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

Richard W M Jones and Paul H J Kelly
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ

Abstract

This paper presents a new approach to enforcing array bounds and pointer checking in the C language. Checking is rigorous in the sense that the result of pointer arithmetic must refer to the same object as the original pointer (this object is sometimes called the 'intended referent'). The novel aspect of this work is that checked code can inter-operate without restriction with unchecked code, without interface problems, with some effective checking, and without false alarms. This "backwards compatibility" property allows the overheads of checking to be confined to suspect modules, and also facilitates the use of libraries for which source code is not available. The paper describes the scheme, its prototype implementation (as an extension to the GNU C compiler), presents experimental results to evaluate its effectiveness, and discusses performance issues and the effectiveness of some simple optimisations.

1 Introduction and related work

C is unusual among programming languages in providing the programmer with the full power of pointers. Languages in the Pascal/Algol family have arrays and pointers, with the restriction that arithmetic on pointers is disallowed. Languages like BCPL allow arbitrary operations on pointers, but lack types and so require clumsy scaling by object sizes.

An advantage of the Pascal/Algol approach is that array references can be checked at run-time fairly efficiently, in fact so efficiently that there is a good case for bounds-checking in production code. Bounds checking is easy for arrays because the array subscript syntax specifies both the address calculation and the array within which the resulting pointer should point.

A pointer in C can be used in a context divorced from the name of the storage region for which it is valid, it's "intended referent", and this has prevented a fully satisfactory bounds checking mechanism from being developed. There is overwhelming evidence that bounds checking is desirable, and a number of schemes have been presented. The main difference between our work and Kendall's `bcc`[13] and Steffen's `rtcc`[7] is that in our scheme the representation of pointers is unchanged. This is crucial, since it means that inter-operation with non-checked modules and libraries still works (and much checking is still possible). Compared with interpretative schemes like *Sabre-C*[14], we offer the potential for much higher performance. Patil and Fischer [10, 11] present a sophisticated technique with very low overheads, using a second CPU to perform checking in parallel. Unfortunately, their scheme requires function interfaces to be changed to carry information about pointers, so also has the inter-operation problem.

Another approach is exemplified by the commercially-available checking package Purify [6]. Purify processes the binary representation of the software, so can handle binary-only code. Each memory access instruction is modified to maintain a bit map of valid storage regions, and whether each byte has been initialised. Accesses to unallocated or uninitialised locations are reported as errors. Purify catches many important bugs, and is fairly efficient. However, Purify does not catch abuse of pointer arithmetic which yields a pointer to a valid region which is not the intended referent. Fischer and Patil [10, 11] provide evidence for the importance of this refinement.

Our goals in this paper are to describe a method of bounds checking C programs that fulfills the following criteria:

- Backwards compatibility — the ability to mix checked code and unchecked libraries (for which the source may be proprietary or otherwise unavailable)
- Works with all common C programming styles

Presented at AADEBUG'97, Linköping, Sweden.

- Rigorously rejects violations of the ANSI C standard
- Checks static and stack objects as well as objects dynamically allocated with `malloc`
- Understands scope of automatic variables
- Performance — including the ability to be able to distribute programs with checks compiled in

There remain some circumstances in which checking is incomplete; as we describe later, these are fairly uncommon in practice. The main shortcoming of the implementation described in this paper is that the performance is currently poor. However, the approach has fundamental performance advantages over previously-published work. Because checked code inter-operates easily with unchecked code, the performance penalty is confined to those modules where it is needed. Furthermore, there is substantial scope for optimisation of loop-invariant pointers and pointers which are induction variables. Because the pointer representation is unchanged, there is no residual overhead once checking code is eliminated. We return to this issue in Section 5.

1.1 Overview of this paper

The next section reviews the problem of bounds checking for C, and the limitations the language places on the checking that can be done. In the following section, the new approach is introduced, and we explain how, unlike earlier schemes, our bounds checking scheme allows inter-operation with unchecked code. Then we give some details of our implementation, and discuss some optimisations and their effectiveness. Finally, we discuss the effectiveness of the scheme in the light of our experience with some large and well-known C programs.

2 Objects, bounds checking in C, and its limitations

ANSI C conveniently allows us to define an *object* as the fundamental unit of memory allocation. Objects are created by declarations or allocations such as those shown in Table 1, which may be static, automatic (i.e. stack-allocated), or dynamically allocated.

Objects are stored sequentially in memory and cannot overlap. Operations are permitted which manipulate pointers within objects, but pointer operations are not permitted to cross between two objects. There is no ordering defined between objects, and the programmer should never be allowed to make assumptions about how objects are arranged in memory.

Bounds checking is not blocked or weakened by the use of a cast (i.e. type coercion). Casts can properly be used to change the type of the object to which a pointer refers, but cannot be used to turn a pointer to one object into a pointer to another. A corollary is that bounds checking is not type checking: it does not prevent storage from being declared with one data structure and used with another.

More subtly, note that for this reason, bounds checking in C cannot easily validate use of arrays of `structs` which contain arrays in turn.

Casts and unions can be used to create a pointer from an object of any other type, in a machine-dependent way. This cannot be checked using our technique, nor by earlier approaches to bounds checking, since there is no object for the pointer to be derived from.

3 The technique and its advantages

In this section we review earlier approaches and explain the basis for the new approach.

3.1 Earlier approaches to carrying bounds information

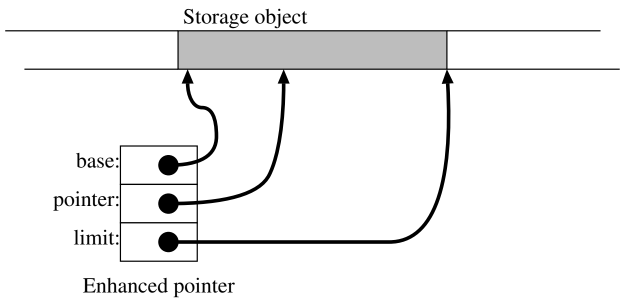

Figure 1: Modified pointer representation: pointer–base-address–extent triple

In earlier work in this area[5, 14, 13, 7, 8, 10, 11], bounds information is carried with each pointer at run-time. A simple approach is to represent each pointer as a triple: the pointer, together with the storage region's base address and limit or extent. Checking is then straightforward. The larger size of pointers requires changes in storage allocation, and the code generator must be modified to copy pointers correctly. The change in pointer size can be avoided by replacing each pointer with an index into a table, which contains the pointer-base-limit triple.

The net effect of both methods is the same. When the program, at runtime, comes to use a pointer, it must first verify that the operation that is about to be

`int a;`	A simple variable
`int a[10];`	An array
`struct { /*...*/ } a;`	A single record
`struct { /*...*/ } a[10];`	An array of records
`malloc(10);`	A single unit of memory allocated with `malloc`

Table 1: Typical objects.

performed is correct. It uses the information about the base and size of the array or structure being pointed to to decide if a particular index is legal.

3.2 Unchanged pointer representation

The problem with both these schemes is that the modified pointer representation is not interpreted correctly by code compiled without bounds checking enabled. This is a problem wherever a pointer is passed to or from an unchecked procedure, whether as a parameter, a result, or in a global variable. It is, of course, often possible to translate pointers where necessary (called *encapsulation* in `bcc`[13] and `rtcc`[7]), but this is inconvenient and difficult to do reliably (e.g. where a function pointer may refer either to checked or an unchecked routine). Because of these difficulties, in `rtcc` only operating system calls are encapsulated – all libraries must be recompiled.

In this paper we show that the pointer representation need not be changed. This avoids the need either for encapsulation or recompilation. The result is improved functionality (e.g. to work with modules and libraries provided in binary-only form), and potentially also improved performance, since well-tested modules can run without checking.

3.3 Checking pointer use: how the scheme works

Given these considerations, in our method pointers are represented as simple addresses, as in ordinary C programs. We maintain a table of all known valid storage objects. Using the table we can map a pointer to a descriptor of the object into which it points, which contains the base, extent and additional information to improve error reporting.

We have to check both pointer arithmetic and pointer use. Pointer arithmetic must be checked because the result must never be allowed to refer to an object different from the one from which it is originally derived. This is because the object for which the pointer is valid can only be determined by checking the pointer itself, by looking it up in the object table.

Every valid pointer-valued expression in C derives its result from exactly one original storage object. If the result of the pointer calculation refers to a different object, it is invalid.

Although it sometimes useful to know where an invalid pointer has been calculated, reporting every instance can yield many false alarms. We therefore replace such incorrectly-derived pointers with a pointer value which is always invalid, called `ILLEGAL` (Defined as `(void *)-2` in our implementation). This ensures that a bounds error is reported when the pointer is actually used.

3.4 Example: pointers to objects

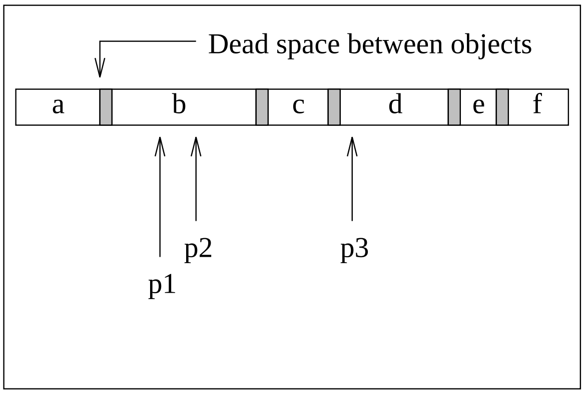

Figure 2: Objects arranged in memory.

Figure 2 shows an example layout for several objects of various sizes, perhaps arising from static allocations, or from calls to `malloc`. Suppose we have pointers `p1`, `p2` and `p3` referring to the objects, or perhaps to their internal components (their type is immaterial since casts may have been used). Table 2 shows permissible pointer operations given the rule that pointer operations are only permitted to take place within an object, and not between objects.

`p2 - p1`	Permitted. Both pointers are within the same object.
`p3 - p2`	Not permitted. Makes assumptions about the layout of objects in memory.
`Increment p2 until p2 == p3`	Not permitted. As soon as `p2` is incremented beyond the end of object `b`, a bounds error will be reported.

Table 2: Permissible operations on pointers `p1`–`p3` in Figure 2

3.5 Problem: legal out-of-range array pointers

An awkward complication arises with arrays. Consider the (correct) code in Figure 3.

```
f()
{
  int *p;
  int *a = (int *) malloc (100 * sizeof(int));
  for (p = a; p < &a[100]; ++p)
    *p = 0;
  return a;
}
```

Figure 3: Iterating over an array.

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 used. According to the definition of permissible pointer operations above, this should be flagged as an error since `p` may now point to a different object.

The ANSI C standard[1] (section 3.3.6, lines 24–27) states that for an array declared `Type a[N];`, a programmer may only generate pointers to elements `a[0]`, `a[1]`, up to `a[N]`. The last element does not literally exist, and any attempt to dereference a pointer to `a[N]` will result in undefined behaviour (or in our case, a bounds error). It is not permissible to create a pointer to, for instance, element `a[-1]` of an array, and such programs will not be portable to architectures where all objects are stored in separate segments.

To overcome this problem, we place at least one byte of dead space between objects in memory (allocations are often aligned to 4 or 8 byte boundaries in memory so there may be several bytes between adjacent objects). A pointer to `a[N]` can now be distinguished from a pointer to the next adjacent object in memory[1] (see the Appendix for an example).

Unfortunately we cannot pad parameters passed to functions (since this would mean that the parameter layouts assumed by checked and non–checked code would be incompatible). This results in a small ambiguity. We resolved this partially in our implementation by flagging function parameters and treating them specially. Essentially, when looking up pointers to parameters, we treat a reference to "`a[N]`" as a possible pointer to the next object in memory. If there is an adjacent parameter, then the pointer will point to the next object.

This is an instance where checking is incomplete: a pointer to an array passed as a parameter can be incremented to point to the later parameters without an error being reported. Using the pointer to refer to earlier parameters or elsewhere will be trapped correctly.

In practice this solution was satisfactory, since although it is possible to pass actual structures and structures containing arrays as parameters, this is very rare, and even then most cases can be caught. The infrequency of use, and the fact that we catch many cases anyway, make this potential loophole an extremely minor concern.

3.6 Objects originating in unchecked code

When an object is allocated in checked code, it is entered in the object table. When the resulting pointer is used in checked code, bounds checking works fully. If the pointer is passed to unchecked code, unchecked accesses can occur.

When a pointer is passed from unchecked to checked code, it may originate either from a checked or unchecked allocation (note that dynamically-allocated objects are always registered since even unchecked code must call the checked `malloc` function).

There are two cases:

1. The pointer passed from unchecked to checked code points into a checked object.

   This may be correct, as it may have been derived from a pointer passed to it, or it may be the result

[1] There is a subtle assumption here: if the size of the object were not an integer multiple of the array element size, then a[N] could lie more than one byte beyond its limit (depending on the size of the element type). However, this case is a bounds error since there is insufficient space for a[N-1].

from a call to the (modified) `malloc` storage allocator. In this case, checking will proceed normally.

It may be incorrect: the pointer may be improperly derived from some other object. This case is indistinguishable and no error will be reported.

2. The pointer passed from unchecked to checked code points into an object which does not appear in the object table because the space was allocated in unchecked code.

   This is detected when the pointer is used. Although it may be helpful to issue a warning message and to perform basic sanity checks, the program can proceed without false alarms. This is because the key check is whether, in pointer arithmetic, the result refers to the same object as the pointer from which it was derived. If the original pointer is not registered, the result should not be. Accidental use of unchecked pointers in checked code to damage checked objects is thereby prevented.

3.7 Maintaining the object table: tracking creation and deletion of objects

At run–time, we track objects as they are created and deleted. We maintain an ordered list of objects in memory, and employ a fast method to convert pointers to objects. Several suitable structures are available for this purpose. We used a splay tree in our implementation[4, 3] but other structures such as tries and skiplists might be suitable.

Static objects (global variables, variables declared as `static` in functions and string constants) persist over the lifetime of the program. A simple modification to the compiler and/or the linker can be made to produce a list of these objects. As indicated above, it is not necessary to find objects in the unchecked parts of the code.

Dynamically allocated objects — those declared with `malloc` and destroyed with `free` — can be tracked by a simple modification to the C library. Although `malloc` often introduces padding anyway, care is needed with objects allocated dynamically by other means (such as `mmap` and `sbrk`).

Stack objects present greater difficulties, since the C `goto` command may mean that they are created and destroyed at several different places in the code (see Figure 4).

In this code fragment, `b` is in scope between the inner set of curly brackets. The `goto label1;` statement has the side effect of creating `b` and `goto label2;` destroys it. In addition, `b` must be created and destroyed if and when control passes the inner curly brackets.

```
f ()
{
  int a;
  if (...) goto label1;
  {
    int b;
    /* ... */
  label1:
    /* ... */
    if (...) goto label2;
    /* ... */
  }
label2:
  /* ... */
}
```

Figure 4: Stack objects created and destroyed by `goto`.

In our implementation, we used the C++ constructor/destructor mechanism of our compiler (GCC) to track such variables. This is fairly common since many C compilers are built to handle C++ too. Details lie beyond the scope of this paper.

Parameters are a special form of stack object. Care must be taken to ensure that parameters are created once on entry to the function, and deleted on exit, even if the procedure exits with `return` early on. The C++ constructor/destructor mechanism can handle this too.

Ordinary stack objects must be padded as described in section 3.4. Parameters are not padded, so that checked and unchecked functions have compatible parameter layouts. ANSI C prevents using the return value of a function as an lvalue immediately. Since all return values are therefore copied into a variable in the calling function, there is no need to take special action checking or padding aggregate function results.

4 Implementation in an existing compiler

We implemented our bounds checking scheme in the GNU C compiler (GCC). In this section we briefly explain how this was done. The resulting program is freely available from a variety of sources[12].

4.1 Checking pointer operations

We altered GCC to replace pointer operations with calls to a library of checking functions. Typically when the programmer writes `p + i`, where `p` has a pointer type and `i` is an integer, the compiler replaces it with:

```
(T *) __bounds_check_ptr_plus_int(p, i, sizeof(T),
                                  __FILE__, __LINE__);
```

`T *` is the type of the pointer `p`, `__FILE__` and `__LINE__` are macros that expand to the current file and line number, and are used to locate errors when they occur.

operator/operand types	
pointer [integer]	*(array reference)*
pointer -> element	*(reference to record field)*
pointer + integer	*(yields pointer)*
pointer - integer	*(yields pointer)*
pointer - pointer	*(yields integer)*
pointer < pointer	*(comparisons)*
pointer > pointer	
pointer <= pointer	
pointer >= pointer	
pointer == pointer	
pointer != pointer	
*pointer	*(dereference),*
pointer++	*(post-increment)*
pointer--	*(post-decrement)*
++pointer	*(pre-increment)*
--pointer	*(pre-decrement)*

Table 3: Operators requiring checking

Table 3 shows the operators where checking code has to be added. Note that we must check pointer arithmetic as well as pointer use. We also check pointer comparisons and subtractions since the result is valid only if the operands refer to the same aggregate.

In order to handle compound operators correctly and efficiently, we specifically detect and replace the following patterns:

- `&*`*pointer* is replaced with *pointer*
- `&`*pointer*[*integer*] is replaced with *pointer* + *integer*
- `&`*pointer* `->` *element* is replaced with *pointer* + *offsetof(element)*.

As described above, certain pointer operations silently return the special representation `ILLEGAL` (Defined as `(void *)-2` in our implementation) when they fail. This allows programmers to make illegal pointers, and only have them caught later if the programmer attempts to dereference them. For instance, in an array declared `int a[10];`, attempting to generate `a+15` results in an `ILLEGAL` pointer which is caught when used later. All pointer operations catch `ILLEGAL` pointers passed and throw bounds errors.

4.2 Using existing C++ mechanisms to track stack objects

Ordinary stack objects (not function parameters) are padded by tricking GCC into believing they are one byte larger than they really are. A patch to the GCC `alloca` function catches variable–sized stack objects.

In order to de-register stack-allocated objects on block exit, we used the constructor/destructor mechanism built into GCC and designed to handle C++ objects, even where they may be created or destroyed by uses of `goto`. The code shown in Figure 5 contains several stack variables in different scopes. The code is compiled as if the user had written the version in Figure 6.

4.3 Finding statically allocated objects at compile and link time

We modified the back-end of GCC slightly to construct a table of statically allocated objects, such as global variables and string constants. Each source file compiled with bounds checking enabled will contain such a table, and this is automatically loaded at run-time before the program starts running. The design of GCC enabled this to be done in a straightforward manner.

It is desirable to track down objects declared in unchecked code too, although not strictly necessary as described earlier. A simple tool was written that takes a library archive or object file, and writes out a table of static objects contained therein. This table can then be linked to the program.

Static objects are padded by asking the linker to allocate one extra byte after each object.

4.4 Minimal modifications to `malloc` and `free`

We modified the GNU `malloc` library to register dynamically allocated objects as they are created, and deregister them as they are freed. A single extra byte of padding is added to each object when it is allocated.

The new library is linked automatically and replaces all calls to the previous `malloc` family of functions.

4.5 Modifications to C library functions

Unlike many other C compilers, GCC usually works with the system–installed C library on whatever operating system it runs. In most instances, the source to these libraries is not freely available, so users will be forced to run them without bounds checking. This implies that a call to a function such as `strcpy`, passing a bad pointer,

```
int sum (int n, int *a)
{
  int i, s = 0;
  for (i = 0; i < n; ++i)
    s += a[i];
  return s;
}
```

Figure 5: Vector sum example with stack objects.

```
int sum (int n, int *a)
{
  /* __bounds_push_function enters a function context. A
   * matching call to __bounds_pop_function will
   * delete parameters.
   */
  __bounds_push_function ("sum");
  __bounds_add_parameter_object (&n, sizeof (int), ...);
  __bounds_add_parameter_object (&a, sizeof (int*), ...);
  /* Extra scope created around the function. GCC will
   * call __bounds_pop_function when leaving this
   * scope.
   */
  {
    /* Declare stack objects, and use GCC's destructor
     * mechanism to ensure __bounds_delete_stack_object is
     * called for each variable however we leave scope
     * (even if we leave with goto).
     */
    int i;
    __bounds_add_stack_object (&i, sizeof (int), ...);
    int s = 0;
    __bounds_add_stack_object (&s, sizeof (int), ...);

    for (i = 0; i < n; ++i)
      s += *(int*)
           __bounds_check_array_reference (a, i,
                                           sizeof (int), ...);
    __bounds_delete_stack_object (&s);
    __bounds_delete_stack_object (&i);
  }
end:
  __bounds_pop_function ("sum");     /* Delete a, n. */
  return s;
}
```

Figure 6: Vector sum example with stack object management using the C++ constructor/destructor mechanism.

will not result in a bounds error, but in a segmentation fault, or in random damage to memory.

To detect such errors, we replaced many C library functions, with efficient bounds–checked versions. Calls to the ANSI `str*` and `mem*` functions are checked in this way. The implementations of `memcpy` and `strcpy` also check for illegal copying of overlapping memory segments.

4.6 Splay trees to look up pointers quickly

In order to reduce the overhead of converting pointers to objects on the occasions when that is necessary, we store the object list as a splay tree[4, 3]. Splay trees are binary trees where frequently used nodes migrate towards the top of the tree. In tests it was found that the look–up function was iterated on average 2.11 times per call on a typical large program. We unrolled the first two iterations of the loop to optimise these cases.

5 Performance and optimisations

For the bounds checking scheme outlined above to be useful, careful consideration must be given to optimising the code produced. In particular, it is possible to reduce the number of accesses to the splay tree that are required quite considerably. In the next few paragraphs we describe some simple optimisations we have implemented, some further optimisations which should be straightforward to add, and we briefly discuss the problematic cases which remain.

5.1 Eliminating calls to register unused variables

If the programmer never takes the address of a stack variable, then no pointer can ever be generated that refers to that variable, and so it is unnecessary even to consider that variable for bounds checking purposes. This is extremely effective, as addressable local variables are rare in typical programs.

5.2 Eliminating look–ups in loops over arrays

For further significant gains in performance, we suggest a simple scheme for optimising loops over arrays using code motion. Consider the fragment of code in Figure 7 after bounds checking code has been added in a simple-minded way. In Figure 8 we have made the pointer-to-object conversion explicit by inlining part of the procedure call.

```
int a[10], i;
for (i = 0; i < 10; ++i)
  /* This is the code substitution for 'a[i] = i;' */
  *(int*)
  __bounds_check_array_reference(a, i, sizeof(int), ...) = i;
```

Figure 7: Code after simple-minded substitution of a checking function.

```
int a[10], i;
for (i = 0; i < 10; ++i)
{
  object *obj = __bounds_find_object (a);
  if (obj && obj->base <= &a[i]
        && &a[i] < obj->extent)
    a[i] = i;
  else
    /* throw a bounds error and exit */
}
```

Figure 8: Code after partially inlining the checking function.

Clearly the call to do the pointer-to-object conversion (`__bounds_find_object`) should be moved outside the loop in the code motion phase of the optimiser. An efficient compiler would then be able to remove the bounds checking tests (`obj->base <= &a[i]` and `&a[i] < obj->extent`) entirely and replace them with two tests outside the loop.

This may be done if there is a way to specify that the call is a constant function (ie. has the same return value when called multiple times) provided that objects are not added or deleted in between calls. GCC does not provide a way to encapsulate this subtlety, and so our implementation does not yet make this optimisation.

Loops which iterate through arrays using pointers (instead of incrementing an array subscript as above) are more difficult: `__bounds_find_object` will be applied to the pointer, which is not loop invariant. Here a more specialised optimisation for induction variables should help.

5.3 Difficulties optimising loops over linked structures

Loops over linked lists, tree structures and the like provide a greater challenge. We were not able to devise an efficient method of optimising loops that traverse linked data structures, although the splay tree we used to implement the object table will tend to cache frequently used objects like the elements in the list near the top.

6 Evaluation

We have used the modified compiler to recompile a wide variety of applications software. In this section we review our experience with reference to some substantial and freely-available C programs. We comment on the problems we encountered, the effectiveness of the scheme in finding errors, and the performance of the resulting code with bounds checking enabled for the entire program (excluding libraries).

We compiled the scripting and GUI language Tcl/Tk[9] in its entirety (around 120,000 lines of code). We made 11 changes to the source code (see table 4).

	no. of instances
Contravening ANSI standard by pointing to negative array offsets.	2
Fixing pointer nasties, such as adding offsets to `NULL` pointers.	3
Using pointers that refer to objects freed in a `realloc`.	2
Changes to support `goto` restriction caused by using C++ constructor and destructor mechanism.	4

Table 4: Changes made to the source of Tcl/Tk.

The resulting interpreter ran all the Tk demos correctly, although noticably more slowly than without checking. The interactive scripts were still quite usable and responsive, but the authors would not recommend using bounds checking in production code until the further optimisations suggested above have been made.

We also compiled Ghostscript, a freeware PostScript™ interpreter. We needed to fix the non–ANSI implementation of stacks that Ghostscript uses (it initializes pointers to the –1 element of each stack), but the changes involved were relatively minor, and the program ran without error. Again, there was a noticable slowdown when drawing complex graphical images, but the program was by no means unusable.

GCC itself compiles with the bounds checking patches. Unfortunately, GCC makes extensive use of *obstacks*, which are large singly–allocated areas of memory that may contain many variable–sized objects. Since the bounds checking library treats these areas of memory as single objects, simple bounds errors between the elementary objects contained inside are not detected. In hindsight, we should have modified GCC's obstack library very slightly to interact correctly with the bounds checking library (by allocating and deleting the simple objects explicitly).

MicroEMACS, a simple text editor that has been ported and used widely, actually has bounds errors which this program picked up immediately.

Although it is possible to construct programs that perform very badly indeed when bounds checking is added — such as programs that solely iterate over long linked lists, doing almost no work at each node — real programs are for the most part quite usable. Nevertheless, a good implementation of this technique must consider optimisation issues very carefully. It is unlikely that we could ever achieve the 10–15% performance loss that would be acceptable if programs are to be distributed with bounds checks compiled in. In practice, most programs showed a 5–6 times slowdown, which is comparable to other commercial bounds checking packages.

Fischer and Patil [10, 11] provide interesting evidence for the practical importance of checking pointers are used to refer only to the intended referent, compared with the checking provided by tools such as Purify.

7 Further work

We plan to investigate optimisation techniques further, and when we have done so we will present benchmark performance comparisons. While we hope to achieve fairly good performance using conventional data flow analysis as described earlier, there is also scope for inter-procedural optimisation, and ultimately it may be possible to validate non-trivial examples at compile-time, using, for example, partial evaluation [2].

The range query lookup on which checking is based is critical to performance, and there is scope for experimental work to tune our splay tree approach and to study alternatives.

There remain some loopholes in our checker. The most serious in practice is that it is possible to manufacture erroneous pointers using unions, casts and unitialised data. At considerable performance cost, we could maintain a shadow of the accessible store, indicating whether it has been initialised and whether it is a pointer. There is scope for optimisation, and doing so would be a substantial project.

8 Summary and conclusions

We have shown how bounds checking can be provided in a convenient form, with recompilation confined to the files where problems are suspected. The execution time penalty for code compiled with bounds checking enabled is substantial, but in many cases this can be alleviated by optimisation, and this is the most pressing direction for further enhancements. The technique has been applied to a wide variety of C programs with generally good results.

Acknowledgements We would like to acknowledge the helpful comments of our anonymous referees.

References

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

[2] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/19.

[3] D.Clark. Splay trees. *Dr. Dobb's Journal*, page 56ff, December 1992.

[4] D.D.Sleator and R.E.Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

[5] D.W.Flater, Y.Yesha, and E.K.Park. Extensions to the C programming language for enhanced fault detection. *Software – Practice and Experience*, 23(6):617–628, June 1993.

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

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

[8] M.V.Zelkowitz, P.R. McMullin, K.R.Merkel, and H.J.Larsen. Error checking with pointer variables. In *Proceedings of the 1976 ACM National Conference*. ACM, New York, USA, 1976.

[9] John Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[10] Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. In *2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG'95), St Malo, France*, May 1995.

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

[12] R.W.M.Jones. Bounds checking patches for the GNU C compiler. Available via the world-wide web from `http://www-dse.doc.ic.ac.uk/~rj3/bounds-checking.html` and via anonymous `ftp` from `ftp://dse.doc.ic.ac.uk/pub/misc/bcc`.

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

[14] S.Kaufer, R.Lopez, and S.Pratap. Saber–C: an interpreter–based programming environment for the C language. In *USENIX San Francisco 1988 Summer Conference Proceedings*. USENIX Association, El. Cerrito, California, USA, 1988.

Appendix: Examples

This appendix presents a number of small examples which illustrate the technique's power and limitations.

Basic example illustrating simple bounds checking

```
#include <stdio.h>
void main() {
	char A[10]={'1','2','3','4','5','6','7','8','9'};
	char B[10]={'a','b','c','d','e','f','g','h','i'};
	char *p = A;
	while(1)
		putchar(*p++);
}
```

Output from the bounds-checking run-time system:

```
ShowItWorks.c:10:Bounds error: attempt to reference memory overrunning the end of an object.
ShowItWorks.c:10:  Pointer value: 0xeffffae2
ShowItWorks.c:10:  Object 'A':
ShowItWorks.c:10:    Address in memory:    0xeffffad8 .. 0xeffffae1
ShowItWorks.c:10:    Size:                 10 bytes
ShowItWorks.c:10:    Element size:         1 bytes
ShowItWorks.c:10:    Number of elements:   10
ShowItWorks.c:10:    Created at:           ShowItWorks.c, line 6
ShowItWorks.c:10:    Storage class:        stack
123456789
```

Simple example showing A[N] is a valid pointer

```
/* A pointer is allowed to refer to the byte after the object from which it
* is derived.  The array is padded by one byte, if necessary, so that this
* is distinguishable from an illegal operation.
*/

#include <stdio.h>

main()
{
	int a[10], *p;

	/* Initialize array 'a' to 0. */
	for (p = &a[0]; p < &a[10]; p++)
		*p = 0;

	/* Now 'p' points to &a[10], which is a valid address, but if we
	* try to use it, we'll get an error.
	*/
```

```
        *--p = 1;       /* OK => sets a[9] to 1 */
        *++p = 1;       /* Bounds error => tries to set a[10] to 1 */
}
```

Output from the bounds-checking run-time system:

```
OneBeyondArrayBounds.c:20:Bounds error: attempt to reference memory overrunning the end of an object.
OneBeyondArrayBounds.c:20:   Pointer value: 0xeffffae0
OneBeyondArrayBounds.c:20:   Object 'a':
OneBeyondArrayBounds.c:20:     Address in memory:     0xeffffab8 .. 0xeffffadf
OneBeyondArrayBounds.c:20:     Size:                  40 bytes
OneBeyondArrayBounds.c:20:     Element size:          4 bytes
OneBeyondArrayBounds.c:20:     Number of elements:    10
OneBeyondArrayBounds.c:20:     Created at:            OneBeyondArrayBounds.c, line 10
OneBeyondArrayBounds.c:20:     Storage class:         stack
```

Checking of out-of-range automatics

```
#include <stdio.h>

char *G;

void f() {
        char A[10]={'1','2','3','4','5','6','7','8','9'};
        G = A+3;
}

void main() {
        f();
        putchar(*G);
}
```

In this example, the global variable `G` is used to capture a pointer into a stack-allocated array. The pointer is invalid after the function `f` has returned. Output from the bounds-checking run-time system:

```
OutOfRangeAutomatics.c:16:Bounds warning: unchecked stack object used at address 0xbffff6ef.*/
```

Arrays within structures are not checked

```
struct {
        int obj1 [10];
        int obj2 [10];
} s;
```

```
main() {
	int i;
	for (i = 0; i < 20; ++i)
		s.obj1[i] = i;	/* no bounds error; reference is within allocation object */
}
```

This example illustrates a limitation on bounds checking as we have defined it. The variable s consists of a single storage object, and the bounds checking does not verify that its use is consistent with the type declaration. To do so would considerably add to the system's complexity, but, more importantly, would lead to false reports in situations where casts are used quite legitimately.

Arrays within arrays are not checked

```
/* Abuse of subarrays of a multidimensional array cannot be checked.
*/
int i;
double a[10][10];

main() {
	for (i = 0; i < 20; ++i)
		a[0][i] = i;	/* No bounds error; reference is within allocation object */
}
```

As in the previous example, the array a consists of a single object, and bounds errors are reported only when a reference outside the whole array is derived.

Pointer to unchecked object passed to checked code

```
/* In file 'unchecked.c' ...
*/
int *unchecked_fn (void)
{
	static int a[10];
	return a;
}

/* In file 'checked.c' ...
*/
extern int *unchecked_fn (void);

int main ()
{
	int *a = unchecked_fn (), i;

	for (i = 0; i < 20; ++i)
		a[i] = i;	/* No bounds error. */
}
```

When the variable `a` is used, it is found to have no corresponding object table entry. Although a warning can be issued here, it is not necessarily an error since the pointer may have been imported from an unchecked module (Note that this problem can be overcome by adding the object to the object tree by hand, using `__bounds_note_constructed_object (...);`).

Correct inter-operation with non-trivial system calls

```
/* Example to show interworking with system calls etc (under SunOS 4.1).
*
* Allocate a 3-page region, set VM protection to disallow access, install a handler to
* catch the resulting faults, re-enable access and continue.  Loop runs over end of region.
*/
#include <stdio.h>
#include <signal.h>
#include <sys/mman.h>
char *region;
int pagesize;

void SEGVHandler(sig, code, scp, addr)
        int sig, code; struct sigcontext *scp; char *addr;
{
    /* Reinstate the page in question */
    char *pagebase = (char *)((int)addr / pagesize * pagesize);
    mprotect(pagebase, pagesize, PROT_READ | PROT_WRITE);
    /* Now we should return and restart the faulting instruction */
}
void main() {
    char *p;
    signal(SIGSEGV, SEGVHandler);
    pagesize = getpagesize();
    region = valloc(pagesize*3);
    mprotect(region, pagesize*3, PROT_NONE);
    for (p = region; p<=&region[pagesize*3]; p+=pagesize)
        *p = 'p';
}
```

Output from the bounds-checking run-time system:

```
Signals.c:27:Bounds error: attempt to reference memory overrunning the end of an object.
Signals.c:27:  Pointer value: 0x23000
Signals.c:27:  Object `(unnamed)':
Signals.c:27:    Address in memory:    0x20000 .. 0x22fff
Signals.c:27:    Size:                 12288 bytes
Signals.c:27:    Element size:         1 bytes
Signals.c:27:    Number of elements:   12288
Signals.c:27:    Storage class:        heap
```

This example is intended to demonstrate that bounds checking can be used even in quite sophisticated contexts with subtle inter-operation with the operating system. Although the actions of the system calls themselves are not checked (of course they could be), the fault address `addr` passed to the signal handler is checkable with no special arrangement.