Bounds checking for C and C++

Andrew Suffield

Project supervisor: Paul Kelly <phjk@doc.ic.ac.uk>
Second marker: Olav Beckmann <ob3@doc.ic.ac.uk>

This project added code to the GNU Compiler Collection to provide run-time checking pointer and array accesses for various bounds errors in compiled code. The primary objectives were to handle C++, to avoid changing the ABI (so that checked and unchecked code can be freely mixed), and to avoid throwing errors on correct code.

It is a continuation of the work done by Richard Jones in 1995, extending it by supporting C++, eliminating the extra padding it added to objects, and correctly handling loops which increment in steps other than one unit.

It has been tested and evaluated on a range of contemporary free software, and the performance impact measured. Unlike previous efforts along similar lines, it operates for both C and C++, and can reliably trap most kinds of bounds errors involving heap, stack, and global objects.

A number of possible extensions to improve accuracy and performance were considered but not explored experimentally; these are discussed at the end of this report.

Table of Contents

1. Introduction

One of the primary features of the C and C++ languages is that they permit the use of pointers without restrictions - a pointer references a location in memory, and effectively arbitrary operations can be performed on it. This is also one of the primary causes of errors in such programs. These errors are particularly unpleasant because their effects often appear to be random - a function can unintentionally modify unrelated data, thus causing obscure problems elsewhere in the program. These errors are generally very difficult to locate and correct. Historically, they have been responsible for a significant number of problems in common utilities[Miller, Lars & So, 1990], and remotely exploitable security holes ([Seeley, 1989] and many similar worms).

There are numerous approaches to solving this problem, most of which have differing advantages and limitations; these will be discussed in detail later. This project was intended to investigate and implement one particular approach, where the target program is recompiled with run-time checking embedded in every instance of pointer arithmetic, to see if it goes out of bounds. It takes the form of an extension to the GNU Compiler Collection (GCC)[1], modifying the generated object code to check for certain kinds of bounds errors. It builds upon the work done by Richard Jones, in his 1995 project entitled *A bounds checking C compiler*.

The primary contributions of this project are:

- A functional implementation of dynamic, compile-time bounds checking for C++. This proved to be quite difficult due to the flexibility of scoping rules in C++.
- An effective method for handling pointers which are currently invalid, without changing the language ABI.
- An evaluation of the performance of this approach.
- An evaluation of the effectiveness of this approach, with particular attention to what things it can and cannot detect.
- A comprehensive analysis of how type-sensitive bounds checking could be used to detect more errors. This would allow the detection of bounds errors internal to an allocation object.

The primary objective of applying the bounds-checking system created in [Jones, 1995] to C++, was accomplished successfully. This was done by reimplementing similar logic in the processing of the language-independant internal representation, while [Jones, 1995] modified the C parser.

However, in the process it became apparent that the added complexity of C++ left scope for a wide range of other possible checks. In addition, several new ideas were raised that could be applied equally to both C and C++. These are discussed in the "Possible extensions" section (6.2). Here is a summary of some of the more interesting ideas:

- By drawing the boundaries for pointers at the edges of allocation objects, some potential checks are lost. In C, this is strictly correct, since it is explicitly permitted by the language specification to assume that structs are laid out in a certain manner, although the programmer may still prefer stricter rules. In C++, no such guarantees are given; it is an error for a pointer to one member variable to be incremented past the end of that member. It should be possible to check for such errors using the same bounds-checking logic.

  However, investigation determined that the necessary support would prove complicated. Due to the limited time available, it was decided to forego implementing this functionality.

- Some of the features provided by other systems for checking program correctness could be added to this one. They were skipped since they are not directly relevant to the bounds-checking problem, but could probably be implemented without too much effort. Examples include checking for use of uninitialized memory, leaking memory, and using pointers to freed memory.
- It would be fairly easy to add run-time type checking, to prevent type-punning. However, it is already difficult to accidentally access an object with the wrong type in C++, and C++ already has a run-time type system, so this is probably not a widely useful extension. It is likely to be significantly more useful for C.
- Numerous optimizations to improve the performance of code compiled with bounds-checking enabled were considered, but were again too time-consuming to implement. A range of performance problems have been identified with the current implementation.

This project focuses on ISO C99[iso-c99] and ISO C++[iso-c++], rather than any earlier revisions of the languages.

2. Background

2.1. Bounds errors

Bounds errors are characterized by an attempt to access elements of an array which don't exist.

Example 1. Trivial bounds error

```
int main(void)
{
  int i[2] = {0, 1};
  return i[2];
}
```

Here an array of two elements is defined, and the third is returned. The behavior of this program is undefined; it may crash, or it may return some unknown value.

A somewhat more subtle problem arises in the case of C strings. These are stored as a sequence of characters terminated by a NULL. If the terminating NULL gets overwritten somehow, then any code which expects it to be present will behave unpredictably; it could run over a huge quantity of data before finding a NULL byte and stopping, or it could crash outright.

Example 2. String bounds error

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char s[] = "Hello world";
  s[strlen(s)] = '!'; Try to append a ! to the string
  printf("%s\n", s);
  return 0;
}
```

In line 7, the trailing NULL character is overwritten with a !, and no new character is added. Line 8 attempts to send this string to stdout - it will probably print the string followed by some garbage, but it might crash.

This problem is distressingly common due to a historical flaw in the strncpy() function in the standard C library. strncpy(a, b, n) copies no more than n bytes of string b to a. This means that if strlen(b) > n, the resulting string in a will not be NULL terminated. The programmer must take special care to terminate it manually when using this function.

One particularly insidious kind of bounds error is the kind which never crashes at all, directly. It occurs when objects exist which are adjacent in memory, and a pointer to one of them is incremented past the end so that it now points to the other object.

Example 3. Bounds error with adjacent object

```
#include <string.h>
struct x
{
  char s[10];
  int a[4];
};

void bang(struct x *d)
{
  strcat(d->s, "!");
}

int main(void)
{
  struct x d;
  strcpy(d.s, "012345678");
  d.a[0] = 3;
  d.a[1] = 2;
  d.a[2] = 1;
  d.a[3] = 0;
  bang(&d);
  return a[0];
}
```

In this example, struct x contains a 10-byte string immediately followed by a 4-integer array. d is initialized with a 9-character string (occupying 10 bytes because of the trailing NULL) and four integers. bang() appends a ! to the string, making it "012345678!" plus a trailing NULL. The NULL byte at the end of the string will overwrite the first byte of d.a[0]. On a big-endian host, this will have no effect because that byte was already zero. On a little-endian host, this will change the value of d.a[0] to zero.

All of the above examples give code which compiles with no warnings, but which will behave "incorrectly" when run. Worse than this, in some scenarios these kinds of errors can form the basis of buffer overrun attacks, where a malicious user deliberately feeds in more data than was expected, overwriting other regions of memory. By carefully controlling what data is overridden, the attacker can run arbitrary code.

2.2. Approaches

There are several ways to approach the problem of checking for bounds errors. Some of the more interesting ones are described here.

2.2.1. Static checking

The most obvious approach is to modify C in a fashion that makes such errors easier to detect. There have been various attempts to create such modified languages, including Robust C[Flater, Yesha & Park, 1993], CCured, and Cyclone. Unfortunately this requires rewriting existing programs in order to make use of the new features - sacrificing portability and the wide range of powerful tools that can operate on regular C programs.

Given the constraint that existing C programs need to be usefully checked, there have been a lot of attempts to create partial static checkers - which look only at the source code, and do not attempt to run the program. The various forms of the classic ’lint’ tool (now mostly obsoleted by splint) fall into this category, along with [Wagner, Foster, Brewer & Aiken, 2000]. All of these suffer one of two problems - they are either too conservative, and issue warnings for code that is correct, or they are too liberal, and fail to detect some genuine errors.

2.2.2. Fat pointers

Fat pointer systems work by replacing every pointer value with a structure that describes the valid range for the pointer, and replacing every pointer read/write with code that checks or updates this structure as necessary. The canonical form of this stores a ’base’ and ’extent’ value with each pointer, describing the lower and upper bounds that it is permitted to point into; if it is outside this range when it is dereferenced, an error is reported. There have been numerous variations on this theme, attaching different meta-data to the pointers, such as [Austin, Breach & Sohi, 1994].

Fat pointer systems incur a notable memory usage penalty and a measurable performance penalty. Somewhat more inconveniently, they change the language ABI - it is not possible to pass pointers freely from checked code to unchecked code, since they have different representations in memory. This means that either the compiler has to be informed about every external call so that it can generate a translation thunk, or wrappers must be written by hand, or all code that is intended to be put into the target application must be rebuild with bounds checking added.

This requirement can make it quite inconvenient to add bounds checking to a given application for debugging purposes - you can’t simply rebuild that application with bounds checking.

2.2.3. Guards

[Patil & Fischer, 1996] discusses a system of pointer "guards" for C. Guards are a variation on the theme of fat pointers. Rather than keep the extended data in the pointer itself, a new object is generated which can be located given the location of the pointer it is associated with. This object stores the meta-data that was stored in the pointer.

The difference here is that rather than inserting code into the program itself to check the bounds, a separate process or thread monitors the program. When running a single-processor application on a multi-processor host, this eliminates some of the performance overhead associated with fat pointers.

2.2.4. Fence posts

Fence posts are an approach to bounds checking based on placing regions at the end of every allocated objects, and trapping any access to those regions. This system is commonly implemented in hardware; as such it is one of the fastest mechanisms, but due to limitations of most modern memory models, can be quite inefficient in memory usage.

The basic principle is to place a region of memory around each pointer which will generate a hardware exception if any read or write attempt is made. This is often implemented on UNIX platforms by positioning objects next to page boundaries, and modifying the memory protection rules for that page to prohibit reading or writing. A significant side-effect of this sort of implementation is that every object will consume an exact number of pages. Since pages are commonly several thousand bytes long, this is expensive if a lot of small objects are used.

Fence posts are of limited use in C, since structural types are required to be a single contiguous unit of memory, so that functions like memcpy() can operate on them. C++ classes do not have this limitation, since memcpy() can’t operate on them anyway - classes have their own methods and conventions for copying. However, using fence posts to delimit members of a structure would be prohibitively expensive in its use of memory.

Hardware fence posts also have one serious limitation - they can usually only check one of two bounds at once. Unless an object happens to fit a page exactly, overruns at one end will not be detected unless they go a significant distance past the end. There is also a less serious limitation that placing a fencepost at the upper bound of an object may break some alignment restrictions it has. Most platforms have no trouble emulating unaligned access, but it can impose a significant performance penalty.

Software-based fence posts are also possible. These do not have the memory or page alignment restrictions of a hardware fencepost implementation - but the speed advantage is also lost.

2.2.5. Tracking objects

This approach was first introduced in [Jones & Kelly, 1997], and is based on an invariant that states "Every pointer is valid", and keeps track of where the objects are located instead. Code is added to each function ensuring that, if the invariant is true on entry to the function, it will still be true on exit. If every function in a program is instrumented in this manner, no uncaught bounds errors are possible. If only some of them are checked, then no uncaught bounds errors are possible in that part of the code - which is a convenient result.

This is accomplished by noting when every object is constructed (regardless of whether it is global, lexical, or dynamic) and building a data structure that can identify the object located in a given memory address. When pointer arithmetic occurs, the original object is looked up. If the pointer goes out of bounds, it is marked as invalid. This can impose a significant performance penalty if not optimized sufficiently.

It is not inherently an error to construct an illegal pointer, as long as this pointer is never dereferenced. One common instance of this happens when iterating over an array - the pointer can end pointing one item past the end. It is therefore necessary to delay flagging errors until the pointer is dereferenced.

2.3. Existing implementations

There are numerous systems in existence which have applications to the bounds-checking problem. It should be readily apparent that they all have different feature sets, and none of them provide a "complete" solution; various things that are desirable are contradictory (static vs. dynamic, compile-time vs. run-time, and similar). These are discussed for comparison purposes; this project was aimed at creating a specific feature set that nothing else does.

2.3.1. CCured

CCured is a close variant of C with modified semantics. It is implemented as a translator (like the original cfront) that takes CCured code as input and emits C code as output, based upon a generic C transformation system called CIL.

CCured is freely available from its web site[2].

CCured is focused on pointer semantics. It introduces four attributes which can be applied to pointer types to describe the intended use of the pointer. These vary the trade-offs between flexibility and performance. It is based around fat pointers.

- SAFE pointers cannot be cast to different types. Arithmetic is not permitted on a SAFE pointer, and a non-zero integer value cannot be assigned to it.
- SEQ pointers are like SAFE pointers, but simple arithmetic may be performed on them as long as they are still within the bounds of the object that they were last *assigned* to point into. They can store integer values, and it is an error to dereference such a stored integer.
- FSEQ pointers are like SEQ pointers, but subtraction is prohibited. This means they can be stored in 2/3 of the memory used by SEQ pointers (only an upper bound is needed).
- WILD pointers are like SEQ pointers, but may be typecast to other WILD pointers regardless of type. Certain aspects of their type (relating to what is and is not a pointer) will be dynamically checked when they are dereferenced to permit this, unlike the other pointer variants which are statically checked. They have considerable memory and performance overheads.

CCured appears to provide safe pointer semantics, but existing C programs will generally require some modification in order to use it effectively. It is possible to have valid C programs which are also valid CCured programs, but all pointers will be WILD and thusly incur size and performance overheads. Converting programs which make extensive use of typecasts to use SAFE and SEQ pointers where appropriate can be a complicated task. Calling C functions from CCured requires writing wrapper functions to convert any pointers.

There is no support for anything resembling C++.

2.3.2. Cyclone

Cyclone is a C-like language which cannot express a range of invalid code which can be problematic in C. The stated goal of Cyclone is: “ To give programmers the same low-level control and performance of C without sacrificing safety, and to make it easy to port or interface with legacy C code. ”

Cyclone is freely available from its web site[3].

The full semantics of Cyclone are beyond the scope of this document, but summarising the features related to bounds checking:

> **Note:** This information was accurate as of cyclone version 0.3, but the language is still somewhat developmental, and may change in some details.

- Cyclone provides C-style pointers, but extends their type with data about the scope of the thing they point to. It is impossible to assign a value to such a pointer which comes from a scope which may not be valid at the time the pointer is dereferenced; attempting to do so is a syntax error. In addition, pointer arithmatic is forbidden on these pointers.

- There is also a fat pointer system with similar semantics, but using completely independant types. It permits pointer arithmatic and performs run-time bounds checking on it as necessary.
- There is a ’never-null’ pointer system, again with a discrete type, which is like the C-style pointer but is guaranteed never to have a null value.
- A C-style pointer can be qualified as being zero-terminated, like a C string, in which case this termination will be enforced.
- While objects with lexical or global scope are constructed and destroyed like C objects, those on the heap cannot be explicitly freed; a garbage collector is used to release their memory when it is no longer needed. This prevents dangling pointers.

Cyclone appears to be a safe system, but only resembles C; most valid C programs are not also valid Cyclone programs. Porting an application from C to Cyclone is possible, but this would lose the portability and wide range of tools available which manipulate C programs. It is an interesting system, but one with very different goals to this project.

There is no support for anything resembling C++.

2.3.3. TinyCC

TinyCC is a C compiler that eschews portability and run-time performance in favour of compiler speed and size. It is intended for scenarios such as rescue disks and scripting in C. It implements bounds checking based on tracking objects. The absence of support for multiple languages or target platforms makes both the compiler, and the implementation of bounds checking, very simple. TinyCC performs only a few trivial optimisations on the generated code, which can have a severe impact on performance.

There is no support for C++.

2.3.4. Valgrind

Valgrind is a runtime environment that JIT-compiles i386 object code to an internal representation, and then executes it on a simulated processor. By monitoring the simulation, it can detect some bounds errors and memory leaks, and provide some performance profiling of the code.

Valgrind is freely available from its web site[4].

Valgrind operates on binaries, not source, so nothing has to be recompiled. Due to the way it works, large parts of valgrind are inherently non-portable, so it is unlikely that it will ever work on any platform other than i386. It is also inherently *very* slow.

Valgrind cannot detect the case where a pointer was incremented past the end of an object so that it points into a different valid object, and it is then used to modify the contents of this object.

Valgrind works as effectively on C++ as it does on C.

2.3.5. BCC

At one point there was a C compiler called bcc[Kendall, 1983], which was based on fat pointers. It was one of the first such compilers ever created, and it now appears to have been abandoned, since I could find no trace of it.

2.3.6. Purify

[Purify, 1995] is a commercial software fencepost system that operates on binaries. No source is required, although debugging symbols greatly improve the usefullness of its output. Since it operates on binaries, it is largely language-agnostic, and thusly works for both C and C++.

Purify works by modifying the object code to replace every memory access with a function call, and to intercept every malloc() and free(). As such it is highly platform-dependant (currently only i386 and sparc are supported), and can operate only on objects stored on the heap. It also checks for some other errors related to memory access, such as reading of uninitialised data.

2.3.7. Electric fence

Electric fence is a simple, free, malloc-based hardware fencepost system. It exhibits the characteristic low overheads and high memory consumption of all such systems, and has no support for C++. Again, it operates only on objects allocated using malloc() - those which are stored on the heap - and not stack objects.

2.3.8. Insure++

Insure++ is a commercial C/C++ compiler and runtime debugger that has limited bounds checking abilities, along with various other analysis and testing tools. No details are available from the company that produces it on how it operates; it is included here for completeness only.

It is likely that US patents #5581696 (all forms of bounds-checking, and identifying access to uninitialised or unallocated memory by means of modifying the code at compile time), #6085029 (various trivial extensions on the former patent to improve diagnostics) and #5842019 (all forms of reference counting and mark-and-sweep scanning for memory leaks) form a significant part of the product, but this is speculation based on marketing materials. As such it may have some overlap with this project.

3. Design

Due to time constraints, it was decided that the project would be concerned only with allocation objects - that is, those objects which are created as a unit by an allocator such as malloc() or new. This means that every struct and class is considered a single object, including any nested classes. Furthermore, types are effectively ignored by the bounds-checking code - it considers all objects to be simple strings of bytes.

3.1. Outline

[Jones & Kelly, 1997] operated by modifying GCC's first phase, in the language-specific front end, to insert code wrapping all interesting pointer operations. This has the significant disadvantage of only working for the C front end. It would have been possible to do something similar for the C++ front end, but then there would have been two implementations of highly similar code. To avoid such unnecessary duplication, all the instrumentation was moved into the RTL generation phase.

In order to handle the tracking of objects, the support library from [Jones & Kelly, 1997] was taken and extended to handle C++, rather than reimplementing the common functionality. It was also modified to eliminate another problem - in order to reduce false positives, the original implementation padded objects with an extra unused byte. Unfortunately this changes the language ABI, which would cause significant problems passing objects between checked and unchecked code in C++, so a different approach was taken. See the section on emulated pointers for details.

3.2. GCC internals

The design of the bounds-checking code has been heavily influenced by the design and limitations of GCC. This section gives a brief overview of how GCC compiles programs. For more detail, see the *GCC internals manual*. Significant parts have been omitted (such as type handling, debugging information, and most of the optimization), since they are not directly relevant to this project.

GCC is a compiler suite that supports multiple input languages, and can generate object code for multiple target environments. In order to support this, it uses two intermediate representations for programs. The first of these representations is called *tree*. As its name suggests, it is tree-structured, with one tree for each function and global object. These trees are high-level representations which closely match the original source code. The nomenclature used in GCC refers to nodes as either "expr" (expressions, which return a value), "decl" (declarations, which could be variables, callable functions or similar) and "stmt" (statements, which have no value). A function which is defined in the current file will have both a decl and a stmt.

Example 4. Tree representation

```
if (a > 2)
  a = a + 1;
else
  b();
```

This C statement would be translated into an IF_STMT node, with three children. The IF_COND child would be a GT_EXPR node, with the VAR_DECL for ’a’ as its first operand and an INTEGER_CST for 2 as its second. The THEN_CLAUSE child would be a MODIFY_EXPR node with the VAR_DECL for ’a’ and a PLUS_EXPR as its operands, and the ELSE_CLAUSE would be a CALL_EXPR with the FUNCTION_DECL for ’b’ as its operand.

GCC operates on one function at a time - it runs all the passes on the first function, then moves on to the next one. The first phase of compilation is to parse the source code, converting it into tree structures directly. This is the only language-specific phase [5]. After this, tree-based optimizations are performed, notably constant folding and function inlining, along with some simplification and homogenization of arithmetic expressions.

The next phase is to generate RTL (Register Transfer Language) from the tree structures. RTL is the second intermediate representation used in GCC. It is structured like a simple LISP variant, and is a low-level representation of the object code that will be generated. Unlike the tree representation, RTL is target-specific. Target features such as register size greatly influence the generated RTL. Decisions about how best to convert language structures into machine-specific instructions are made here. This is the last phase where the tree representation is used. No later phase will significantly alter the algorithms generated, although they may still be reordered or unrolled.

RTL generation is driven by large switch statements in the recursive functions expand_stmt() and expand_expr(), which walk the tree in execution-order, generating RTL as they do so. This means that when examining a node, only limited information is available about its parents, and no information is available about any child nodes which have not yet been expanded. *It is not possible to process a child recursively without committing to generating its RTL,* although the RTL can be later modified or moved. This turned out to be quite inconvenient, as will be discussed later.

After this, the bulk of the optimizations are performed, updating the RTL structures in-place, and then the RTL is converted into assembly by the target-specific machine description. None of this is directly relevant to bounds-checking, so it is not discussed here.

3.3. Details

3.3.1. Program instrumentation

The desired behavior is that any pointer access (read or write) that it outside the bounds of the object it is supposed to point to, will be trapped and an error reported. This means it is necessary to intercept every pointer access, and everything that could generate a pointer which goes out of bounds. This is relatively easy, since every instance of a particular kind of expression will pass through the same place in the expand_expr() function when generating RTL.

There are precisely five operations which can generate pointers in C/C++.

- The address-of operator, ’&’. This cannot generate an invalid pointer; it is a syntax error to apply it to any object without an lvalue[6], and the address of an object with an lvalue is valid by definition.
- The addition operator, ’+’, with one pointer operand and one integer operand. It can generate bounds errors.
- The subtraction operator, ’-’, with the a pointer as the left operand and an integer as the right operand. GCC internally translates this into an addition operator, and applies a unary negation operator to the right-hand operand.
- The array index operator, ’[]’. This is internally translated into an addition operation.
- Type casts to pointer types. If these are made from pointer types, then the resulting pointer will still be valid, although accessing it may fail. If these are made from non-pointer types, then they can generate invalid pointers, but are beyond the scope of this project.

Since the subtraction and index operators are effectively variations on addition, they will henceforth be ignored; anything that applies to addition will apply equally to them, and be handled by the same part of GCC.

In addition, there is one other operations which can generate bounds errors without involving pointers.

- The field selection operators, ’.’ and ’->’. These are more of a problem in C, where dynamic allocation is made in a typeless manner with malloc(), than in C++, where dynamic allocation is performed with new, but it can still happen (for example, if two different compilation units were built with different definitions of a type, and an object of that type is passed from one to the other at run time, or if an object has been type-punned).

Addition and field selection operations need to be instrumented with calls to the bounds-checking library. Type casts can be safely ignored, since they never change the value of a pointer. Pointer read and write operations also need to be instrumented, so they can be checked for validity.

Pointer accesses themselves all take the form of the dereference operator, ’*’; the array index operator [] is interpreted as an add followed by a dereference, and the indirect field selection operator ’->’ is interpreted as a dereference followed by a direct field selection.

Here are the two basic transformations needed:

Table 1. Arithmetic expressions

Source code	`extern void *p;`
	`p + 1`
	`p - 1`
Generated code	`__bounds_check_ptr_plus_int(p, 1);`
	`__bounds_check_ptr_plus_int(p, -1);`

Table 2. Field selection

Source code	`struct s`
	`{`
	`  int x;`
	`};`
	`extern struct s *a, b;`
	`a->x`
	`&b.x`
Generated code	`*__bounds_check_reference(`
	`  __bounds_check_ptr_plus_int(a,`
	`                            offsetof(struct s, x)),`
	`  sizeof(a->x)`
	`)`
	`__bounds_check_ptr_plus_int(b, offsetof(struct s, x))`

3.3.2. Object tracking

There are several things that create and destroy objects in C/C++.

- Automatic objects are created on the stack at or near the point in the source code where they are defined. They are destroyed when they go out of scope - this will be at the end of the block they were declared in.

- Static objects are created during global construction, when the binary is first loaded. They are destroyed during global destruction.
- Dynamic objects are created with a call to malloc(), or the operator

  `new`

  . They are destroyed with a call to free(), or the operator

  `delete`

  .

All of these occasions need to be instrumented, so that the bounds-checking library knows what objects currently exist, where they are located, and how big they are.

Dynamic objects are by far the easiest to track. The system library functions malloc() and free() are replaced with versions that call the bounds-checking library. The new and delete operators are similarly modified, inside the compiler.

Static objects require a little more subtlety. Since they are often never constructed at all (for example, if they have no initialization value, or if the native binary format can fill in their values without running any program code), they can't be directly intercepted. It would be possible to add a constructor to all of these objects, but that would be unnecessarily slow and complicated. Instead, the compiler builds a table of all the static objects, and creates a global constructor function that passes this table to the bounds-checking library.

Automatic objects are the most complicated to track. The point at which they are created can be moved by optimization passes. Some experimentation indicated that simply tracking entry and exit of block scopes, and noting which scope each variable occurred in, was unreliable. One of the key problems is with code of this form:

```
void foo(void)
{
  int a;
  a = 5;
  int b;
  b = 1 + a;
  int c;
  c = b + 2;
}
```

Ignoring the effects of other optimizations (like dead code elimination and register allocation), the flow analysis pass will identify that a and c can be safely stored in the same location. It is therefore incorrect to assume that a is destroyed at the end of the scope it is declared in - it is effectively destroyed at the point where c is constructed.

Eventually, a compromise that handles most cases was chosen. A fake constructor and destructor is created for every object, that calls the bounds-checking library. GCC then emits the calls at appropriate places - almost (see below). This has the effect of adding a constructor to every object, which causes certain previously legal code to be rejected.

Example 5. Code that is broken by added constructors

```
{
  int a;
  goto foo;
  {
    char b[5] = {0,0,0,0,0};
  foo:
    a = b[0];
  }
}
```

This code is normally compiled correctly by GCC. When the bounds checking code is activated, it gives the following obscure error:

```
error: label ‘foo’ used before containing binding contour
```

This error will refer to the line where label ’foo’ is defined, but means "This label was used before the binding contour that contains it" - referring to the ’goto foo’ preceding it. It is an error to jump into a binding contour after some non-trivial constructors have run. The problem is caused by the invisible constructor that has been attached to the array ’b’. If this error were not given, then the constructor for ’b’ would not be run, and so the assignment

```
a = b[0];
```

would, to the bounds-checking library, to be a reference to an unchecked array.

This isn’t quite complete, because new objects can still be constructed before the old one stored at the same location is destroyed. To handle the last few cases, the bounds-checking library was modified so that if a new stack object is allocated at an address already occupied, the object already in that location is assumed to be released (previously, an internal error was reported).

3.3.3. Emulated pointers

One question has been deliberately avoided until now. What is the appropriate action when addition or field selection operations go out of bounds? It would be possible to immediately report an error, but such operations are not intrinsically wrong.

Example 6. Valid use of invalid pointers

```
int mogrify(char *d, char *end)
  {
    char *a = d, *b = d;
    while (a < end)
      {
        switch (*a)
          {
          case ’\\’:
            switch (*++a)
              {
              case ’n’:
                *b++ = ’\n’;
                break;
              case ’t’:
                *b++ = ’\t’;
                break;
              default:
                *b++ = *a;
              }
            break;
          default:
            *b++ = *a;
          }
        a++;
      }
    return a != b;
  }
```

This function scans the characters between d and end, replaces any occurrence of "\n" with a newline and "\t" with a tab, and returns true if any changes were made and false otherwise. a will always point one item past the end, and b might. Note that it works in-place, rather than creating a copy. It is a simplified variation of a function used in the Dancer IRC server, modified to show the problem more clearly.

[Jones & Kelly, 1997] dealt with invalid pointers by setting them to a fixed value, so that any operations which went out of bounds would return the same result, and then trapping every occasion where such pointers were used and flagging them as errors. This would cause the example above to fail on the line ’return a != b;’, so objects were padded with an extra byte to handle the common cases. Unfortunately, this approach has a number of problems. The most significant one in this example is that it makes no attempt to deal with the case where pointers are incremented more than one byte past the end of an object; if the input string contained two escape characters instead of one, that would have been flagged as an error. In addition, it can only work on allocation objects; it precludes the possibility of finer-grained checks, because the internal structure of objects cannot be modified to insert such padding.

Instead, this project emulates pointers which go out of bounds, giving the appearance that every object was allocated in its own virtual address space. This is accomplished as follows:

1. Whenever a pointer arithmetic operation would generate a pointer that is outside the bounds of the object it started in, the offset from the end of the object is computed (for pointers that have been decremented before the start, the offset is computed from the start instead, and is negative).
2. A one-byte region of memory is allocated on the heap. This is then inserted into the object tree with some extra meta-data indicating that it is an emulated region, the object it was derived from, and the offset from the end of that object.
3. The address of this region is returned as the result of the arithmetic operation. This address is the emulated pointer.

Whenever pointer arithmetic occurs, the pointer is first looked up in the object tree to see if it is part of an emulated region. If it is, then the "real" pointer value is put in its place, computed from the meta data stored in the object tree, and the arithmetic operation is performed on that instead. If this operation results in a value which is still out of bounds with respect to the original object, then a new emulated pointer is generated. If it is no longer out of bounds, then a pointer to the appropriate location in the original object is returned. This means that pointers can freely go out of bounds, and then back into bounds, and still function as expected.

One side-effect of this process is that pointer comparisons do not behave as expected on emulated pointers. In order to correct this, all comparisons between two pointer values are instrumented in the compiled code, so they are made via calls to the bounds-checking library. The library then performs the same lookup for comparisons that it does for arithmetic, replacing emulated pointers with their "real" values, before making the comparison. Likewise, pointer difference operations have to be adjusted. Since these operations are being instrumented anyway to support emulated pointers, we also check that both pointers refer to the same object - if they do not, the behavior is undefined, so the bounds-checking library flags it as an error.

Unfortunately there are problems when interfacing with unchecked code. If an emulated pointer is passed to unchecked code, then odd things may happen. Comparing such a pointer with another pointer in unchecked code will probably lead to unexpected results, and arithmetic on it would certainly do so. In order to mitigate the effects of performing arithmetic on emulated pointers in unchecked code, the size of the emulated region allocated is increased, so emulated pointers are padded on both sides. The logic to find the "real" pointer is then modified, so that if the emulated pointer is not at the base address for its region, then the offset from that base address is also taken into account.

4. Implementation

4.1. Tracking object creation

In order to check pointer bounds, it is obviously necessary to know where the valid objects are located. In order to do this, code was added to GCC that emits calls to the bounds-checking library when new objects are created and destroyed.

4.1.1. Automatic variables

Automatic variables turned out to be relatively easy to handle. They are always represented by the tree expression ’VAR_DECL’, and the code for them is always generated in the expand_decl function, so a call is emitted to __bounds_add_stack_object immediately after instantiating them.

4.1.2. Objects passed as parameters

These are handled separately from automatic variables, since they have special initialization rules to efficiently fill in their values from the caller. However, in other respects they are like automatic variables. Their tree expression is ’PARM_DECL’, and they all pass through assign_parms, so a call is emitted to __bounds_add_param_object there.

4.1.3. Static objects

Static objects are usually generated in a different relocation section to the program code, so they can’t be noted immediately after being created. Instead, a note is made whenever creating a static object inside a function, and when work has finished on the current compilation unit (one source file in C/C++), a new function is generated that makes calls to __bounds_note_constructed_private_table and __bounds_note_constructed_object as necessary based on the list of global declarations, and the list of static objects that was noted earlier.

This code is largely the same as it was in [Jones & Kelly, 1997]; the only significant difference is that constant strings are now noted as well.

4.1.4. Heap (dynamic) objects

Heap objects are created at runtime, so the only way to handle them is to instrument calls to the functions which create them. The malloc() and free() functions are replaced by overriding the weak symbol definition in the system C library with a strong one. In addition, the ’new’ and ’delete’ operators are instrumented in the compiler so that they make similar calls to the bounds-checking library. The system allocation functions are still used, but encapsulated with functions that note the size and address of objects as they are created and destroyed.

4.2. Instrumenting expressions

In order to fully understand how expressions are handled, it is necessary to understand how GCC represents them in RTL (register transfer language). RTL is a language with a similar look and feel to lisp. It is used to represent the semantics of program code in a manner that makes certain translations/optimizations relatively easy. More complete documentation can be found in the and the rtl.def file in the GCC source tree.

An RTX (register transfer expression) is the basic unit used in RTL.

```
(reg:SI 67)
```

is an RTX which represents register number 67, accessed in SI (single integer - an integer type which is 4 bytes long) mode.

```
(mem:SI (reg:SI 67) [0 S1 A8])
```

represents the memory location referenced by the value of (reg:SI 67), also accessed in SI mode. This would be used, for example, if register 67 contained a pointer.

```
(mem:SI (plus:SI (reg:SI 67) (const_int 4 [0x4])))
```

represents the memory location which occurs 4 units after the value of (reg:SI 67); if the pointer x is stored in register 67, this RTX is equivalent to the C expression

```
x[4]
```

.

```
(mem:SI (symbol_ref:SI ("x")))
```

represents the memory location referenced by the symbol "x".

When a new variable is declared, an RTX is generated to contain it. This would generally be either a reg expression (for variables which are put directly into registers), a mem expression containing a symbol_ref (for static or exported variables), or a mem expression containing a plus expression that finds an offset from the frame pointer (for automatic variables). Code was added to the functions which generate RTXs for variables, so that their type is checked and any RTX which represents a pointer value is tagged so that we can later distinguish between pointer and non-pointer values.

When a tree expression of code PLUS_EXPR or MINUS_EXPR is converted into RTL, it would normally be turned into a (plus ...) or (minus ...) expression. Trees are converted into

RTL in a recursive manner, so the first step in processing a tree expression is to convert its arguments into RTL. This descends down the tree until it reaches a leaf - either a literal ('1'), a variable, or a function call.

A flag has been added to every RTX that indicates whether or not it represents a bounded pointer value. Every memory address, and every variable that has pointer type, is flagged as being a bounded pointer when the RTX for it is generated. In addition, a PLUS_EXPR that has one pointer and one integer operand is also flagged as being a bounded pointer. In this fashion, the flag propagates up the tree. When such a PLUS_EXPR is processed, instead of emitting a (plus ...) RTX, a call to __bounds_check_ptr_plus_int is emitted instead. MINUS_EXPR is internally handled as a PLUS_EXPR with a unary minus applied to the right-hand operand, and therefore needs no special treatment.

A range of other special cases are covered in a similar manner - array references, and accesses to members of compound types can pass through different branches of code under some circumstances, so these are also caught and replaced with calls to __bounds_check_ptr_plus_int as appropriate.

Every dereference and field access operation is instrumented with first a call to __bounds_check_ptr_plus_int (if necessary), and then a call to __bounds_check_reference. Assignment operations are handled in the same manner.

The simple recursive descent of the tree structure that GCC uses to convert trees into RTL complicated matters. Using this method, it is not possible to tell whether a given expression will return a bounded pointer or not until its RTL has already been generated. At this point, it is too late to change the way in which the RTL is generated. One particular case where this became a problem was with static initializers. A static variable can be initialized in two ways - it can have a precomputed value, stored in the .data section of the generated object, or it can be initialized with the result of a constructor function. If this value is to be bounds-checked, then it *must* be initialized via a constructor - it is necessary to make a run-time call to the bounds-checking library. Unfortunately, there are a range of cases where this does not happen, such as when a pointer variable is initialized by taking the address of another static object. Right now, these are simply ignored and not checked; this could conceal a bounds error.

4.3. Bounds-checking library

The behavior covered so far requires the following interfaces to be presented by the bounds-checking library, libcheck:

Table 3. Interface between libcheck and instrumented code

Function	**Purpose**
Basic bounds checking	

Function	**Purpose**
__bounds_check_ptr_plus_int_obj	Perform a checked pointer addition and return the result.
__bounds_check_reference	Check that referencing the given pointer is safe, and abort the program with a suitable error message if it is not. Returns the pointer.
Object tracking	
__bounds_add_heap_object	Informs the bounds-checking library that a new object has been created on the heap, giving its base address and size.
__bounds_delete_heap_object	Informs the bounds-checking library that a heap object is about to be freed.
__bounds_cxx_new	Counterpart to __bounds_add_heap_object, for the C++ new operator. This is a different function in order to simplify the compiler instrumentation; it returns the base pointer, while __bounds_add_heap_object returns nothing.
__bounds_cxx_delete	Counterpart to __bounds_delete_heap_object, for the C++ delete operator.
__bounds_add_stack_object	Counterpart to __bounds_add_heap_object, for stack objects. This takes an extra argument describing the scope of the object (after optimization, these functions may not necessarily be called in the same sequence, so this has to be passed explicitly).
__bounds_delete_stack_object	Counterpart to __bounds_delete_heap_object, for stack objects.
__bounds_add_param_object	Counterpart to __bounds_add_stack_object, for function arguments. It has a slightly different type signature, but otherwise behaves the same.
__bounds_note_constructed_private_table	Called at most once per compilation unit, from a global constructor, to pass in the table of static objects. There is no provision for deallocating static objects.
__bounds_push_function	Informs the bounds-checking library that a new scoping level has been entered. The name is not ideal; a scoping level is not necessarily a function. This returns an identifier for the scoping level to the instrumented code, which will be passed in any calls to __bounds_add_stack_object.

Function	**Purpose**
__bounds_pop_function	Informs the bounds-checking library that the current scoping level has been left, and therefore any stack objects which were allocated within it should be forgotten.
Pointer emulation and enhanced checking	
__bounds_check_ptr_diff_obj	Takes two pointers, and returns the difference between them, compensating if either of them is emulated. If they do not point to the same object, abort with an error. This is not strictly a bounds error, but is useful to check anyway.
__bounds_check_ptr_lt_ptr __bounds_check_ptr_le_ptr __bounds_check_ptr_ge_ptr __bounds_check_ptr_gt_ptr __bounds_check_ptr_ne_ptr __bounds_check_ptr_eq_ptr	These functions handle the pointer comparison tests: <, <=, >=, >, !=, ==. They return a boolean (integer) value, and abort if the two pointers refer to different objects.

Most of these functions also take arguments describing the filename and line of the source code from which they were called. These are used to generate better error messages.

Note: These interfaces are largely unchanged from [Jones & Kelly, 1997]; much of the implementation is the same, but extended where necessary to handle C++ and emulated pointers.

Objects are tracked using a simple unbalanced binary tree, which stores the base address and extent of each object. Originally this was a splay tree, but the implementation provided with [Jones & Kelly, 1997] turned out to have a bug, and time constraints meant that it could not be easily fixed.

Emulated pointers are handled by storing the allocated region on the splay tree like a normal object, along with a field that says which real object they are emulating. All operations on pointers first normalize them to a tuple comprising the object they refer to, and a pointer; this pointer may be outside the bounds of the object. The operation is then performed on the normalized pointer. If the result is a pointer value, it will be compared to the object; if it is out of bounds, it will be replaced with a new emulated pointer instead.

5. Experimental evaluation

5.1. Testing

There are basically two things which need to be true for this project to be considered stable and reliable.

- It should not cause any correct code to fail, or to behave differently in any manner other than performance.
- It should not allow any of the bounds errors it checks for to pass undetected.

It is, obviously, difficult to tell whether these conditions have been satisfied. Compilation errors are often subtle, and can go undetected for years, and subtle changes in behaviour are very hard to detect. Even worse, it is probably impossible to tell whether bounds errors have been missed or not. As such, reliability testing for this project has been done on a somewhat ad hoc basis. The only thing that will ever really tell whether it works adequately or not, is regular usage in a variety of circumstances.

There are currently no known problems of either form. A series of varied tests were performed; in all cases, the code was *at least as reliable as the same version of GCC with bounds checking turned off*. It should be noted that GCC undergoes rapid development, and frequently causes correct code to fail; this is independant of and unrelated to the bounds checking support.

- GCC’s own test suite was run with bounds-checking enabled. There were no significant failures, although a few test cases were reported as "unresolved"; this is due to the tests being highly sensitive to changes in GCC internals, such that they need enhancing to run properly with bounds checking enabled.
- A series of simple, artificial tests were constructed for each individual modification to GCC, as it was implemented; these help to ensure that later changes do not break earlier ones. This is a variation on the theme of the GCC test suite, specifically intended to exercise the bounds-checking code. The tests are highly artificial and bear no resemblence to real code.
- One of my own projects, the as-yet-unreleased version 1.1 of the dancer IRC server, has a test suite that exercises around 85% of the lines of code (according to test coverage analysis). As such, it can be relatively certain that it is functioning correctly. This test suite runs cleanly and passes all tests. Also, during the process of testing, a bounds error was trapped by the system, which was easily corrected. This tree is regularly analysed with Electric Fence, so it should be largely free of heap errors; the newly discovered error was an overflow of a stack object.

- A bounds error was discovered in all bison-generated parsers which use verbose error reporting; this turned out to be an obscure bug in the error handling, which was reported to the bison developers and promptly fixed. (This does not affect most parsers; verbose error reporting is disabled by default)
- A selection of C++ coursework exercises from Imperial College students was briefly tested when compiled in bounds-checking mode, to see if there were any bounds errors detected. From a set of 60 submissions:
  - 25 failed to compile, both with and without bounds checking. This was largely caused by differences between g++ 2.95 as used by the students, and 3.3 as used here.
  - 4 had bounds errors that were detected by the system. All of these were errors in string handling, three of them were stack objects, and only one caused the program to crash outright; the others ran and gave apparently correct output. Electric fence was unable to detect any of these errors; valgrind detected three of them, as well as some occasions where an uninitialised value was used.
  - None of the others gave different output when compiled with bounds checking, compared to without it. Electric fence found no other errors; valgrind found several uses of uninitialised memory, but no further bounds errors.
- A range of free software has been successfully compiled with bounds checking, and runs without errors. However, no particularly stressful testing was performed on the applications, due to the limited availability of test suites; only superficial functionality was checked. This process was primarily intended to check that the bounds checking system could compile a variety of complicated code, often poorly written and abusing the C type system, without breaking in an obvious fashion.

  The following applications were tested in this manner: dash, tar, gzip, povray, e2fsprogs, easytag, mpg321, groff and gkrellm.

None of this is firmly conclusive; much more testing would be required for that. However, it does form a compelling demonstration that the system works *at least* as reliably as some other similar systems.

5.2. Performance

The bench++ suite was selected as the only available benchmark suite that exercised C++, and worked when testing. Several other benchmark suites were tried, but they were all so old that they did not even compile correctly with a current version of GCC. One, Todd Austin’s ptrdiff suite for exercising pointer operations, actually contained a bounds error introduced by an old version of yacc. Unfortunately the suite was so old that it would not function with a newer version.

For tests with real world applications, POV-Ray (a C++ ray tracer) was selected, since it is computationally expensive and entirely predictable in its behavior.

All of the tests were bound by core processing speed - they were not affected by IO or device latencies. Unless otherwise stated, they all ran well within the limits of available memory. All times are given in seconds, to one decimal place.

Table 4. Experimental system

	cyclone
Processor	AMD Athlon (T-Bird)
CPU clock speed	700MHz
Bogomips	1400
Motherboard	Asustek A7N8X

5.2.1. Bench++

Bench++ is a self-contained benchmark suite that runs a series of tests exercising both the compiler and C++ STL. It was run with and without bounds checking, using its own driver and result collating tools. A few of the benchmarks failed to compile, due to changes in g++ since the suite was released; this was independent of whether bounds checking was enabled or not, and is therefore discarded as irrelevant. The benchmark suite provided output in the form of ratios showing the relative time taken for each test. Where a ratio appears as ’inf’, that means one of the benchmarks completed so rapidly that it was measured as taking no time. Times were measured internally by the benchmark suite, based on the system nanosecond timer. The test driver also computed the accuracy over a number of runs; these ranged from 10% to 0.1% depending on the benchmark.

Table 5. Bench++ time ratios

Benchmark	**cyclone**	
	Original	**Bounds-checked**
A000094d	1.00	748.74
A000094e	1.00	730.39
A000094f	1.00	488.12
A000094g	1.00	347.83
A000094h	1.00	836.36
A000094i	1.00	12620.87
A000094j	1.00	666.67

Benchmark	**cyclone**	
	Original	**Bounds-checked**
B000002b	1.00	1149.40
B000003b	1.00	1142.29
B000004b	1.00	1001.73
B000010	1.00	336.31
B000011	1.00	2258.06
B000013	1.00	191.14
D000001	1.00	26.15
D000002	1.00	743.32
D000003	1.00	24.20
D000004	1.00	673.08
D000005	1.00	551.35
D000006	1.00	1208.70
E000001	1.00	1.56
E000002	1.00	1.75
E000003	1.00	1.89
E000004	1.00	1.91
E000007	1.00	1.25
F000001	1.00	2604.40
F000002	1.00	3236.84
F000003	1.00	107.49
F000004	1.00	81.68
F000005	1.00	11.27
F000007	1.00	26.35
F000008	1.00	221.98
G000001	1.00	4.77
G000002	1.00	0.65
G000003	1.00	4.15
G000004	1.00	4.54
G000007	1.00	3.05
H000005	1.00	inf
H000006	1.00	1009.01
H000007	1.00	111.65
H000008	1.00	1555.94
H000009	1.00	538.69
L000001	1.00	4300.00
L000002	1.00	3383.33

Benchmark	**cyclone**	
	Original	**Bounds-checked**
L000003	1.00	2687.50
L000004	1.00	inf
O000001a	1.00	403.39
O000001b	1.00	406.47
O000002a	1.00	419.23
O000002b	1.00	367.52
O000003a	1.00	603.64
O000003b	1.00	574.36
O000004a	1.00	898.83
O000004b	1.00	742.19
O000005a	1.00	386.08
O000005b	1.00	1727.27
O000006a	1.00	2681.82
O000006b	1.00	2406.93
O000007a	1.00	362.07
O000007b	1.00	529.33
O000008a	1.00	692.58
O000008b	1.00	1024.08
O000009a	1.00	1383.62
O000009b	1.00	1591.40
O000010a	1.00	409.64
O000010b	1.00	1086.31
O000011a	1.00	914.63
O000011b	1.00	660.61
O000012a	1.00	4.26
O000012b	1.00	12.40
P000001	1.00	inf
P000002	1.00	615.72
P000003	1.00	638.39
P000004	1.00	inf
P000005	1.00	431.01
P000006	1.00	245.45
P000007	1.00	250.68
P000008	1.00	179.31
P000010	1.00	1285.11
P000011	1.00	1837.27

Benchmark	**cyclone**	
	Original	**Bounds-checked**
P000012	1.00	1159.16
P000013	1.00	1346.43
P000020	1.00	253.90
P000021	1.00	305.46
P000022	1.00	296.34
P000023	1.00	239.22
S000001a	1.00	233.86
S000001b	1.00	186.61
S000002a	1.00	1604.10
S000002b	1.00	531.48
S000003a	1.00	3879.60
S000003b	1.00	530.61
S000004a	1.00	357.35
S000004b	1.00	774.04
S000005a	1.00	7483.05
S000005b	1.00	6194.92
S000005c	1.00	6296.61
S000005d	1.00	5305.08
S000005e	1.00	5720.34
S000005f	1.00	4752.14
S000005g	1.00	5033.90
S000005h	1.00	3991.53
S000005i	1.00	4432.20
S000005j	1.00	3466.10
S000005k	1.00	1728.81
S000005l	1.00	802.54
S000005m	1.00	762.39

The full list of what all these benchmarks sample, is included as an appendix. Some notable ones are:

- A000094i, sorting a list of 5000 integers by inserting them into a tree. This is inevitably pointer-intensive, which probably explains why the performance penalty was an order of magnitude larger than most of the others.
- G000002, testing the speed of iostream. This is the only test which ran faster with bounds-checking enabled. Since there are no pointers used here, this is probably an

optimization quirk (which likely indicates a bug in GCC, causing it to miss a potential optimization when the process isn’t distorted by the bounds-checking code).

Looking at the results more broadly, the performance impact of the bounds checking code is clearly dependant on the program being compiled.

- The ’A’ series, which is a group of classic artificial benchmarks, and the ’B’ series, which is a selection of CPU-intensive real-world code, performed particularly poorly, ranging from several hundred to several thousand times slower.
- The ’D’ series, which measures dynamic memory allocation, varied. Simple allocation of small-sized objects is approximately 25 times slower, while initialising those objects one at a time is several hundred times worse with bounds-checking enabled. Allocating objects on the stack was significantly worse, but this is not abnormal; both stack and heap objects take the same (long) time to be processed by the bounds checking library, but native allocation for stack objects is much faster than it is for heap objects.
- The ’E’ series, which tests exception handling, was largely unaffected, being less than twice as slow; this is unsurprising, since there are few pointer accesses involved.
- The ’F’ series, which tests control flow structures, was moderately slower with bounds checking, but not as much as many of the other tests. The large slowdown in F1 and F2 was caused by an optimisation that only worked with bounds checking disabled - a boolean can be tested without the use of an explicit comparison instructon on i386. This is an unnecessary limitation; the compiler modifications should detect that there are no pointer accesses involved, and avoid instrumenting operations that do not require it.
- The ’G’ series tests IO using the standard library. Performance here is affected only because some parts of the standard library are inlined into the program, and thusly get compiled with bounds checking enabled.
- The ’H’ series tests various bit-packing operations; these are invariably compiled into a simple machine instruction without bounds checking, and get replaced with a function call when it is enabled.
- The ’L’ series tests loops. The current bounds checking implementation handles loops poorly, putting slow and invariant operations in the loop body. See the "Possible extensions" section (6.2) for further details.
- The ’O’ series tests how well a selection of typical optimisations are performed. Since bounds checking has a tendancy to disrupt GCC’s optimisation passes, these have erratic but poor performance.
- The ’P’ and ’S’ series test function calls and common C++ idioms. All the delays here are extra function call overheads; these should be eliminated in the bounds checking code.

5.2.2. POV-Ray

POV-Ray was run several time with some sample scenes provided in the version 3.5 distribution, and the times compared with and without bounds checking. There were no problems found or differences in the output. All images were rendered at a resolution of 60x45 unless otherwise stated, selected because it fitted neatly in the available memory. Times were measured by the system scheduler, with a theoretical resolution of about 10us. Disk latencies to load the input data give a practical accuracy of around +/- 0.02s, although this was not experimentally tested.

Table 6. POV-Ray benchmark times

Benchmark	**cyclone**	
	Original	**Bounds-checked**
radiosity.pov	0.2	142.2
lathe2.pov	0.2	412.9
simple.pov	0.6	129.7
crystal.pov	0.2	678.2

Notes on the scenes rendered

radiosity.pov

Rendered with quality set to 3. This is a scene with complicated lighting.

lathe2.pov

Rendered with quality set to 9. These scene is comprised of a number of lathed shapes.

simple.pov

Rendered with quality set to 9, and anti-aliased with a threshold of 0.3. This scene contains a single light source, a sphere, and a chequered plane.

crystal.pov

Rendered with quality set to 9, at a resolution of 40x30. This scene contains transparent objects, so is rather more complex than earlier ones.

In general, POV-Ray is around 200 to 3000 times slower when compiled with bounds-checking enabled. Further investigation into precisely why this is the case were not possible due to time constraints.

5.2.3. Summary

The performance of programs compiled with bounds checking is acutely and universally poor.

Most programs run several hundred times slower; some are as much as several thousand times slower. This is primarily a problem of implementation rather than design; there are many opportunities for improving the performance significantly by making relatively simple changes to the code, such as replacing the object tree with a more efficient data structure.

The testing also revealed a related but less complex problem - the bounds checking library increases the memory usage a lot. This is not a hard problem to solve; it is really a simple bug. The library never frees emulated pointers, and it also prevents the application from freeing memory with free(); the latter could be changed fairly easily, but emulated pointers are trickier. See the "Possible extensions" section (6.2) for further discussion of this issue.

This is difficult to usefully measure, but the performance penalty for interactive applications is less significant. They already use very little processing power, and so the added overheads, while significant, do not cause a problem; the applications still respond quickly enough to the user.

However, in general the effect of bounds checking is to make binaries significantly larger and programs significantly slower, so it is not appropriate to build applications in this fashion for general purpose use - this is primarily a debugging tool.

6. Concluding remarks

6.1. Quirks

In most cases, the bounds-checking support works in an obvious and easily understood fashion, but there are a few things discovered which were unexpected.

6.1.1. Padded objects

In order to follow the restrictions laid down by the C and C++ standards regarding alignment of fields, structural types are sometimes internally padded with unused space. This can result in apparently incorrect code being accepted by the bounds-checking system.

```
struct a
{
        int x,y;
        char z[10];
};
```

At first glance, this struct seems simple enough. However, alignment restrictions can complicate matters. Taking i386-linux as an example, ints are 4 bytes in size and must be aligned on a 4-byte boundary. In a single struct a, this is clearly satisfied, but in an array of struct a, it is not - each struct would be 18 bytes in length, so the second array element would not be correctly aligned. To compensate for this, two bytes of padding are added to the end of every struct a, so it is 20 bytes in length instead.

All of this has an interesting effect. Accessing z[10] and z[11] within a struct a will not only be accepted by the bounds-checking library, it will work without problems - the struct has been artificially enlarged by the compiler, so the last field is longer.

This would not appear to be a problem; the bounds-checking library will not report any bounds errors, and the code will work correctly, even if some access should venture into the two bytes of padding. However, there is a problem. If we were to take the same code and compile it on a platform where ints are 2 bytes in size and must be aligned on a 2-byte boundary, then the struct would not need padding, and so accessing z[10] and z[11] would be (correctly) flagged as an error.

What all of this means, is that bounds-errors can be platform-dependent. If you thoroughly check every possible branch and path of a particular program on one platform, that does not mean that it will function correctly on a different one. This is not specific to bounds errors - many other portability issues behave the same way (particularly those regarding the size of types) - but needs to be taken into consideration.

6.1.2. Mixing checked and unchecked code

If checked and unchecked code is compiled into the same binary, some abnormalities can result. While these have been reduced to a small set of uncommon code, it is nonetheless possible for bounds-checking to cause problems in such cases. This section is intended to give an exhaustive list of those cases where it does not.

All the problems are related to passing data between checked and unchecked code, either by function arguments or return values, or by global variables. Furthermore, they relate only to passing pointers, not any other data types. It should be observed that passing a reference to a pointer, or a structural type which contains a publically accessible pointer, are also equivalent, as are inline class member functions that return pointers. Lastly, any pointer may be freely passed between checked and unchecked code *if it is within bounds at the time*.

It is when a pointer is currently outside the bounds of the object that it should point to - that is, when it is being emulated - that problems can occur. There are basically two cases:

- When passing such a pointer from unchecked code to checked code, the bounds-checked code will either assume this is a pointer to an unchecked object, and ignore it, or it will observe that this pointer lies within the bounds of a checked object, and thusly assume it is supposed to refer to that object. In the latter case, things will probably break. There is nothing that can be done about this - the unchecked code presumably has a bounds error. The best solution is to recompile the unchecked code with bounds-checking enabled.
- When passing such a pointer from checked code to unchecked code, the pointer will have an emulated value. Any attempt to manipulate such a value from unchecked code will have unexpected and undefined results. In an attempt to mitigate them, emulated pointers are padded with a few bytes on either side; this allows them to be incremented or decremented by small values, and small accesses to the memory surrounding them to be handled without actually causing a secondary bounds error (by overwriting some memory belonging to another object). A potential extension would be to use platform-specific memory management abilities, such as the UNIX mprotect() syscall, to deny read and write access to the emulated regions entirely, so that such accesses are trapped by the hardware.

Since it is highly abnormal for correct code to pass pointer values which are out of bounds, these corner cases are unlikely to ever cause problems in real programs. They are covered here for completeness only.

6.2. Possible extensions

In the process of designing this project, several ideas were pursued that were not eventually implemented, mostly due to time constraints.

6.2.1. Freeing emulated pointers

One problem with the emulated pointer scheme is that it allocates a lot of memory, and never frees it again. This was done because freeing the memory allocated for an emulated pointer will invalidate that pointer, which can result in some unexpected side-effects. Here are some of the other approaches that were considered, but rejected as being too unreliable or too complicated to implement at this time:

- The most obvious way is to free the memory for emulated pointers at the same time as that for the objects they emulate - so all pointers to array ’x’, whether in bounds or out of bounds, are invalidated at the same time.

  However, this does result in an effect which could silently change the behavior of code, even in the absence of bounds errors. When the emulated region is freed, obviously it is no longer possible to emulate comparisons between the pointers any more.

  Example 7. Freeing emulated regions too early can have side-effects

  This is a contrived example; I have not yet found any real code which would suffer this problem.

  ```
  {
    char *a = malloc(5);
    char *b = a + 8;
    char *c = a + 6;
    free(a);
    return b < c;
  }
  ```

  Assuming that malloc returns monotonically increasing values, this will result in the heap containing first the five bytes explicitly malloc-ed, then the emulated region for b, then the emulated region for c. Without bounds-checking, it will return false, since (a+8) is greater than (a+6). With bounds-checking, it would return true, since the emulated region for b is below that for c.

  This approach was discarded because it could introduce subtle differences in the behavior of the compiled program, without any way to detect this has happened.

- A partial solution is possible by detecting pointers which are never copied anywhere but in local automatic variables, and thusly can be freed when they go out of scope. This will trap the common case of loop control variables which are left one byte past the end of a string, but do no good in more complex cases.

This principle can be extended by instrumenting all pointer store/copy/return operations, so that the bounds-checking library can track which scopes know about a given pointer at a given point in time. This is effectively reference counting - when all references have been overwritten or gone out of scope, the emulated pointer no longer exists anywhere, and its memory can be freed.

Unfortunately, studying the C/C++ standards[iso-c99][iso-c++] carefully revealed that this can't work. It is explicitly allowed that a pointer can be copied into an integer variable, and stored there. No arithmetic may be performed on it, but it can later be copied back into a pointer variable and treated as if it were a direct copy of the original pointer. In order to track pointers through such operations, it would be necessary to instrument *every* integer copy operation in the program. This would make the compiled program prohibitively slow.

- A better variation on the same theme would be one of the garbage collection algorithms. Since all the allocated regions are known, they can be scanned for values which are valid pointers to known emulated regions, so a mark-and-sweep garbage collector can be created.

  It is possible for there to be false positives - an integer value at a given location might coincidentally be equal to such a pointer, but this is unlikely. A more likely false positive is when a pointer to an old emulated region, that was stored in a region which has since been freed, reappears when a new object is allocated in the same location. This can be solved by clearing memory regions when freeing them.

  Other garbage collection techniques may perform better here.

6.2.2. Structural types

Structural types (classes, structs and arrays) can contain multiple fields. It is possible for a bounds error to occur between these fields, rather than between allocation objects. It would be useful if the bounds-checking code could detect such errors. This is a problem specific to C++ - in C, it is allowed by the language specification to let a pointer go past the end of one member into another, although alignment and padding may leave a gap between them. In C++, no such operation is allowed in a class or non-trivial struct.

In order to detect such errors, a few things are needed.

- The type of every object allocated.

  This is simple for static or automatic objects, and those allocated with new. For those allocated with malloc(), it is harder. I hypothesize that the best approach would be to note the expected type of every pointer access, and assume a malloc()ed object has the same type as its first access.

- The structure of all types used.

  This requires the compiler generate a suitable representation of all the types, store it in the generated program, and pass it to the bounds-checking library during global construction.

  Since types are not uniquely identified by their name (type names are only unique within a single compilation unit), it will be necessary to implement some mechanism by which they can be recognized. The easiest way is to emit each type once for every compilation unit it is used in, and reference it only with a pointer to that record. This may result in significant memory consumption, particularly in the presence of templated classes.

  This can be simplified by observing that the structure as represented in the source is not the best form for the purposes of bounds checking. All that is really needed is a description of where the boundaries within the object occur - the hierarchical structure is not significant. Therefore, types can be easily represented as a list of field sizes. In addition, some further optimizations are possible. Take the following structural types:

```
struct a
{
  int x;
  int y;
};

struct b
{
  char n[20];
  struct a z;
};

struct c
{
  int n[5];
  char x[4];
  char y[4];
};
```

  Assuming that char is one byte, int is four, and there are no alignment restrictions, this results in:

  - An 8-byte type ’struct a’, comprised of two 4-byte fields.
  - A 28-byte type ’struct b’, comprised of a 20-byte field followed by two 4-byte fields.
  - A 28-byte type ’struct c’, comprised of a 20-byte field followed by two 4-byte fields.

  Which means that structs b and c are equivalent, and so they can be described as the same type to the bounds-checking library. If the type ’struct a’ is never used outside other type definitions, then it can be discarded as irrelevant.

Given this data, the bounds-checking library simply has to check each pointer arithmetic operation in terms of the nearest internal boundaries, instead of the allocation ones.

One problem that this raises, is how to distinguish between multiple overlapping objects. Take the following code:

```
struct a
{
  char m[10];
  char n[10];
  int o, p;
};
struct a x;
for (char *c = &x; c < (&x + sizeof(x)); c++)
  *c = 0;
```

The problem here is the question of how the bounds-checking library can tell whether this loop should be bounded by the limits of ’x’, or ’x.m’. One obvious approach is to say that &x should be bounded to x, and &x.m should be bounded to x.m. However, this would require being able to tell the difference between those two values. Since the only place to store this information is in the value itself, one of the pointers (probably &x.m) would have to be emulated. This requires instrumenting all address-of operations, and may have further problems; due to the complexity, this approach was discarded.

Another, simpler variation on this approach is to assume that the largest object with its base pointer at a given address is used - so the above loop is bounded to x, because x was the largest object starting at the address &x; if &x.m had been used, it would still have been bounded to x, since they start at the same address. Once the bounds have been determined in this fashion, they are encoded into an emulated pointer - so the variable c will carry an emulated value after the first increment, saying that it should always be bound to x. This approach would require changing the instrumentation of field selection operations - &x.n must not bind the returned value to x, so it can’t be treated the same as a regular addition.

Neither of these approaches are very good. Ideally a strong distinction would be made between field selection and pointer arithmetic, and the bounds-checking library would become type-aware, but that goes far beyond the scope of this project. It should be noted that such a system would add new constraints not present in C - you could no longer directly manipulate the in-memory structure of an object, or use type punning to access it in strange ways. However, these constraints are probably desirable.

6.2.3. Flow-based optimization

It should be possible to leverage the data-flow analysis already performed by GCC to improve performance significantly. It is not necessary to check every operation in order to detect all the bounds errors - only those where the result is new or otherwise uncertain. Take the following function:

```
int f(char *a)
{
  a[0] = 0;
  a[1] = 1;

  return a[0] + a[1];
}
```

Since the value of a has not changed between the assignments and the return statement, it is not necessary to check that a[0] and a[1] are valid accesses a second time - if they were within bounds the first time, they still will be, and if they were not, the program will have terminated with an error. In addition, it is not necessary to look up ’a’ on the object tree more than once. It would probably be significantly more efficient if cases like these could be detected and optimized away.

Another variation on the same theme leverages the fact that objects are formed of a single continuous region of memory, with no gaps.

```
void f(char *a)
{
  a[0] = 0;
  a[5] = 5;
  a[1] = 1;
}
```

Here, it is not necessary to check all three accesses. If a[0] and a[5] are both in bounds, then a[1] must also be, since it lies between them. This should be independent of the sequence of the instructions - flow analysis should identify groups that can be handled in this manner and replace them with a single query that covers the entire range. In this case, checking for an access starting at ’a’ that is 6 bytes in length would be appropriate.

The same principle can be extended to loops.

```
void f(char *a)
{
  int i;
  for (i = 0; i < 5; i++)
    a[i] = ’\0’;
}
```

This should be compiled with a single check, rather than five.

There is another optimisation that would help significantly for many programs. Currently, the expression instrumentation is indiscriminate in how it modifies the compiled code. Often it does things that are unnecessary.

```
int f(int d)
{
  int i, t = 1;
  for (i = 0; i < d; i++)
    t = t * 2;
  return t;
}
```

This function will be instrumented with a range of calls to the bounds checking library, noting the creation of stack objects and entry/exit of scope. None of these are necessary; the address of the stack objects is never taken and the function manipulates no pointers.
[Jones & Kelly, 1997] added an optimisation pass that looked for such calls and eliminated them, but this would be significantly more complicated for C++ (merely due to how the language features have structured parts of the compiler). In addition, it only works for simple cases - where no pointers are used at all. Ideally, this would be implementing using the data flow analysis, so that the bounds checking library is only called for things that definitely need to be checked.

6.3. Overview

The system fundamentally works and is useful. However, it could be significantly _more_ useful with a moderate amount of further work. One of the most significant issues is the performance penalty it imposes on checked code. I estimate that a few weeks of work could reduce the penalty by at least an order of magnitude. The priorities here are to replace the object tree with a more efficient algorithm, and to eliminate spurious calls to the bounds checking library when constructing objects that never have their address taken.

The emulated pointer method is an effective solution to the joint problems of allowing pointers to progress past the end of the associated objects and still maintaining normal pointer comparison semantics, but it has more potential past that. It is a necessary precondition for implementing checking of the boundaries of fields within structural types without changing the layout of those types.

The memory usage of the system is unnecessarily high, and this needs to be fixed; it should be relatively simple to do so.

Last, but not least, the bounds checking library needs rewriting; it is rapidly reaching the point where it will not be maintainable. There is a lot of dead code in it, and it is not well structured or commented, since it has been modified several times over the past few years to add features which were not originally planned. Nonetheless, it functions adequetely for the current system.

Bibliography

[Jones & Kelly, 1997] “Backwards-compatible bounds checking for arrays and pointers in C programs”, R W M Jones and P H J Kelly, *Third International Workshop on Automated Debugging, Edited by M Kamkar, Edited by D Byers, 1997.*

[Jones, 1995] *A bounds checking C compiler, Richard Jones, Department of Computing, Imperial College, June 1995.*

[Miller, Lars & So, 1990] *Communications of the ACM, Volume 33, Issue 12, December, 1990, “An empirical study of the reliability of Unix utilities”, B P Miller, F Lars, and B So, pp. 32-44.*

[Seeley, 1989] *A Tour of the Worm, January 1989, USENIX 1989 Winter Conference.*

[Patil & Fischer, 1996] *Software - Practice and Experience, Volume 27, Issue 1, January, 1997, Online: 1097-024X, Print: 0038-0644, “Low-cost, Concurrent Checking of Pointer and Array accesses in C programs”, Harish Patil and Charles Fischer, June 6, 1996, pp. 87-110.*

[Flater, Yesha & Park, 1993] *Software - Practice and Experience, Volume 23, Issue 6, June, 1993, Online: 1097-024X, Print: 0038-0644, “Extensions to the C Programming Language for Enhanced Fault Detection”, David W Flater, Yelena Yesha, and E K Park, August 10, 1992, pp. 305-316.*

[Kendall, 1983] *Bcc: run-time checking for C programs, 1983, USENIX 1983 Summer Conference.*

[Wagner, Foster, Brewer & Aiken, 2000] *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities, February 2000.*

[gccint] *GCC internals manual, Free Software Foundation, 1988-2003.*

[iso-c99] *Programming language C, ISO 9899:1999, International Organisation for Standardisation (ISO), 1999.*

[iso-c++] *Programming language C++, ISO/IEC 14882:1998, International Organisation for Standardisation (ISO), 1998.*

[Purify, 1995] *Purify: Fast Detection of Memory Leaks and Access Errors, 1995-1999.*

[Austin, Breach & Sohi, 1994] *Efficient Detection of All Pointer and Array Access Errors, 1994, Todd M Austin, Scott E Breach, and Gurindar S Sohi, SIGPLAN Conference on Programming Language Design and Implementation.*

A. Bench++ benchmark descriptions

This list is copied directly from the bench++ documentation.

a000090

Measure clock resolution by second differences

a000091

Dhrystone

a000092

Whetstone

a000094a..k

Hennesy benchmarks

b000002b

Tracker: float

b000003b

Tracker: double

b000004b

Tracker: float & int

b000010

Orbit

b000011

Kalman

b000013

Centroid

d000001

malloc & free: 1000 ints

d000002

malloc & init & free: 1000 ints

d000003

new & delete: 1000 ints

d000004

new & init & delete: 1000 ints

d000005

alloca: 1000 ints (optional test)

d000006

alloca & init: 1000 ints (optional test)

e000001

Local exception caught

e000002

Class method exception caught

e000003

Procedure exception caught: 3-deep

e000004

Procedure exception caught: 4-deep

e000006

Declared Procedure exception caught: 4-deep

e000007

Procedure exception caught: 4-deep re-thrown at each level

e000008

Procedure exception 4-deep: Implemented using setjmp/longjmp

f000001

Boolean assignment

f000002

Boolean if

f000003

2-way if/else

f000004

 2-way switch

f000005

 10-way if/else

f000006

 10-way switch

f000007

 10-way sparse switch

f000008

 10-way virtual function call

g000001

 iostream.getline: 20 char buffer

g000002

 iostream.>> : 20 chars in loop

g000003

 iostream.<< : 20 char buffer

g000004

 iostream.<< : 20 chars in loop

g000005

 istrstream.>> : int

g000006

 istrstream.>> : float

g000007

 fstream.open/fstream.close

h000001

 packed bit arrays

h000002

 unpacked bit arrays

h000003

packed bit ops in loop

h000004

unpacked bit ops in loop

h000005

int conversion

h000006

10-float conversion

h000007

bit-fields

h000008

bit-fields and packed bit array

h000009

pack and unpack class objects

l000001

"for" loop

l000002

"while" loop

l000003

inf. loop w/break

l000004

5-iteration loop

o000001a

Constant Propagation (including math functions)

o000001b

Same, Hand Optimized

o000002a

Local Common Sub-expression (including math functions) o000002b

Same, Hand Optimized

o000003a

Global Common Sub-expression

o000003b

Same, Hand Optimized

o000004a

Unnecessary Copy

o000004b

Same, Hand Optimized

o000005a

Code Motion (including math functions)

o000005b

Same, Hand Optimized

o000006a

Induction Variable

o000006b

Same, Hand Optimized

o000007a

Reduction in Strength (including math functions)

o000007b

Same, Hand Optimized

o000008a

Dead Code

o000008b

Same, Hand Optimized

o000009a

Loop Jamming

o000009b

Same, Hand Optimized

o000010a

Redundant Code

o000010b

Same, Hand Optimized

o000011a

Unreachable Code

o000011b

Same, Hand Optimized

o000012a

String Ops

o000012b

Same, Hand Optimized

p000001

Procedure Call: No Args

p000002

Procedure Call: No Args: Catches Exceptions

p000003

Static Class Method Call: No Args: Catches Exceptions

p000004

Inline Procedure Call: No Args

p000005

Static Class Method Call: 1-int Arg: Catches Exceptions

p000006

Static Class Method Call: 1-int *Arg: Catches Exceptions

p000007

Static Class Method Call: 1-int &Arg: Catches Exceptions p000008

Procedure Call: No Parameters: Called thru pointer, Catches Exceptions

p000010

Procedure Call: 10-int Args: Catches Exceptions

p000011

Procedure Call: 20-int Args: Catches Exceptions

p000012

Procedure Call: 10-(3-int) Args: Catches Exceptions

p000013

Procedure Call: 20-(3-int) Args: Catches Exceptions

p000020

Class Method Call: 1-"this" Arg: Catches Exceptions

p000021

Virtual Class Method Call: 1-"this" Arg: Catches Exceptions

p000022

Virtual Const Class Method Call: 1-"this" Arg: Catches Exceptions

p000023

Same as p000022: called in loop to see if lookup is optimized

s000001a

Max: C++ Style

s000001b

Max: C Style

s000002a

Matrix: C++ Style

s000002b

Matrix: C Style

s000003a

Iterator: C++ Style

s000003b

Iterator: C Style

s000004a

Complex: C++ Style

s000004b

Complex: C Style

s000005a

Stepanov: C++ Style Abstraction Level 12

s000005b

Stepanov: C++ Style Abstraction Level 11

s000005c

Stepanov: C++ Style Abstraction Level 10

s000005d

Stepanov: C++ Style Abstraction Level 9

s000005e

Stepanov: C++ Style Abstraction Level 8

s000005f

Stepanov: C++ Style Abstraction Level 7

s000005g

Stepanov: C++ Style Abstraction Level 6

s000005h

Stepanov: C++ Style Abstraction Level 5

s000005i

Stepanov: C++ Style Abstraction Level 4

s000005j

Stepanov: C++ Style Abstraction Level 3

s000005k

Stepanov: C++ Style Abstraction Level 2

s000005l

Stepanov: C++ Style Abstraction Level 1

s000005m

Stepanov: C++ Style Abstraction Level 0

Notes

1. http://gcc.gnu.org/
2. http://manju.cs.berkeley.edu/ccured/
3. http://www.research.att.com/projects/cyclone/
4. http://developer.kde.org/~sewardj/
5. Although there are a few language-specific hooks used later, for things like function calling conventions
6. An object has an lvalue if it can appear on the left-hand side of an assignment - it has storage allocated. A variable ’x’ has an lvalue, but the expression ’x + 1’ does not.