irq が発生するべきかがチェッカでは判断できないので検査できない．この問題は free_irq の呼び出し時の状態チェックを行う #D, #E, #F, #G でも検査の見落としを起こす．

4.3 ターゲットとなるドライバ

free_irq が必ず発生するべきポイントはドライバの終了時である．しかし，ハンドラの登録・解除は終了時以外も発生する場合が多く，ドライバの生存期間中ハンドラの登録・解除が正しくされていることを検査する必要がある．そのため，ドライバの動作モデルがどのようになるのかが重要となり，ターゲットドライバを決める必要がある．

そこで頻繁に割り込みハンドラの登録・解除を行うドラ

イバを調査し，最も多いドライバをターゲットとする．ドライバのエントリポイントは割り込みハンドラ以外はコールバック関数になることがほとんどである．そこで調査では clang を利用して， request_threaded_irq, free_irq をコールグラフに含むコールバック関数を調査する．ここでは path-sensitive 解析ではなく AST を辿る方法を用いる．また，前述のようにファイルをまたぐ場合に解析ができないため，ファイル外から呼ばれる可能性のある non-static 関数も同様に調査し，.c ファイルの解析が終わってからファイル外関数の呼び出しによるコールグラフを補完する．そして request_threaded_irq や free_irq を呼び出しているコールバック関数の構造帯とメンバ名の数をカウントする．

表 8 が調査結果である．調査対象は linux のソースツリーの drivers/ 以下を対象とした．free_irq については request_threaded_irq を呼んでいないコールバックのみカウントした．結果から，PCI デバイスドライバが最も割り込みハンドラの登録・解除を行うことがわかり，またデバイスのプローブ時とレジューム時にハンドラは登録され，取り出し時やシステムシャットダウン時，サスペンド時にハンドラは削除されることがわかる．他にもネットワークデバイスドライバは e1000 や bnx2 など PCI デバイスドライバになることもあるものの，割り込みハンドラの登録解除は net_device_ops の ndo_open/ndo_stop でしていることがわかる．

4.4 コードの instrumentation

調査結果から，本研究では PCI デバイスドライバにターゲットを絞り割り込みハンドラの登録解除忘れがないかを検査する．ドライバの動作シナリオとして，デバイスのプローブ時，取り出し時以外にもシステムのシャットダウンやサスペンド・ハイバネーションがあった場合も正しくハンドラの解放が行われているか検査する．しかし，clang の制約としてドライバの生存期間に渡るパスを辿ることは難しい．

そこで検査の対象が割り込みハンドラの登録・解除のみで，シンボル値を利用できることを利用して問題を回避する．割り込みハンドラの登録には irq 番号と dev_id の組を覚えておかねばならないため，ドライバの状態に必ず irq 番号と dev_id の情報が残っている．観察した限りドライバの状態はほとんどがコールバックの引数に渡される struct pci_dev, struct device に用意されているドライバ固有のデータ領域に保存されていることが多い．

そのため，それら引数の型の変数をダミーとして確保し，ドライバの動作モデルを再現するようにコールバックを直接呼び出す解析専用の関数を実装し，そこから解析を始めるようにする．解析は翻訳単位で行う必要があるため，ドライバ実装の C 言語コードに直接 instrument する．

```
char sval[4096];
char *s = sval;

void TestPCIDriver(struct pci_dev *pdev,
     const struct pci_device_id *id) {
  int loop = 0;
  enum TEST_PCI_STATE state;
reprobe:
  if (x_probe(pdev, id)) return;
  state = PCI_STATE_PROBED;

normal_operation:
  switch(*(s++) % 4) {
  case PCI_EVENT_PM:
    x_power_management(pdev, &state);
    break;
  case PCI_EVENT_REMOVE:
    x_remove(pdev);
    state = PCI_STATE_REMOVED;
    if (*(s++) % 2 && loop++ < 10)
      goto reprobe;
    break;
  default:
    ; /* do nothing */
  }
  if (*(s++) % 2 && loop++ < 10 &&
      state == PCI_STATE_PROBED)
    goto normal_operation;

  x_shutdown(pdev);
}
```

図 2 instrument したコード例
x_probe, x_remove などは pci_driver::probe, remove を呼び出すようにする．x_power_management はサスペンドやハイバネーションの状態遷移のモデルを再現する関数で，ここでその実装は省略している．

PCI ドライバの動作モデルは linux のドキュメントを参考に実装した．実装例は図 2 である．実装ではドライバの状態を指す変数 (pdev, id) は関数引数として渡されたものとしてシンボル値を設定し，複数の状態に遷移する可能性がある場合はグローバルスコープに置いた変数の値で分岐するようにする．するとシンボリック実行の性質からランダムに分岐が起こり，goto でランダムな回数ループした後でシステムのシャットダウン時の処理を呼び出し，解放忘れがないかチェックする．また，probe に失敗した場合は即座に終了することで，その時に割り込みハンドラの解放忘れがないかチェックする．実際の動作モデルは割り込みハンドラが porbe 後の処理間に入るものの，検査対象はハンドラの登録解除なので無視した．

5. 結果：バグの検査

チェッカ及び instrumentation を用いて，clang でコン

表 8　割り込みハンドラ登録・解除をするコールバック

request_threaded_irq を呼ぶコールバック		free_irq を呼ぶコールバック	
構造体::メンバ名	数	構造体::メンバ名	数
pci_driver::probe	324	pci_driver::remove	238
platform_driver::probe	226	net_device_ops::ndo_stop	105
i2c_driver::probe	128	platform_driver::remove	92
net_device_ops::ndo_open	111	comedi_driver::detach	84
spi_driver::probe	65	i2c_driver::remove	60
platform_driver::remove	45	pcmcia_driver::remove	46
pci_driver::resume	40	spi_driver::remove	41
pcmcia_driver::probe	34	pci_driver::suspend	40
work_struct::func	33	pci_driver::shutdown	33
comedi_driver::auto_attach	32	file_operations::release	23

パイルエラーなしでビルドできる PCI デバイスドライバ 593 件を検査した．対象は 2014 年 6 月にリリースされた linux 3.15 を対象としている．結果，230 件のバグ報告が得られている．ただし，静的解析の性質上，false positive は避けられないためマニュアルで確認する必要がある．現在調査中であるものの，少なくともバグの可能性が高い報告 2 件について詳細に述べる．

図 3 は PCI ベースの PC カードのホストデバイスとなる CardBus のバスドライバの例となる．このバグは request_irq の後の pcmcia_register_socket が失敗したケースで free_irq を忘れている．この場合，ドライバの持つ資源が unmap ラベル以後で全て解放されるため，デバイスが割り込みを起こすと割り込みハンドラが解放済みの資源にアクセスする恐れがある．このように，非決定的なバグもチェッカで検査することができることがわかる．

2 件目はより単純なケースで，Atheros 社提供ネットワークカードに対するデバイスドライバ (net/ethernet/atheros/alx/main.c) において，シャットダウン時の処理を定義していなかったパターンであった．デバイスの終了処理をしないため，割り込みと DMA が電源が切れるその瞬間まで起こり，予期しない動作が起きるかもしれない．シャットダウン時の処理を定義しない原因を調べたところ，過去実装はあったものの削除されていることが判明した．その実装を見ると確かにシャットダウン時に free_irq は呼ばれており，ハンドラの解放はすべき操作であったことがわかる．このパターンは典型的な regression で，このようにチェッカは退行テストとしても利用できることがわかる．また，ユーザの入力依存でドライバの生存期間に渡るパスを検査する必要のあるパターンであるため， instrument が効果的であることがわかる．

まとめ: クラスタリングによりより発生しやすいバグを抽出することが可能になり，実際に効果的なバグの検査をすることが可能になることがわかる．例えエラー処理と非同期的実行が絡む極めて複雑なバグでさえ，パターンとして認識さえできれば既存の静的解析を利用して発見することも難しくないことがわかる．

```
static
int yenta_probe(struct pci_dev *dev,
        const struct pci_device_id *id)
{
  struct yenta_socket *socket;
  ....
  if (!socket->cb_irq ||
    request_irq(socket->cb_irq,
          yenta_interrupt,
          IRQF_SHARED,
          "yenta", socket)) {
    ....
  } else { /* 成功パス */
    socket->socket.features |= SS_CAP_CARDBUS;
  }
  ....
  ret =
    pcmcia_register_socket(&socket->socket);
  if (ret == 0) { /* 成功パス */
    /* Add the yenta register attributes */
    ret = device_create_file(&dev->dev,
        &dev_attr_yenta_registers);
    if (ret == 0)
      goto out; /* 成功パス */
    .....
  }
unmap: /* pcmcia_register_socket 失敗パス */
  iounmap(socket->base);
release:
  pci_release_regions(dev);
  ....
out:
  return ret;
}
```

図 3　検知されたバグの例 (場所 : drivers/pcmcia/yenta_socket.c)

6. 関連研究

バグの実態調査は過去数十年に渡って行われてきており，調査結果はバグ対策の研究に大きく貢献している．典

型的な実態調査は特定のシステムについての障害報告やバグ報告を多数集めてきて特定の指標に基づいて調べ上げることである．最近の調査として，Lu らのファイルシステムの調査 [5] があげられる．彼女らは linux 2.6 のファイルシステム数種類についてのパッチ 5,000 件以上に渡ってマニュアルで調査を行い，実際に ext3 や vfat などの代表的なファイルシステムにおいて発生してきたバグや変化の傾向を示している．Cotroneo らの研究 [6] では bohrbug, aging-related bug, non-aging-related bug の 3 種類に分けて linux や MySQL などのオープンソースソフトウェアのバグを調査している．2008 年の Lu らの研究 [9] では報告された concurrency bug 105 件に注目し， atomicity violation, order violation などの性質について定義し，詳細に報告している．本研究で挙げた自然言語処理やクラスタリングはこれらのような指標を探し出すための方法で，ケーススタディとして挙げた割り込みハンドラの登録・解除のような流れで同様の調査を行うことが可能である．ただし，特定のシステムに特化した指標になるためより広範囲のソフトウェアへの適応が難しいものの，対象が絞られるためバグの検査にとっては効果が高いと考えられる．

異なるバグの調査方法として対象のバグを先に定義し，コードの静的解析などで検査する方法があげられる．Wang ら [1] [2] は C 言語で未定義動作を引き起こすパターンを定義し，llvm IR の解析をすることで，linux カーネルを含む多数のソフトウェア検査して 100 件以上のバグを発見している．Palix ら [3] は Chou [10] らの研究で定義されたバグパターンについて再調査を行い，10 年に渡って linux がバグを取りきれていないことを示している．Kadev ら [11] はデバイスの異常を正しくハンドルせずに問題が起きるパターンなどを定義して検査し，多量のバグを発見している．我々のケーススタディでは過去によく発生していた linux カーネル API に特化した問題に焦点を当てて検査を行い，バグを発見しているものの，これらのように新たに定義した問題を発見できるとは限らない．しかし，多数の開発者が過去誤りやすかった問題も同じバグである以上検査はしていく必要があり，本研究のようなアプローチも彼らのようなアプローチも両方必要である．

本研究の検査でも clang の static analyzer を用いたように，検査を簡単にするフレームワークはバグ対策にとって不可欠である．SymDrive [12] は S2E [13] を用いてデバイスの入力をシンボル値とし，デバイスなしでドライバの検査をできるようにする手法である．S2E はバイナリを用いたシンボリック実行を提供し，システム全体に渡るパスの検査を可能にする手法である．これらのフレームワークもやはりバグの性質によっては対策が難しいパターンも存在する．しかし，我々の方法でバグの実態が得られれば，コードを instrument してフレームワークをより効果的に使うことができるようになった例のように，よりよい使い方を発見できるかもしれない．

我々は linux カーネルを対象としている．linux のバグはシステムの仕様が明確でないために問題が発生している側面もある．最もバグ対策として効果があるのは API の提供者がシステムの仕様を明確に定めてバグ対策を最初からしやすいシステムを提供することである．Windows では Static Driver Verifier [4] を用いて Windows カーネル API の使用誤りをデバイスドライバがしていないかを検査することができる．seL4 [14] は仕様を満たすことを形式検査し，カーネル全体に渡ってバグがないことを保証している．しかし，これらのソフトウェアでもどのように仕様を定めるかを決定するときに，どのようなバグが OS カーネルにおいて発生しやすいのかを知ることはより効率的なバグ対策につながるかもしれない．

7. 結論

本研究は現実に発生しやすかったバグを調べ，バグを抽象化・モデル化し，検査するという基本的なバグ対策の一連の流れを実践したものである．このケーススタディから，バグの対策において重要なのはどのようにしてより発生しやすかったバグの実態を得るか，ということであることがわかる．それさえ得られれば，例えそのバグが顕在化するために複雑な条件を持っていたとしても，何を検査項目とし，どのように検査を設計・実装するかは簡単に決定することができる．バグの実態を得るための方法として本研究はパッチに対する自然言語処理とクラスタリングを用いることで，過去のパッチ全体から見てより発生しやすかったバグの実態を抽出することに成功した．今後も本論文で挙げた以外のバグのパターンについて調査し，バグ対策をするという一連の流れを継続し，より linux の信頼性の向上に貢献していく必要がある．

参考文献

[1] Wang, X., Zeldovich, N., Kaashoek, M. F. and Solar-Lezama, A.: Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior, *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, ACM, pp. 260–275 (2013).

[2] Wang, X., Chen, H., Jia, Z., Zeldovich, N. and Kaashoek, M. F.: Improving Integer Security for Systems with KINT, *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, USENIX Association, pp. 163–177 (2012).

[3] 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)*, ACM, pp. 305–318 (2011).

[4] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K. and Ustuner, A.: Thorough Static Analysis of Device Drivers, *Proceedings of the 1st ACM SIGOPS/EuroSys*

European Conference on Computer Systems 2006 (EuroSys '06), ACM, pp. 73–85 (2006).

[5] Lu, L., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H. and Lu, S.: A Study of Linux File System Evolution, *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, USENIX Association, pp. 31–44 (2013).

[6] Cotroneo, D., Grottke, M., Natella, R., Pietrantuono, R. and Trivedi, K.: Fault triggers in open-source software: An experience report, *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE '13)*, pp. 178–187 (2013).

[7] Manning, C. D., Raghavan, P. and Schuetze, H.: *Introduction to Information Retrieval*, Cambridge University Press (2008).

[8] Blei, D. M., Ng, A. Y. and Jordan, M. I.: Latent Dirichlet Allocation, *Journal of Machine Learning Research*, Vol. 3, pp. 993 – 1022 (2003).

[9] Lu, S., Park, S., Seo, E. and Zhou, Y.: Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, ACM, pp. 329–339 (2008).

[10] 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)*, ACM, pp. 73–88 (2001).

[11] Kadav, A., Renzelmann, M. J. and Swift, M. M.: Tolerating Hardware Device Failures in Software, *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*, ACM, pp. 59–72 (2009).

[12] Renzelmann, M. J., Kadav, A. and Swift, M. M.: SymDrive: Testing Drivers Without Devices, *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, USENIX Association, pp. 279–292 (2012).

[13] Chipounov, V., Kuznetsov, V. and Candea, G.: S2E: A Platform for In-vivo Multi-path Analysis of Software Systems, *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*, ACM, pp. 265–278 (2011).

[14] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S.: seL4: Formal Verification of an OS Kernel, *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*, ACM, pp. 207–220 (2009).