# 広範な実用 C プログラムに適用可能かつ高精度な動的境界検査ツール*

荒堀　喜貴[†a)]　　権藤　克彦[†b)]　　前島　英雄[††]

# A Precise Dynamic Bounds Checker Applicable to a Wide Range of Real C Programs*

Yoshitaka ARAHORI[†a)], Katsuhiko GONDOW[†b)], and Hideo MAEJIMA[††]

**あらまし**　C プログラムの境界検査手法は現在まで継続的に提案されている．それらのうち，実用コードとの互換性が高くかつ誤検出率の低い手法は，実行時に全有効オブジェクトの境界を heap 領域上の表を用いて管理する手法である．しかし，この手法は現状，低レベルな C プログラムに適用することができない．我々はこの問題を解決する検査機構及びその実装を提案する．実験の範囲内で，我々の検査機構に基づく境界検査器は Linux カーネルを含む広範な実用 C プログラムを高精度に検査できることを確認した．

**キーワード**　ソフトウェア工学，ソフトウェア開発環境，テスト，デバッグ

## 1. まえがき

境界違反とはプログラム実行時に有効メモリオブジェクトの境界を越えて行う不正アクセスであり，C プログラムの最も危険なバグの一つである．悪意のある攻撃者は C プログラムの境界違反を利用して重要なシステムへの攻撃や侵入を試みる場合が多い．彼らは通常，境界検査が正しく行われていないプログラムバッファへのアクセスを発見し，特殊な入力を与えて境界違反を発生させる．その結果，バッファの隣接領域に格納されたリターンアドレスや関数ポインタが不正な値に書き換わり，プログラムは不正な動作を引き起こす．CERT [1] などのセキュリティ機関は 2009 年現在も境界違反に起因する脆弱性を報告し続けている．したがって，実用 C プログラムの境界検査は依然として重要な課題である．

† 東京工業大学情報理工学研究科計算工学専攻，東京都
Department of Computer Science, Tokyo Institute of Technology, Tokyo, 152–8552 Japan
†† 東京工業大学総合理工学研究科物理情報システム専攻，横浜市
Department of Information Processing, Tokyo Institute of Technology, Yokohama-shi, 226–8502 Japan
a) E-mail: arahori@sde.cs.titech.ac.jp
b) E-mail: gondow@cs.titech.ac.jp
* 本論文はシステム開発論文である．

### 1.1 既存の境界検査手法

C プログラムの境界検査手法はこれまでに多数提案されており，様々な長所と短所をもつ．

静的手法は検査対象プログラムのソースコードを解析して境界違反が発生し得る位置を予測する．いくつかの静的手法は大規模実用 C プログラムを比較的効率良く検査することに成功している [2]〜[6]．静的手法は対象コードの境界違反を網羅的に検出できる反面，検査時間が大きい，ソースコードの大幅な変更を要求する，誤検出率が高いなどの欠点をもつ．

動的手法は検査対象プログラムに検査コードを挿入し，実際にプログラムを実行して検査を行う [7]〜[25]．静的手法に比べ，動的手法は検査の網羅性に劣る反面，検査時間が短く誤検出率が低い．様々な動的手法が提案されているが，それらは検査精度の低い方から順に次のように分類できる：

（1） static，heap，stack 領域のうち 1 種類の領域上で順次的な境界違反[(注1)] を検出するもの．

（2） 複数種類の領域上で順次的な境界違反を検出するもの．

（3） 複数種類の領域上で順次的でない境界違反や

(注1)：オブジェクトの有効領域内を順次アクセスしていき，有効領域の最上位/最下位アドレスをアクセスした直後に 1 バイト外をアクセスする境界違反．

ダングリングポインタも含む一般的な境界外アクセスを検出するもの．

（4） 複数種類の領域上で一般的な境界外アクセスを含む様々な不正メモリ操作（不正な関数ポインタ，メモリリークなど）が発生しないことを言語レベルで保証するもの．

これらの動的手法の中で（3）に該当し，実用コードとの互換性が高くかつ効率的な実装が知られている検査手法が，オブジェクト表に基づく手法である [22]〜[25]．この手法では，実行時に対象プログラムが使用する各メモリオブジェクトの情報（開始アドレス，サイズなど）を，検査コードが heap 領域上の表（オブジェクト表）で追跡管理する．また，対象プログラムの各メモリ操作の直前で，検査コードがそれらの情報を参照して操作の安全性を検査する．この手法は，static 領域，heap 領域，stack 領域などすべての領域上のオブジェクトの境界を把握でき，かつ，多くの種類のメモリ操作を検査できるため，高精度である．また，ポインタの内部表現を変更しないので検査コードと対象コードの互換性が高い．更に，自動プール割当 [26] などの最適化手法の適用が可能であるため，検査の高速化が可能である．

しかし，従来のオブジェクト表に基づく境界検査は，検査コードが多くのライブラリ関数や OS のシステムコールに強く依存しているため，それらを利用できない低レベルな C プログラムに適用することが困難である．低レベルな C プログラムには，仮想マシンモニタや OS のカーネルや組込みプログラムや標準 C ライブラリなどの多くの重要なプログラムが含まれる．更に，これらのプログラムでも境界違反は過去に何度も報告されている [1]．したがって，これらを検査できないことは深刻な問題である．

### 1.2 提案手法

そこで，我々はトラップキャッシュ機構と呼ぶ新たな境界検査機構を提案する．この機構は static 領域上の小容量かつ固定サイズのバッファを有効活用することで検査コードの実行環境依存部分を可能な限り削減する．その要点と効果は以下のとおり：

- 対象プログラムが最近アクセスしたオブジェクトの境界情報のみを static 領域上の小容量かつ固定サイズのバッファ（トラップキャッシュ）で追跡管理する．その結果，境界情報の追跡のために検査コードがメモリを動的に割り当てる必要がなくなり，検査コードのメモリ管理関数への依存部分が解消される．
- 検出報告用のバッファを static 領域上に配置し，検出した境界違反の内容をすべてそのバッファに保存する．その結果，検出報告のために検査コードが入出力用のライブラリ関数やシステムコールに依存する必要がなくなる．

トラップキャッシュ機構は，低レベルな C プログラムへの適用が可能であるという点において，従来のオブジェクト表に基づく検査方式より優れる．ただし，その利点を得ることと引換えに検査精度を犠牲にしている．特に，順次的でない境界違反の検出が不可能となるため，検査精度の観点からは，トラップキャッシュ機構は前述の（2）に該当する．

我々の先行研究 [18] では，従来のオブジェクト表に基づく検査コードが（各オブジェクトの境界情報の追跡管理のために）メモリ管理用のライブラリ関数やシステムコールに強く依存することを指摘し，それに対する解決法の原型を示した．本研究では，従来の検査コードがメモリ管理関数のほかにも多くのライブラリ関数やシステムコールに依存することを明らかにし，先行研究の手法を拡張することでその依存を解消する．これらの問題点と解決策は先行研究の範囲では明らかではなかった．

### 1.3 実験結果の要約

我々はトラップキャッシュ機構に基づく境界検査器を GCC [27] に実装し，Linux [30] や Apache [28] を含む 13 種類の実用 C プログラムを対象として各種の実験を行った．その結果，トラップキャッシュ機構に基づく検査コードは Linux カーネルを含む低レベルな C プログラムに対しても容易に適用できることが確認できた．また，境界検査を文字列に限定する最適化を行った場合，たかだか 4 kByte 程度のキャッシュサイズでそれらのプログラムが引き起こす境界違反を高精度に検出できることが分かった．更に，実行時間のオーバヘッドは平均 17%と十分に小さかった．

### 1.4 論文の構成

以降，**2.** でオブジェクト表に基づく境界検査法とその問題点を詳細に説明する．**3.** で問題点に対する解決手法としてトラップキャッシュ機構を提案する．**4.** で実験結果を示し，**5.** で関連研究との比較を行う．**6.** で結論と今後の課題を述べる．

## 2. オブジェクト表に基づく境界検査とその問題点

JK [22]，RL [24]，DA [25] は検査対象プログラム

が実行時に割り当てた各オブジェクトの境界情報をheap 領域上のオブジェクト表で管理する．オブジェクト表が管理する境界情報はプログラムの各メモリアクセスの直前に参照され，アクセス範囲が対象オブジェクトの境界内に収まるかどうかの判定に利用される．この方式に基づく境界検査器を OTBC（Object-Table-based Bounds Checker）と呼ぶ．本章では，既存の OTBC の実現法とその問題点を説明する．

### 2.1 検査コードの挿入と処理内容

OTBC は検査対象プログラムのコンパイル時に境界検査コードを挿入する．検査コードはオブジェクト追跡コードと境界検査コードに分類される．

#### 2.1.1 オブジェクト追跡コード

オブジェクト追跡コードは，プログラムが割り当てた/解放したオブジェクトの境界情報をオブジェクト表に登録/削除する．OTBC はオブジェクトの割当位置/解放位置にオブジェクト追跡コードを挿入する．例えば，malloc/free の呼出しに対し次の挿入を行う：

```
  p = malloc (size);
# if (p != NULL) reg_obj (p, size);
  ...
  free (p);
# if (p != NULL) unreg_obj (p);
```

ここで，#で始まる行が，挿入したオブジェクト追跡コードである．関数 reg_obj は新しく割り当てられたオブジェクトの境界情報（ベースアドレス，サイズ等(注2)）をオブジェクト表に登録する．関数 unreg_obj は解放されたオブジェクトの境界情報をオブジェクト表から削除する．

static オブジェクトまたは stack オブジェクトの割当と解放についても OTBC はオブジェクト追跡コードを挿入する：

```
  char g_buf[64];
# void init_global_objs (void) {
#   reg_obj(&g_buf[0], sizeof(g_buf));
# }
  void func (void) {
    char buf[32];
#   reg_obj(&buf[0], sizeof(buf));
    ...
#   unreg_obj(&buf[0]);
    return;
  }
```

ここで，関数 init_global_objs はプログラムの開始直後に一度だけ呼ばれてグローバル変数の境界情報をオブジェクト表に登録する．

#### 2.1.2 境界検査コード

境界検査コードはメモリアクセスまたはポインタ操作の境界検査を行う．OTBC はメモリアクセスまたはポインタ操作に対し境界検査コードを挿入する．例えば，配列参照に対し次の挿入を行う：

```
# chk_bnd(buf, buf+i);
  buf[i] = val;
```

また，ポインタ演算に対し次の挿入を行う：

```
# chk_bnd(buf, buf+i);
  p = buf + i;
```

ここで，関数 chk_bnd(注3) はまず，オブジェクト表を探索して，与えられたベースポインタ（buf）に対応するオブジェクト（バッファbuf）の境界情報を取得する．次に，アクセス先のアドレスまたは演算結果のポインタ（buf+i）が対象オブジェクトの境界内に収まるかどうかを検査する．境界を越える場合は境界違反を報告する．

また，OTBC は不正なメモリ操作を引き起こし得る外部ライブラリ関数を，境界検査を行うラッパ関数に置換する．例えば，関数 memcpy の呼出し：

```
  memcpy(dst, src, n);
```

はラッパ関数 wrap_memcpy の呼出しに置換する：

```
# wrap_memcpy(dst, src, n);
```

ここで，関数 wrap_memcpy はポインタ dst，src の指すオブジェクトのサイズが $n$ 以上であるかどうかを検査し，検査を通れば元の関数 memcpy を呼び出す．

なお，本章の冒頭では説明の便宜上，メモリ管理関数 malloc/free の呼出しに検査コードを挿入する例を示したが，OTBC の実際の実装ではこれらのメモリ管理関数のラッパを用意し，ラッパ内で境界情報の登録/削除を行っている場合が多い．

以上の境界検査手法は JK によるものである．RL（及び DA）は JK と異なり，オブジェクト表以外に

(注2)：既存の OTBC はこのほかにオブジェクトの領域種別（static, heap，stack 等）や割当実行位置（ファイル名と行番号）などの情報も追跡管理する．

(注3)：実際の OTBC は関数 chk_bnd の呼出しに対して他の引数も渡す．典型的には，アクセス種別（read または write）やアクセス実行位置（ファイル名と行番号）などのデバッグ用情報を渡す．

OOB（Out-Of-Bounds）オブジェクト表と呼ばれる表を用いて一時的に境界を越えるポインタを追跡管理する．その結果，ポインタ演算結果が一時的に境界外を指すがメモリアクセスには使用されない場合であっても誤検出が発生しない [24].

### 2.2 問 題 点

従来のオブジェクト表に基づく境界検査手法は比較的高互換(注4)かつ高精度(注5)であるが，検査コードの実装が多くのライブラリ関数や OS のシステムコールに強く依存しているという問題点をもつ．その結果，それらを利用できない環境で動作するプログラムに対し，検査コードを適用することができない（適用しても検査コードは機能しない）．

オブジェクト表に基づく検査コードは多くのライブラリ関数やシステムコールで実装される．例えば，オブジェクト表のエントリの割当と解放はメモリ管理用のライブラリ関数（`malloc`，`free` など）やシステムコール（`SBRK`，`MMAP` など）で実装される．また，不正メモリ操作の検出報告は入出力用の関数（`fprintf`，`syslog` など）やシステムコール（`WRITE` など）で実装される．更に，検出報告に有益なデバッグ情報（例えば，コールチェインなど）を付加するために，デバッグ情報操作用の専用ライブラリ（DWARF ライブラリ [5], [29] や BFD ライブラリ [15] など）が使用される．このように，従来のオブジェクト表に基づく検査コードは，多くのライブラリ関数やシステムコールに依存している．

一方，C 言語は，それらのライブラリ関数やシステムコールを利用できない環境で動作するコードの記述に使用されることも多い．本論文では，そのようなコードを低レベル C コードと呼び，低レベル C コードを含むプログラムを低レベル C プログラムと呼ぶ．OS のカーネル（Linux [30] など）や仮想マシンモニタ（Xen [43] など）や標準 C ライブラリ（GLIBC [11], Newlib [17] など）やハードウェアを直接操作する種々の組込みプログラムなど，多くの重要なプログラムが低レベル C プログラムに該当する．また，それらの低レベル C プログラムにおいても不正メモリ操作はこれまでに多数報告されてきた [1], [36], [37]．したがって，低レベル C プログラムのメモリ検査は重要な課題である．しかしながら，上述のとおり，従来のオブジェクト表に基づく検査コードは特定のライブラリ関数や OS のシステムコールに強く依存しているため，低レベル C プログラムを検査することができない．

## 3. 提案手法：トラップキャッシュ機構

### 3.1 概　　要

本研究の目的は，従来のオブジェクト表に基づくメモリ検査手法の問題点（前章を参照）を解決することである．すなわち，検査コードを低レベル C プログラムに容易に適用できるようにすることである．

我々はこの課題の解決に向けて，トラップキャッシュ機構と呼ぶ新たな検査機構を提案する．この機構はオブジェクト表に基づく検査コードを低レベルな環境でも動作できるように改変したものである．トラップキャッシュ機構の要点と効果は以下のとおり．

- **Point 1**：最近アクセスされたオブジェクトの境界領域のみを小容量の固定サイズのバッファ（トラップキャッシュと呼ぶ）で追跡管理する．ここで，オブジェクトの境界領域とは，そのオブジェクトに隣接する 1 Byte の領域（2 個）を意味する（図 1）．このように追跡管理する境界領域を限定することにより，検査によるメモリオーバヘッドが大幅に低減され，次の **Point 2** が実現可能となる．
- **Point 2**：トラップキャッシュを検査対象プログラムの static 領域に配置する．これにより，検査コードの環境依存部分が大幅に減る．特に，トラップキャッシュの管理を環境依存度の高いメモリ管理関数（`malloc` や `free` など）やシステムコール（`SBRK` や `MMAP` など）を使用せずに実現できる．
- **Point 3**：検出報告用のバッファを検査対象プログラムの static 領域に配置し，不正メモリ操作の検出時に検査内容をバッファに保存する．また，検査内容の生成にはデバッグ情報操作用のライブラリは使用しない．これにより，検査コードの環境依存部分が大幅に減る．特に，検査内容の報告を環境依存度の高い入出力関数（`fprintf` や `syslog` など）やシステムコール（`WRITE` など）を使用せずに実現できる．

我々は，トラップキャッシュ機構の設計において，低

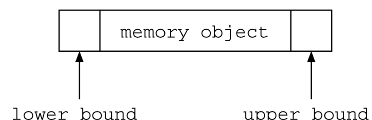


図 1　有効メモリオブジェクトの境界領域
Fig. 1　Boundary region of a valid memory object.

(注4)：ポインタの内部表現を変更しないため．
(注5)：static，heap，stack 領域上の全アクセスを検査できるため．

レベル C コードへの適用可能性の獲得と引換えに検査精度を（ある程度）犠牲にする方針をとった．すなわち，トラップキャッシュに基づく境界検査は，従来のオブジェクト表に基づく検査に比べて検査精度が低い．特に，従来のオブジェクト表に基づく検査では，順次的でない境界違反やダングリングポインタを含む一般的な境界外アクセスが検出可能であるのに対し，トラップキャッシュに基づく検査ではそれらを検出できない．また，オブジェクト表ではすべてのオブジェクトの情報を追跡可能であるのに対し，トラップキャッシュで追跡管理できる境界領域の数には上限がある（トラップキャッシュは固定サイズ）．これらの制限事項（詳細は **3. 5** で後述）は検査精度の低下を意味する．ただし，**4.** の実験結果が示すとおり，順次的な境界違反に対しては，たかだか 4 kByte 程度のトラップキャッシュで高精度な検査が可能であることが確認されている．

### 3. 2 トラップキャッシュ機構による境界検査の例

本節では，トラップキャッシュに基づく境界検査の実例として，低レベルコードの検査例を述べる．図 2 は Linux カーネル 2.6.20.4 の関数 do_dccp_getsockopt に検査コードを挿入したものである（注6）．

```
      static int
      do_dccp_getsockopt(..., char *optval,
                         int *optlen)
      {
#1      reg_obj(&optval, sizeof(optval));
#2      reg_obj(&optlen, sizeof(optlen));
        ...
        int val, len;
#3      reg_obj(&val, sizeof(val));
#4      reg_obj(&len, sizeof(len));
        ...
        if (... ||
#5          wp_copy_to_user(optval, &val,
                            len))
          return -EFAULT;
        return 0;
        ...
#6      unreg_obj(&optval);
#7      unreg_obj(&optlen);
#8      unreg_obj(&val);
#9      unreg_obj(&len);
      }
```

図 2 Linux カーネル 2.6.20.4 の関数 do_dccp_getsockopt の検査
Fig. 2 Checking the function do_dccp_getsockopt in Linux kernel 2.6.20.4.

プログラムの実行時にこの関数が呼び出されると，まず，関数のスタック変数の宣言に挿入された検査コード（行#1 から行#4 の関数 reg_obj）が順次実行される．このとき，関数 reg_obj の各呼出しはターゲットのメモリオブジェクトの境界領域を計算し，それをトラップキャッシュに登録する．例えば，行#1 の reg_obj の呼び出しは変数 optval の境界領域（&optval-1 と &optval+sizeof(optval)）を求め，それをトラップキャッシュに登録する．ここで，トラップキャッシュは検査対象プログラムの static 領域に配置された小容量かつ固定サイズのキャッシュであるため，境界領域の登録時に既に満杯になっている場合がある．その場合は，LRU（Least Recently Used）エントリを廃棄し，新規のエントリとして再利用する．このようにして，最近アクセスされたオブジェクトの境界領域のみをトラップキャッシュに保存する．後に **4.** の実験で説明するように，非常に小さなキャッシュサイズ（4 kByte）でも大規模実用 C プログラムの境界違反の検出に十分な効果を示す．したがって，トラップキャッシュによるメモリ使用量のオーバヘッドは非常に小さい．

一連の関数 reg_obj の呼出しによる境界領域の登録後，プログラムの実行は関数 do_dccp_getsockopt の本体に進む．本体の実行中は，各メモリアクセスの実行位置に挿入された検査コード（関数 chk_bnd）が境界検査を行う．ここでは，行#5 のラッパ関数 wp_copy_to_user が行う境界検査に焦点を当てて解説する． 図 3 が示すとおり，ラッパ関数は元の関数 copy_to_user を実行する前に，関数 chk_bnd を呼び出してポインタ from，to が指す領域の境界を検査する（注7）．このとき，我々の検査方式はトラップキャッシュに保存されている境界領域へのアクセスを境界違

（注6）：説明の便宜上，境界違反の検出に関係のないコードは省略した（図中の...）．

（注7）：**3. 4** で後述するとおり，ラッパ関数の本体は開発者が手動で記述する．この際，元の関数の仕様に基づき，検査内容を決定する．典型的には，元の関数のメモリアクセスの範囲が境界違反を引き起こすかどうか（境界領域とオーバラップするかどうか）を検査する．例えば，関数 copy_to_user は，アドレス from から始まる n バイトの領域の内容を読み出し，アドレス to から始まる n バイトの領域に書き込む．したがって，ラッパ関数 wp_copy_to_user は，読出し範囲及び書込み範囲が境界領域とオーバラップするかどうかを chk_bnd で検査する．

```
    unsigned long
    wp_copy_to_user(void *to,
                    const void *from,
                    unsigned long n)
    {
#10   chk_bnd(from, n, loc);
#11   chk_bnd(to, n, loc);
      return copy_to_user(to, from, n);
    }
```

図 3　関数 copy_to_user のラッパ
Fig. 3　The wrapper of the function copy_to_user.

反と判定する．より正確には，メモリアクセス(注8)のベースアドレス（BASE）とサイズ（SIZE）を用いて，関数 chk_bnd が次のステップを実行する：

- **Step 1：**トラップキャッシュ内を検索して，アドレス BASE または BASE+SIZE-1 を囲む境界領域が保存されたエントリを探す．
- **Step 2a：**検索がヒットした場合，メモリアクセスの範囲が境界領域とオーバラップするかどうかを検査する．オーバラップする場合，次のいずれかの場合に限り境界違反を報告する：

（1） 境界領域が他のオブジェクト（隣接オブジェクト）の有効領域と重なっていない．

（2） 境界領域が隣接オブジェクトの有効領域と重なっているが，アクセス範囲が隣接オブジェクトの境界領域ともオーバラップしている（つまり，アクセス範囲が 2 個の隣接するオブジェクトを横断している）．

- **Step 2b：**検索がミスした場合，境界違反を報告せずに検査を終了する．

例えば，&val，len がそれぞれ BASE 引数，SIZE 引数として渡されると，行#10 の chk_bnd の呼出しはトラップキャッシュ内を検索して，アドレス&val または&val+len-1 を囲む境界領域が保存されたエントリを探す．このとき，検索はヒットし，境界領域&val-1 と&val+sizeof(val) を保持するエントリが見つかる（図 2 の行#3 の reg_obj の呼出しがそのエントリを登録しているため）．

ここで，行#5 でラッパに渡される変数 len の値について，2 通りの場合を考える．まず，0<len かつ len<=sizeof(val) が成り立つ場合，関数 chk_bnd は境界違反を報告しない（**Step 2a**）．しかし，それ以外の場合，例えば，len==-1 が成り立つ場合，chk_bnd に渡される SIZE 引数は ((unsigned int)-1) になる（ラッパ関数 wp_copy_to_user（及び，元の関数）の第 3 仮引数の型が unsigned int であるため）．この場合，ベースアドレス&val とサイズ ((unsigned int)-1) で規定されるアクセス範囲が境界領域&val+sizeof(val) とオーバラップする（((unsigned int)-1) は sizeof(val) よりもはるかに大きな値であるため）．その結果，関数 chk_bnd は境界違反を報告する（**Step 2a**）．ただし，ここでの報告とは，検出報告用のバッファに検査内容を書き込むことを意味する（前述の **Point 3** を参照）．なお，この境界違反は実際に CVE-2007-1730 として報告されているものであり，複数種類の攻撃コードが存在する．

境界違反が検出されなかった場合，プログラム実行は最終的に図 2 の行#6 から行#9 の関数 unreg_obj の呼出しへと進む．ここで，関数 unreg_obj の各呼出しは，トラップキャッシュから変数 optval，optlen，val，len の境界領域を保持するエントリを削除する．

これまで各種の検査コードがトラップキャッシュ機構に基づき図 2 のプログラムを検査する過程を説明してきたが，トラップキャッシュ機構には低レベル C プログラムの検査を可能にする大きな特長がある．それは，キャッシュ本体と検出報告用バッファが対象プログラムの static 領域に配置されているため，キャッシュエントリの登録/検索/削除と境界違反検出時の検出報告がすべて static 領域上の単純なポインタ操作で実現できる点である．すなわち，トラップキャッシュ機構に基づく検査コードは，標準 C ライブラリ関数（malloc，free，fprintf など）や OS のシステムコール（SBRK，MMAP，WRITE など）に依存しない．その結果，検査コードはそれらの関数やシステムコールを利用できない低レベル C コードに対しても適用できる（有効に機能する）．

### 3. 3　トラップキャッシュの実装

次に，トラップキャッシュ機構の実装について述べる．図 4 が示すとおり，トラップキャッシュ機構は四つの構成要素からなる：

- キャッシュエントリを保持する splay 木．
- LRU エントリを選択する自己調整型リスト．
- 空きエントリを保持するプール．

(注8)：なお，我々の手法では，メモリアクセス（メモリの読み書き）のみを境界検査の対象とし，ポインタ演算には検査コード chk_bnd を挿入しない．したがって，オブジェクトの最後のアドレスの 1 バイト外を指すポインタを生成し，元に戻すなどのポインタ演算（ANSI C で合法）は検査の対象外である．

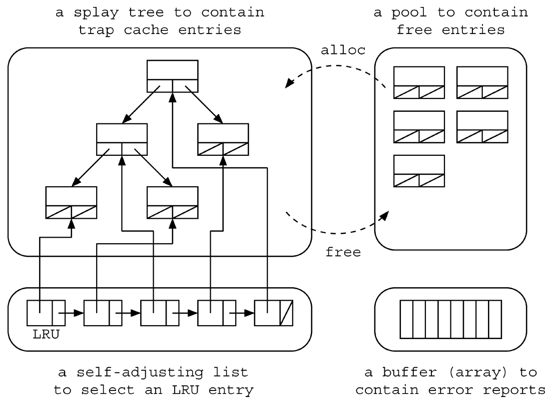


図 4 トラップキャッシュ機構の構成要素
Fig. 4 Components of trap caching.

- 検出報告用のバッファ（配列）.

以下で，これらの各構成要素に関する設計上の判断について説明する．

トラップキャッシュの有効エントリを格納するデータ構造として，我々は splay 木 [32] を採用した．なぜなら，我々の知る限り，エントリへのアクセスを最も高速化できるデータ構造が splay 木だからである．プログラムのメモリアクセスは時間的局所性をもつため，メモリアドレスを探索キーとするエントリの探索も時間的局所性をもつ．すなわち，最近アクセスしたエントリは近々再度アクセスする傾向がある．一方，splay 木は二分探索木の一種であり，アクセスしたノードを splaying と呼ぶ操作で根に移動しつつ木全体のバランスを維持する．したがって，splay 木では最近アクセスしたノードが根の付近に集まるので，最近アクセスしたノードへのアクセスは他のバランス木より高速に行える．また，splaying は，最近アクセスしたノードを単純に根に移動するだけの操作とは異なり，アクセスの時間計算量が最悪でも $O(\log n)$ になるように木全体の形をバランスする．このように，splay 木はトラップキャッシュエントリのアクセスパターンを高速化するのに最適である．したがって，我々はキャッシュエントリを splay 木で管理することにした．

トラップキャッシュは有効エントリを格納する splay 木のほかに，LRU エントリを選択する自己調整型リストをもつ．前述のとおり，境界領域の新規登録時にキャッシュが既に満杯になっている場合，LRU エントリを新規のエントリに置換しなければならない．ところが，splay 木単独では LRU エントリを高速に特定することができない．そこで，我々はキャッシュエントリをアクセス順で保持する自己調節型のリストを導入した．このリストでは，キャッシュエントリがアクセスを受けるたびに，対応するリストエントリがリスト末尾に移動される．その結果，キャッシュの LRU エントリは常にリストの先頭に配置され，LRU エントリの特定と置換を高速に実行できる．

次の構成要素は，キャッシュの空きエントリを保持するプールである．我々はこのプールを static 領域に配置し，キャッシュエントリの割当と解放をすべてこのプール上で行うようにした．その結果，トラップキャッシュに対するすべての操作（エントリの登録，検索，削除）は static 領域上の単純なポインタ操作で実現することができる．すなわち，トラップキャッシュの管理は従来のオブジェクト表の管理と異なり，メモリ管理用のライブラリ関数やシステムコールに依存しない．この特長により，トラップキャッシュ機構に基づく検査コードは低レベルコードへの適用が容易である．

最後の構成要素は，検出報告用のバッファ（配列）である．このバッファも static 領域に配置し，境界違反検出時の検出内容はすべてこのバッファに格納する．その結果，検査コードは従来のオブジェクト表に基づく検査コードと異なり，入出力用のライブラリ関数やシステムコールに依存しない．この特長も，検査コードの低レベルプログラムへの適用を容易にしている．

### 3.4 トラップキャッシュ機構の適用

#### 3.4.1 低レベル C プログラムへの適用

低レベル C プログラムにトラップキャッシュ機構を適用し比較的高精度な検査を実現するには，対象プログラムが使用するメモリ操作関数やメモリ割当/解放関数等のラッパを開発者が手動で記述する必要がある．再び，**3.2** の Linux カーネルの検査例を用いて説明する．この例では，メモリ操作関数 `copy_to_user` がコンパイル時にラッパ関数 `wp_copy_to_user` に置換されているが，このラッパは開発者があらかじめ手動で記述しておかなければならない．また，関数 `copy_to_user` に限らず，境界検査の対象としたいメモリ操作関数にはラッパが必要となる．我々の検査例では，`linux-2.6.20.4/include/asm/uaccess.h` 等で定義されるメモリコピー関数（`copy_to_user`, `copy_from_user`, `strncpy_from_user`, `put_user` 等）やメモリ参照関数（`strnlen_user` 等）を検査の対象と決め，各々のラッパ関数を記述した．ラッパ関数の記述コストは，検査対象とする関数の種類や数に応じて変化する．

一方，heap 領域を対象に境界検査を行うには，heap オブジェクトの境界情報を追跡管理しなければならない．この追跡管理には，メモリ管理関数のラッパの記述が不可欠である．我々の Linux カーネルの検査例では，linux-2.6.20.4/include/linux/slab.h 等で定義されるメモリ割当関数（kmalloc，kcalloc，kzalloc 等）やメモリ解放関数（kfree 等）（が割当/解放するオブジェクト）を追跡管理の対象と決め，各々のラッパ関数を記述した．ただし，Linux カーネルも含め，実用 C プログラムが使用するメモリ管理関数には複数のレベルが存在する場合がある．例えば, (1) システムコール SBRK や MMAP （に対応するライブラリ関数 sbrk や mmap）で大まかにメモリ領域を確保し, (2) 確保した領域をライブラリ関数 malloc や calloc 等で細かく切り分け, (3) 更に，その領域を別のメモリ管理関数で細分化して割り当てることなどは一般的によく行われる．このような場合，どのレベルのメモリ管理関数（の割当/解放するオブジェクト）を追跡管理の対象とするかによって，ラッパの記述対象や記述コストが変わる．一方，我々のトラップキャッシュに基づく検査コードはどのレベルのメモリ管理関数にも依存しないため，任意のレベルでラッパ関数を柔軟に記述できると予想している．

なお，検査対象プログラムのコンパイル時に元の関数からラッパ関数への置換を自動で行うには，一連のラッパ関数の記述に加え，元の関数と置換後のラッパ関数のペアのリストを検査器の設定ファイルに記述しておく作業も必要である．

**3.4.2 高レベル C プログラムへの適用**

高レベル C プログラム(注9)へのトラップキャッシュ機構の適用は，低レベル C プログラムの場合に比べて容易である場合が多い．なぜなら，我々の検査器はデフォルトで，標準 C ライブラリ関数（に含まれるメモリ操作/管理関数）のラッパを提供し，開発者のラッパ記述コストを抑えているからである．例えば，string.h 等で定義されるメモリ操作関数（memcpy，strcpy 等）や stdlib.h 等で定義されるメモリ管理関数（malloc，calloc，free 等）のラッパはデフォルトで提供されるため，開発者による手動記述は必要ない．ただし，検査対象プログラムが独自のメモリ管理関数（前節のレベル (3)）を使用し，かつ，それらの関数（が管理するオブジェクト）を追跡管理の対象としたい場合には，開発者によるラッパの記述（及び，置換規則の設定ファイルの変更）が必要となる．また，標準 C ライブラリ以外のライブラリ関数の検査についても同様である．

## 3.5 制限事項

本節では，トラップキャッシュ機構に基づく境界検査器のいくつかの制限事項を述べる．

**3.5.1 非同期割込みへの対応**

トラップキャッシュの管理はオブジェクト表の管理に比べて実行環境からの独立性が高いが，完全に独立しているわけではない．実行環境に依存する部分はトラップキャッシュ操作中に割込みを禁止する関数である．トラップキャッシュは static 領域上に配置され，すべての検査コード間で共有される．したがって，ある検査コードがトラップキャッシュを操作している間に割込みが発生し，ハンドラ内の検査コードがその操作を割り込むことが起こり得る．この場合，トラップキャッシュのデータ構造は整合性を失ってしまう．これを防止するには，トラップキャッシュの操作を割込みマスクで保護しなければならないが，割込みマスクの設定関数は実行環境によって異なる．したがって，検出器は検査対象のプログラムの実行環境に応じて適切なマスク設定関数を使用しなければならない．例えば，前述の Linux カーネル 2.6.20.4 の検査では，我々はマスク設定関数として spin_lock_irqsave を使用するよう検出器に若干の修正を加えた．ただし，適切なマスク設定関数を使用するよう検出器を修正する作業は，従来のオブジェクト表ベースの検査器を低レベルな環境に移植する作業に比べてはるかにコストが低い．実際，我々の検出器にマスク設定関数を指定する作業は 10 行程度の修正で済んだ.

**3.5.2 スレッドへの対応**

マルチスレッドプログラムの検査では，トラップキャッシュへのアクセスを同期する必要がある．ただし，我々は現状，低レベル C プログラムではスレッド切換が割込みによって発生する場合が多いと予想し，トラップキャッシュへの操作を割込みマスクで保護する以外の措置はとっていない．したがって，トラップキャッシュへの複数スレッドからのアクセスをマスク以外で同期することは今後の課題である．

**3.5.3 トラップキャッシュミス**

トラップキャッシュは static 領域に配置され小容量かつ固定サイズであるため，エントリ検索時にキャッ

(注9)：低レベル C プログラム以外の C プログラム，すなわち，OS のシステムコールやユーザ空間内のライブラリ関数に依存する C プログラムを高レベル C プログラムと呼ぶ.

シュミスが起きる場合がある．そのようなキャッシュミス（したがって，不正メモリ操作の検出漏れ）が頻繁に発生するなら，トラップキャッシュに基づく検査法は受け入れられない．実際，**4.** の大規模実用 C プログラムを対象とした境界違反の検出実験では，キャッシュサイズを 1 kByte にした場合に検出漏れが多数発生した．しかし，幸運にも，キャッシュサイズを徐々に増加していくにつれ検出漏れは著しく減少し，最終的にはたかだか 4 kByte のキャッシュで 10 万行を超える大規模プログラムに対しても検出漏れが発生しなくなった．したがって，キャッシュミスによる検出漏れの大部分は，キャッシュサイズを適度に設定することで解決可能であると考えられる．

#### 3.5.4　境界領域をスキップする境界違反への対応

**3.2** で説明した境界検査手順は，アクセス範囲が境界領域とオーバラップしない場合に検出漏れを引き起こす．すなわち，順次的でない境界違反を検出できない．例えば，あるオブジェクトへのアクセスの途中で境界領域をスキップして別のオブジェクトの領域をアクセスするような境界違反は我々の手法では検出できない．このような境界違反を検出するには，RL [24] や DA [25] と類似の OOB（Out-Of-Bounds）オブジェクトの追跡管理が必要となる．ただし，それらの追跡管理に必要なテーブルサイズは大規模 C プログラムの検査においてもたかだか 1 kByte 程度であると報告されている [24]．したがって，トラップキャッシュ機構の枠組みでも OOB オブジェクトを追跡管理することは可能であると予想し，その実現と評価は今後の課題とする．

#### 3.5.5　隣接するオブジェクトへの対応

**3.2** の境界検査手順の **Step 2a** の（2）では，メモリアクセス範囲があるオブジェクトの境界領域とオーバラップし，かつ，その境界領域が隣接オブジェクトの有効領域と重なる場合の処理について述べた．しかし，この処理は完全ではなく，境界違反の検出漏れが発生する場合がある．例えば，次のコード片を考える：

```
char a1[64], a2[64], *p;
for (p = a1; p <= a1 + 64; p++)
  *p = 0;
```

配列 `a1` の直後に配列 `a2` が配置される（`a1+64==a2`）とすると，このコード片は `sizeof(char)` バイトの境界違反を引き起こす．しかし，デリファレンス `*p` に対して挿入される検査コードは `chk_bnd(p, sizeof(*p))` であり，境界違反を検出できない．なぜなら，`p` が `p==a1+64` を満たすときに，`*p = 0` はオブジェクト `a1` の境界領域へのアクセスであるが，境界領域が隣接オブジェクト `a2` の有効領域と重なり，かつ，アクセス範囲（アドレス `p` から始まる `sizeof(*p)` バイト）が隣接オブジェクト `a2` の境界領域とオーバラップしないからである．すなわち，検出手順の **Step 2a** の（1）と（2）のいずれのケースにも該当しないため，境界違反を検出できない．

このような検出漏れへの対策として，我々の検出器は現状，static，heap，stack の各領域においてオブジェクトの配置を変更する（追跡対象のオブジェクトの割当時に 1 Byte の暗黙のパディングを付加する）オプションを提供している．しかし，一部の実用 C プログラムではオブジェクトの配置変更が許容されない場合もあり，その場合にこの対策は使用できない（**4.** の実験でも，使用していない）．上記の検出漏れへのより完全な対策として，RL や DA と類似の OOB オブジェクトの追跡管理が有効であると予想するが，その実現と評価は今後の課題である．

#### 3.5.6　トラップキャッシュのアクセス衝突への対応

共有メモリ型のマルチコア環境で各コアに個別のスレッドを割り当てて動作するプログラムにトラップキャッシュ機構を適用した場合，各スレッドによるトラップキャッシュアクセスの衝突が頻繁に発生し，検査時間が大幅に増加することが予想される．我々は現状，このような衝突への対策を行っていない．しかしながら，上記のようなプログラムは今後増加すると考えられるため，対策は必要である．対策候補の一つとして，同期不要アルゴリズムの適用を検討しているが，その実現と評価は今後の課題である．

## 4.　実　　験

本章の実験の目的は，トラップキャッシュ機構の有効性を示すことである．すなわち，トラップキャッシュ機構の導入により，オブジェクト表ベースの検査コードが低レベルコードに適用可能になり，かつ，検査精度とオーバヘッドに深刻な悪影響が生じないことを示したい．そのための手段として，トラップキャッシュ機構に基づく境界検査器を GCC 4.1.1 の拡張として実装した（境界判定方式は **3.2** で説明したとおり）．この検出器を TCBC と呼ぶ．我々は境界検査の対象

表 1　検査コードの適用可能性
Table 1　Applicability of check code.

| Program | Type | Level | # Lines | RL | TCBC |
|---|---|---|---|---|---|
| apache_1.3.37 | web server | high | 74 K | yes | yes |
| cvs-1.12.6 | version control tool | high | 74 K | yes | yes |
| gawk-3.1.0 | text processing language | high | 27 K | yes | yes |
| glibc-2.7 | standard C library | low | 791 K | no | yes |
| gzip-1.2.4 | compression tool | high | 5 K | yes | yes |
| httpd-2.2.2 | web server | high | 206 K | yes | yes |
| kgdb-2 | built-in kernel debugger | low | 3 K | no | yes |
| linux-2.6.24 | operating system | low | 5327 K | no | yes |
| newlib-1.16.0 | embedded standard C library | low | 337 K | no | yes |
| openssl-0.9.6 | SSL and TLS toolkit | high | 116 K | yes | yes |
| php-4.4.4 | web development language | high | 345 K | yes | yes |
| proftpd-1.3.0 | FTP server | high | 58 K | yes | yes |
| sendmail-8.12.7 | mail server | high | 82 K | yes | yes |

を文字列操作(注10)に限定するという最適化を検出器に施した上で，本章の実験を行った．

この最適化では，対象コードのコンパイル時に抽象構文木を走査し，各メモリアクセスに対し，型が文字列操作に関連する場合に限り，境界検査コードを挿入する．例えば，対象コード中にポインタのデリファレンス`*p`が含まれる場合，`p`の型が`[unsigned] char *`または`[unsigned] char **`であれば境界検査コードを挿入するが，それ以外の型（`int *`等）であれば挿入しない．ただし，ラッパ関数への置換はラッパが用意されている関数すべてに対して行う．また，オブジェクト追跡コードも基本的には文字列オブジェクトの割当/解放部に対してのみ挿入するが，ラッパ関数の呼出しの引数となり境界検査を受けるオブジェクト(注11)も追跡管理の対象とする．これと類似の最適化はRuwaseとLamによる検出器 [24] やProPolice [9] でも採用されており，実用Cプログラムの境界違反の大部分は文字列操作に起因するという実情を有効活用している．

実験環境はIntel Core2 Duo 1.33 GHz×2と2 GBのRAMを搭載したLinux 2.6.24ワークステーションであり，コンパイル時の最適化レベルには-O2を使用した．

### 4.1　低レベルCコードへの適用可能性

我々はトラップキャッシュに基づく検査コードの低レベルCコードへの適用可能性を調べるために，13種類の実用Cプログラムに対してTCBCの適用を試みた．表1がその結果である．表中の列LevelはCプログラムのレベルを表し，高レベル（high）と低レベル（low）の2種類の値をとる．高レベルなCプログラムとは，OSのシステムコールと標準Cライブラリを利用するCプログラムを意味し，低レベルなCプログラムとは，それらを利用できない環境で動作する低レベルCコードを含むプログラムを意味する．表中の列# Linesは，対象プログラムのソースコード行数を表す．表中の列RLはRuwaseとLamの手法に基づく従来のオブジェクト表ベースの検出器 [24] を対象プログラムに適用できたかどうかを示す．列TCBCは我々のトラップキャッシュに基づく検出器の適用結果である．

表中の結果が示すとおり，RLは13個の実用Cプログラムのうちの高レベルな9個のプログラムには適用できたが，低レベルコードを含む4個のプログラムには適用できなかった．これはRLの検査コードが標準Cライブラリ関数及びOSのシステムコールに強く依存しているためである．同様の理由で，従来のオブジェクト表ベースの検査器であるJK [22] やDA [25] やMudflap [23] なども低レベルなプログラムには適用できない．これに対し，我々の検出器（TCBC）は13個の実用Cプログラムすべてに適用できた．トラップキャッシュに基づく検査コードは標準Cライブラリ関数やシステムコールに依存しないため，それらを利用できない低レベルな環境でも稼動するからである．

以上の実験結果から，トラップキャッシュ機構の導

(注10)：我々が意図する文字列操作とは，単純な文字配列へのアクセスのほかに，`[unsigned] char *`や`[unsigned] char **`型のポインタを介したメモリアクセスや（`void *`型でアクセス先の領域を指定する）`memcpy`などのライブラリ関数によるメモリアクセスを含む．

(注11)：ただし，我々の検出器の現状の実装では，ポインタ解析を行っておらず，ラッパ関数の呼出しに直接渡されるオブジェクトのみを追跡管理の対象に加えている．

表 2　順次的な境界違反の検査精度
Table 2　Accuracy of checking sequential buffer overflows.

| Program | Buffer Overflow | Region | RL | TCBC(1 KB) | TCBC(2 KB) | TCBC(4 KB) |
|---|---|---|---|---|---|---|
| apache_1.3.37 | htpasswd.c:421 | stack | succeeded | succeeded | succeeded | succeeded |
| cvs-1.12.6 | CERT VU#192038 | heap | succeeded | failed | failed | succeeded |
| gawk-3.1.0 | io.c:1961 | stack | succeeded | succeeded | succeeded | succeeded |
| gzip-1.2.4 | CVE-2001-1228 | static | succeeded | failed | succeeded | succeeded |
| httpd-2.2.2 | CERT VU#395412 | stack | succeeded | succeeded | succeeded | succeeded |
| linux-2.6.20.4 | CVE-2007-1730 | stack | N/A | failed | succeeded | succeeded |
| openssl-0.9.6 | CERT VU#102795 | heap | succeeded | failed | failed | succeeded |
| php-4.4.4 | zip.c:302 | heap | succeeded | succeeded | succeeded | succeeded |
| proftpd-1.3.0 | CVE-2006-6563 | stack | succeeded | succeeded | succeeded | succeeded |
| sendmail-8.12.7 | CERT VU#398025 | static | succeeded | failed | failed | succeeded |

入により，検査コードはレベルの高低を問わず，様々な種類の実用 C プログラムに適用可能になることが確認できた．

### 4.2 検査精度

次に，前節の 13 種類のプログラムのうち，境界違反の脆弱性を含む 10 種類のプログラム対象として，トラップキャッシュに基づく検査コードの検査精度を評価した．この実験の目的は，トラップキャッシュに基づく検査コードがどの程度のキャッシュサイズを使用すればどの程度の検査精度を得られるかを調べることである．検査精度の達成目標を示す目的で，Ruwase と Lam によるオブジェクト表ベースの検査器 RL [24] を使用した．境界違反は SecurityFocus [36] などで一般に公開されている攻撃コードを利用して発生させた．

表 2 が実験結果である．表中の列 Program は検査対象のプログラムを示し，列 Buffer Overflow は該当の境界違反に対して，CERT [1] や MITRE [37] などのセキュリティ機関が割り当てた脆弱性番号（CERT VU#... や CVE-...）または境界違反の発生位置（ファイル名と行番号）を示す．列 Region は境界違反が発生したメモリ領域の種類である．列 RL は検出器 RL [24] による境界違反の検査精度（達成目標）を表す．列 TCBC（xKB）は我々の検出器（TCBC）による検出結果であるが，括弧内の xKB は実験時に使用したトラップキャッシュのサイズである．

まず，列 RL が示すとおり，従来のオブジェクト表ベースの検査コードは低レベルプログラムである linux-2.6.20.4 を除き，すべての境界違反を検出できている．一方，列 TCBC（1 kByte）が示すとおり，1 kByte のトラップキャッシュに基づく検査コードは，static，heap，stack の全領域上で境界違反の検出漏れを引き起こしている．しかし，列 TCBC（2 kByte）が示すとおり，キャッシュサイズを 2 kByte に設定するとそれらの検出漏れはある程度解消できた．更に，列 TCBC（4 kByte）が示すとおり，キャッシュを 4 kByte まで増やすと検出漏れは全く発生しなくなった．

以上の実験結果から，トラップキャッシュに基づく検査コードは，大規模な実用 C プログラムに対してもたかだか 4 kByte のキャッシュサイズで，境界違反を高精度に検出できることが分かった．ただし，この実験で発生させ検出できたのはすべて順次的な境界違反である．**3.5** でも述べたとおり，オブジェクト表に基づく検査法（RL やその拡張である DA）は順次的でない境界違反も検出できるが，我々の手法は現状，検出できない．

なお，本節の実験結果や Ruwase と Lam による報告 [24] が示すとおり，境界検査を文字列操作に限定する最適化を行った場合でも，実用上は良い精度で検査を行えると予想できる．ただし，我々の検査手法（及び最適化）によって，対象コード中の文字列操作をすべて検査できるわけではない．例えば，文字配列に対する int * 型のポインタを用いたアクセスなどは検査の対象外である．また，アセンブリコードで記述された文字列操作も検査できない．我々は現状，これらの検査漏れは比較的少ないと考え，対策は今後の課題としている．

### 4.3 オーバヘッド

最後に，前節の 10 種類のプログラムを対象に，トラップキャッシュに基づく境界検査のオーバヘッドを調べた．表 3 がオーバヘッドの計測結果である．表中の列 Benchmark は計測に使用したベンチマークプログラム名を示す．ベンチマークプログラムが存在しない場合は，対象 C プログラムのパッケージに付属するテストスイートを使用し，その実行コマンド名（make check または make test）を明記した．列 Description はベンチマークプログラムが対象プログラムに課した

表 3　オーバヘッド
Table 3　Overheads.

| Program | Benchmark | Description | Time |
|---|---|---|---|
| apache_1.3.37 | httperf [39] | Response time to 15 K tcp connections at the rate of 90 per second. | 3% |
| cvs-1.12.6 | make check | Time to run the test suite. | 12% |
| gawk-3.1.0 | make check | Time to run the test suite. | 3% |
| gzip-1.2.4 | make check | Time to run the test suite. | 33% |
| httpd-2.2.2 | httperf [39] | Response time to 15 K tcp connections at the rate of 90 per second. | 17% |
| linux-2.6.20.4 | iozone [14] | Time to read and write 4 KB records from a 64 MB file on ext3. | 23% |
| openssl-0.9.6 | speed [40] | Time to sign and verify 2048 bit keys using rsa. | 9% |
| php-4.4.4 | make test | Time to run the test suite. | 51% |
| proftpd-1.3.0 | curl [41] | Latency of 225 MB file transfer via the network loop back interface. | 8% |
| sendmail-8.12.7 | smtp-source [42] | Time to send 1 K messages running 10 smtp sessions in parallel. | 14% |
| Average | | | 17% |

処理内容を示す．列 Time はトラップキャッシュに基づく境界検査の実行時間のオーバヘッドであり，ベンチマークを 10 回実行して得た平均値である．

表中の列 Time が示すとおり，検査による実行オーバヘッドは 3%から 51%の範囲に収まり，平均は 17%である．この結果から，我々のトラップキャッシュに基づく境界違反検出器は，Ruwase と Lam によるオブジェクト表ベースの検出器 RL [24] と同程度に高速であることが分かった．したがって，トラップキャッシュの導入により，検査による実行オーバヘッドが著しく増加する危険性はないと考えられる．

なお，Ruwase と Lam の検査手法は OOB オブジェクトの追跡管理を行うため，追跡管理しない手法（我々の手法を含む）に比べて実行オーバヘッドが大きくなる場合がある．ただし，Ruwase と Lam の報告 [24] によると，8 種類の実用プログラムを対象とした実験では，OOB オブジェクトの追跡管理による検査時間の増加は非常に小さい．（最悪のケースで 15%増加し，残りのケースでの増加は無視できるほど小さい）．これは，多くのプログラムにおいて OOB オブジェクトへのポインタ操作が実行される頻度が非常に少ないことを反映している．したがって，TCBC と RL のオーバヘッドの比較は妥当であると考えられる．

## 5. 関 連 研 究

C プログラムの境界検査手法はこれまでに多数提案されている．本章では，それらの手法を主に実装方式に基づいて分類し，各手法の適用範囲/適用コスト，検査精度，オーバヘッドなどの側面を議論するとともに我々の手法と比較する．

### 5.1 静的境界検査

静的境界検査は一般に，動的検査に比べて検査の網羅性に優れる反面，誤検出が多く，現実的な時間内で大規模なプログラムを検査することが難しい．しかし，比較的効率の良い静的手法もいくつか存在する．Wagner らは文字列バッファを操作するライブラリ関数による境界違反を検出する手法を提案した [3]．Larochelle らは LCLint [2] を拡張し，開発者のアノテーションとヒューリスティクスを用いて軽量な境界検査を行う手法を提案した [4]．Dor らは検査対象プログラムの各関数の事前事後条件と副作用を開発者に記述させ，その記述をもとにして文字列バッファの境界違反を低い誤検出率で検出する手法を提案した [6]．

これらの静的手法は我々の手法に比べ，検査の網羅性に優れる（検出漏れが少ない）が，誤検出が多く，多量のソースコードの変更を必要とする．

### 5.2 動的境界検査

動的手法は一般に，大規模なプログラムを現実的な時間内で検査できる反面，網羅的な検査が困難である．

#### 5.2.1 バイナリー検査

対象プログラムのバイナリーに検査コードを挿入し，他の不正メモリ操作とともに境界違反を検出するツールがある．Purify はオブジェクトコードに対して検査コードを静的に挿入する [13]．Valgrind は実行可能ファイルを動的に解釈しながら検査を挿入する [16]．Purify と Valgrind は各メモリ位置に対し，不正メモリ操作を検出するためのメタデータを関連づける．

これらの手法は我々の手法と異なり，(1) ソースコードがなくても検査を行える，(2) 境界違反以外のバグ（例えば，未初期化変数の読込みやメモリリーク）も検出できるという利点をもつ．しかし，ソースコードに検査コードを挿入する手法に比べて検出精度が低く，実行オーバヘッドが非常に大きい（平均で少なくとも 1000%以上）．また，低レベル C プログラムへの適用

も困難である．

**5.2.2** ライブラリ関数の置換

いくつかのツールは外部ライブラリ関数を境界検査機能をもつ関数に置換する．Electric Fence は `malloc` を置換し，置換後の `malloc` は割り当てた領域の隣接領域を OS のメモリ保護を利用してアクセス不可にする [10]．その結果，`malloc` で確保した領域の境界違反は OS のメモリ保護が検出する．Libsafe と Libverify は `strcpy` などのメモリ操作用のライブラリ関数を置換し，置換後の関数は引数を検査して stack 領域上の境界違反を検出する [12]．

これらの手法は我々の手法に比べ，対象プログラムのソースコードを必要としない点が優れる．しかし，static，heap，stack の全領域で境界検査を行えるわけではないため，我々の手法より検査精度が低い．

**5.2.3** スタック保護

スタック破壊は最も危険な攻撃の一つであるため，スタック保護に特化した手法が提案されている．StackGuard はカナリーワードをリターンアドレス格納位置の直前に挿入する事でスタック破壊を検出する [7]．StackShield は関数開始直後にリターンアドレスのコピーを安全な場所に保存し，関数終了直前に復元することでリターンアドレスの書換えを無効化する [8]．ProPolice は PFP（Previous Frame Pointer）格納位置の直前にカナリーワードを挿入することで，リターンアドレスと PFP の両方を保護する [9]．また，ProPolice はローカル変数（関数のパラメータも含む）をスタックフレーム上のバッファの直前に配置する事で，境界違反からローカル変数を保護する．

これらの手法は我々の手法に比べ，実行オーバヘッドが小さい．また，ProPolice は低レベル C プログラム（Linux カーネル）への適用実績がある．しかし，これらの手法は static 領域や heap 領域の境界違反を検出できないため，検査精度は低い．

**5.2.4** 拡張ポインタ表現

ポインタの内部表現を拡張してポインタ値のほかにターゲットオブジェクトのベースアドレスとサイズなどの検査用情報を含める手法が複数提案されている．拡張ポインタ表現は一般に fat pointer と呼ばれる．SafeC は検査対象プログラムのポインタ表現を fat pointer に変換し，fat pointer に基づく検査コードを挿入する [19]．Cyclone も fat pointer と実行時検査を使用するが，静的型解析によりメモリ安全性を言語レベルで保証しつつ不要な検査を除去する [20]．更に，Cyclone は C 言語の文法を拡張し，各ポインタの fat pointer への変換方針を開発者に決定させる．CCured は独自の型システムを利用してポインタを安全なものと非安全なものに分類し，非安全なポインタをポインタ演算に関連するもの（SEQ ポインタ）とキャストに関連するもの（WILD ポインタ）に分類する [21]．分類後，CCured は SEQ ポインタに対して境界検査を挿入し，WILD ポインタに対して動的型検査を挿入する．その際に，WILD ポインタを fat pointer に変換し，SEQ ポインタに関連づけるメタデータはポインタ変数とは別の場所に保管する．このようにして CCured は従来の fat pointer 方式の互換性を改善している．

fat pointer に基づく境界検査手法は我々の手法と同様，static，heap，static の全領域と多くのメモリ操作を検査できるため高精度である．また，順次的でない境界違反を検出できる点や構造体の各フィールドの境界を正確に把握できる点などで我々の手法より検査精度が高い．更に，境界違反のみならず，ダングリングポインタや不正な関数ポインタなど様々な不正メモリ操作を検出する事ができる．一方，ポインタのターゲットオブジェクトのベースアドレスとサイズを fat pointer から瞬時に取得できる（表を検索する必要がない）ため検査オーバヘッドも我々の手法より小さい．しかし，fat pointer は従来のポインタ表現と互換性がないため，大規模実用 C プログラムの検査時に多量のソースコードの修正が必要となる事が多い．典型的には，対象プログラムがコンパイル済みの外部ライブラリ関数とポインタを介してデータを授受する場合やインラインアセンブリを用いてポインタ操作を行っている場合に修正が必要となる．我々の手法はこれらの場合でも修正を要求しない．

**5.2.5** オブジェクト表

**2.** で説明したとおり，いくつかのシステムは対象プログラムのコンパイル時に検査コードを挿入し，実行時に heap 領域上のオブジェクト表を用いて有効オブジェクトの境界情報を管理する．JK [22] はオブジェクト表に基づく初期の検査手法であり，RL [24] は JK を拡張してポインタが一時的に境界外を指す場合の誤検出を低減した．更に，RL は境界検査を文字列操作に限定する事で実行オーバヘッドを大幅に削減した．DA [25] は自動プール割当 [26] と呼ばれる手法を用いてオブジェクト表を複数の小さな表に分割することで RL の実行オーバヘッドを削減した．また，DA は OS

のメモリ保護を活用してオーバヘッドを更に低減した．

これらの手法は高精度であり，fat pointer 方式に比べると対象プログラムとの互換性が高い．更に，DA は非常に高速である．しかし，従来のオブジェクト表に基づく検査コードは，多くのライブラリ関数やシステムコールに依存するため，低レベル C プログラムへの適用が非常に困難である．これに対し，我々のトラップキャッシュ機構に基づく検査コードは，実行環境依存部分がほとんどないため，低レベル C プログラムへの適用が容易である．しかし，その反面，順次的でない境界違反を検出できない，管理できる境界の数に上限がある，境界違反以外の一般の領域外アクセスを検出できないなど，検査精度が低い．

## 6. むすび

従来のオブジェクト表に基づく境界検査器は，検査コードが多くのライブラリ関数やシステムコールに依存するため，低レベル C プログラムへの適用が困難である．この問題に対する解決手法として，我々はトラップキャッシュ機構と呼ぶ新たな検査機構を提案した．この機構は，static 領域上の小容量の固定サイズのバッファを活用することで検査コードの実行環境依存部分を大幅に削減する．その結果，検査精度の低下と引換えに，検査コードの低レベルコードへの適用が非常に容易になる．実験の結果，トラップキャッシュ機構に基づく検査コードは，低レベル C プログラムに対して容易に適用できた．また，検査精度と検査オーバヘッドの計測においても良い結果を示した．今後の課題は検査精度の向上，及び，より高度な最適化手法を導入して実行オーバヘッドを低減することである．

**謝辞** 本研究は，ルネサステクノロジ，日立製作所，早稲田大学，東京工業大学の共同プロジェクトである NEDO（New Energy and Industrial Technology Development Organization）P05020 から一部支援を受けた．

## 文 献

[1] CERT/CC, http://www.cert.org/advisories/
[2] D. Evans, J. Guattag, J. Horning, and Y.T. Lclint, “A tool for using specifications to check code,” Proc. 2nd ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (SIGSOFT 1994/FSE-2), pp.87–96, ACM, 1994.
[3] D. Wagner, J. Foster, E. Brewer, and A. Aiken, “A first step towards automated detection of buffer overrun vulnerabilities,” Proc. 7th Annual Network and Distributed System Security Symp. (NDSS 2000). ISOC, 2000.
[4] D. Larochelle and D. Evans, “Statically detecting likely buffer overflow vulnerabilities,” Proc. 10th Conf. on USENIX Security Symp. (USENIX Security ’01), pp.177–190, USENIX Association, 2001.
[5] DA’s DWARF Page: http://reality.sgiweb.org/davea/dwarf.html
[6] N. Dor, M. Rodeh, and M. Sagiv, “Towards a realistic tool for statically detecting all buffer overflows in C,” Proc. 2003 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2003), pp.155–167, ACM, 2003.
[7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beatie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, “Stack-guard: Automatic adaptive detection and prevention of buffer-overflow attacks,” Proc. 7th Conf. on USENIX Security Symp. (USENIX Security ’98), pp.63–78, USENIX Association, 1998.
[8] Vendicator, Stack Shield technical info file v0.7, http://www.angelfire.com/sk/stackshield, 2001.
[9] H. Etoh and K. Yoda, GCC extension for protecting applications from stack-smashing attacks, http://www.trl.ibm.com/projects/security/ssp, 2000.
[10] Electric Fence, http://perens.com/works/software/
[11] Free Software Foundation (FSF). GNU C Library: http://www.gnu.org/software/libc/
[12] A. Baratloo, N. Singh, and T. Tsai, “Transparent run-time defense against stack-smashing attacks,” Proc. USENIX Annual Tech. Conf. (USENIX ’00), pp.251–262. USENIX Association, 2000.
[13] R. Hastings and B. Joyce, “Purify: A tool for detecting memory leaks and access errors in C and C++ programs,” Proc. 1992 USENIX Winter Tech. Conf., pp.125–138. USENIX Association, 1992.
[14] IOzone, http://www.iozone.org/
[15] LIB BFD, the Binary File Descriptor Library, http://www.cs.utah.edu/dept/old/texinfo/bfd/bfd.html
[16] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” Proc. 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2007), pp.89–100. ACM, 2007.
[17] Redhat, Inc. Newlib: http://sourceware.org/newlib/
[18] Y. Arahori, K. Gondow, and H. Maejima, “TCBC: Trap caching bounds checking for C,” Proc. 2009 IEEE Conf. on Dependable, Autonomic and Secure Computing (DASC 2009), pp.49–56, IEEE Computer Society, 2009.
[19] T. Austin, S. Breach, and G. Sohi, “Efficient detection of all pointer and array access errors,” Proc. 1994 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 1994), pp.290–301, ACM, 1994.
[20] T. Jim, G. Morriset, D. Grossman, M. Hicks, J.

Cheney, and Y. Wang, “Cyclone: A safe dialect of C,” Proc. USENIX Annual Tech. Conf. (USENIX ’02). USENIX Association, 2002.

[21] G.C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, “Ccured: Type-safe retrofitting of legacy software,” ACM Trans. Program. Lang. Syst., vol.27, no.3, pp.477–526, 2005.

[22] R. Jones and P. Kelly, “Backwards-compatible bounds checking for arrays and pointers in C programs,” Proc. Third Int. Workshop on Automatic Debugging (AADEBUG 1997), pp.13–26. Linköping University Electronic Press, 1997.

[23] F. Eigler, “Mudflap: Pointer use checking for c/c++,” Proc. GCC Developers’ SUMMIT, pp.57–69, 2003.

[24] O. Ruwase and M. Lam, “A practical dynamic buffer overflow detector,” Proc. 11th Annual Network and Distributed System Security Symp. (NDSS 2004), pp.159–169. ISOC, 2004.

[25] D. Dhurjati and V. Adve, “Backwards-compatible array bounds checking for C with very low overhead,” Proc. 28th Int. Conf. on Software Engineering (ICSE 2006), pp.162–171, ACM, 2006.

[26] C. Lattner and V. Adve, “Automatic pool allocation: Improving performance by controlling data structure layout in the heap,” Proc. 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2005), pp.129–142, ACM, 2005.

[27] Free Software Foundation (FSF), GCC, the GNU Compiler Collection, http://gcc.gnu.org/

[28] The Apache Sotware Foundation, Apache HTTP SERVER PROJECT, http://httpd.apache.org/

[29] The DWARF Debugging Standard, http://dwarfstd.org/

[30] The Linux Kernel Project, http://www.kernel.org/

[31] The Sendmail Consortium. http://www.sendmail.org/

[32] D.E. Sleator and R.E. Tarjan, “Self-adjusting binary search trees,” J. ACM, vol.32, no.3, pp.652–686, 1985.

[33] D. Wheeler. SLOCCount, http://www.dwheeler.com/sloccount/

[34] H.T. Brugge. boundschecking, http://sourceforge.net/projects/boundschecking/

[35] Valgrind-project. Crocus: A signal-handler checker, http://valgrind.org/downloads/variants.html?njn

[36] SecurityFocus. http://www.securityfocus.com/

[37] MITRE, http://cve.mitre.org/

[38] The Open Group. SUSv3: The Single UNIX Specification, Version 3. http://www.unix.org/what_is_unix/single_unix_specification.html

[39] httperf. http://www.hpl.hp.com/research/linux/httperf

[40] OpenSSL: The Open Source toolkit for SSL/TLS. http://www.openssl.org/

[41] cURL. cURL and libcurl, http://curl.haxx.se/

[42] Postfix. http://www.postfix.org/

[43] Xen.org. The Xen hypervisor, the powerful open source industry standard for virtualization. http://www.xen.org/

（平成 22 年 1 月 8 日受付，5 月 10 日再受付）

**荒堀　喜貴**

2010 東京工業大学情報理工学研究科博士課程計算工学専攻了．同年同大学情報理工学研究科計算工学専攻特別研究員．博士（工学）．ソフトウェア開発環境・システムプログラミングに興味をもつ．

**権藤　克彦**　（正員）

1994 東京工業大学理工学研究科博士課程情報工学専攻了．同年同大学情報理工学研究科情報工学専攻助手，講師を経て，1998 より北陸先端科学技術大学院大学助教授．ブラウン大学客員研究員（2000-2001）．2003 より東京工業大学助教授（現在は同准教授）．博士（工学）．ソフトウェア開発環境・システムプログラミングに興味をもつ．著書「例解 UNIX プログラミング教室」「Java によるプログラミング入門」．ACM，日本ソフトウェア科学会．

**前島　英雄**　（正員：フェロー）

1973 東京工業大学理工学研究科修士課程制御工学専攻了．同年（株）日立製作所入社，部長，主管研究員を経て，1999 より東京工業大学大学院総合理工学科教授．工学博士．マイクロプロセッサ，特にマルチコアやリコンフィギャラブル・アーキテクチャに興味をもち，最近はソフトウェア統合開発環境の研究も行っている．IEEE，情報処理学会各会員．