Vol.2020-OS-149 No.5
2020/5/28

エラーコード解析を用いた構造体メンバのメモリリークの解析

鈴木慶汰[1]　窪田貴文[1]　河野健二[1]

概要：Linux Kernel のような大規模なソフトウェアでは構造体のメンバのメモリリークは大きな問題になりうる。実際に過去２年間の Linux Kernel のパッチを調査したところ、メモリリークに関するパッチのうち、54.6% が構造体メンバに関するものであった。しかしながら、これらのバグを網羅的に発見するのは難しい。これは、それぞれのコードパスによって確保、もしくは解放される構造体メンバーがことなってしまうためである。

本稿では、Linux Kernel を対象に、構造体メンバのメモリリークを静的に解析するツールを作成する。本解析ではエラーコード解析という解析を行い、それぞれの関数から返されるエラーコードに注目することで、各ベーシックブロックでの確保/解放されているメモリ領域の情報を正しく伝搬する。今回は特に解放責任の所在が明確なものを対象に解析を行う。本稿で提案したツールを使って、Linux Kernel で新たに 2 種類のバグを発見し、修正パッチを作成した。これらのパッチは開発者に承認された。

1. はじめに

Linux Kernel などのシステムソフトウェアではソフトウェアエージングは大きな問題になりうる [15]．これは，ソフトウェアを実行する中で，時間の経過とともに，これらのソフトウェアが劣化し，最悪の場合システムを異常終了させてしまう可能性があるためである．そのため，ソフトウェアエージングの特徴として，問題が発生するタイミングと，実際にそれらの問題が表面化するタイミングが大きく異なっている．

ソフトウェアエージングの原因の１つにメモリリークがある．メモリリークは，動的に確保されたメモリ領域を解放せず，参照する変数がなくなることにより発生する．その特徴から，確保されたままのメモリ領域が時間とともに増えていくことで，ソフトウェアが使用できるメモリ領域が圧迫されていき，ソフトウェアエージングが引き起こされる．このため，システムソフトウェアでメモリリークが発生してしまうと，システム全体のパフォーマンス低下，もしくは異常終了をしてしまう．

メモリリークの中には**構造体メンバのメモリリーク**という特定のパターンのリークがある．構造体を用いて関連するデータをまとめて扱うことは一般的であり，動的に確保された領域を構造体メンバに格納することが多い．動的に確保された構造体に，これらのメンバが存在する時，各構造体を解放する際にメンバのメモリも解放される必要がある．しかし，構造体メンバの解放をすることなく，構造体を解放してしまう事例が多く，これを構造体メンバのメモリリークと呼ぶ．

```
1  int main() {
2    struct node* p;
3    p = (struct node*)malloc(...);
4    p->data = (struct data*)malloc(...);
5    ...
6    // Need p->data to be freed
7    free(p);
8  }
```

コード 1　構造体メンバのメモリリークの例

コード 1 は構造体メンバのメモリリークの簡単な例である．この例では３行目で構造体 `node` を確保している．この構造体のメンバ `data` に動的に確保された領域を格納している (４行目)．この後に，７行目で構造体 `node` を解放している．これにより，構造体 `node` は正しく解放されているが，依然として `p->data` は確保されたままとなっている．構造体 `node` は解放されてしまっているため `p->data` の指す領域はどこからも参照されなくなっており，構造体メンバのメモリリークが発生している．

実際に Linux Kernel の修正パッチを調査したところ，メモリリークを修正するパッチの内，54.6% のパッチが構造体メンバに関するリークであった．これらのパッチの 9

[1] 慶應義塾大学
Keio Uniersity

割以上は開発者本人により発見，もしくは動的解析のツールを用いてコード実行時に発見された．しかし，これらの手法では，問題の発見が遅くなってしまう．これは，どちらの手法も，実際に問題のあるコードを開発者の目を通る，もしくは実際に実行されない限りは発見ができないからである．実際に発見されたバグでは，発見まで 6 年半かかっているケースもあった．そこで，網羅的にソースコードを解析できる静的解析のアプローチも必要である．

本稿では，特に Linux Kernel をターゲットにし，構造体メンバのメモリリークを静的解析の手法を用いて検知する．各構造体メンバの型と構造体の先頭からのオフセット (Field) のみに注目し，構造体解放時に各メンバに該当する Field に対して動的にメモリが確保されているにも関わらず，解放されていなかったら警告を出す．各 Field は構造体の解放までのいずれかのコードパスで確保されていたら確保されているとみなす．また，構造体の解放までのすべてのコードパスで解放されている場合のみに解放されているとみなす．

メモリの動的確保と解放が同一関数内で行われていることは稀であり，確保の情報を正しく伝搬するには，関数間にまたがったすべてのコードパスを考慮する必要がある．しかし，Linux Kernel のような大規模なソフトウェアでは，すべてのとりうるコードパスを考慮すると，組み合わせ爆発が発生し，スケールしない．

そこで，関数間で伝搬する情報を最小限にすることで解析を簡略化し，大規模なソフトウェアでもスケール可能な手法を考える．その手法として，各関数が呼び出されるコンテキストを考慮し，それぞれのコンテキストによって伝搬する情報を変化させる．今回は，特に関数の呼び出されるコンテキストを成功状態と，エラー状態の 2 種類に分類る．これを実現するため，Linux Kernel の returning convention に注目したエラーコード解析を行う [13].

各関数の返すエラーコードに注目することで，関数が正しく実行された状態 (成功状態) と異常終了した状態 (エラー状態) を区別する．これは，各関数の返り値を条件分岐等で処理している部分に注目する．もし返り値をエラーとして処理を行っていた場合，呼び出し先の関数がエラーを返す直前の各 Field の情報を渡す．それ以外のときは成功状態の Field を返すようにする．このように各関数の返り値がどう扱われるかを考慮することで，それぞれのコードパスで関数のどの状態が期待されているかを判断することができる．これにより，それぞれの期待されている状態に対応する情報のみを渡すことができ，それぞれのコードパスで必要な Field の情報を渡すことができる．

作成したツールを用いて実際に Linux Kernel を解析した．本解析では，特にエラーコード解析の精度，そして Linux Kernel で実際にバグが発見できるかどうかに注目した．エラーコード解析の精度は，78.0% の関数間のエラーコード解析が正しくエラー処理を行うブロックを検知できていた．また，実際の Linux Kernel で 2 つの新しいバグを発見し，Linux Kernel の開発者に実際にバグであると確定された [19], [20].

2. Linux Kernel におけるメモリリーク

2.1 構造体メンバのメモリリーク

Linux Kernel などの大規模なソフトウェアは，開発段階で全てのバグを取り除くことは現実的に不可能であり，多くのバグが残っている [15]．メモリリークはソフトウェアエイジングを引き起こすバグとしてであり，その発見は困難であるにも関わらず，システム全体に影響を与え，最悪の場合システムの異常終了につながる可能性があるからである．

メモリリークの中でも構造体メンバのメモリリークは，発見が難しく，特に問題になりやすい．これは，動的に確保された構造体のメンバに，さらに動的に確保された領域を格納することで発生する．構造体自体が解放される時，各構造体メンバに格納されている領域も解放しなければいけない．しかし，コード 1 のように，構造体メンバのみから参照されているメモリ領域があるにも関わらず，構造体自体を解放してしまうことがある．これにより，構造体メンバより参照されていた領域のメモリリークが発生してしまう．

コード 2 は Linux Kernel で実際に発見された構造体メンバのメモリリークである．このコードは wlcore デバイスドライバ内で発見されたもので，エラー処理による解放忘れである [7]．関数 `wl12xx_fetch_firmware` 以外は `goto out` でのエラー処理で問題なかった．そのため，8 行目のエラー処理も同様に行なっていた．しかし，`wl1271_setup` でメモリ領域を確保している．もし `wl1271_setup` がエラーだった場合，その関数内で確保された領域は解放されるが，それ以降の `wl12xx_fetch_firmware` のエラー処理では確保されたリソースの解放が必要である．修正部分ではこれを考慮していなかったため，メモリリークが発生している．

2.2 Linux Kernel のパッチ調査

構造体メモリリークが存在するかどうかを示すため，Linux Kernel の過去 2 年分のパッチを調査する (計 2,462,924 件)．Linux Kernel version 5.3-rc4 (commit d45331b00ddb179e291766617259261c112db872) を対象とし，‘memory leak’ などのキーワードを含んだものを調査する．

表 1 にこの結果を示す．全パッチの内，メモリリークに関するパッチは合計 580 件存在した．その中でも構造体メンバのメモリリークに関するものは 317 件存在した．従ってメモリリーク関連のパッチのうち，54.6% が構造体のメ

```
1  int wl12xx_chip_wakeup(struct wl1271 *wl)
2  {
3    ret = wlte1271_setup(wl);
4    if (ret < 0)
5      goto out;
6  ...
7    ret = wl12xx_fetch_firmware(wl);
8    if (ret < 0) {
9  -       goto out;
10 +       kfree(wl->fw_status);
11 +       kfree(wl->raw_fw_status);
12 +       kfree(wl->tx_res_if);
13   }
14 out:
15   return ret;
16 }
17
18 int wl1271_setup(struct wl1271 *wl)
19 {
20   wl->raw_fw_status = kzalloc(...);
21   wl->fw_status = kzalloc(...);
22   wl->tx_res_if = kzalloc(...);
23   if (!wl->tx_res_if)
24     goto err;
25   return 0;
26 err:
27   kfree(wl->fw_status);
28   kfree(wl->raw_fw_status);
29   return -ENOMEM;
30 }
```

コード 2 Bug found in Linux Kernel Git Log

表 1 Linux Kernel 内で発見されたバグの割合

Period	Leak Related	Struct Related	Rate
2017/9 ~ 2018/8	225	106	47.1 %
2018/9 ~ 2019/8	355	211	59.4 %
total	580	317	54.6 %

表 2 バグ発見で使用されていたツール

Method	Tool Name	Tool Total	Total
Static	Coccinelle[14]	12	26
	Coverity[2]	10	
	LDV[3]	4	
Dynamic	Syzkaller[4]	15	45
	KASan[5]	30	
Manual	-	-	246

ンバのメモリリークであった．また，各年の割合を見ても 2018 年 9 月からの 1 年間は 59.4%, 2017 年 9 月からの 1 年間は 47.1%と同程度存在していることが確認できる．

また，実際に現在構造体メンバのメモリリークがどのような手法で検知されているかを調べるため，どのようなツールを用いているかを調査する．表 2 にメモリリークを検知する手法を示す．この表からわかるように 9 割以上のバグは手動でのチェック，もしくは動的検査ツールで発見されている．これらの動的な手法では，実際にコードが実行されないとバグを発見できない．そのため，実際に発見されるまでに長い時間がかかる場合がある，実際，コード 2 は kmemleak[11] を使用して発見されており，バグが発見されるまでに 6 年以上かかっていた．

3. 提案手法

本稿では Linux Kernel を対象に静的解析の手法を用いて，構造体メンバのメモリリークを検知するツールを作成する．

3.1 概要と課題

基本的なアプローチとして，構造体の解放を検知した時，各構造体メンバの状態を確認し，その時点で解放されていないメンバに対して警告を出す．この時，各構造体メンバの確保と解放の情報を収集し，構造体解放時にそれぞれのメンバの状態を確認する．

また，本解析では，各情報を追跡するために型とメンバのインデックスをみる Field-based [21] な解析を行う．これは，各構造体とそのメンバを効率的に判別することができるためである．構造体のメンバは構造体そのものの先頭アドレスとそこからのオフセットとして表現される．そのため，構造体の各メンバ変数として扱われることがない．これにより，Value-based な解析では，構造体メンバについて正しい情報を得ることができない．また，Value-based なアプローチでは，関数呼び出しごとに，呼び出し元の構造体との対応を取る必要があり Alias 解析が必要となる．そこで，Field-based なアプローチを取ることで， Alias 解析を回避しつつ構造体の各メンバについて情報が得られるようにする．

確保の情報については，構造体解放までに確保されている可能性があれば，確保されているとみなす必要がある．これは，確保の情報を見落とすことで発生する未検知を防ぐためである．したがって，各構造体メンバはいずれかのコードパスで確保されていたら確保されているとみなす．

解放の情報については，構造体解放までに解放されていることが保証できる場合に解放されているとみなす．これは，解放忘れを見逃すことを防ぎ，未検知を減らすためである．したがって，構造体の各メンバはすべてのコードパスで解放されていたら解放されているとみなす．

構造体の各メンバは，確保・解放の状況に応じて，３つのいずれかの状態を持つ．それぞれの状態について以下のように定義する．

- 未確保: 構造体メンバに動的な領域が割り当てられていない状態．すべての構造体メンバはこの状態から始まる (初期状態).
- 確保済み: 構造体メンバに動的にメモリ領域が割り当てられた状態．取りうるコードパスのいずれかのパスで構造体メンバが確保されていた場合，構造体メンバ

```
1  int main() {
2    struct node* p;
3    p = (struct node*)malloc(...);
4    ...
5    int err = allocate_member(p);
6    if (err < 0) {
7      free(p);
8    }
9    ...;
10 }
11 int allocate_member(struct node* p){
12   struct data* dat;
13   dat = (struct data*)malloc(...);
14   if (!dat)
15     return -ERROR;
16   p->data = dat;
17   return 0;
18 }
```

コード 3 関数の終了状態を考慮しなければいけない例

は確保済みの状態にあるとする．

- 解放済み: 構造体メンバが解放された状態．取りうる全てのコードパスのすべてのパスで構造体メンバが解放されていた場合，構造体メンバは解放済みの状態であるとする．

各 Field の状態を集めるため，本解析では関数内での解析と，関数間での解析を行う．関数内解析では各関数内の基本ブロックごとに確保されている可能性のある Field，解放されている可能性のある Field を集める．これは，始めに各基本ブロックの直前の基本ブロックから情報を伝搬する．その後に，各基本ブロック内での動的な領域を確保／解放する関数の呼び出しを検知し，該当の Field を追加する．呼び出された関数が直接領域を確保／解放する関数出なかった時，関数間の解析を行う．

関数間解析では，呼び出された関数が返る時に各 Field が取りうる状態を呼び出し元に渡す．この時，各 Field が呼び出し先の関数で取りうる全ての状態を呼び出し元に反映してしまうと，大量の誤検知が発生してしまう．これは，関数の状態によっては，確実に確保されていない／確実に解放されている Field が存在するからである．

例 3 は関数の状態によって Field が取る状態が変わる例である．この例では構造体 `struct node` のメンバ `data` を別の関数 (`allocate_member()`) で動的に確保しょうとしている．もし領域の確保に成功したら呼び出し先の関数は `struct node` のメンバに確保した領域を格納し return する．確保に失敗したら負の値のエラーコードを返して終了する．ここで，関数の状態を考慮しなかった場合，7 行目の構造体の解放の処理で，メンバが確保されていると考え，解放処理が必要と警告を出してしまう．しかし，実際にはこの部分はメンバが確保できなかったときのためのエラー処理であるため，そもそもメンバは確保されていない．このように各関数の状態を一色端に扱ってしまうと誤検知が発生してしまう．そこで，各関数が取りうる状態を考える．

しかし，関数が取りうる全ての状態を考慮するのは全てのコードパスを解析する必要があり，Linux Kernel のような大規模なソフトウェアでは不可能である．これは，関数間で伝搬させる情報が複雑になってしまうからである．このため，効果的にこれらの誤検知を防ぐ方法として，呼び出し元の関数が期待している呼び出し先の関数の終了状態を考える．ここで，関数が呼び出されるコンテキストによって渡す状態を変更する．

本解析では関数間で伝搬する情報を最小限にすることで Linux Kernel でもスケールする方法を考える．そこで，関数が実行後に取りうる状態を成功状態とエラー状態の 2 つ状態に分類し，Linux Kernel の 関数の returning convention を用いて各関数のエラーコードを解析する．

また，本解析では解放された構造体にそのメンバの解放責任があることが自明な場合のみを対象とする．動的に確保された領域によっては，複数の別の構造体が同じ領域をメンバで参照することがる．このような場合は，どの構造体がその領域を解放するかの判定が必要である．しかし，この判定を行うのは容易ではない．今回の解析は，特に基本的な構造体メンバのメモリリークを解析する上での基本的なアプローチに注目しているため，これらの複雑な解放責任の判定が必要なメンバは対象外とし，そのメンバの解放責任が自明な場合のみを対象とする．

3.2 関数内での解析

アルゴリズム 1 は関数内での基本的なアルゴリズムを示す．各関数はベーシックブロックごとに確保済みの Field と解放済みの Field を管理するリストを持つ．各ベーシックブロック解析時，はじめに直前のベーシックブロック (predecessor) からの情報を伝搬させ，初期化を行う．具体的には，確保済リストは各 predecessor の確保済みの情報の和集合を取る．これによりベーシックブロック B までに確保されている可能性があるすべての確保の情報をあつめることができる．これに対して解放済リストは，各 predecessor の解放済情報の積集合を取る．これにより，ベーシックブロック B までに確実に解放されているものの情報をあつめることができる．

各ベーシックブロック内の命令について，関数呼び出しを行っている命令を探す．もし動的な領域を確保する関数呼び出しだった場合，確保された Field を新たに確保済リストに追加する．表 3 が確保を行う関数の一覧である．同じように解放を行う関数呼び出しだった場合，解放された Field を解放済リストに追加する．この時，構造体を解放していた場合，後ほど評価を行うため各関数ごとに管理する解放済構造体リスト (`FreedStructList`) に追加する．表 4 が解放を行う関数の一覧である．もしその他の関数を

Algorithm 1: 各関数内での情報収集

```
Result: Collect Intra-process Information
foreach Function do
    FreedStructList ⟵ φ;
    foreach B in BasicBlock do
        AllocList_B ⟵ φ;
        FreeList_B ⟵ φ;
        foreach Pred in PredBlocks(B) do
            AllocList_B ⟵ AllocList_B ∪ AllocList_Pred
            FreeList_B ⟵ FreeList_B ∩ FreeList_Pred
        end
        foreach I in Instructions do
            if isCallInst then
                if is Alloc Function then
                    AllocList_B.add(getField(I))
                else if is Free Function then
                    FreeList_B.add() if is Struct Free then
                        FreedStructList.add(getField(I))
                    end
                else CopyFromCalleeFunction() ;
            else if isStoreInst then
                if stores allocated field then
                    AllocList_B.add(getField(I));
                end
        end
    end
end
```

表 3 Alloc Functions

kzalloc	kmalloc	vmalloc
kcalloc	vzalloc	kzalloc_node
kmalloc_array	kmem_cache_alloc	kmem_cache_alloc_node

表 4 Free Functions

kfree	kzfree	vfree	kvfree

呼び出していたら，呼び出し先の情報をコピーする必要がある．そのため，関数間解析を行う．

確保された Field が別の Field に格納される場合，値が格納された Field も確保済と扱う必要がある．そのため，各ベーシックブロックの Store 命令について，もし確保済リストにある Field を格納していたら，値が格納されていた Field もリストに追加する．

3.3 エラーコード解析と関数間の解析

3.3.1 エラーコード解析

関数間の解析で，どの状態を期待されているかを解析するため，Linux Kernel の関数の返すエラーコードに注目する．Linux Kernel では，関数の返り値として正常に終了したかどうかを判別するエラーコードを用いることが往々にしてある．このエラーコードは returning convention によってエラー発生時は負の数，もしくはポインターの場合は NULL で返され，正常終了した時にはゼロ，もしくは NULL 以外を返す [13]．これらのエラーコードを解析することで，呼び出し元に渡す情報を変更する．

エラーコード解析は呼び出し先の関数内で返しうるエラーコードとその時の各 Field の状態を判定する関数内解析部と，呼び出し元の関数で返り値がどのようなコンテキストで使用されているかを確認する関数間解析部に分けることができる．

Algorithm 2: 関数内エラーコード解析

```
Output: ErrorReturnBlockMap_F
foreach Function F do
    Map ErrorReturnBlockMap_F ⟵ φ;
    Queue PendingErrorCode_F ⟵ φ;
    foreach BasicBlock B in F do
        if assigns return value R then
            if IsErrorValue(R) then
                PendingErrorCode_F.add(R);
            end
        end
        if returns then
            while PendingErrorCode_F not empty do
                Code ⟵ PendingErrorCode.front();
                ErrorReturnBlockMap_F[Code] = B;
                PendingErrorCode_F.pop();
            end
        end
    end
end
```

アルゴリズム 2 は関数内解析部の解析を示す．はじめに各関数について返り値 R を指定する部分を探す．これらの R についてにエラーにあたる負の数，もしくは NULL を格納していた時，エラーを返すとみなす．ここで，返り値を指定したベーシックブロック B 内でそのまま return されている時，関数内での解析を行ったあとの B の最終的な状態がそのエラーコードが指定されたときの Field の状態となる．

処理によっては返り値を指定した部分と実際に関数から return する部分が異なることがある．これはエラー処理などが複数ブロックに渡って行われているときなどに起きる．そのため，各エラーコードをキューに追加し，実際に return するまで待つ．Return する Successor ブロックを発見した時，これらのキューにあるエラーコードを return するブロックとともにマップに格納する．これによってエラーコードと最終的な field の状態をとりだすことが可能になる．成功状態も同様の手法で取り出すことができる．

上記の手法で関数内でのエラーコード解析を行うことができた．この次のステップとして関数呼び出しが行われた時，関数の呼び出し元のコンテキストによって渡す情報を切り替える．これは各返り値がどのようにして評価されているかによって渡す情報を切り替える．返り値 R に対して評価の仕方の方法は整数 $C(C \leq 0)$ に対して次の 3 通りある．

(1) $R \leq C$: R が整数 C 以下と評価する．True である時，関数内エラーコード解析で求めた C 以下の全て

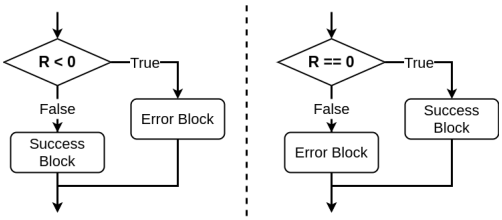

図 1 関数間エラーコード解析の例

のエラーコードの情報を渡す．逆に False である時，C 以上の全てのエラーコードもしくは成功状態の情報を渡す．

(2) $R == C$: R が整数 C であると評価する．C を負の数と評価するときは該当のエラーコード R のみの情報を渡す．C を 0 と評価するときは成功状態の情報を渡す．

(3) $R != C$: R が整数 C でないと評価する．C 以外のエラーコードや成功状態の情報を渡す．

図 1 は C=0 のときの関数間エラーコード解析の例を示している．左は 評価方法 1 の例である．この時 True と評価したパスはエラーパスとなる．そのため，続くベーシックブロックはエラー処理を行うベーシックブロックとなる．逆に False のパスは成功状態であるため，成功時の処理を行うベーシックブロックとなる．同じように右は評価方法 2 の例である．この時は True の時は成功パスとなり，False がエラーパスとなる．このように評価を行うことでエラーパスと成功パスの 2 つが生じるため，エラーパスと評価されたベーシックブロックには関数内エラーコード解析で求めたエラーコードの情報を，成功パスには成功状態の情報を渡す．これらの評価方法に基づいて，該当のエラーコードの情報を集める．

3.3.2 関数間解析

関数間のエラーコード解析では，実際にどの情報を渡す必要があるかを求めた．関数間の解析では，実際にこれらの情報をまとめ，渡す．関数間で渡す情報としては各エラーコードもしくは成功状態の確保リスト，解放リスト，そして解放済構造体リストである．また，関数がエラーコードを返すにもかかわらず，返り値についてなにも評価を行わない場合，暗黙的に成功状態のみを渡すとする．

エラーコードの情報を集めるにあたって，複数のエラーコードの情報をまとめて渡す必要がある時，関数内での情報伝搬と同様に確保の情報は和集合を，解放の情報は積集合を渡す．ここで，解放済構造体リストについては，確保済みリストと同様和集合を取る．これは，構造体そのものの解放情報は各メンバの情報と異なり，次の解析を行う上で必要な情報である．そのため，構造体の解放に関する情報は必ず渡す必要がある．

3.4 メモリリークの検知

実際に収集した情報からメモリリークを検知するため，

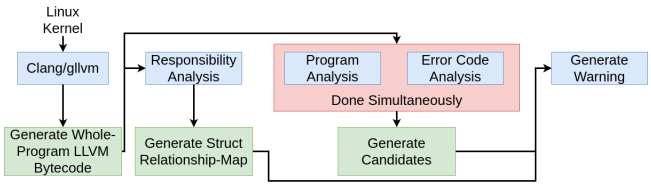

図 2 解析のフロー

本ツールでは各関数の解析終了時，解放済構造体リストに追加されている構造体を評価する．解放済構造体リストの各構造体に対して，それぞれのメンバを確認する．

本ツールでは各メンバの解放責任が明確な場合にのみ警告を出す．そのため，予め各メンバの解放責任を知る必要がある．これは，全ての構造体に対してそのメンバの型が，他の構造体のメンバに含まれているかを確認することで知ることができる．これにより各構造体のメンバで，他の構造体がメンバとして持っていないものに関しては，確かにそのメンバは構造体が解放する必要があるということがわかる．しかし，これでは構造体メンバが暗黙的に共有されていた時に誤検知をしてしまう．これは `void*` などにキャストした時に現れる．そこで，今回は各メンバの Field が `void*` にキャストされていた時も，解放責任が不明と考える．

構造体に解放責任があるメンバの Field が確保リストにある時，そのメンバは，確保済状態である．そのため，解放済リストも確認をする．もし，メンバの Field が解放済リストにあった場合は，解放済状態であるため，正しい状態である．しかし，もしなかった場合はメモリリークとなるため，警告を出す．

4. 実装

前節での解析手法を用いて，実際のツールの実装を考える．本ツールは LLVM[12] と Clang[1] を用いて実装する．図 2 が本ツールの流れとなっている．本ツールはまずはじめに Clang を用いて Linux Kernel をコンパイル及び LLVM Bytecode を生成する． また，gllvm[18] を用いることで Linux Kernel を一つの大きなモジュールとして扱う．作成した LLVM Bytecode を用いて，構造体と各メンバの解放責任を判定する．今回は解放責任が明示的なもののみを対象とするため，構造体メンバが他の構造体から参照されていない型であることが保証できる時のみ，解析対象のメンバとする．同時に，前節の解析手法で関数内及び関数間解析をおこなう．これと同時にエラーコード解析も行う．解放済構造体リストが生成できたら，各関数について構造体メンバの解放責任の情報と照らし合わせ，確保済み状態のものを検知し警告を出す．これによりメモリリークを検知することが可能にある．

5. 評価

実際に本ツールを用いて Linux Kernel の解析を行い，

表 5　実験環境

LLVM/Clang	10.0.0
gllvm	1.2.5
Target Kernel	Linux Kernel version 5.3-rc4 (commit d45331b00ddb179e29 -1766617259261c112db872)
OS	Ubuntu 18.04
CPU	Intel(R) Xeon(R) CPU E5-2620
RAM	96 GB

表 6　解析結果

分類	エラーコード解析なし	エラーコード解析あり
全警告	34	23
誤検知	32	21
正検知	2	
精度	5.9 %	8.7 %

本手法の評価を行う．今回は特にエラーコード解析でどの程度正しくエラーコードを解析できているのかという点，そして実際にメモリリークは検知できているのかという点について評価を行う．

表 5 は実験環境を示している．

5.1　エラーコード解析の精度

関数間でのエラーコード解析がどの程度正しいかを評価するため，エラー処理を行っていると判定したベーシックブロックについて警告を出した．これらの警告の内，自明なものを上から 100 件取り出し，実際にエラーであったかどうかを確認した． 100 件のうち，実際に 78 件の警告がエラーコードから正しくエラー処理を行っているベーシックブロックをみつけだしていた．これは，全体の 78.0% の警告にあたる．誤検知として現れたものは，そもそもエラーコードを用いていないようなものが多く存在した．これは独自の関数間で決められた規約で返り値をやり取りしているものが多く存在した．

また，実際に Linux Kernel で本ツールを用いて解析を行った．表 6 はその結果を示したものである．今回は，エラーコード解析ありと，エラーコード解析なしの 2 パターンで解析をこない，その結果を示している．どちらのパターンも同じメモリリークを 2 つ検知した．また，エラーコード解析なしでは全警告数が多く出ていた．これはエラーコードを考慮しなかったことにより間違って確保の情報がエラー処理に流れてしまった結果の誤検知であると考えられる．

エラーコード解析ありの場合，検知の精度は 8.7% となっていた．これは，関数間解析で，カウンタなどの変数で解放を管理している時，一部のコードパスで解放処理が必要ないないため全体として解放情報が正しく伝搬しなくなってしまうなどの理由が挙げられる．実際にこれらの情報を正しく扱うためには変数同士のコンテキストを理解する必要がある．そのため，正しく解析するのは難しい．

```
1  int process_system_preds(...)
2  {
3    struct event_filter *filter = NULL;
4    filter = kzalloc(sizeof(*filter), ...);
5    if (!filter)
6      goto fail_mem;
7    ...
8    process_preds(filter, ...);
9      // allocs filter->prog
10   ...
11   if (!filter_item)
12     goto fail_mem;
13   ..
14   return 0;
15 fail_mem:
16   kfree(filter);
17     // leak on filter->prog
18   return -ENOMEM;
19 }
```

コード 4　新たに発見されたバグ (kernel/trace/trace_event_filter.c)

5.2　発見できたバグ

本解析では，Linux Kernel で新たに 2 件のバグを発見した．どちらのバグも Linux Kernel のデベロッパによりレビューされ，バグであると確定された [19], [20].

コード 4 は，実際に Linux Kernel で見つかったメモリリークである [20]．これは trace ディレクトリにある trace_event_filter.c で発見された．このコードでは構造体 `struct event_filter` を確保している (4 行目)．その後関数 `process_preds` で構造体のメンバ `filter->prog` に動的な領域を格納している．この後の処理でエラーだった時 (11 行目)， `filter` は解放処理が必要となる．ここで，16 行目で正しく `filter` は解放しているが，そのメンバである `filter->prog` は解放を忘れている．そのためメモリリークとなる．このバグは約 2 年間存在していた．

6.　関連研究

静的解析を用いた手法　メモリリークを検知する手法として静的な解析を用いる手法がある [8], [9], [10], [14], [16]. Linux Kernel のエラー処理時のメモリリークを検知する手法として Hector [16] がある．Hector は関数内でのエラー処理のパターンに注目する．各関数のエラー処理内でどのようにリソースが解放されているかに注目し，パターンに外れて解放を行っていないものに対して警告を出す．複雑なコードパスの解析が必要ないため，効率的な解析が行える一方，全てのパターンにおいて同様にリークしていた時，未検知となってしまう．また，関数間でのコンテキストを考慮しないため，実際には確保されていない領域も警告の対象となってしまう．

コードパスの解析をおこなう手法として SMOKE [8] がある．SMOKE は Use Flow Graph を用いることである Value の状態を解析する．UFG を用いて，必ずメモリリー

クが起こりえないものを省略することで Constraint Solver が必要になるパスを減らしていく．既存の手法に比べて効果的に解析できる一方，依然として解析時間が長くなってしまう．また，構造体メンバは構造体に付随した Value として認識されないされずさらに Value として認識されないため，UFG の解析対象にならない．

動的解析を用いた手法　静的解析を用いた手法以外にも動的解析を用いた手法がある [6], [11], [17]．Kmemleak[11] は Linux Kernel で用いられるメモリリークを検知するツールである．幅広く利用できる一方，実際にメモリリークが発生しないと検知が不可能である．そのため，検知ができない，もしくは検知まで長い時間がかかってしまう可能性がある．

また，LLVM ASan も動的な手法を用いたツールである．これは Shadow Memory を用いてオーバヘッドを減らしている一方，Kmemleak と同様，実際に該当のパスを通らない限り検知ができない．

7. まとめ

本稿では Linux Kernel などの大規模ソフトウェアでよく発生するメモリリークである構造体メンバのメモリリークを示した．実際に Linux Kernel を調べたところ，過去 2 年間のメモリリークに関するパッチのうち， 半数以上のパッチが構造体メンバに関するものであった．今回は Linux Kernel を対象に静的解析のアプローチで構造体メンバのメモリリークを検知するツールを作った．各関数内で確保されている Field，また解放されている Field の情報をあつめ，構造体が解放された時にこれらの構造体メンバに該当する Field が正しく解放されているかを確認する．また関数間での解析を簡略化するためにエラーコード解析を行い，関数の返り値の扱いによって渡す情報を効率的に切り替えた．本解析で Linux Kernel で実際に新たに 2 件のメモリリークを検知できた．

謝辞　本研究は，JST，CREST，JPMJCR19F3 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] Bruening, D. and Zhao, Q.: Practical memory checking with Dr. Memory, *International Symposium on Code Generation and Optimization (CGO 2011)*, pp. 213–223 (2011).
[7] Chen, Z.: "wlcore: Fix memory leak in case wl12xx_fetch_firmware failure", https://patchwork.kernel.org/patch/10736827/ (2018).
[8] 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).
[9] Heine, D. L. and Lam, M. S.: A practical flow-sensitive and context-sensitive C and C++ memory leak detector, *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, New York, NY, USA, Association for Computing Machinery, pp. 168–181 (2003).
[10] Heine, D. L. and Lam, M. S.: Static detection of leaks in polymorphic containers, *Proceedings of the 28th international conference on Software engineering*, ICSE '06, New York, NY, USA, Association for Computing Machinery, pp. 252–261 (2006).
[11] Kernel.org: Kernel Memory Leak Detctor(Kmemleak), https://www.kernel.org/doc/html/v4.17/dev-tools/kmemleak.html (2019).
[12] 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).
[13] Linux.org: Linux Kernel Document, https://www.kernel.org/doc/html/latest/kernel-hacking/hacking.html#return-conventions (2020).
[14] Padioleau, Y., Lawall, J., Hansen, R. R. and Muller, G.: Documenting and automating collateral evolutions in linux device drivers (2008).
[15] 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).
[16] 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).
[17] Shi, Q., Xiao, X., Wu, R., Zhou, J., Fan, G. and Zhang, C.: Pinpoint: fast and precise sparse value flow analysis for million lines of code, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, New York, NY, USA, Association for Computing Machinery, pp. 693–706 (2018).
[18] SRI-CSL: gllvm, https://github.com/SRI-CSL/gllvm (2020).
[19] Suzuki, K.: "block/genhd: Fix memory leak in error path of __alloc_disk_node()", https://lore.kernel.org/linux-block/20191211091049.11080-1-keitasuzuki.park@sslab.ics.keio.ac.jp/T/ (2019).
[20] Suzuki, K.: "tracing: Avoid memory leak in process_system_preds()", https://lore.kernel.org/lkml/20191211091258.11310-1-keitasuzuki.park@sslab.ics.keio.ac.jp/ (2019).
[21] 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).