Vol.2018-SE-198 No.27
2018/3/10

# 高脅威メモリリークのバイナリレベル動的検出法

小泉 雄太　荒堀 喜貴　権藤 克彦

**概要**：メモリリークは，不要なオブジェクトが将来増えることはない低脅威リークと，不要なオブジェクトが将来増え続ける高脅威リークに脅威度の観点から分類できる．この分類は一定の基準でまとめられたオブジェクト群（グループ）にも適用可能である．従来の Staleness 解析のような全てのオブジェクトに対して特定の指標でリークの評価/報告をするリーク検出手法では，高脅威リークが低脅威リークの報告に埋没する可能性や，高脅威リークの即時判定ができないなどの問題がある．また，バイナリレベルでの解析においては，型情報のようなグループ化に適した情報の取得が困難な現状がある．

本研究の提案手法である Pikelet は，バイナリコードを対象に高脅威のリークを高精度に検出することを目的とした動的メモリリーク検出手法である．高脅威リークの漸次的な特性から，グループサイズの成長過程を測定することで高脅威リークのグループを高精度で検出する．また，バイナリ解析で実現可能なグループ化の手段としてオブジェクト割り付け時の calling context を用いる．オブジェクトのグループ化と，グループの成長から脅威度を導出することで，Pikelet は新たに生成されたオブジェクトの危険度を即時判断し，プログラムに差し迫った脅威をより正確に報告する．実用プログラムを対象とする実験の結果，Pikelet は既存研究に比べて高脅威リークオブジェクト群の検知において精度の向上を示した（同一実行内での計測結果の平均で Recall は 22 ポイント，Precision は 80 ポイントの精度向上を達成した）．また，実行オーバーヘッドは既存手法と同程度に収まることを確認した．

## 1. 問題背景

### 1.1 メモリリークとは

昨今の主要なプログラミング言語は，メモリの動的割り付け (allocation) によって柔軟なオブジェクトの確保を可能にしている [5]．オブジェクトは割り付け関数（malloc や new）によってヒープ領域内に動的に確保される．割り付け関数は確保したオブジェクトの先頭アドレス（ポインタ）を返す．割り付けられたオブジェクトは自動では解放されないため，明示的に解放関数 (free や delete) を使って解放する必要がある．この解放が割り付けと正しく対応していない場合[*1]にメモリリークが発生する．

メモリリークの主な症状はプログラムのパフォーマンスの低下，クラッシュの誘発などである．Web サーバ/データベースサーバに代表される長期間の実行が要求されるプログラムにとっては，メモリリークは死活問題となる．プログラムのパフォーマンス低下やクラッシュという重大な障害を引き起こすにも関わらず，それら障害の原因がメモリリークであると特定することとリークを再現することは困難を極める．メモリリークの主原因は開発者のメモリ管理の不備，より正確には，メモリ解放の書き忘れである．しかし，大規模プログラムの例外/エラー処理のように複雑な制御フローを開発者が詳細に理解し，全ての制御フロー上でメモリ解放を漏れなく記述することはあまりにも困難である．

### 1.2 既存研究

メモリリークの自動検出を目的とする研究がこれまでに数多く提案されてきた．本節では，これらのうち本研究に最も関連する研究を紹介する．

SWAT[4] はオブジェクトの *Staleness* を評価し，それが設定された述部（*Staleness predicate*）を満たせばリークと判定する．*Staleness* はオブジェクトへの最終アクセスからリーク測定までに経過したクロック数で表現される．オブジェクト個々を追跡することは大きな実行オーバーヘッドを伴うという観点から，[4] ではサンプリング技術によって実行オーバーヘッドの削減を達成した．オブジェクト個々に対する *Staleness* 評価は，どんなリーク (仮にそれが差し迫った脅威) でも一定時間の経過が必要になる．また，述部は絶対値的に設定されるため，その精度はプログラム特性に大きく左右される．故に汎用性に優れていると

†1 現在，東京工業大学
Presently with Tokyo Institute of Technology

*1 例えば，例外処理によって割り付けられたオブジェクトの解放パスへ到達しないなどがある．https://cwe.mitre.org/data/definitions/401.html

は言えず，各プログラムに最適な述部を使用者が選択する必要がある．

Cork[6] はオブジェクトを型情報によりまとめ，garbage collection（GC）後の個々の型についてサイズの増加を測定するアプローチを用いた．GC 間での各型の合計サイズの成長を測定することで，ヒープを圧迫している型を特定/報告する．より脅威度の高いリークを検知するのに適しているが，成長を見せないリークの検出に対しては積極的ではない．

### 1.3 本論文の貢献

本研究の提案する Pikelet は，メモリリークを脅威度の観点から分類し，高脅威のリークをバイナリレベルで高精度に検出することを目的とした動的リーク検出手法である．さらに Pikelet は，脅威度の低いメモリリークについても積極的に判定を行い，潜在的な危険を検知する．Pikelet はバイナリレベルでの脅威度分類を実現すべく，割り付け時の calling context に基づきオブジェクトをグループ化する．オブジェクト群の合計サイズの増加が続くグループに対し，高脅威リークの判定を与える．一方で増加もせず，アクセスも見られないグループに対しては低脅威リークと判定する．新たに生成されたオブジェクトがリークしているグループに所属する場合，そのオブジェクトが孕むリークの危険性を即時判定することが可能になる．

よって本研究の貢献は以下の点である．

- 既存のリーク検出手法は脅威度の考慮が不十分であるという問題の提起
- バイナリコードを対象に，脅威度に応じてリークを検出するアプローチの提案
- calling context に基づくオブジェクトのグループ化およびグループ単位での成長（Growth）と Staleness の測定によりバイナリレベルでの脅威度を考慮したリーク検出が実装可能と示した点
- 提案手法の精度・効率面での有効性を実験により定量化した点

## 2. 問題設定

### 2.1 既存研究の問題点

メモリリークは脅威度の観点からの分類が可能である．不要なオブジェクトが将来増えることはない低脅威リークと，不要なオブジェクトが将来にわたり増え続ける高脅威リークである．

Web サーバに代表されるメモリリークが致命的となるプログラムは図 1 のような入力ループ構造を必ず持つ．高脅威リークが発生する可能性があるのはこのループ内部である．特定の入力によってリークオブジェクトが生成されるとき，ループを繰り返すことで徐々にヒープメモリが圧迫されていく．一方でプログラムのスタートアップ時に生

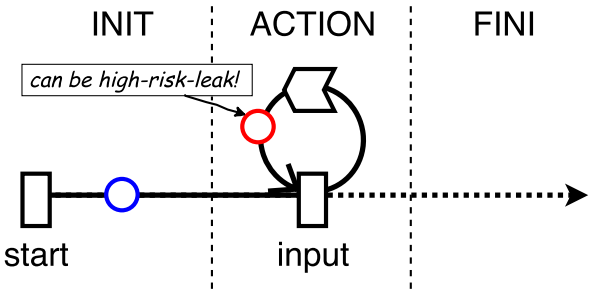


図 1 無期限実行が要求されるプログラムの簡易モデル

成されるようなオブジェクトは，通常ループ外に存在するためリークになったとしても低脅威となる．

従来の Staleness 解析のようなオブジェクト個々に対する特定の指標 (例えば Staleness のような) でリークを一元的に評価/報告する手法では，低脅威リークと高脅威リークの区別は不可能である．これは，一元的評価がリークか否かの二択を主眼に置いているためである．結果として，真に脅威として開発者に報告されなければならない高脅威リークが低脅威リークと混ざり埋没する可能性が高い．また，とりわけ Staleness 解析ではどんなリークオブジェクトも時間経過を待たねばならず，即時判定ができない．さらに，[4] のモチベーションに示されているように，個々のオブジェクトを監視することは実行オーバーヘッドが増大しがちである．

これらを解決するアイデアの一つがオブジェクトのグループ化である．グループに対して脅威度を求めることで，高脅威のグループに属するオブジェクトをまとめて判定することが可能となり，検出の効率化も図れる．高脅威リークを検出する手法として，Java/C#を対象言語とした [2], [6] が存在するが，グループ化は型情報により実現している．残念ながら，C/C++のバイナリ解析においては型情報が欠落している[*2]ため，これらの手法をそのまま再現することは不可能である．

本論文が指摘する既存研究の問題点は次の通りである．

- バイナリコードを対象に脅威度を考慮したリーク検出ができない

### 2.2 問題点に対する改善案の仮説

**高脅威リークの検出技術の不足** オブジェクトたちは個々に独立しているわけではない．同一の用途を想定して生成されているオブジェクト群（グループ）が存在する．つまり，あるオブジェクトの一つがリークしたのなら，同一用途の他全てのオブジェクトにリークの可能性がつきまとう．リーク判定は個々のオブジェクトではなく，共通する背景を持つオブジェクト群に対して行うべきである．

また，リークが症状として外部に現れるということは，

[*2] デバッグ情報を付与すれば取得可能だが，そのためには通常リコンパイルが必要である．そのような状況ではバイナリ解析の意義は薄れる．

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define INIT 100
4  int *p=NULL;
5  int *q=NULL;
6  void *malloc_wrapper(unsigned int size){
7      void *ptr = (void *)malloc(size);//tgt
8      return ptr;
9  }
10 void init(){
11     p = (int *)malloc_wrapper(4);
12     *p = INIT;
13 }
14 void action(int a){
15     q = (int *)malloc_wrapper(4);
16     *q = a;
17 }
18 void input(){
19     int a;
20     while(1){
21         scanf("%d", &a);
22         action(a);
23     }
24 }
25 void fini(){//never call in this program...
26     free(p);
27     free(q);
28 }
29 int main(){
30     init();
31     input();
32     fini();
33     return 0;
34 }
```

**図 2** sample.c

ヒープを圧迫しているグループがいるということの裏付けでもある．グループ数が有限ならば，いずれかのグループがヒープを圧迫しているのである．そして，メモリリークによるヒープの圧迫は突発的ではなく漸次的である．すなわち，いずれかのグループがヒープを圧迫していく過程が存在する．グループの成長，要するにグループを構成するオブジェクトたちの合計サイズの増加を計測 [6] することで，高脅威リークをあぶり出すことが可能であると考える．

### 2.3 仮説に対応するコード例とその説明

図 2 に示したソースコードは，入力ループを伴う Web サーバのような無期限の実行を期待されるプログラムを抽象化した (図 1) コードである．このプログラムで生成されるオブジェクトは 2 種類である．一つは 11 行目で呼ばれる malloc（のラッパー関数）によって確保される[*3]オブジェクトで，8 行目に値を格納された後は 26 行目で解放されるのを待つだけだが，入力ループによって解放は無期限に延期され続けるためリークとなる．もう一つは 15 行目の malloc（のラッパー関数）によって確保されるオブジェクトで，こちらは入力ループ内で解放されずに，入力毎に生成される．ポインタ q による参照がなくなったオブジェクトはどこからも参照されず，リークとなる．もちろん，入力ループの内部で生成されたからといって，高脅威リークと見なされるわけではない．

[*3] 実際に確保されるのは 7 行目のラッパー関数内である．

既存の Staleness 解析では，一定時間が過ぎた後に，前者も後者も順々にリークとなったオブジェクトを報告するだけである．しかし，その判定にかかる時間の間にもリークオブジェクトは増え続ける．Staleness 解析では，生成直後のリークオブジェクトを判定することは不可能である．

## 3. 提案手法

### 3.1 概説

***Grouping*** 我々はオブジェクトを allocation 時の calling context によってグループ化した．以下，グループという単語は同一の allocation calling context（ACC）を持つオブジェクト群と定義する．例えば図 2 からは 11 行目を経由するグループと 15 行目を経由するグループの 2 つのグループが得られる．グループ単位に脅威度を導出することで生成されたオブジェクトの即時リーク判定が可能になる．calling context はアドレス値から導出が可能であるため，動的バイナリ解析でも実現可能である．また，アロケーション時，同一のポインタに代入される性質上，同一 ACC を持つオブジェクト群は同一の型を持つ．すなわち型情報によるグループ化を近似的に達成できる．

***Growth approach*** メモリリークが生じている場合，ヒープメモリを圧迫している原因となるグループ（group/-groups）が存在している．そしてメモリリークは突発的なバグではなく，漸次的である．すなわち，あるメモリリークの原因となるグループは時間をかけてそのオブジェクトサイズの総和が増加する．ここで我々はグループの各オブジェクトサイズの総和が時間経過で増加することを成長 (growth) と定義する．この成長をスナップショットを通して検知を試みる．スナップショットとは特定のタイミングで各グループの状態を計測することを指す．スナップショットについての詳しい解説は次節で行う．

図 2 のプログラムに対して 3 回目の入力直後，7 回目の入力直後，11 回目の入力直後にスナップショットの実行を仮定した場合のヒープの状態と計測されるグループの成長を図 3 に示す．このとき得られるグループは前述の通り 2 つである（init() 経由と action(経由)）．

成長が続くグループ（例えば図 3 の via action()）を高脅威リークと判定し，開発者へ報告する．前述の通り，グループは ACC によって区別されるので，開発者は allocation

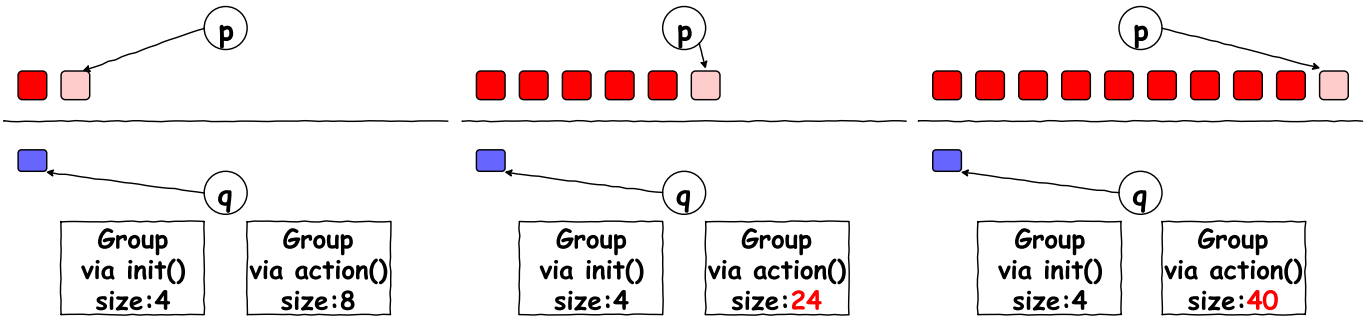


図 3　グループの成長

の位置を容易に特定することができる．

一方で成長せずに残り続ける場合がある（例えば図 3 の via init()）．これらは低脅威リークであり許容が可能である場合もある．だが，入力ループ内に存在していながら低脅威リークと判定されるグループの中には潜在的に高脅威となるものが存在する．このような成長しない/成長が乏しいリークパターンについても積極的に判定を行う．詳しくは次節で説明をする．

### 3.2　リーク判定

***Snapshot*** スナップショットとは，特定のタイミングでの各グループの状態を計測することと定義される．我々のアプローチでは，スナップショットのタイミングは精度に大きく関わってくるので，重要な議題である．我々はスナップショットを一定数の外部入力後に実行すると定める．以下に外部入力数を選んだ理由を述べる．

前提として，最も望ましいタイミングは GC の実行時であった．判定アルゴリズムの基礎となった Cork[6] は Full Heap GC の実行を一つのタイミングとしていたためである．しかしながら，前述の通り C/C++には GC が標準サポートされておらず，また GC の実行と対応するようなタイミングは自明ではない．

他方，シンプルなスナップショットのタイミングは一定の経過命令数や経過時間毎に実行することである．しかし，プログラム特性の影響が出やすく一般性に欠ける．外部入力数によるスナップショットは，プログラム特性によるスナップショットへの影響 (特に実行間隔) をある程度緩和する．入力 1 回に対しての重みはプログラム特性に依るが，処理に必要な経過命令数や時間などよりプログラム特性に左右されやすい要素を排除が可能である．

$i$ 回目スナップショットで未解放オブジェクトを持つ各グループに対して以下の計測結果を提出する．

$$SnapshotResult_i(G) = 8 * "NotFreed"ish + 4 * "Freed"ish + 2 * Grow\_num\_of\_objects + 1 * Grow\_V^{max}$$

それぞれの述部は真偽を返す（数値的には偽なら 0，真なら 1 である）．”NotFreed”ish と”Freed”ish は G の未解放オブジェクト数と解放済みオブジェクト数の比較結果である．Grow_num_of_objects と $Grow\_V^{max}$ は前回のスナップショットと比較してオブジェクト数/合計サイズの最大値が増加したかを表す．これら 4 つの述部による結果をグループのスナップショットの履歴（$G.SnapshotResultHistory_i$:$SRH_i$）に残す．実装段階で 32 ビット，つまり直近 8 回分の履歴の保存が可能である．8 回という数値は各閾値を定める一つの基準となっている．

***Growth approach*** 前述したグループの成長を計測するために Cork[6] で用いられた RRT（Ratio Ranking Technique）を基礎とした growth rate の計算を行う．我々が対象にしているのは C/C++言語で記述されたプログラムであるため，GC が標準ではサポートされていない．そのため，Cork のように $i$ 回目の GC 後のグループサイズ $V_i(G)$ を計算式に組み込むことは不可能である．代替案として，$i$ 回目のスナップショット時のグループサイズの最大値 $V_i^{max}(G)$ を定義，使用する．これはリークが生じているならば，巨視的にはあるグループの最大サイズの成長が生じているからである．$V_i^{max}(G)$ はスナップショット時に計測/計算が可能である

レート $r_i(G)$ は単純増加し，閾値 $R_{thres}$ を超えたらグループ G は高脅威リーク（Grow）と判定される．リーク未判定のグループ内のオブジェクトが全て解放された場合はレート $r_i(G)$ と $Spawn(G)$ は初期化される．一方で $V_i^{max}(G)$ は初期化せず記録として保持することで，誤検出の抑制に利用する．

$$\begin{aligned} V_i^{max}(G) &= max(V_{i-1}^{max}(G), V_i(G)) \\ Q_i^{max}(G) &= \frac{V_i^{max}(G)}{V_{i-1}^{max}(G)} \\ p_i(G) &= i - Spawn(G) \\ r_i(G) &= r_{i-1}(G) + p_i(G) * (Q_i^{max}(G) - 1) \end{aligned}$$

***Staleness approach*** 低脅威リークについても積極的に判定を行う．成長しないグループについては，Staleness の概念を用いる．我々は新たにグループ単位の Staleness を定義/使用する．グループ単位の Staleness は計測結果の変化とアクセスが一切ないと推定されるスナップショット数である．

$8k-4$ 回目のスナップショットの実行時に，未解放オブジェクトを持つ各グループの SRH を参照する．直近 4 回のスナップショット結果に変化がない，すなわち SRH の 16 進数表示下位 4 桁が 0x0000, 0x4444, 0x8888, 0xcccc となるグループに，リークの容疑をかける．リーク容疑のかかったグループは $8k$ 回目のスナップショットまでに計測結果の変化が変わらず，グループ内のオブジェクトへアクセスがなかった場合，低脅威リークと判定される．

例えば，以下は 12 回目のスナップショットでの 2 つのグループの SRH を仮定したものである．

```
1 Snapshot:12
2 ACC:0x241fdba7 SRH:0x00cccccc //G1 #
3 ACC:0x6f55348e SRH:0x74474744 //G2
```

G1 は SRH のリーク容疑がかかる．G1 が容疑を晴らすには，16 回目のスナップショットまでに，それまでと異なるスナップショットの計測結果が報告されるか，あるいはグループ内のオブジェクトの一つにでもアクセスが観測される必要がある．アクセス監視は [4] で提案された Adaptive Bursty Tracing を用いてサンプリングを行う．高頻度で実行されるルーチン（関数）でのアクセス監視を間引くことで精度を保ったまま，実行オーバーヘッドを削減する．グループ内のオブジェクト全体のアクセスの論理和を取ることで，低レートのサンプリングでもグループに対するアクセスの見落としを抑制することが可能である．

***Intermittent approach*** これまでに解説をしたそれぞれの判定 (Growth, Staleness) は，記録される 8 回のスナップショットの測定結果を基準に分類をされている．だが，長い非成長期間を有する成長パターン，つまり Growth 判定と Staleness 判定の両方を満たし得るパターンが存在する（同時に判定されることはない）．しかも厄介なことに，現実世界で問題になるのはこのパターンであり，発見に必要な時間が長期化する傾向にある．

これらの特徴は，間欠泉（intermittent spring）のように不定期な間隔で成長することである．成長とアクセスがない長い期間が存在すれば Staleness 判定により”低脅威”リークと判定されてしまう．だが実際にはこれは誤りである．そこで，Staleness 判定を受けた後に成長をする，あるいは Staleness 判定を受けた時点で一定の成長レート ($r_i(G) > R_{thres}/2$) を保有しているグループについては，脅威度を再評価する．各グループの状態を示す遷移図を図 4 に示す．再評価を待っている状態が PRE-INTERMITTENT である．Staleness 判定を受けているので，リーク判定の基準は満たしているが，低脅威と判断はできないため，PRE-INTERMITTENT の脅威度は低脅威でも高脅威でもない（実験では便宜上低脅威として扱っている）．STALE 状態を経由してから成長をしたグループは，PRE-INTERMITTENT へ遷移後に閾値 $R_{thres}$ の半分の値を超えた段階で高脅威（INTERMITTENT）と再定義される．一方で ACTIVE 状態から遷移したグループは，閾値の条件を満たしているので，PRE-INTERMITTENT へ遷移後に成長が認められたら高脅威（Intermittent）と再定義される．閾値の基準を下げることで，低頻度の成長に対して素早い判定を行う．

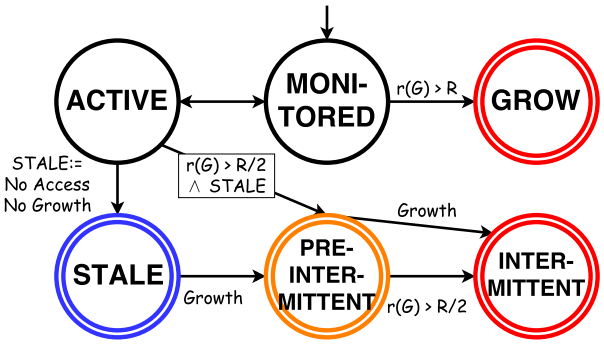


図 4　状態遷移図

## 4. 実験

### 4.1 Methodology

Pikelet と SWAT を Pin[8] を用いて実装/再現した．また，ACC を PCC[1] により表現/管理を行った．PCC[1] は端的に言えば calling context のハッシング表現手法である．実験は MacOS 上に構築した Linux 仮想マシン上で行った．実験環境を表 2 に記載する．

精度実験は複数のリークパターンを内包したスモールベンチと，lighttpd-1.4.19 を対象に行った．実行オーバーヘッドの計測実験には異なるスモールベンチと FFmpeg[*4]を対象とした．

また，一部のプログラム（lighttpd）についてリークの注入を行った．リークの注入は [7] で用いられた free を間引く方式を用いた．間引く割合を 20%とした．外部入力のないプログラムは実行命令数を計測し，いくつかの値で等分した間隔でスナップショットを実行/計測した．

### 4.2 精度評価

以下に精度指標を定義する．オブジェクト数を単位として Recall/Precision を定義する．

また，高脅威リークを積極的かつ高速に判定をするというコンセプトから，Risk Oriented（RO）を冠した ROTP，ROFP，ROFN を定義する．報告されたリークのうち，高脅威リークと正しく判定したオブジェクト数が ROTP，誤検出を ROFP と定義する．また，プログラム終了時までにリークと判定されなかった未解放オブジェクト/グループについては，リークとしては報告しない．

True の基準は手動で各プログラムに対する正解セットを用意し，スナップショット毎の各精度指標を評価する．”

[*4] https://www.ffmpeg.org

| | | | | Groups | | | Objects | | |
|---|---|---|---|---|---|---|---|---|---|
| | Static | Dynamic | # of | # of | | | # of | | |
| program | LOC | instructions | Snap | Group | Leak | High/Low | Object | Leak | High/Low |
| Leak_SEINL | 239 | 95,756,868 | 64 | 4 | 3 | 2/1 | 8,800 | 7,800 | 6,800/1,000 |
| lighttpd-1.4.19 | 54,698 | 35,272,515 | 50 | 445 | 94 | 18 /76 | 7,426 | 2,670 | 2,541/129 |

**表 1** 精度実験：プログラム情報

| | VM Host | VM Guest |
|---|---|---|
| OS | High Sierra 10.13.2 | Ubuntu14.04<br>Linux 3.13.0-137-generic |
| Processor | Intel Core i7 7567u | Intel Core i7 7567u |
| Core / Freq | 2 / 3.5GHz | 2 / 3.5GHz |
| Memory | 16GB | 3.9GB |
| Storage | 500GB | 101.3GB |
| | OS Emulator | |
| name | VirtualBox | |
| version | 5.1.30 r118389 (Qt5.6.3) | |

**表 2** 実験環境

高脅威”の基準は，Pikelet によって高脅威のと報告されたグループに適用し，比較対象の SWAT が報告するリークは全て高脅威と仮定する[*5]．

$$RiskOrientedPrecision = \frac{ROTP}{ROTP + ROFP}$$

$$RiskOrientedRecall = \frac{ROTP}{ROTP + ROFN}$$

また，示されるグラフの縦軸は精度指標（Recall など）を表し，横軸は $i$ 回目スナップショットを表す．すなわちスナップショット毎の精度指標の遷移をプロットしたものである．左側が通常の Recall/Precision，右は ROReca-l/ROPrecision である．また，実線は Recall/RORecall，破線は Precision/ROPrecision である．リークの報告がない場合においては，すべての精度指標の値は 0 とする．

instructions は検査対象プログラムの実行命令数である．Group/Object は報告された総グループ/オブジェクト数であり，Leak は報告されたグループ/オブジェクトの内正解セットと合致した数で，High/Low はその内の脅威度の内訳である．精度の向上（表 3）は Pikelet の精度指標値と SWAT（IdleGt10Million[4][*6]）の精度指標値との差（ポイント）である．Ave は時間平均での精度差を示し，Max は精度改善の最大値である．

#### 4.2.1 small benchmark

Leak_SEINL は自作したスモールベンチプログラムである．4 種類のグループ（高脅威 2，低脅威 1，非リーク 1）を内包している．高脅威リークは 3 章で解説した Grow と Intermittent が一つずつである．結果を図 5 に示す．外部入力を設けていないため，実行命令数から 64 等分した命令数毎にスナップショットを実行した．

赤が Pikelet，青が SWAT（IdleGt10Million），緑が SWAT（IdleGt50Million）である．述部の比較のために IdleGt50Million[*7]を実装した．

SWAT の両述部は低脅威リークグループのオブジェクトは述部条件を満たせば正しくリークと判定できた．だが，サンプリングによってアクセスを見逃すことで非リークグループのオブジェクトをリークと誤判定した．また，述部条件を満たすまでに時間がかかるため生成間もないリークオブジェクトについては見逃した．IdleGt50Million は Precision/ ROPrecision の上昇率は高いが，短時間の実行では償却が間に合わなかった．

Pikelet はグループ単位の判定により新たに生成されるオブジェクトについても即時判定が可能で見逃しを抑制した．また，アクセスの論理和を取ることでサンプリングされてもアクセスを判別し，誤検出を抑制した．47 回目のスナップショットで Intemittent を判別し，全ての指標について 100%を達成した．

#### 4.2.2 lighttpd-1.4.19

リークの注入によって入力ループ内に存在する allocate-deallocate の均衡が崩れるため，意図的に高脅威リークを発生させることができる．Pikelet/SWAT を適用して起動し，画像/テキストを含む html ページの閲覧を約一時間行った．また，スナップショットを lighttpd へのページ要求 10 回毎に行うように設定した．SWAT の述部は実験時間を考慮し IdleGt10Million を選択した．正解セットはサンプリングを行わずに実行した記録から生成した．

赤が Pikelet，青が SWAT である．グラフ（図 6）にプロットしているのは 50 回目までのスナップショットの記録で，無期限実行の観点から正常終了時を考慮していない．

図 6 から分かるように，Pikelet は全ての指標で SWAT より常に高い精度を示した．着目すべきは Precision と ROPrecision である．残念ながら Precision は Pikelet も決して高い値を出せなかったが，ROPrecision は高い精度を示している．これは Pikelet の誤検出は低脅威リークでのみ生じているという事実を示している．また，SWAT は ROPrecision の低さは，高脅威リークの報告が低脅威リー

[*5] 一度高脅威/低脅威問わずリークと判定されたオブジェクトは誤検出と検出器側で判定が可能な場合（例えば，STALE と判定した後にアクセスがあった場合など）でもリーク判定のフラグを立てたままとする（つまり，FP から TN へ変化することはない）．ただし，低脅威から高脅威への脅威度の更新は可とする（つまり，ROFN から ROTP へ変化することはある）．

[*6] 1000 万命令を超える期間アクセスが確認できなかったらリークと判定する述部．

[*7] 5000 万命令を超える期間アクセスが確認できなかったらリークと判定する述部．

| | Recall | | Precision | | RORecall | | ROPrecision | |
|---|---|---|---|---|---|---|---|---|
| program | Ave | Max | Ave | Max | Ave | Max | Ave | Max |
| Leak_SEINL | +10.95 | +27.32 | +16.99 | +42.37 | +13.18 | +50.08 | +34.37 | +79.67 |
| lighttpd-1.4.19 | +20.68 | +51.88 | +7.74 | +40.03 | +22.29 | +61.04 | +80.37 | +100.00 |

**表 3** 精度実験：精度の向上（単位：ポイント）

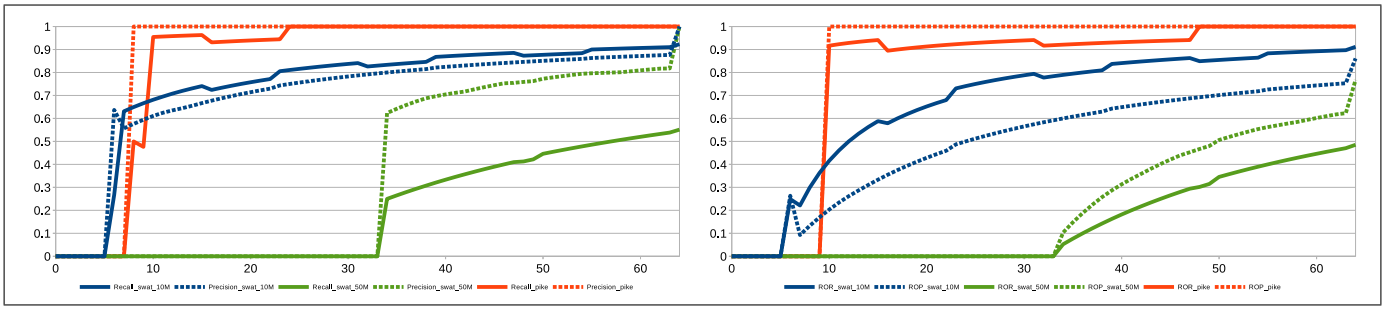


**図 5** Accuracy vs SWAT on small benchmark

クの報告に埋没していることを示している．

表 3 から分かるように，総じて SWAT と比較して全スナップショットの平均値で精度の向上を示したが，より精密に精度を比較するのであれば，現実的には検査器の精度が乱高下しない（と予想される）十分な実行時間を経た後の精度を計測/比較をすべきである．

### 4.3 実行オーバーヘッド評価

Pikelet が十分低オーバーヘッドで実行が可能であることを確かめるため，実行オーバーヘッドの測定を行った（図 7）．グラフは左から native，inscount，Pikelet，SWAT の順でプロットした．native は Pin を介さずに実行した場合であり，inscount は実行命令数を数える pintool を計装して実行した場合である．inscount は Pikelet，SWAT 内でも経過命令数を計測するために実装されている．それぞれ対象プログラムに対して 5 回計測し，その平均値を採用した．グラフの y 軸は，SWAT の時間を 1 とした正規化された値である．また，実行時間を 32 分割したタイミングでスナップショットを実行した（表 4）．

Leak_S は自作したスモールベンチプログラムである．Leak_S は一万回の malloc と個々のオブジェクトに対するアクセスを数回行った後，オブジェクトへアクセスしない期間（実行時間全体の約 9 割）を経て，生成された一万個のオブジェクトを全て未解放の状態のままにして終了する．

FFmpeg は C 言語で記述された動画/音声の編集用フリーソフトウェアである．プログラムの性質上，オブジェクトのアロケーション，アクセス，実行命令数が多いため測定用として採用した．約 10 秒の音声データをコピーするという操作に対して測定を行った．

一般に SWAT/Pikelet において最大のオーバーヘッドを生じるのはアクセス監視である．図 7 に示されるように Leak_S と FFmpeg の両方で Pikelet の方がオーバーヘッドが小さくなった．これはスナップショットのコストは Pikelet の方が小さくなることで，グループの管理などの追加コストが償却されたと考えられる．

以上から，Pikelet は SWAT と同程度のオーバーヘッドにより実現しており，多数のオブジェクトが少数のグループへ集約される場合，SWAT よりも僅かに高速化される．アクセス監視が低脅威リークの判定に使用されている点，また，グループ全体のアクセスの論理和を取ることで低脅威リークの誤検出を抑えられる事実から，Pikelet はサンプリングレートの下限を引き下げることで精度を維持したまま実行オーバーヘッドをより削減できる可能性を有している．

## 5. 関連研究

***Staleness*** SWAT[4] はオブジェクト個々の Staleness（最終アクセスからスナップショットまでの経過クロック数と定義）を評価し，述部（Staleness predicate）の条件を満たしていた場合はリークと報告する．サンプリングによってメモリアクセスの実行オーバーヘッドを軽減させる．[7] は SWAT の述部を SVM（Support Vector Machine）を通して学習により定義することを試みた．学習によって得られた各オブジェクト群への述部の条件を満たしたオブジェクトをリークと判定する．

Leak Pruning[2] は Staleness を最終アクセスから経過した Full Heap GC の数[*8]と定義し，それに加えて型情報から最もヒープを圧迫していると判定された型間のエッジ（ポインタ）を切り，実際にオブジェクトを解放させる．このアプローチはリークの精密な検出よりも，リークによって終了してしまうプログラムの延命に焦点を当てている．

***Growth*** Cork[6] は型を一つのグループとし，GC 間のグループの成長を RRT（Ratio Ranking Technique）によって評価する．また，TPFG（Types Points-from Graphs）

[*8] 正確にはヒープ分割数の対数．

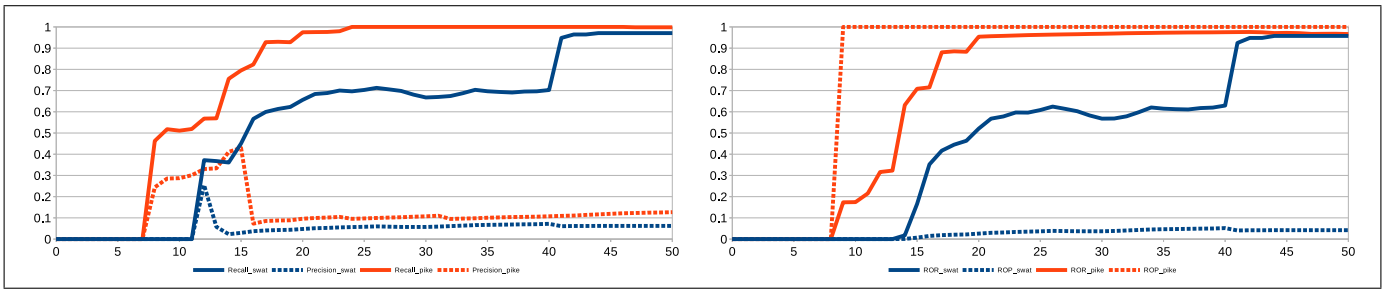


図 6　Accuracy vs SWAT on lighttpd-1.4.19

| program | LOC | instructions | # of Snap | # of Groups | # of Objects | average time(s) native | average time(s) inscount | average time(s) Pikelet | average time(s) SWAT |
|---|---|---|---|---|---|---|---|---|---|
| Leak_S | 107 | 391,482,939 | 32 | 1 | 10,000 | 0.3368 | 1.2514 | 5.085 | 5.1782 |
| FFmpeg | 1,268,917 | 3,369,223,075 | 32 | 488 | 5,168 | 0.56 | 11.2276 | 793.019 | 801.0892 |

表 4　実行オーバーヘッド実験

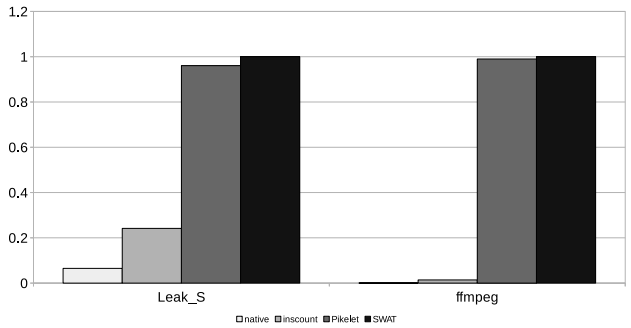


図 7　run-time overhead

[6] により型やエッジの情報を管理しておくことで，リークに関係のあるオブジェクト周辺の情報をレポートとして提供する．

***Taint analysis*** LEAKPOINT[3] は汚染解析による割り付けられたオブジェクトへのポインタを伝搬演算し，参照数を計測することで到達不可能なオブジェクトを特定する．つまるところ，C/C++言語に適用可能な Reference Counter Garbage Collection に近い機能を実現した（実際に到達不可能となったオブジェクトに対する解放機能も付いている）．

## 6. 結論

Pikelet は Growth，Staleness の 2 種類の尺度でメモリリークを脅威度の観点から分類し，高脅威メモリリークを高精度で検出する手法である．また，Pikelet の実装に際して割り付け時の calling context によるオブジェクトのグループ化を行い，それが実行バイナリ解析でも実現可能であることを確かめた．

Pikelet は既存手法（SWAT）よりも早期に高脅威リークを検出し，低脅威リークについても精度で上回る（時間平均で 10 ポイント以上の向上）．ただし，より厳密に精度を比較するのであれば，十分な実行時間を減る実験を行う必要があるであろう．実行オーバーヘッドは SWAT とほぼ同程度に収まり，多数のオブジェクトが少数のグループへ集約される場合は SWAT よりも僅かに高速（-1%程度）になることを確認した．

## 7. 参考文献

### 参考文献

[1] Bond, M. D. and McKinley, K. S.: Probabilistic Calling Context, *SIGPLAN Not.*, Vol. 42, No. 10, pp. 97–112 (2007).
[2] Bond, M. D. and McKinley, K. S.: Leak Pruning, *SIGARCH Comput. Archit. News*, Vol. 37, No. 1, pp. 277–288 (2009).
[3] Clause, J. and Orso, A.: LEAKPOINT: Pinpointing the Causes of Memory Leaks, *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, New York, NY, USA, ACM, pp. 515–524 (2010).
[4] Hauswirth, M. and Chilimbi, T. M.: Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling, *SIGPLAN Not.*, Vol. 39, No. 11, pp. 156–164 (2004).
[5] Jones, R., Hosking, A. and Moss, E.: *The Garbage Collection Handbook: The Art of Automatic Memory Management*, CRC Applied Algorithms and Data Structures, Chapman & Hall (2012).
[6] Jump, M. and McKinley, K. S.: Cork: Dynamic Memory Leak Detection for Garbage-collected Languages, *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, New York, NY, USA, ACM, pp. 31–38 (2007).
[7] Lee, S., Jung, C. and Pande, S.: Detecting Memory Leaks Through Introspective Dynamic Behavior Modelling Using Machine Learning, *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, New York, NY, USA, ACM, pp. 814–824 (2014).
[8] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J. and Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, New York, NY, USA, ACM, pp. 190–200 (2005).