Backwards-compatible bounds checking for arrays and pointers in C programs

Richard W M Jones and Paul H J Kelly

May 1997

Department of Computing
Imperial College, London

Intro:

- Most C and C++ bugs are due to pointer or array bounds errors
- Even for C and C++, fairly good tools have existed for some time which catch bounds errors
- Most programmers don't use them

Why poor take-up?

- Performance
- Convenience
- False positives
  — unnecessary warnings
- False negatives
  — uncaught errors

Performance

- Good enough to use bounds checking in production code?

  Some techniques are quite close

- Good enough for most software development purposes?

  Generally, programmers will accept quite large overheads during debugging

⋆ **Problem:**

  Unlike ordinary arrays, C's pointers make it hard to mix checked code with unchecked code

The bounds checking problem in C

- A pointer in C can be used in a context divorced from the name of the storage region for which it is valid, its *intended referent.* So instead of

  ```
  printf(A[n]);
  ```

  we get

  ```
  int A[10]; ... p = A;
    ... p += n; ... print(*p);
  ```

- To check whether `*p` is valid, we need to find out which storage allocation it was derived from

  For example, consider

  ```
  int A[10], B[10]; ... p = A
    ... p += 10 ... print(*p);
  ```

  Here, `p` probably points to a valid region but is improperly derived.

- We need to check that the storage region has not been de-allocated, either explicitly or by block exit

How to do it . . . 1

- **Change pointer representation:**

  Structure pointer to provide information about the intended referent [S.C.Kendall, 1983, J.L.Steffen, 1992]

- **Add "guard" variables:**

  For each pointer variable or parameter, add a "guard" variable which provides information about the intended
  referent [Patil and Fischer, 1996]

⋆ *Problem:*

  Both fail to inter-operate with code compiled without checking

  E.g. consider function-typed variables, virtual functions, and call-backs.

How to do it . . . 2

- **Maintain shadow bitmap:**

  Maintain a map indicating which storage regions are valid. Update it when stack allocations, `malloc` and `free` occur. Augment each memory access instruction with code to check whether the address is valid [Hastings and Joyce, 1992].

- **Advantages:**

  Fairly efficient

  Doesn't require access to source code, so can (must) be applied to all constituents of application

- *Problem:*

  False negatives - fails to flag accesses to a valid region using an improperly-derived pointer

Summarise requirements:

- **Track intended referent for each pointer**

  It is not good enough just to check that accesses are to valid locations

- **No change to pointer representation**

  In order to inter-operate with unchecked code without restriction, no information can be bundled with the pointer.

How to do it . . . 3: the central idea

Invariant:

Assume all stored pointers are properly-derived pointers to their intended referent

Implementation:

- **Maintain table of valid storage regions**
  - Initialise with global declarations; update with stack and dynamic allocation/deallocations.
  - Given a pointer, find its intended referent by searching the table
- **Check address arithmetic expressions**
  - Check that the result refers to the same storage region as the pointer from which it was derived — i.e. that they have the same intended referent. If not, an error may have occured.

Note: all expressions yielding a pointer result depend on *exactly one* original pointer.

Correctness

Theorem: all stored values of pointer type are always properly-derived pointers to their intended referent.

Proof sketch: By induction:

- **Base case: start of computation**

  Initially, all statically-allocated storage regions are in the object table. All variables are uninitialised.

- **Inductive step:**

  Computation can progress by:

  - **Assignments**
  - **Allocations/de-allocations**
  - **Block entry/exit**

  In each case we maintain the object table to include all valid objects, and we check all assignments to preserve intended referents.

Lemma: Given that intended referents are preserved by address arithmetic, it is easy to check uses of pointers.

Properties of the approach:

- **What if a variable contains a pointer which is not in the table?**
  - An optional warning can be issued immediately
  - The pointer may have originated from unchecked code, so it may be valid to proceed
  - The pointer can be abused to clobber other regions allocated in unchecked code,
  - We can check that it is not used to derive a pointer to a known region, so regions allocated by check code are safe.

  This should never happen if all code is checked.

Another property of the approach:

- **Invalid address arithmetic is detected *before* the result is used**
  - An optional warning can be issued immediately.
  - The pointer is replaced by a dummy so that an error is flagged when it is used.
  - Address arithmetic warnings are sometimes unhelpful false positives.
  - However, it is very useful to be able to detect exactly where the invalid operation occurred.

Another property of the approach:

- **Fragile invariant**

  The result of invalid address arithmetic must not be used to update a pointer.

- Because it may then have a different intended referent, and will be assumed valid.

A fly in the ointment

Some out-of-range pointers are legal

Example:

```
int *p;
int *A = (int *) malloc (100 * sizeof(int));
for (p = A; p < &A[100]; ++p)
   *p = 0;
```

- On exit from the loop, p points to A[100].
- The final ++p increments p beyond the range for which it is valid, although the resulting pointer is never de-referenced.
- According to the definition of permissible pointer operations above, this would be flagged as an error since p may now point to a different object.
- According to the ANSI C standard, this example is legal and further arithmetic on p can be used to yield a valid pointer.

More on legal out-of-bounds pointers

Example B:

```
int *p;
int *A = (int *) malloc (100 * sizeof(int));
for (p = A; p < &A[100]; ++p)
        *p = 0;
while (p > A) {
        p -= 1;
        *p = 0;
}
```

Example C:

```
int *p;
int *A = (int *) malloc (100 * sizeof(int));
for (p = &A[99]; p >= A; --p)
        *p = 0;
```

Solution

- **Pad all storage regions by at least one byte**

  So that, if the object is used as an array, a pointer one item beyond the bound cannot refer to different storage region.

- Cost is minimal, often zero due to word alignment and `malloc` administration records

- No problem for inter-operability since checked module's storage layout is freely chosen.

...Except parameters

```
typedef struct {char A[24];} T;
void A(T p1, T p2)
{   int i;
    char *q;
    printf("&p2 = %d\n", &p2);   /* use addr so in table */
    q = (char *)&p1;
    for (i=0; i<48; i++,q++)   /* no pointer comparison */
        putchar(*q);       /* use pointer not subscripting */
}
```

In certain extremely obscure circumstances, false negatives can occur with parameters:

- We cannot change the storage layout for passing parameters to unchecked code.
- This arises with:
  - Adjacent parameters
  - Whose size means there is no intervening padding
  - Both of whose addresses are used

- Which are traversed as arrays
- Using pointers, not subscripting

Implementation

Compile-time:

- Modification to gcc
- Inserts checking into abstract syntax tree
- Don't register an object if its address is never used
- Exploit gcc's support for C++ constructors/destructors to manage stack allocation/deallocation on block entry/exit
- List statically-allocated objecta for table initialisation

Link-time:

- Process unchecked modules' binary to locate statically-allocated storage

Run-time:

- Object table implemented as splay tree

- Malloc/free modified to update table and catch use of freed objects
- Optimised versions of memcpy, strcpy etc.

Performance

- Extremely robust
- Performance is not good
- Slowdown is highly variable
- Worst case 100×

But:

- Slowdown only for checked code
- Some simple optimisations will help a lot
  - Loop invariants: repeated lookup of same object
  - Induction variables: course of values is known and can be checked in loop header
- We will characterise benchmark performance when these optimisations have been implemented.

Summary

- Few bounds checkers for C avoid false negatives by tracking intended referents
- Only ours does so without changing the pointer representation
- This makes inter-operation with unchecked modules, libraries, the OS, and devices much more convenient
- Performance is currently poor but could get much better
- Take-up is still surprisingly low

Further work:

- Optimisation; intra-procedural, inter-procedural
- Improving run-time system, object table data structure
- Checking for accesses to uninitialised data

- Checking bounds errors *within* storage regions

References

[American National Standard for Information Systems, 1990] American National Standard for Information Systems (1990). Programming language C. Technical Report ANSI X3.159-1989, ANSI Inc., New York, USA.

[Hastings and Joyce, 1992] Hastings, R. and Joyce, B. (1992). Purify: fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136.

[J.L.Steffen, 1992] J.L.Steffen (1992). Adding run–time checking to the portable C compiler. *Software – Practice and Experience*, 22(4):305–316.

[Patil and Fischer, 1996] Patil, H. and Fischer, C. (1996). Low-cost, concurrent checking of pointer and array accesses in C programs. *Software Practice and Experience*.

[S.C.Kendall, 1983] S.C.Kendall (1983). Bcc: run–time checking for C programs. In *USENIX Toronto 1983 Summer Conference Proceedings*. USENIX Association, El. Cerrito, California, USA.