情報処理学会論文誌 Vol.53 No.3 1150–1165 (Mar. 2012)

競合回避機構を備えた高互換かつ高精度な境界検査手法

荒堀 喜貴[1,a)] 権藤 克彦[2] 前島 英雄[3]

受付日 2011年2月12日, 採録日 2011年12月16日

概要：ネットワークプログラムなどの重要なシステムにおいて，境界違反は深刻な脆弱性となりうるため，（C 言語の）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，シグナル処理中に深刻な互換性の損失または検査精度の低下を引き起こしてしまう．これらの問題を回避するために，我々は（1）間接シグナル処理，（2）検査バッファリングと呼ぶ 2 つの技術からなる検査制御方式を提案する．間接シグナル処理は検査対象プログラムの実行コンテキストを追跡管理し，検査バッファリングはシグナルハンドラ内の検査コードの実行をハンドラの終了まで保留する．我々は提案方式の実装と評価実験を行った．実験の範囲内で，Apache や Sendmail などのシグナル処理を含む実用 C プログラムに対し，提案方式は互換性を維持したまま高精度な境界検査を実現できた．

キーワード：ソフトウェア開発ツール，動的プログラム解析，バグ検出，境界検査，競合回避

Highly Compatible and Precise Bounds Checking with a Race-avoiding Machinery

Yoshitaka Arahori[1,a)] Katsuhiko Gondow[2] Hideo Maejima[3]

Received: February 12, 2011, Accepted: December 16, 2011

Abstract: For important systems such as network programs, boundary errors can be the source of severe vulnerabilities so that bounds checking techniques (for C) have been continuously proposed. Among them, the only approach to maintain high backwards compatibility and low false-positive rate is the one which uses heap-allocated tables to dynamically track the bounds of every valid object. However, this approach seriously loses compatibility or decreases accuracy during the handling of signals. To avoid these problems, we propose a scheme for controlling checks that consists of two techniques: (1) *indirect signal handling*, which keeps track of the execution context of the checked program, and (2) *check buffering*, which suspends the execution of check code within a signal handler until the handler finishes. We have implemented our scheme and conducted experimental evaluation. Our experimental results show that, without losing compatibility, our scheme was able to perform bounds checks precisely on real C programs including Apache and Sendmail which employed signal handlers.

Keywords: software development tool, dynamic program analysis, bug finding, boundary checking, race avoidance

[1] 電気通信大学大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[2] 東京工業大学学術国際情報センター
Global Scientific Information and Computing Center, Tokyo Institute of Technology, Meguro, Tokyo 152–8550, Japan
[3] 東京工業大学大学院総合理工学研究科
Interdisciplinary Graduate School of Science and Engineering, Tokyo Institute of Technology, Yokohama, Kanagawa 226–8502, Japan
[a)] arahori@spa.is.uec.ac.jp

1. 導入

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 1 つである．C 言語で記述されたネットワークプログラムなどの重要なシステムに境界違反の脆弱性が存在する場合，悪意のある攻撃者はそれを利用してシステムへの攻撃や侵入を試みる．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現在もなお，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，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．

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持つ．

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うる位置を予測する．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], [3], [4], [5], [6]．静的手法は一般に，対象コードの境界違反を網羅的に検出できる反面，検査時間が長い，誤検出率が高い，または，ソースコードへの多量のアノテーションの付加（またはソースコードの変更）を必要とするという欠点を持つ[*1]．

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], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19]．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．これらの動的手法の中で Jones と Kelly による手法（JK と呼ぶ）が比較的高度な互換性と検査精度を両立させた初期の手法である [17]．ここで，検査コード（または検査手法）が検査対象コードと互換性を持つとは，検査コード挿入後のプログラムが元のプログラムと（境界検査を除いて）等価に動作する性質を意味する．JK は heap 領域上の表（オブジェクト表と呼ぶ）を用いて，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（ベース，サイズなど）を追跡管理する．対象プログラムがポインタを介してオブジェクトをアクセスする際に，JK はオブジェクト表から対象オブジェクトの境界情報を取得して境界違反を検査する．この手法はポインタの内部表現を変更せずに各オブジェクトの境界情報を追跡管理できるため，対象コードとの互換性が高い．また，static 領域，heap 領域，stack 領域上のすべてのオブジェクトへのアクセスを検査できるため，検査精度が高い．Ruwase と Lam の手法（RL と呼ぶ）は JK の互換性を改善し [18]，Dhurjati と Adve の手法（DA と呼ぶ）は RL の検査時間を改善した [19]．

これら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かつ高精度であるが，オブジェクト表に対する操作が非同期シグナル安全（async-signal-safe）でないため，シグナルハンドラに挿入された検査コードが競合状態を引き起こしてしまう．典型的には，オブジェクト表エントリの割当て/解放に使用するメモリ管理関数が非同期シグナル安全でなく，競合状態を引き起こす．また，検査対象プログラムがマルチスレッドである場合は，オブジェクト表に対する複数スレッドによる操作を同期させる必要がある．しかし，スレッド同期関数も一般に非同期シグナル安全でないため，シグナル処理中に使用すると競合状態が発生する．この問題に対する現状の回避策は，(1) シグナルハンドラとそこから呼び出される関数群に対して検査コードの挿入を抑制するか，あるいは，(2) 対象プログラムを大幅に変更してシグナルを同期的に処理できるようにすることである．しかし，これら回避策は (1) 検査精度の低下を招く，(2) 開発者に多量の手動作業を要求するなどの欠陥を持つ．

1.2 提案手法

そこで，我々はシグナル処理中の境界検査による競合状態を防止しつつ，検査精度の低下やプログラムの手動変更も回避できる検査制御方式を提案する．我々の方式の要点は次のとおり：

- 各スレッドの実行コンテキスト（シグナル処理中か否か）を追跡管理する．
- シグナル処理中は境界検査を実行せず，各メモリアクセスの開始アドレスとサイズをスレッド固有のバッファ[*2]に記録する．
- シグナル処理後にバッファに記録された各アクセスの境界検査を実行する．

この方式に基づく境界検査を BBBC（Buffer-Based Bounds Checking）と呼ぶ．我々の先行研究 [20] では，JK，RL，DA などの既存手法がマルチスレッドプログラムの境界検査で問題を引き起こすことを指摘し解決した．本研究では，既存手法がシングルスレッドでも問題を起こすこと，および，それが解決可能であることを示す．これらのこと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 BBBC のプロトタイプを GCC [21] に実装し，Apache [22] や Sendmail [23] を含む 11 種類の実用 C プログラムに適用して各種の実験を行った．その結果，BBBC はシグナル処理中のメモリアクセスに対しても境界検査を実行でき，かつ，互換性の問題（競合状態）を回避できることが分かった．また，BBBC はそれらのプログラムに含

[*1] 本論文では，検査対象プログラム中に存在しないバグ（境界違反を含む）を検査手法またはツールが誤って検出してしまうことを誤検出と呼ぶ．また，検査対象プログラム中に存在するバグを検査手法またはツールが検出し損なうことを検出漏れと呼ぶ．検出漏れの多い/少ない検査手法はそれぞれ，（検査の）網羅性に劣る/優れるとも表現する．

[*2] マルチスレッドプログラムも検査できるよう，スレッド固有のバッファを使用した．

まれる既発見と未発見の境界違反を合計 11 個検出でき，高度な検査精度を示した．さらに，境界検査を文字列に限定する最適化を行った場合，実行時間のオーバヘッドは平均 20%であった．

1.4 論文の構成

以降，2 章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 章で我々の検査制御方式を提案する．4 章で実験結果を示し，5 章で関連研究との比較を行う．6 章で結論と今後の展望を述べる．

2. オブジェクト表に基づく境界検査とその問題点

JK [17]，RL [18]，DA [19] 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．オブジェクト表は通常，オブジェクトのアドレスとサイズを探索キーとする splay 木 [24] で実装される．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 OTBBC（Object-Table-Based Bounds Checker）と呼ぶ．本章では，既存の OTBBC 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

OTBBC 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．OTBBC はオブジェクトの割当て位置/解放位置にオブジェクト追跡コードを挿入する．たとえば，`malloc`/`free` の呼び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，サイズなど[*3]）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てと解放についても OTBBC 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 1 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．OTBBC 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．たとえば，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` [*4]はまず，オブジェクト表を走査して，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，OTBBC は不正なメモリ操作を引き起こしう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．たとえば，関数 `memcpy` の呼び出し：

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

はラッパ関数 `wrap_memcpy` の呼び出しに置換する：

```
  # 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．

以上の境界検査手法は JK によるものである．RL（および DA）は JK の手法を拡張して，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の誤検出を低減した [18]．ただし，その拡張においても関数 `reg_obj`，`unreg_obj`，`chk_bnd` に渡す引数は JK のものと同一である．

[*3] 既存の OTBBC はこのほかにオブジェクトの領域種別（static, heap, stack など）や割当て実行位置（ファイル名と行番号）などの情報も追跡管理する．

[*4] 実際の OTBBC は関数 `chk_bnd` の呼び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．

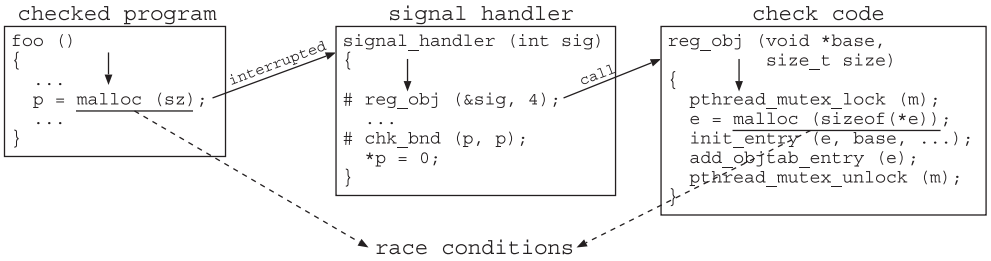

図 1 既存の OTBBC の検査コードが引き起こす競合状態
Fig. 1 Race conditions caused by existing OTBBC's check code.

2.2 問題点：検査コードの競合状態

既存の OTBBC はポインタの内部表現を変更しないため比較的高互換であり，また，static，heap，stack 領域上の全アクセスを検査できるため高精度である．しかし，実用 C プログラムへの適用においては依然として重大な互換性の問題を引き起こす．すなわち，シグナルハンドラやスレッドなどの非同期処理に対する境界検査が競合状態を誘発する．これは次の事実の衝突に起因する：

- オブジェクト表のエントリの割当て/解放はメモリ管理関数（`malloc`/`free` など）で実現されている．また，オブジェクト表への複数スレッドによるアクセスの同期は `lock`/`unlock` 関数で実現されている．
- しかし，これらの関数は非同期シグナル安全（async-signal-safe）[*5]でないため，シグナルハンドラに挿入された検査コードがシグナル処理中に競合状態を引き起こしてしまう．

図 1 は既存の OTBBC の検査コードが引き起こす競合状態の一例である．この例では，検査対象プログラムが関数 `malloc` の実行途中でシグナルを受信し，起動したシグナルハンドラが検査コード（`reg_obj`）を実行し，検査コードが `malloc` を呼び出してオブジェクト表のエントリを割り当てている．ここで，`malloc` は非同期シグナル安全でないため，検査コード内の `malloc` は競合状態を引き起こす．検査対象プログラムが POSIX の `lock`/`unlock` 関数の実行途中でシグナルを受信した場合も同様の競合状態が発生しうる．

2.3 既存の競合回避手法とその問題点

2.3.1 検査コード挿入の抑制

検査コードの競合状態に対する回避手法の 1 つは，検査対象プログラムのシグナルハンドラをすべて特定し，各ハンドラが呼び出す関数もすべて特定し，それらに対する検査コードの挿入を抑制することである．この回避策は実際に，RL [18] のプロトタイプ実装 [26] でも採用されている．しかし，この手法は次の問題点を持つ：

(1) 検査対象プログラムが大規模または複雑である場合，全シグナルハンドラおよびそれらが呼び出す全関数の特定が非常に面倒な作業となる場合がある（適用コストの問題点）．
(2) 全シグナルハンドラおよびそれらが呼び出す全関数の境界違反を検査できない（検査精度の問題点）．

以下で，各問題点について事例に基づき考察を行う．

適用コストの問題点 メールサーバとして広く利用されている Sendmail [23] のバージョン 8.13.5（sendmail-8.13.5 と呼ぶ）に対し，従来の OTBBC を適用する場合を考える．この場合，適用コストの問題は深刻である．実際に我々が sendmail-8.13.5 のソースコード全体（80K 行を超える）から特定した SIGINT ハンドラ，および，ハンドラが実行する関数群を**表 1** に示す[*6]．表 1 において，列 Function は sendmail-8.13.5 が SIGINT の処理中に呼び出す関数の名前を表し，列 Kind はその関数がハンドラであるか（H），ハンドラでないか（NH）を示す．列 LOC は関数のソースコード行数を示す．表 1 のとおり，sendmail-8.13.5 は SIGINT に対するハンドラを少なくとも 3 種類（`sem_cleanup`, `intsig`, `intindebug`）持ち[*7]，SIGINT の処理で実行する関数は約 50 個に及び，それらのソースコード行数は合計で 2,800 行を超える．SIGINT 以外のシグナルの処理も含めると，（検査コード挿入の抑制のために）確認すべき関数の個数と合計行数はさらに大きくなる．このため，従来の OTBBC の適用は非常に面倒である．

原則として，シグナルハンドラの実装では，(1) スタック変数の操作，(2) `volatile sig_atomic_t` 型のグローバル変数の操作，(3) 非同期シグナル安全な関数の呼び出しのみが許されており，この原則を遵守（しようと）するハ

[*5] 非同期シグナル安全（async-signal-safe）な関数とは，リエントラント関数（reentrant function）またはシグナルが割り込めない関数である．リエントラント関数とは，実行途中に非同期的に自分自身を呼び出しても正しい結果が得られる関数である．シグナルハンドラで使用できるのは非同期シグナル安全な関数だけであり，それ以外の関数を使用すると競合状態や不正な実行結果が生じる．非同期シグナル安全な関数の一覧は SUSv3 [25] などで得られる．

[*6] 表 1 に示す関数は，sendmail-8.13.5 のパッケージ内でソースコードが利用可能な関数に限定した．したがって，シグナルハンドラが実行する標準 C ライブラリなどの外部ライブラリの関数は表中に現れない．

[*7] これらの SIGINT ハンドラは実行コンテキストに応じて使い分けられる．我々が特定できた SIGINT ハンドラは 3 種類であるが，合計 80K 行を超える sendmail-8.13.5 のソースコード中には，ほかにも SIGINT のハンドラが存在するかもしれない．

表 1 sendmail-8.13.5 が SIGINT の処理時に呼び出す関数群
Table 1 Functions for sendmail-8.13.5 to call when handling SIGINT.

Function	Kind	LOC	MO	SO
sem_cleanup	H	8	0	0
sm_sem_stop	NH	6	0	0
intsig	H	47	7	0
sm_signal	NH	54	0	0
pend_signal	NH	65	0	0
sm_allsignals	NH	57	0	0
sm_syslog	NH	151	20	20
sm_vsnprintf	NH	39	1	1
sm_io_vfprintf	NH	618	43	33
sm_print	NH	18	4	0
sm_fvwrite	NH	225	47	6
sm_wsetup	NH	48	14	0
sm_init	NH	14	0	0
sm_makebuf	NH	32	10	0
sm_whatbuf	NH	64	10	0
sm_malloc	NH	7	0	0
sm_free	NH	11	0	0
sm_realloc	NH	12	0	0
sm_flush	NH	64	10	0
sm_abort_at	NH	22	2	0
sm_exc_match	NH	10	2	0
sm_match	NH	94	15	15
sm_io_fprintf	NH	20	3	0
sm_exc_print	NH	9	3	1
sm_wbuf	NH	49	6	0
sm_io_flush	NH	26	3	0
sm_io_getinfo	NH	71	17	2
sm_exc_raise_x	NH	49	5	0
sm_exc_free	NH	32	8	3
sm_find_arguments	NH	257	43	10
sm_grow_type_table_x	NH	24	9	5
sm_malloc_x	NH	13	0	0
sm_realloc_x	NH	14	0	0
sm_bprintf	NH	39	5	0
str2prt	NH	82	13	13
sm_malloc_tagged_x	NH	42	2	0
sm_debug_loadactive	NH	7	0	0
sm_debug_loadlevel	NH	23	7	1
sm_heap_register	NH	42	10	0
ptrhash	NH	41	13	1
sm_abort	NH	17	0	0
sm_snprintf	NH	42	1	1
intindebug	H	14	0	0
sm_exc_raisenew_x	NH	19	0	0
sm_exc_vnew_x	NH	157	28	0
sm_vstringf_x	NH	16	0	0
sm_vasprintf	NH	50	3	1

ンドラは単純で，他の関数をほとんど呼び出さない傾向にある．実際に，sendmail-8.13.5 でも，表 1 の SIGINT ハンドラ sem_cleanup や SIGPIPE ハンドラ sigpipe などがこのケースに該当する．

しかし，大規模または複雑な実用プログラムでは，前述の原則が遵守されないことも多く，特定のハンドラが多数の関数群を実行する場合がある．実際，sendmail-8.13.5 では，SIGINT ハンドラ intsig や intindebug が原則に反し，表 1 に示す多数の関数群を実行する．また，SIGCHLD ハンドラ reapchild や SIGUSR1 ハンドラ sigusr1 なども数十種類の関数を実行する．sendmail-8.13.5 の事例に限らず，**表 2** に示すバージョン管理システム cvs-1.12.6 や FTP サーバ proftpd-1.3.0 などでも，特定のシグナルハンドラが多数の関数群を実行する．このようなプログラムに対する従来の OTBBC の適用コストは非常に高い．

検査精度の問題点 次に，検査精度の問題について議論する．再び，sendmail-8.13.5 の事例を用いて，シグナル処理中に実行される全関数に検査コードの挿入を抑制した場合の検査精度の低下を考察する．

この場合，検査精度の低下は無視できないほど大きい．実際，少なくとも，表 1 のすべての関数で境界検査が実行不能となる．このことは，検査不能に陥る（1）関数の個数およびコード行数と（2）関数の種類の両観点からみて危険性が高い．まず，SIGINT の処理に関連する関数に限定しても，表 1 に示す約 50 個の関数（合計 2,800 行以上）が検査不能となる．これらの関数群には，境界検査の対象となるメモリ操作が多数含まれる．表 1 の列 MO は各関数中で境界検査の対象となる行数[*8]を示し，列 SO はそれらのうち文字列操作を含む行数を示す．表 1 のとおり，sendmail-8.13.5 では，SIGINT の処理に関わる約 50 個の関数のうちの約 30 個が境界検査の必要なメモリ操作を実行する．検査対象のメモリ操作を含む行数は合計で 350 行を超え，文字列操作をともなう行数だけでも 100 行を超える．SIGINT 以外のシグナルの処理で実行される関数群も考慮に入れると，sendmail-8.13.5 のソースコード全体で検査不能に陥る関数の個数およびコード行数はさらに大きくなるため，検査精度の低下は無視できない．

検査不能に陥る関数の種類の観点からも，検査精度の低下は深刻である．なぜなら，シグナルハンドラはログ生成や各種のデータ構造のクリーンアップなどを行う場合があり，これらの処理はプログラム全体で多用される文字列操作関数やデータ構造操作関数などで実現されるからである．それらの関数に検査コードの挿入を抑制した場合，プログラム全体で（対応する）文字列処理やデータ構造操作に対し，境界検査を実行できなくなってしまう．このような深刻な検査精度の低下は，sendmail-8.13.5 でも実際に確認される．たとえば，表 1 に示す関数群のうち，sm_syslog

[*8] 1 行中に検査対象となるメモリ操作が複数個含まれる場合でも，検査対象の行数に重みを付けず，1 行として数えた．また，マクロの展開によって検査対象の行数が複数行になる場合であっても，展開前のマクロ呼び出し（を含む行）を検査が必要な 1 行として数えた．

表 2 互換性の実験結果
Table 2 Results of compatibility experiment.

Program(version)	Type	LOC	Signal	Thread	RL	BBBC
aget(0.4.1)	FTP client	1K	no	yes	yes	yes
apache(2.2.2)	web server	206K	yes	yes	no	yes
ctrace(1.2)	debug library	1K	no	yes	yes	yes
cvs(1.12.6)	version control system	74K	yes	no	no	yes
gawk(3.1.0)	text processing language	27K	yes	no	no	yes
gzip(1.2.4)	compression tool	5K	yes	no	no	yes
openssl(0.9.6)	SSL and TLS toolkit	116K	yes	yes	no	yes
php(4.4.4)	web development language	345K	yes	yes	no	yes
proftpd(1.3.0)	FTP server	58K	yes	no	no	yes
sendmail(8.12.7)	mail server	82K	yes	yes	no	yes
smtprc(2.0.2)	SMTP relay checker	3K	no	yes	yes	yes

はログ生成を行う関数であり．この関数は `sm_vsnprintf`，`sm_io_vfprintf`，`sm_fvwrite`，`sm_match` などの文字列操作関数によって実現される．これらのログ生成関数および文字列操作関数は，sendmail-8.13.5 のソースコードの至るところで使用されるため，これらの関数が検査不能に陥ることは深刻な検査精度の低下を引き起こす．

このように，プログラム全体で多用される関数がシグナルハンドラ中でも使用され，それらの関数に検査コードの挿入を抑制することによって検査精度が大幅に低下するケースは，表 2 に示す cvs-1.12.6 や proftpd-1.3.0 などの他の実用プログラムでも確認されている．

2.3.2 シグナルの同期的処理

別の競合回避手法は，シグナル処理専用のスレッドと関数 `sigwait` [27] を用いて，シグナルを同期的に処理するように対象プログラムを書き換えることである．ただし，この回避策もプログラムの規模や複雑さに応じて手動作業の量と難易度が増し面倒である（適用コストの問題点）．また，シグナル処理専用のスレッドとそうでないスレッドの間で共有データへのアクセスを（ロックで）同期する必要があるため，プログラミングの誤りにより，データ競合やデッドロックなどの競合状態が新たに混入する危険性がある．

2.3.3 トランザクショナルメモリ

シグナルに限らない一般の（たとえば，スレッドの）競合状態の回避手法として，トランザクショナルメモリと呼ばれる機構 [28], [29] が提案されている．この手法は，共有データへのアクセスを投機的に実行し，競合の発生を検出した場合に限り，競合前の状態にロールバックしてアクセスを再実行する．この手法を検査対象プログラムに適用すれば，競合回避をある程度実現できる可能性がある．しかし，トランザクショナルメモリは一般に完全ではない．なぜなら，ファイル処理やネットワーク通信など入出力をともなう操作はロールバックと再実行が非常に困難だからである．

2.3.4 ライブラリ関数の非同期シグナル安全化

理想的な競合回避手法は検査コードが呼び出すメモリ管理関数や `lock`/`unlock` 関数をすべて非同期シグナル安全化することである．しかし，これらのライブラリ関数を元の機能や性能を維持したまま非同期シグナル安全化することは我々の知る限りきわめて困難である．

3. 提案手法

本章では，オブジェクト表に基づく境界検査の問題点（シグナル処理中の競合状態）に対し，我々の解決手法と実装を提案し，その効果を述べる．提案手法のアイデアは，各スレッドの実行コンテキスト（シグナル処理中か否か）を追跡管理すること，および，シグナル処理中の境界検査の実行をシグナル処理後まで保留することである．我々はこのアイデアをそれぞれ，間接シグナル処理，検査バッファリングと呼ぶ 2 種類の処理によって実現する．

3.1 検査バッファリング

我々はシグナル処理中の各メモリアクセスに対し，境界検査を即座には実行せず，境界検査に必要な情報（開始アドレス，サイズなど）をスレッド固有のバッファに保存する．バッファに保存した各検査は，シグナル処理完了後にまとめて実行する．この手法を検査バッファリングと呼ぶ．

検査バッファリングの実装は以下のとおり単純である：

- **要点1**：（メインスレッドを含む）スレッドの開始時に，スレッド固有領域にコンテキスト変数と検査バッファを割り当てる．各スレッドのコンテキスト変数は現在そのスレッドがシグナル処理中であるかどうかを示す．コンテキスト変数の管理は 3.2 節で説明する間接シグナル処理によって実現する．各スレッドの検査バッファはシグナル処理中の各検査関数（`reg_obj`，`unreg_obj`，`chk_bnd`）の呼び出しの引数（と関数名）を保存する．
- **要点 2**：各検査関数は本体の先頭でコンテキスト変数

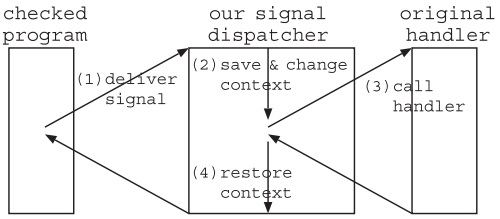

図 2 間接シグナル処理
Fig. 2 Indirect signal handling.

を参照し，現在のスレッドがシグナル処理中であるかどうかを判定する．

– **シグナル処理中である場合**：検査関数の引数（と関数名）を検査バッファに保存し，関数本体の実行をスキップしてリターンする．
– **シグナル処理中でない場合**：(1) 検査バッファに保存された各引数を適切な検査関数に与えて実行し，(2) 現在の検査関数の本体を実行する．

3.2 間接シグナル処理

検査バッファリングを実現するには，各スレッドの実行コンテキスト（現在シグナル処理中であるか否か）を追跡管理しなければならない．我々は各スレッドのスレッド固有領域にコンテキスト変数を割り当て，間接シグナル処理と呼ぶ手法でこの追跡管理を実現する．間接シグナル処理は次のステップからなる：

- **Step 1**：シグナル発生時に，対応するシグナルハンドラではなく検査器のシグナルディスパッチャにシグナルを配送する（**図 2** (1)）．
- **Step 2**：ディスパッチャは現在のスレッドのコンテキスト変数の値を（スタックフレームに）保存し，コンテキスト変数を “シグナル処理中” に変更する（図 2 (2)）．
- **Step 3**：ディスパッチャは受信したシグナルに対応する元のハンドラを呼び出す（図 2 (3)）．
- **Step 4**：ハンドラ実行終了後，ディスパッチャはコンテキスト変数を元の値に復元してリターンする（図 2 (4)）．

このように，間接シグナル処理はシグナルハンドラ起動前にコンテキスト変数を “シグナル処理中” に切り替え，ハンドラ終了後に元のコンテキストに戻す．その結果，各スレッドは検査コードの実行時にコンテキスト変数の値から現在シグナル処理中であるかどうかを判定でき，シグナル処理中の場合に検査バッファリングを実行できる．

次に，間接シグナル処理の実装上の課題をいくつか述べ，我々の解決法を説明する．

3.2.1 シグナルリダイレクション

間接シグナル処理の **Step 1** では，各シグナルを検査器のディスパッチャにリダイレクトしなければならない．我々はこの課題をシグナルハンドラ登録関数（`sigaction` または `signal`）の置換によって実現する．たとえば，関数 `sigaction` の呼び出し：

```
sigaction(SIGINT, &act, &oact);
```

はラッパ関数 `wrap_sigaction` に置換する：

```
# wrap_sigaction(SIGINT, &act, &oact);
```

ここで，ラッパ関数 `wrap_sigaction` は引数 `act` のフィールド `sa_handler` を上書きしてディスパッチャを設定する．関数 `sigaction` は通常，第 1 引数で指定されたシグナルに対するハンドラとして，第 2 引数のフィールド `sa_handler` に設定された関数を登録する．したがって，ラッパの `sa_handler` の設定変更により，SIGINT に対するハンドラは元のハンドラからディスパッチャになる（SIGINT のリダイレクション）．

3.2.2 シグナルディスパッチ

間接シグナル処理の **Step 3** を実行するには，各シグナルと元のハンドラの対応関係を管理しなければならない．我々はこの対応関係の管理をディスパッチ表で実現する．ディスパッチ表はシグナル番号をインデックスとし構造体 `struct sigaction` を要素とする配列である．ラッパ関数 `wrap_sigaction` は各シグナルのリダイレクションの設定時に，元のハンドラをディスパッチ表の該当レコードに記録する．その結果，ディスパッチャは受信したシグナルに対応する元のハンドラをディスパッチ表から取得できる．

3.2.3 ディスパッチ表の管理

ここで，我々はオブジェクト表の管理と類似の課題に直面する．すなわち，ディスパッチ表の管理は非同期シグナル安全な操作で実現しなければならない．幸い，この課題はオブジェクト表の場合に比べて扱いが容易である．なぜなら，ディスパッチ表は次の特徴を持つからである：

- **特徴 1**：表のサイズが上限固定であるため，（`malloc`/`free` などによる）エントリの割当て/解放は不要である．したがって，表に対する操作で非同期シグナル非安全になりうるのは表へのアクセスの同期関数のみである．
- **特徴 2**：表へのアクセス頻度が（オブジェクト表に比べて）非常に少ない．アクセスが発生するのはシグナルハンドラ登録時またはシグナル受信時のみである．

これらの特徴により，ディスパッチ表の管理を非同期シグナル安全化するには，表へのアクセスの同期関数のみを安全化すればよく，さらに，同期関数は高速でなくてもよい（低速であっても検査時間に大きな悪影響は出ない）．そこで，我々はアトミックな CAS（Compare-And-Swap）命令を用いて単純なスピンロック関数を実装し，ディスパッチ表へのアクセスを同期する：

```
static inline void
spin_lock(volatile sig_atomic_t *lock)
{
  while (1)
```

```
    if (CAS(lock, 0, 1))
      break;
  }
```

ここで，CAS はアトミックな CAS 命令であり，現在の*lock の値が 0 である場合に*lock に 1 を書き込み真を返すという操作を途中で割り込まれずに実行する．したがって，関数 spin_lock は非同期シグナル安全である．一方，この関数は複数スレッドから頻繁に呼ばれる場合にビジーウェイトを引き起こす可能性が高い．しかし，ディスパッチ表へのアクセスが発生するのはシグナルハンドラ登録時とシグナル受信時だけであるため，実用 C プログラムではこのビジーウェイトはほとんど起きない．したがって，上記のスピンロック関数は我々の目的には十分である．なお，オブジェクト表へのアクセス頻度はディスパッチ表に比べて非常に高い（検査対象プログラムがメモリアクセスを行うたびにオブジェクト表へのアクセスが発生する）ため，上記の（低速な）スピンロック関数はオブジェクト表へのアクセスの同期には向かない．

3.3 議論

3.3.1 提案手法の効果

検査バッファリングと間接シグナル処理によって得られる効果は以下のとおりである．

- **効果 1**：境界検査による競合状態を回避できる．なぜなら，シグナル処理中は検査バッファリングにより検査関数の本体を実行せず，その結果，非同期シグナル非安全な関数の実行を回避できるからである．
- **効果 2**：シグナル処理中のメモリアクセスの境界違反も検出できる．なぜなら，バッファに保存した境界検査はシグナル処理後に実行するからである．ただし，3.3.2 項で述べるとおり，検出は完全であるわけではなく，検出漏れが発生する場合もある．
- **効果 3**：適用コストが低い．なぜなら，検査バッファリングと間接シグナル処理は手動による作業（対象プログラムへの検査コード挿入の抑制やソースコードの修正など）を要求しないからである．
- **効果 4**：対象プログラムが入出力操作を行う場合でも競合回避が機能する．
- **効果 5**：実現が容易である．実際，4 章で使用する境界検査ツールのプロトタイプでは，検査バッファリングと間接シグナル処理を 400 行程度で実装できた．

以上の複数の効果は，従来の競合回避手法（2.3 節を参照）では同時に達成することはできない．特に，JK，RL，DA などの既存の OTBBC は，競合回避のために，全シグナルハンドラおよびそこから呼ばれる全関数に対し，検査コードの挿入を抑制する（実際の抑制は開発者が手動で行う）．その結果，シグナルハンドラやそこから呼び出される関数のメモリアクセスについて境界検査を実行できない．これに対し，我々の境界検査では，検査バッファリングによりシグナル処理中のメモリアクセスの境界検査をシグナル処理後に実行できる．したがって，我々の境界検査は既存の OTBBC（で検査コードを抑制した場合）に比べて検査精度が高い（検出漏れが少ない）といえる．

3.3.2 提案手法の制限事項

検査バッファリングと間接シグナル処理にはいくつかの制限事項がある．第 1 の制限は検査バッファとコンテキスト変数に対する操作が非同期シグナル安全でなければならないことである．したがって，これらの操作の実現に，POSIX スレッドが提供するスレッド固有領域操作用の関数[*9]を使用することはできない．そこで，我々は GCC や一般的な商用コンパイラ（Intel 製，Microsoft 製，Boland 製など）で利用可能な__thread キーワードを使用して，検査バッファとコンテキスト変数をスレッド固有領域に割り当て，かつ，これらを非同期シグナル安全に操作する．

__thread キーワードは通常の変数アクセスと同様にスレッド固有領域をアクセスすることを可能にする．__thread 宣言された変数はスレッド固有領域に割り当てられ，かつ，その変数へのアクセスはスレッドレジスタを介したメモリアクセス命令にコンパイルされる．ここで，コンパイラとスレッドライブラリの連携により，スレッド切替え時にスレッドレジスタは次のスレッドの固有領域を指すように変更される．スレッド実行中はスレッドレジスタはつねに現在のスレッドの固有領域を指し，シグナル処理に割り込まれても値は変化しない．

そこで，我々は検査バッファとコンテキスト変数を__thread 変数として宣言することでスレッド固有領域に割り当てた．さらに，これらに対する操作（通常の変数アクセス）をシグナルマスクで保護することで非同期シグナル安全化した．したがって，検査バッファやコンテキスト変数の操作の途中で他のシグナルが発生した場合でも，操作が完了しシグナルマスクが解除されるまでは，他の（つまり，発生したシグナル処理にともなう）検査バッファやコンテキスト変数の操作が割り込むことはない．

第 2 の制限は__thread 宣言した検査バッファが固定長であるため，シグナル処理中に境界検査を行えるメモリアクセス数が限られていることである．幸いにも通常のシグナル処理は非常に短いので，バッファサイズを十分大きくとれば深刻な検出漏れの大部分を防止できるはずである．ただし，実用 C プログラムにおいてシグナル処理中に境界違反が発生する報告例，および，バッファサイズの制限により検出漏れが発生する例を我々は現状確認できていない．したがって，バッファサイズと検出漏れの関係についての詳細な分析は今後の課題である．

なお，__thread キーワードは現状，すべてのコンパイラ

[*9] pthread_key_{create,delete} や pthread_{get,set}specific など．これらの関数は非同期シグナル安全ではない．

やアーキテクチャで利用できるわけではない．しかし，マルチコアの普及とともにスレッドの利用が増加しており，__thread キーワードは今後，多くの環境でサポートされるので問題はない．

第 3 の制限はシグナル処理中の検査をシグナル処理後に実行することにより，検査結果に違いが生じる場合があることである．違いが生じるケースは主に 2 つ考えられる：

- **Case 1**：シグナル処理中の境界違反がオブジェクト表そのものを破壊する．
- **Case 2**：マルチスレッドプログラムの検査において，(1) あるスレッドがシグナル処理中にある heap オブジェクトの境界を違反し，(2) シグナル処理の終了直後にスレッド切替えが発生し，(3) 切替え後のスレッドが上記の heap オブジェクトを解放する．

前者のケースでは，シグナル処理後に検査の続行が不可能となってしまう．後者のケースでは，境界違反を起こしたスレッドが次に検査コードを実行する際に，該当の heap オブジェクトの境界情報がオブジェクト表中に残っていないため違反を正確に報告できない．しかし，我々はこれらのケースが発生するのは非常に稀であると予想し，対策は今後の課題としている．

3.3.3 遅延評価との関連性

我々の提案手法は，遅延評価 [30], [31] と呼ばれる式の評価方式と共通点を持つ．提案手法はシグナル処理中に呼ばれた検査関数の本体の実行タイミングをシグナル処理後まで遅延させる．これに対し，遅延評価は関数呼び出し式の評価タイミングや関数呼び出しに渡されたパラメータの評価タイミングをそれらの評価結果が（他の式の評価のために）必要になる時点まで遅延させる．したがって，両者は検査実行または式評価のタイミングを遅延させる点が共通している．

一方，提案手法と遅延評価は以下の相違点を持つ：

- **相違点 1**：提案手法と遅延評価は主目的が異なる．遅延評価は我々の知る限り，性能向上（不必要な式の評価の回避）を主な目的としている．これに対し，我々の提案手法は互換性の向上（境界検査の競合状態の回避）を目的としている．
- **相違点 2**：提案手法と遅延評価では，検査実行または式評価の実行タイミングが異なる．遅延評価では，遅延させていた式評価の実行タイミングは，評価結果が他の式の評価に必要になった時点である．これに対し，提案手法では，遅延させていた検査の実行タイミングは，シグナル処理が終了して次のメモリ管理またはメモリアクセス（の検査コード）に到達した時点である．
- **相違点 3**：提案手法と遅延評価は，実装方式が異なる．上記の実行タイミングを実現するために，我々の提案手法は間接シグナル処理により，各スレッドの実行コンテキストを追跡管理する．また，間接シグナル処理（にともなうコンテキスト変数の更新とディスパッチ表の参照）と検査バッファリング（にともなう検査バッファの更新）はシグナル処理中に実行する操作であるため，非同期シグナル安全に実現する．これに対し，従来の遅延評価の実装はコンテキストを追跡管理しておらず，非同期シグナル安全性も考慮していない．

以上の 3 つの相違点の組合せが，従来の遅延評価と比較した場合の我々の提案手法の新規性である．

上記の**相違点 2** と**相違点 3** に関連し，提案手法の実装時に我々が行ったいくつかの選択について議論する．まず，遅延中の検査の実行タイミングについて論じる．3.1 節でも説明したとおり，我々は，遅延中の検査をシグナルハンドラの終了直後に実行する．すなわち，シグナル処理の終了後に最初に到達した検査コードで，検査バッファに保留中の検査を実行する．遅延中の検査の実行タイミングをこのように実装した理由は，検出漏れを低減させるためである．実際，他の実行タイミングでは検出漏れが増加してしまう．

他の実行タイミングとしては考えられるのは，遅延中の検査をシグナルハンドラの終了直後ではなく，さらに遅延させた後に実行する方式である．しかし，この方式は，たとえば次のシナリオで検出漏れを引き起こす：

- **Step 1**：検査対象プログラムがメモリオブジェクト m を割り当て，検査コードが m の境界情報をオブジェクト表に登録する．
- **Step 2**：シグナル S が発生し，シグナルディスパッチャがコンテキスト管理を行った後に S のハンドラを起動する．
- **Step 3**：起動したハンドラが m へのアクセスで境界違反を引き起こし，検査コードがそのアクセスの検査情報を検査バッファに保存する（検査の実行を保留する）．
- **Step 4**：ハンドラの終了直後に，検査対象プログラムが m を解放し，検査コードが m の境界情報をオブジェクト表から削除する（ここで，保留中の検査を実行せず，さらに遅延させる）．
- **Step 5**：検査対象プログラムがメモリアクセスを実行し，検査コードがその境界検査を実行するとともに，**Step 3** で保留していた検査を実行する．

遅延中の検査の実行をハンドラ終了直後よりさらに遅延させる方式では，**Step 3** で保留した検査（m に対する境界違反の検査）を **Step 4** の検査コード内で実行せずに，**Step 5** の検査コード内で実行する．しかし，**Step 4** で m の境界情報がオブジェクト表から削除されてしまっているため，**Step 5** の検査コードは m 上の境界違反（**Step 3**）を検出できない．

これに対し，遅延中の検査をシグナルハンドラの終了直後に実行する方式では，検査コードは **Step 4** において，

m の境界情報を削除する前に遅延中の検査を実行する．この場合，上記の検出漏れは発生しない．

このように，遅延中の検査の実行タイミングは検出精度に影響する．我々は特定のシナリオで発生する検出漏れを回避するために，遅延中の検査をシグナルハンドラの終了直後に実行する方式を選んだ．

次に，**相違点 3** に関連し，間接シグナル処理と検査バッファリングの非同期シグナル安全性を議論する．まず，これまで説明してきたとおり，我々が問題視するのは，検査対象プログラムが非同期シグナル安全でない操作の実行中にシグナルハンドラによって割り込まれる場合である．この場合，従来の（オブジェクト表に基づく）検査コードは競合状態を引き起こしてしまうが，我々の提案手法に基づく検査コードは競合を回避する．すなわち，間接シグナル処理と検査バッファリングは，ハンドラ（およびハンドラが呼び出す関数）内の検査コードが引き起こす競合を防ぐ．この競合回避を実現すべく，我々は，間接シグナル処理にともなう処理（コンテキスト変数の更新やディスパッチ表の参照）と検査バッファリングにともなう処理（検査バッファの更新）のすべてを非同期シグナル安全な操作で実装している（3.2.3 項と 3.3.2 項を参照）．その結果，検査対象プログラムがどのようなタイミングでシグナルハンドラに割り込まれても，ハンドラ実行中に検査コードは競合状態を引き起こすことなく正常に動作する．

一方，間接シグナル処理や検査バッファリングの実行中にシグナルが発生するケースも考えられる．このような場合でも，両処理は正常に動作する．まず，間接シグナル処理において，シグナルディスパッチャがディスパッチ表を参照している最中に他のシグナルが発生した場合を考える．この場合，ディスパッチャはディスパッチ表を保護するロックをすでに獲得しているので（3.2.3 項を参照），新たなディスパッチャを即座に起動するとデッドロックが発生する（起動したディスパッチャも同一のロックを獲得しようとするため）．

この問題に対処すべく，我々はディスパッチ表へのアクセス全体（ロックの獲得/解放を含む）をシグナルマスクで保護している．その結果，ディスパッチ表の参照中にシグナルが発生しても，そのシグナルは，ディスパッチ表の参照後にロックが解放されマスクが解除されるまでブロックされる．したがって，上記のデッドロックが発生することはない．

次に，検査バッファリングにおいて，検査バッファの更新中に他のシグナルが発生した場合を考える．この場合，ディスパッチャおよびハンドラを即座に起動してしまうと競合状態が発生する（起動したハンドラ内の検査コードが同一の検査バッファを更新するため）．

この問題に対処すべく，我々は検査バッファの更新期間中は全シグナルをブロックしている（3.3.2 項を参照）．その結果，検査バッファの更新は他の更新によって割り込まれることはない．同様に，間接シグナル処理におけるコンテキスト変数の更新もシグナルマスクで保護しているため（3.3.2 項を参照），更新途中でシグナルが発生しても，他の更新の割込みにより競合状態が発生することはない．

このように，検査対象プログラムのシグナルハンドラが多重に呼び出される状況（シグナル処理中に他のシグナルが発生する状況）においても，間接シグナル処理と検査バッファリングは競合状態を引き起こさず正常に動作する．

4. 実験

我々は提案手法のプロトタイプ（BBBC と呼ぶ）を GCC 4.3.2 の拡張として実装した．プロトタイプ実装では，境界違反の判定方式として我々の先行研究 [32] の方式を採用した．この方式は有効オブジェクトに隣接する 1 バイトの領域（合計 2 バイト）を trap 領域と定め，trap 領域とオーバラップするメモリアクセスを境界違反と判定する．

我々は BBBC を実用 C プログラムに適用して各種の実験を行った．実験の目的は提案手法の（1）互換性，（2）検査精度，（3）実行オーバヘッドを計測することである．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 互換性

我々はまず，提案手法（BBBC）と実用 C プログラムの互換性を評価し，Ruwase と Lam の手法（RL）の互換性と比較した．表 2 はその結果を示す．列 Program は検査対象プログラム，列 Type はプログラムの種類，列 LOC は SLOCCount [33] で計測したソースコード行数を示す．列 Signal は対象プログラムがシグナル処理を非同期的に行うかどうかを示す．列 Thread はマルチスレッド処理の有無を示す．列 RL と列 BBBC はそれぞれ，RL と BBBC の検査コードの互換性の有無を示す．我々は検査コードが次の条件を満たした場合に，互換性がある（“yes”）と判定した：

- **条件 1：** 検査コード挿入後のプログラムがパッケージ付属のテストスイートをパスする．テストスイートが存在しない場合，マニュアル中の使用例に基づく単純な動作テストをパスする．
- **条件 2：** シグナルハンドラ実行中に検査コードが非同期シグナル非安全な関数を呼び出さない．

なお，RL の実装には文献 [26] を使用した．また，シグナル処理中の非同期シグナル非安全関数の呼び出しは Crocus [34] と手動確認の両方で検出した．

表 2 が示すとおり，我々の境界検査手法（BBBC）は 11 種類のプログラムすべてと互換性を維持できた．一方，Ruwase と Lam の手法（RL）はシグナルを非同期的に処

理しない 3 種類のプログラムとは互換性を維持できたが，他の 8 種類のプログラムとは維持できなかった．これらの互換性の損失はすべて，シグナル処理中に検査コードが非同期シグナル非安全な関数を呼び出したことによる．JK，DA の実装もシグナル処理中にオブジェクト表を用いて境界検査を行うため，RL と同様に互換性を失う．以上の実験結果から，我々の手法は既存のオブジェクト表ベースの境界検査手法より対象コードとの互換性が高いと考えられる．

4.2 検査精度

次に，我々の境界検査手法の検査精度を評価した．我々は表 2 の 11 種類のプログラムのうち，脆弱性が報告されている 8 種類のプログラムに対して攻撃を行い，実際に境界違反を発生させた．これらの攻撃は SecurityFocus [35] などが公開している攻撃コード（exploit code）を用いて実行した．

表 3 はそれらの境界違反に対する BBBC の検出結果である．列 Boundary Violation は攻撃対象のプログラム名とその境界違反を示す．CERT [1] や MITRE [36] などのセキュリティ機関が境界違反を脆弱性として報告している場合は脆弱性番号（CERT VU#... や CVE-...）を示し，そうでない場合は境界違反が発生するソースコード位置（ファイル名と行番号）を示した．列 Access と列 Area はそれぞれ，境界違反を引き起こすメモリアクセスとメモリ領域を示す．列 Res は発生した境界違反を BBBC が検出できたかどうかを示す．列 Res が示すとおり，BBBC は 8 個の境界違反をすべて検出できた（その際に誤検出は発生しなかった）．

表 3 の既報告の境界違反のほかに，BBBC は ctrace-1.2 と smtprc-2.0.2 の未報告の境界違反も検出した．これらの境界違反は前節の互換性の実験中に判明した．ctrace-1.2 は ctrace.c の 61 行目のマクロ `HASH` の型変換の誤りが原因となり，マクロ `HASH` を使用して配列 `_thread` のインデックスを計算し配列参照を行う数箇所で境界違反を引き起こしていた．smtprc-2.0.2 は parse_config_files.c の 281 行目で malloc が割り当てた領域に対し，288 行目で構造体フィールドアクセスが境界違反を引き起こしていた．また，同ファイルの 352 行目で `malloc` が割り当てた領域に対し，516 行目で `strcat` による書き込みが境界違反を引き起こしていた．これらの境界違反は我々の知る限り，未報告である．

表 3 BBBC の検査精度
Table 3 Accuracy of BBBC.

Boundary Violation	Access	Area	Res
apache(CERT VU#395412)	`token[++c]`	stack	yes
cvs(CERT VU#192038)	`*cp`	heap	yes
gawk(io.c:1961)	`strcpy`	stack	yes
gzip(CVE-2001-1228)	`strcpy`	static	yes
openssl(CERT VU#102795)	`memcpy`	heap	yes
php(zip.c:302)	`zzip_read`	heap	yes
proftpd(CVE-2006-6563)	`read`	stack	yes
sendmail(CERT VU#398025)	`*bp++`	static	yes

本節の実験結果より，我々の境界検査はあらゆる領域（static，heap，stack）の境界違反に有効であることが分かる．また，配列参照やデリファレンスや外部ライブラリ関数呼び出しを含む多くの種類のメモリ操作を検査できることも分かった．これらの効果は従来のオブジェクト表ベースの境界検査手法（OTBBC）と同様である．

一方，従来の OTBBC ではシグナル処理中の境界検査が競合状態を引き起こしてしまう．そのため，全シグナルハンドラおよびそこから呼び出される全関数に対し，検査コードの挿入を手動で抑制する回避策がとられている（2.3.1 項を参照）．しかし，この回避策を実施した場合，検査コードの挿入を抑制した全関数に対して境界検査を実行できなくなってしまう．実験に使用したプログラムでは確認されなかったものの，それらの関数で境界違反が発生するケースも存在すると考えられる（開発中のプログラムなどでは特に）．我々の提案手法（検査バッファリング）は，それらの関数の検査を可能にする．したがって，提案手法に基づく境界検査は，従来の OTBBC（で検査コードの挿入を抑制した場合）に比べて検査精度が高い（検出漏れが少ない）といえる．

4.3 実行オーバヘッド

最後に，我々の境界検査の実行オーバヘッドを評価した．我々は表 2 の 11 種類のプログラムのうち，標準的なベンチマークツールまたはテストスイートが利用可能な 8 種類のプログラムを対象として，検査コード挿入前後の実行時間を計測した．計測中は境界検査を文字列操作に限定した．この最適化は境界違反の大部分は文字列操作に起因する[*10]という観測に基づくものであり，RL [18] や ProPolice [9] も採用している．

表 4 は計測で得られた BBBC の実行オーバヘッドである．列 Benchmark Tool は計測に使用したベンチマークツール名またはテストスイート実行コマンドを示す．列 What ... はベンチマークの処理内容を示す．列 Overhead は境界検査による実行時間のオーバヘッドを示す．

実行オーバヘッドの平均は 20%であり，我々の境界検査手法は実用プログラムのテストやデバッグに使用できる（あるいは，開発時のビルドプロセスに組み込める）ほど十分小さいといえる．実際，我々の手法は静的な境界検査手法 [2], [3], [4], [5], [6] に比べて検査の網羅率が低い（検出漏

[*10] 実際，境界検査を文字列に限定しても表 3 の境界違反はすべて検出できた．ただし，ctrace-1.2 の境界違反と smtprc-2.0.2 の境界違反の 1 つはこの最適化により検出できなくなった．

表 4 BBBC の実行オーバヘッド
Table 4 Run-time overheads imposed by BBBC.

Program(version)	Benchmark Tool	What the benchmarking tool made the program execute	Overhead
apache(2.2.2)	httperf [37]	Respond to 15K tcp connections at the rate of 90 per second.	19%
cvs(1.12.6)	make check	Run the bundled test suite.	15%
gawk(3.1.0)	make check	Run the bundled test suite.	4%
gzip(1.2.4)	make check	Run the bundled test suite.	34%
openssl(0.9.6)	speed [38]	Sign and verify 2,048 bit keys using RSA.	11%
php(4.4.4)	make test	Run the bundled test suite.	52%
proftpd(1.3.0)	curl [39]	Transfer a 225 MB file via the network loop back interface.	8%
sendmail(8.12.7)	smtp-source [40]	Send 1K messages running 10 smtp sessions in parallel.	15%
Average			**20%**

れが多い）反面，1 回の検査に要する時間は非常に小さい．たとえば，Wagner らの静的な境界検査手法 [3] では，32K 行の C コード（sendmail-8.9.3）の検査に約 15 分を要するという報告がある．これに対し，我々の手法は，実験環境は異なるものの，82K 行の C コード（sendmail-8.12.7）の検査結果（表 3 を参照）を数秒で得ることができた．さらに，意図的に負荷を大きくしたベンチマーク（表 4 の sendmail-8.12.7 の行を参照）の実行でさえ，約 40 秒で完了できた．

他の動的検査手法と比較しても，我々の境界検査のオーバヘッドの低さは有望である．実際，Valgrind [13] や Purify [12] などのバイナリレベルの動的検査手法ではオーバヘッドが平均で 1,000%を超えてしまう[*11]．また，ソースレベルの動的検査であっても，JK [17] などは効果的な最適化を行っておらず，大部分のプログラムで 400%以上のオーバヘッドを示す．これらの動的手法に比べ，我々の手法は低オーバヘッドである．一方，RL [18] は我々と同様，境界検査の対象を文字列操作に限定する最適化を採用している．その結果，オーバヘッドは 14 種類の実用 C プログラムに対して 26%以下であり，我々の手法と同程度に高速である．

なお，我々の手法では，オブジェクトの trap 領域を飛び越す境界違反（たとえば，オブジェクトの有効領域から 2 バイト離れた領域へのアクセス）を検出できない．これに対し，RL では，JK の手法をベースにしつつ，ポインタ演算によってオブジェクトの有効領域から外れたポインタに対しても，そのポインタがかつて指していたオブジェクトを追跡管理する．その結果，RL は，有効領域から 2 バイト以上離れた領域への不正アクセスも検出できるが，我々の手法と比較して，実行オーバヘッドが若干大きくなる．ただし，Ruwase らの報告 [18] によると，上記の追跡管理によるオーバヘッドの増加は無視できるほど小さい（たかだか数%の増加）．

ポインタ解析や型解析を駆使してより高度な最適化を行う手法 [15], [16], [19] は，境界検査を文字列操作に限定しなくてもオーバヘッドが非常に小さい．たとえば，DA [19] は，9 種類のベンチマークプログラムに対して平均で 12%のオーバヘッドを示しており，我々の境界検査より高速である．DA と同様の最適化（自動プール割当て [41]）を導入することで，我々の検査も同程度に高速化できると予想するが，その実現と評価は今後の課題である．

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分類し，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，その記述を基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 [5]．

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，ソースコードへの多量のアノテーションの付加（またはソースコードの変更）を必要とする．

一方，Clarke らは有界モデル検査に基づく C プログラ

[*11] Purify と Valgrind は，各メモリアクセスに対し，境界違反以外の不正メモリ操作の検査も行うので，境界検査のみに着目した厳密なオーバヘッドの比較は難しい．ただし，5.2.1 項でも述べるとおり，バイナリレベルでの動的検査の実行オーバヘッドの大部分は，(1) 全メモリアクセス命令を検査対象とすること，および (2) 各命令の検査時にアクセス範囲に含まれる全メモリバイトまたは全メモリビットのメタデータを参照することに起因する．この事を考慮に入れると，境界検査のみに限定しても，我々の手法のオーバヘッドはバイナリレベルでの動的検査のオーバヘッドよりはるかに低いと考えられる．

ムの静的検査ツール CBMC を提案した [6]．CBMC は，C プログラムのソースコードを解析し，プログラムの実行がバグの発生箇所に到達した場合に限り真となる命題論理式を生成する．生成した論理式の充足可能性は Chaff [42] などの高効率な Boolean SAT ソルバで判定し，充足が確認された場合にバグ（と反例）を報告する．CBMC が検出するバグには，配列の境界違反やダングリングポインタやユーザがソースコードに追記したアサーションの違反などが含まれる．CBMC はループ文や関数の再帰呼び出しなどの繰返し構造をユーザが指定した回数を上限として展開（本体を複製）し，展開部のみを検査する．これにより，検査対象の状態空間および検査時間の削減を図る．

CBMC は，他の静的検査手法と同様，我々の手法に比べ検査の網羅性に優れる．また，CBMC は検出に成功したバグについて反例を報告するので，ユーザはバグの発生原因をより効率的に特定することができる（我々の手法には原因特定を支援する仕組みはない）．しかし，他の静的検査手法と同様，CBMC は検査時間が長い．また，有界モデル検査技術一般の問題点として，ユーザが指定した展開回数の上限を超える繰返しで発生するバグを検出できない．我々の手法にはそのような制限はない．

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 バイナリ検査

対象プログラムのバイナリ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 [12]．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 [13]．Purify と Valgrind は，検査対象プログラムが使用するメモリ領域の各バイト（Purify）または各ビット（Valgrind）に対してメタデータを関連付け，メモリの割当て/解放やアクセスに応じて各バイトまたは各ビットの状態を追跡管理する．たとえば，Purify は，各メモリバイトに対して 2 ビットのメタデータを関連づけ，各々のバイトが未割当ての状態，割当て済みかつ未初期化の状態，割当て済みかつ初期化済みの状態のいずれの状態であるかを追跡管理する．Purify と Valgrind は両者ともに，命令単位でメモリアクセスを検査する．すなわち，検査対象プログラムの各メモリアクセス命令の実行直前に，アクセス範囲に含まれる全メモリバイトまたは全メモリビットのメタデータを参照し，不正なメモリアクセスを検出する．

バイナリレベルで検査を行う手法は，我々の手法と異なり，(1) 検査対象プログラムの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（たとえば，未初期化変数の読み込みやメモリリーク）も検出できるという利点を持つ．特に，(1) により，対象プログラムが利用するコンパイル済みのライブラリ内部の不正メモリアクセスも検査対象となる（我々の手法ではバイナリ形式のライブラリ関数は検査できない）．

しかし，バイナリレベルでのメモリ検査は，以下に述べる理由により，ソースレベルでの検査（我々の手法を含む）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 1,000%以上）．第 1 に，バイナリコードはスタック領域に割り当てられるメモリオブジェクトについて正確な型情報を持たない．その結果，バイナリレベルでの検査はスタック上で発生する不正メモリ操作（境界違反を含む）を高精度に検出することができない．実際，Purify と Valgrind は，スタックフレームの境界は把握できるものの，フレーム内の個々のメモリオブジェクトの境界を正確に把握することはできず，フレーム内の境界違反に対して検出漏れを引き起こす．

第 2 に，バイナリレベルのメモリ検査は，全メモリアクセス命令を対象とし，各命令の検査時にはアクセス範囲に含まれる全メモリバイトまたは全メモリビットのメタデータを参照する．したがって，ソースレベルの境界検査では検査する必要のなかった変数の参照や代入などのメモリアクセスが，バイナリコードの境界検査では検査対象に含まれてしまう．また，オブジェクト単位でメタデータ（境界情報など）を管理するソースレベルの検査に比べ，バイナリレベルの検査は，より粒度の細かいバイト単位またはビット単位でメタデータ（バイトまたはビットの状態）を追跡管理する．このため，バイナリレベルの検査はソースレベルの検査に比べ，メタデータの追跡管理のコストが大きいだけでなく，各メモリアクセスの検査時に参照するメタデータの数も多い[*12]．

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持つ関数に置換する．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 [11]．

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．

[*12] ソースレベルの検査では，アクセス対象のオブジェクトに関連づけられたメタデータ 1 個を参照するだけでよいが，バイナリレベルの検査では，アクセス範囲に含まれる全バイトまたは全ビットのメタデータを参照しなければならない．

5.2.3 スタック保護

スタック破壊は最も危険な攻撃の 1 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こと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き換えを無効化する [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ことで，バッファオーバフロー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．しかし，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 [14]．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保証しつつ不要な検査を除去する [15]．さらに，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 [16]．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 ポインタに関連付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．さらに，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）ため我々の手法より高速である．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ことが多い．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 章で説明したとおり，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 [17] はオブジェクト表に基づく初期の検査手法であり，RL [18] 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．さらに，RL は境界検査を文字列操作に限定することで実行オーバヘッドを大幅に削減した．DA [19] は自動プール割当て [41] 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さらに低減させた．

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．さらに，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境界検査はシグナル処理中に深刻な互換性の問題（競合状態）を引き起こしてしまう．これに対し，我々の手法は互換性を維持したままシグナル処理中の境界違反を検査することができる．

6. 結論と今後の展望

従来のオブジェクト表に基づく境界検査手法はシグナル処理中に深刻な互換性の問題を引き起こしてしまう．互換性の問題を回避するためにシグナル処理中の境界検査を抑制すると，今度は検査精度が低下してしまう．この問題に対する解決手法として，我々はシグナル処理中の境界検査の実行をシグナル処理後まで保留することを提案し，検査バッファリングと間接シグナル処理によって実現した．実験の結果，我々の手法はシグナル処理を含む実用 C プログラムに対しても，互換性を損なうことなく高精度な境界検査を実行できることが確認できた．今後の課題は，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させることである．自動プール割当ての導入は有望な候補の 1 つであり，導入により DA と同程度に高速な境界検査を実現できると我々は予想している．

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ました．

参考文献

[1] CERT/CC, available from ⟨http://www.cert.org/advisories/⟩.

[2] Evans, D., Guattag, J., Horning, J. and Tan, Y.: LCLint: A Tool for Using Specifications to Check Code, *Proc. 2nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 1994/FSE-*

2), pp.87–96, ACM (1994).
[3] Wagner, D., Foster, J., Brewer, E. and Aiken, A.: A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities, *Proc. 7th Annual Network and Distributed System Security Symposium* (*NDSS 2000*), pp.3–17, ISOC (2000).
[4] Larochelle, D. and Evans, D.: Statically Detecting Likely Buffer Overflow Vulnerabilities, *Proc. 10th Conference on USENIX Security Symposium* (*USENIX Security '01*), pp.177–190, USENIX Association (2001).
[5] Dor, N., Rodeh, M. and Sagiv, M.: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C, *Proc. 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI 2003*), pp.155–167, ACM (2003).
[6] Clarke, E., Kroening, D. and Lerda, F.: A Tool for Checking ANSI-C Programs, *Proc. 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS 2004*), pp.168–176 (2004).
[7] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beatie, S., Grier, A., Wagle, P., Zhang, Q. and Hinton, H.: Stack-Guard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, *Proc. 7th Conference on USENIX Security Symp.* (*USENIX Security '98*), pp.63–78, USENIX Association (1998).
[8] Vendicator: Stack Shield technical info file v0.7 (2001), available from ⟨http://www.angelfire.com/sk/stackshield/⟩.
[9] Etoh, H. and Yoda, K.: GCC extension for protecting applications from stack-smashing attacks (2000), available from ⟨http://www.trl.ibm.com/projects/security/ssp/⟩.
[10] Perens, B.: Electric Fence (1998), available from ⟨http://perens.com/FreeSoftware⟩.
[11] Baratloo, A., Singh, N. and Tsai, T.: Transparent Run-Time Defense Against Stack-Smashing Attacks, *Proc. 2000 USENIX Annual Technical Conference*, pp.251–262, USENIX Association (2000).
[12] Hastings, R. and Joyce, B.: Purify: Fast Detection of Memory Leaks and Access Errors, *Proc. USENIX Winter Technical Conference*, pp.125–138, USENIX Association (1992).
[13] Nethercote, N. and Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation, *Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '07*), pp.89–100, ACM (2007).
[14] Austin, T., Breach, S. and Sohi, G.: Efficient Detection of All Pointer and Array Access Errors, *Proc. 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '94*), pp.290–301, ACM (1994).
[15] Jim, T., Morriset, G., Grossman, D., Hicks, M., Cheney, J. and Wang, Y.: Cyclone: A safe dialect of C, *Proc. 2002 USENIX Annual Technical Conference*, pp.275–288, USENIX Association (2002).
[16] Necula, G., Condit, J., Harren, M., McPeak, S. and Weimer, W.: CCured: Type-Safe Retrofitting of Legacy Software, *ACM Trans. Prog. Lang. Syst.*, Vol.27, No.3, pp.477–526 (2005).
[17] Jones, R. and Kelly, P.: Backwards-compatible bounds checking for arrays and pointers in C programs, *Proc. 3rd International Workshop on Automatic Debugging*, pp.13–26 (1997).
[18] Ruwase, O. and Lam, M.: A Practical Dynamic Buffer Overflow Detector, *Proc. 11th Annual Network and Distributed System Security Symposium* (*NDSS 2004*), pp.159–169, ISOC (2004).
[19] Dhurjati, D. and Adve, V.: Backwards-Compatible Array Bounds Checking for C with Very Low Overhead, *Proc. 2006 International Conference on Software Engineering* (*ICSE '06*), pp.162–171, ACM (2006).
[20] Arahori, Y., Gondow, K. and Maejima, H.: Cache-Based Bounds Checking for Multi-Threaded C Programs, *Proc. 2009 IASTED Conference on Parallel and Distributed Computing Systems* (*PDCS 2009*), IASTED, CD-ROM (2009).
[21] Free Software Foundation (FSF): *GCC, the GNU Compiler Collection*, available from ⟨http://gcc.gnu.org/⟩.
[22] The Apache Sotware Foundation: *Apache HTTP SERVER PROJECT*, available from ⟨http://httpd.apache.org/⟩.
[23] The Sendmail Consortium, available from ⟨http://www.sendmail.org/⟩.
[24] Sleator, D. and Tarjan, R.: Self-adjusting binary search trees, *J. ACM*, Vol.32, No.3, pp.652–686 (1985).
[25] SUSv3: The Single UNIX Specification, Version 3, available from ⟨http://www.unix.org/what_is_unix/single_unix_specification.html⟩.
[26] Brugge, H.T.: boundschecking (2005), available from ⟨http://sourceforge.net/projects/boundschecking/⟩.
[27] IEEE POSIX: Portable Operating System Interface, IEEE Std. 1003.1-2004, available from ⟨http://www.unix.org/single_unix_specification/⟩.
[28] Herlihy, M. and Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proc. 1993 ACM International Symposium on Computer Architecture* (*ISCA 1993*), pp.289–300, ACM (1993).
[29] Shavit, N. and Touitou, D.: Software Transactional Memory, *Proc. 1995 ACM International Symposium on Principles of Distributed Computing* (*PODC 1995*), pp.204–213, ACM (1995).
[30] Henderson, P. and Morris, J.: A Lazy Evaluator, *Proc. 1976 ACM SIGPLAN Symposium on Principles of Programming Languages* (*POPL 1976*), pp.95–103, ACM (1976).
[31] Friedman, D.P. and Wize, D.S.: CONS Should not Evaluate Its Arguments, *Proc. 3rd International Colloquium on Automata, Languages and Programming*, pp.257–284, Edinburgh University Press (1976).
[32] Arahori, Y., Gondow, K. and Maejima, H.: TCBC: Trap Caching Bounds Checking for C, *Proc. 2009 IEEE International Conference on Dependable, Autonomic and Secure Computing* (*DASC '09*), pp.49–56, IEEE Computer Society (2009).
[33] Wheeler, D.: SLOCCount, available from ⟨http://www.dwheeler.com/sloccount/⟩.
[34] Valgrind-project: Crocus: A signal-handler checker (2005), available from ⟨http://valgrind.org/downloads/variants.html?njn⟩.
[35] SecurityFocus, available from ⟨http://online.securityfocus.com/⟩.
[36] MITRE, available from ⟨http://www.mitre.org/⟩.
[37] HP Labs: *The httperf homepage*, available from ⟨http://www.hpl.hp.com/research/linux/httperf/⟩.
[38] The OpenSSL Project: *The Open Source toolkit for SSL/TLS*, available from ⟨http://www.openssl.org/⟩.
[39] The cURL Project: *cURL and libcurl*, available from ⟨http://curl.haxx.se/⟩.

[40] The Postfix Project: *The Postfix Home Page*, available from ⟨http://www.postfix.org/⟩.
[41] Lattner, C. and Adve, V.: Automatic pool allocation: Improving Performance by Controlling Data Structure Layout in the Heap, *Proc. 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '05*), pp.129–142, ACM (2005).
[42] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L. and Malik, S.: Chaff: Engineering an Efficient SAT Solver, *Proc. 38th Design Automation Conference* (*DAC 2001*), pp.530–535, ACM (2001).

荒堀 喜貴

2010 年東京工業大学大学院情報理工学研究科計算工学専攻博士後期課程修了．同年同大学院同専攻特別研究員．2011 年より電気通信大学情報システム学研究科情報システム基盤学専攻助教．専門はシステムプログラミング，プログラム解析，高信頼計算，並列処理．博士（工学）．IEEE，ACM，USENIX 各会員．

権藤 克彦

1994 年東京工業大学大学院理工学研究科博士課程情報工学専攻修了．同年同大学院情報理工学研究科情報工学専攻助手，講師を経て，1998 年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 年）．2003 年より東京工業大学助教授．2011 年より同大学教授．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持つ．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，電子情報通信学会各会員．

前島 英雄 （正会員）

1973 年東京工業大学大学院理工学研究科修士課程制御工学専攻修了．同年（株）日立製作所入社，部長，主管研究員を経て，1999 年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ガラブル・アーキテクチャに興味を持ち，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 会員，電子情報通信学会フェロー．