Linux Kernel における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：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い．Linux Kernel では過去 2 年間のメモリリークバグのうち，約 54.6 % がこのバグであった．本稿では，これらのバグに注目し，静的解析を用いてバグ検知をするツールを提案した．構造体メンバの解放責任を判定するためにそれぞれのメンバの解放回数に注目して解析を行なった．もっとも簡単なケースで 207 件の警告が出た．

1. はじめに

ソフトウェアには数多くのバグが存在する．Linux Kernel のような成熟したソフトウェアにも多量のバグが定期的に存在する [3]．

バグの一種にメモリリークが存在する．これは，動的に確保されたメモリ領域を解放せず，かつ参照している変数が無いことにより生じるバグである．メモリリークはバグの中でもソフトウェアエージングの一因とされていて，時間経過とともに徐々に現れてくるバグの一種である．そのため，メモリリークは発生してからすぐに見つけるのは難しい．

メモリリークの中でも，構造体メンバのメモリリークは特に検知が難しい．これは，動的に確保されたメモリ領域を構造体メンバに格納し，そのまま構造体自体を解放してしまうなどの状況で発生する．メモリリークを検知するツールは様々な研究が行われているが，既存の手法では構造体メンバのメモリリークを検出するのは難しい [1], [4]．

本稿では，実際にどの程度 Linux Kernel で構造体メンバのメモリリークが存在するかを示す．そのために，はじめに Linux Kernel の修正パッチを確認し，どの程度過去にバグが存在していたかを示す．そしてこれらの結果を元に，構造体メンバのメモリリークを検知するツールを作成し，実際に Linux Kernel を解析し，現在のバージョンでどの程度バグが存在するかを示す．

2. 調査

実際にどの程度構造体メモリリークが存在するかを示すため，Linux Kernel の過去 2 年分のパッチを調査する (計 2,462,924 件)．今回は Linux Kernel version 5.3-rc4 (commit d45331b00ddb179e291766617259261c112db872) を使用し，'memory leak' のキーワードを含んだものを調査する．

```
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 }
```

コード 1　Bug found in Linux Kernel Git Log

表 1 はこの結果を示したものである．メモリリーク関連のパッチのうち，54.6 % 以上のものが構造体のメンバのメモリリークであった．また，各年の割合を見ても同程度存在していることが確認できる．

コード 1 は本調査で発見された修正パッチである．このコードはデバイスドライバ内で発見されたもので，エラー処理によるメモリ領域の解放忘れである．この関数呼び出し以外は `goto out` でのエラー処理で問題なかった．そのため，この部分も同様にエラー処理を行なっていた．しかし，`wl1271_setup` でメモリ領域を確保しているため，それ以降のエラー処理ではこれらで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

[1] 慶應義塾大学
Keio Uniersity

表 1　Linux Kernel 内で発見されたバグの割合

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

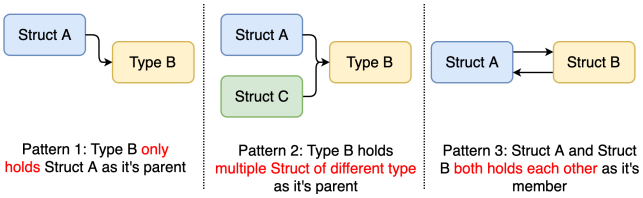

図 1　Patterns of authority

ため，メモリリークが発生した．

このバグは kmemleak[2] を使用して発見された．実際にこのバグが発見されるまでに 6 年程度かかっていた．ここから，動的解析の手法ではバグは発見できるものの，時間がかかってしまうということがわかる．そこで，今回は静的解析のアプローチを用いてバグ解析を行う．

3. 提案

本稿では，Linux Kernel における構造体メモリリークを解析を行う．これを行うために，ソースコードを静的解析するツールを作成する．

構造体メンバのメモリリークを解析する上で，課題となるのが解放責任の所在である．構造体メンバは複数の他の構造体から共有されている可能性がある．そのため，解放責任を正しく判断する必要がある．

構造体メンバが取りうる状態として三つの状態がある．図 1 はそれぞれの状態を示している．最も簡単なものは pattern 1 である．このパターンは，型 B が構造体 A の構造体メンバであり，他の構造体ではメンバとして持たれていない状態である．これはすなわち，型 B の解放責任は常に構造体 A にあると考えることができる．

これに対して，Pattern 2, Pattern 3 はより複雑なパターンである．それぞれ解放責任が型ベースでは判断できないものである．Pattern 2 は型 B が複数の構造体からメンバとして認識されているパターンである．これは，どの構造体に解放責任があるかわからないため，判断が難しくなる．また， Pattern 3 は構造体 B, 構造体 A それぞれがお互いをメンバとして持っているパターンである．

今回はもっとも簡単な Pattern 1 に注目し，解析を行なう．検知をするため，まずはじめに各構造体に関するマップを作成する．このとき，構造体のメンバで，他の構造体には含まれていないものをマークする．このマークされた構造体メンバに関して実際に情報を集める．

マップが作成されたのち，ソースコードを実際にみて，解放されている変数の情報を集める．このとき，解放された変数が構造体であった場合，のちにメンバの解放判定を行うためリストに追加する．

最後に，解放された構造体のリストと解放された変数をマッチさせて，解放されていないメンバについて，解放責任があったら，警告をだす．

4. 現状

本稿では，最も単純な Pattern 1 での解析を行なった．現状 271 件の警告が出ており，そのうち 61 件はバグ候補である．発生している誤検知としては，グローバル変数などが格納されているそもそも解放の必要のないメンバであるケースと使用後にメモリプールに戻しているケースがある．

5. 関連研究

メモリリークの解析の研究として様々な研究がある [1], [4]．WHOOP はメモリリークを Path-Sensitive Analysis によって解析する手法を取っている [1]．

Hector はメモリリークの中でもエラー処理に特化して解析を行なっている．これらは関数内でのエラー処理のパターンに依存するため，コード 1 のような関数内で全てのエラー処理が `goto` で書かれているようなケースでは正しい解析が難しい．

6. 今後の予定

本稿では，もっとも簡単なケースで実験を行い， 207 件の警告が出た．今後はより複雑なケースである Pattern 2 と Pattern 3 での解析を行い，結果をまとめたい．

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] 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).

[2] Kernel.org: Kernel Memory Leak Detctor(Kmemleak) (2019).

[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, New York, NY, USA, ACM, pp. 305–318 (2011).

[4] 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).