Bounds-Checking Entire Programs without Recompiling

Nicholas Nethercote
Computer Laboratory
University of Cambridge
United Kingdom
njn25@cam.ac.uk

Jeremy Fitzhardinge
San Francisco
United States
jeremy@goop.org

ABSTRACT

This paper presents a new technique for performing bounds-checking. We use dynamic binary instrumentation to modify programs at run-time, track pointer bounds information and check all memory accesses. The technique is neither sound nor complete, however it has several very useful characteristics: it works with programs written in any programming language, and it requires no compiler support, no code recompilation, no source code, and no special treatment for libraries. The technique performs best when debug information and symbol tables are present in the compiled program, but *degrades gracefully* when this information is missing—fewer errors are found, but false positives do not increase. We describe our prototype implementation, and consider how it could be improved by better interaction with a compiler.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, monitors, symbolic execution*; D.2.4 [**Software Engineering**]: Software/Program Verification—*assertion checkers, reliability, validation*; D.3.2 [**Language Classifications**]: Multiparadigm languages—*C, C++*

General Terms

Reliability, Verification

Keywords

Bounds-checking, memory debuggers

1. INTRODUCTION

Low-level programming languages like C and C++ provide raw memory pointers, permit pointer arithmetic, and do not check bounds when accessing arrays. This can result in very efficient code, but the unfortunate side-effect is that accidentally accessing the wrong memory is a very common programming error.

The most obvious example in this class of errors is exceeding the bounds of an array. However the bounds of non-array data objects can also be violated, such as heap blocks, C structs, and stack frames. We will describe as a *bounds error* any memory access which falls outside the intended memory range.

These errors are not difficult to introduce, and they can cause a huge range of bugs, some of which can be extremely subtle and lurk undetected for years. Because of this, tools for preventing and identifying them are extremely useful. Many such tools are available, using a variety of techniques. None are ideal, and each one has a different set of characteristics, including:

- the kinds of bugs they can and cannot spot;
- which regions of memory (heap, static, stack) they work with;
- the number of false positives they produce;
- whether they are static or dynamic;
- which parts of a program they cover, in particular how they work with libraries;
- what level of compiler support is needed.

In this paper we describe a new technique for identifying bounds errors. The basic idea is that all data objects that have bounds we wish to check are tracked, as are all pointers; each pointer has a legitimate memory range it can access, and accesses outside this range are flagged as errors. Our tool effectively turns programs pointers into *fat pointers* on-the-fly. However, the pointer ranges are maintained separately from the pointers themselves, so pointer sizes do not change.

The technique spots many, but not all, bounds errors in the heap, the stack, and in static memory; it gives few false positives; it is dynamic, and relies on dynamic binary translation. These characteristics are not particularly exciting.

The main contribution of this paper is that our bounds-checking technique has several unique characteristics: it does not require any compiler support; it works with programs written in any language; and it checks entire programs, without requiring any special treatment for libraries. This last characteristic is particularly important—inadequate treatment of library code is the single major shortcoming of many previous bounds-checking techniques.

Thus, our technique has its advantages and disadvantages, and is unlikely to be a world-beater in its own right. However, it is a useful complement to existing techniques, and can hopefully be part of a future hybrid technique that is a world-beater.

This paper is structured as follows. Section 2 describes the technique at a high-level, and Section 3 describes our prototype implementation, and gives more low-level details. Section 4 discusses the shortcomings of the technique, and ways in which a compiler might co-operate with our tool so that it can give better results. Section 5 describes related work, and discusses how our technique compares to others. Section 6 discusses future work and concludes.

2. THEORY

This section provides a high-level description of our technique. Certain details are kept vague, and subsequently fleshed out in Section 3.

2.1 Overview

The basic idea is simple. Every pointer has a range of addresses it can legitimately access. The range depends on what the pointer originally pointed to. For example, the legitimate range of a pointer returned by `malloc()` is the bounds of the heap block pointed to by that pointer. We call that range the pointer's *segment*. All memory accesses are checked to make sure that the memory accessed is within the accessing pointer's segment. Any violations are reported.

Pointers are often used in operations other than memory accesses. The obvious example is pointer arithmetic; for example, array elements are indexed using addition on the array's base pointer. However, if two pointers are added, the result should not be used to access memory; nor should a non-pointer value be used to access memory. Thus we also need to know which program values are non-pointers. The result is that every value has a run-time type, and we need a simple type system to determine the type of all values produced by the program.

The following sections describe aspects of the technique in more detail.

2.2 Metadata

The technique requires maintenance of *metadata* describing the run-time types of data objects. We say this metadata *shadows* the real values. This metadata has one of the following four forms.

- X, a *segment-type*, describes a segment; this includes its base address, size, location (heap, stack, or static), and status (in-use or freed).
- n, a *non-pointer-type*, describes a value which is known to be a non-pointer.
- $p(X)$, a *pointer-type*, describes a value which is known to be a pointer to segment X;
- u, an *unknown-type*, describes a value for which the type is unknown.

The following data objects will have metadata associated with them.

- Segments: each new segment is given a new segment-type X;
- Memory and registers: each value is shadowed by one of the types n, u or $p(X)$.

In principle, every value produced, of any size, can be assigned a type. However, our most important checking takes place when memory is accessed, which is always through word-sized pointers. Therefore, the shadow value tracking can be done at word-sized granularity.

2.3 Checking Accesses

Every load and store is checked. When accessing a memory location m through a value `x` we look at the type of `x` and behave accordingly.

- n: Issue an error about accessing memory through a non-pointer.
- u: Do nothing.
- $p(X)$: If m is outside the range of X, issue an error about accessing memory beyond the legitimate range of `x`; if X is a freed segment, issue an error about accessing memory through a dangling pointer.

Note that in all cases the memory access itself happens unimpeded.

2.4 Allocating and Freeing Segments

There are four steps in dealing with each segment.

1. Identify when it is allocated, in order to create a segment-type describing it.
2. Upon allocation, set the metadata for the new words in the segment.
3. Identify each pointer to the segment, in order to set the pointer's shadow to the appropriate pointer-type.
4. Identify when it is freed, in order to change the status of its segment-type to "freed".

The way these aspects are handled differs between the heap, static and stack segments. In addition, the exact meaning of the term "segment", and thus the kinds of errors found, differs between the three memory areas. The following sections discuss these differences. Section 3.3.2 describes how the implementation deallocates segment types.

2.4.1 Heap Segments

Heap segments are the easiest to deal with. For the heap, one heap segment represents one heap block. The four steps for dealing with heap blocks are as follows.

1. By intercepting `malloc()`, `calloc()`, `realloc()`, `new`, and `new[]`, we know for every heap segment its address, size, and where it was allocated. We can thus easily create the new segment-type, store it in a data structure, and set the shadow value of the returned pointer to point to it.
2. All the words within a newly allocated heap block have their shadows set to n, since they should definitely not be used to access memory, being either zeroed (for `calloc()`) or uninitialised.
3. The pointer returned by the allocation function has its shadow set to $p(X)$, where X was the segment-type just created. At this point, this is the only pointer to the segment.

4. By intercepting `free()`, `realloc()`, `delete`, and `delete[]`, we know when heap segments are freed; we can look up the segment-type in the data structure by its address (from the pointer argument), and change the segment-type's status to "freed".[1]

Note that our technique as described only detects overruns past the edges of heap blocks. For example, if the heap block contains a struct which contains two adjacent arrays, overruns of the first array into the second will not be caught. This could be improved by using debugging information, which would tell us the type of the pointer. Then with some effort, fields accesses could be identified by looking at offsets from the block pointer. However, see Section 2.4.2 for a discussion of some difficulties in identifying array overruns in these cases.

2.4.2 Static Segments

Identifying static segments is more difficult. First, identifying static segments by looking only at the compiled executable is close to impossible. So we rely on debugging information to do this; if this information is not present, all pointers to static data objects will have the type u, and no accesses to static memory will be checked.

We can create segment-types for each static data object, e.g. arrays, structs, integers; thus for static memory, a segment represents an single data object. The four steps for dealing with static segments are as follows.

1. If debugging information is present, we identify static arrays in the main executable at startup, and for shared objects when they are mapped into memory.

2. Newly loaded static memory can contain legitimate pointers. Those we can identify, from debugging information, are set to point to static segments. The remaining words have their type set according to their value. If a word is clearly a non-pointer—e.g. a small integer—we give it the type n, otherwise we give it the type u.

3. Identifying pointers to static segments is harder than identifying pointers to heap segments. This is because compilers have a lot of control over where static data objects are put, and so can hard-wire absolute addresses into the compiled program. One possibility is to rely on a simple assumption: any value that looks like a pointer to an object, is a pointer to that object (even if it is not to the start of the object). For example, if a static integer array `a[10]` is located at address `0x8049400`, we may see the following x86 instruction snippets used to load elements of the array into register `%edx`:

```
movl $0x8049400, %eax          # load a[0]
movl (%eax), %edx

movl $0x8049400, %eax          # load a[5]
movl 20(%eax), %edx

movl $0x8049414, %eax          # load a[5]
movl (%eax), %edx

movl $0x8049400, %eax          # load a[5]
movl $5, %ebx
movl (%eax,%ebx,4), %edx

movl $5, %eax                  # load a[5]
movl 0x8049400(,%eax,4),%edx

movl $0x8049428, %eax          # load a[10]
movl (%eax), %edx
```

This approach works in all these cases except the last one; it is a bounds error, because the pointer value used falls outside the range of `a`. However, detecting this error is not possible, given that another array `b[]` might lie directly after `a[]`, and so an access to `a[10]` is indistinguishable from an access to `b[0]`.

Unfortunately, this approach sometimes backfires. For example, the program `gap` in the SPEC2000 benchmarks initialises one static pointer to a static array as `p = &a[-1]`; `p` is then incremented before being used to access memory. When compiled with GCC, the address `&a[-1]` falls within the segment of another array lying adjacent; thus the pointer is given the wrong segment-type, and false positives occur when `p` is used to access memory. A similar problem occurs with the benchmark `gcc`, in which an array access of the form `a[var - CONST]` occurs, where `var` $\geq$ `CONST`; the address `a[-CONST]` is resolved by the compiler, giving an address in an adjacent array.

The alternative is to be more conservative in identifying static pointers, by only identifying those that point exactly to the beginning of static arrays. Unfortunately, this means some static array accesses will be missed, such as in the third assembly code snippet above, and so some errors will be missed. In particular, array references like `a[i-1]` are quite common and will not be handled if the compiler computes `a[-1]` at compile-time.

As well as appearing in the code, static pointers can also occur in static data. These can be identified from the debugging information.

4. We identify when static arrays in shared objects are unmapped from memory. With an appropriate data structure, we can mark as freed all the segments within the unmapped range. This is safe because static memory for a shared object is always contiguous, and never overlaps with heap or stack memory.

As with heap segments, arrays within static structs could be identified with some effort. However, as with top-level arrays, direct accesses that exceed array bounds—like the `a[10]` case mentioned previously—cannot be detected.

2.4.3 Stack Segments

The stack is the final area of memory to deal with. One could try to identify bounds errors on individual arrays on the stack, but a simpler goal is to identify when a stack frame is overrun/underrun. In this case, one stack segment represents one stack frame. The four steps in dealing with stack segments are as follows.

[1] Section 3.9 explains how we can deal with programs that use custom allocators.

1. When a function is entered, function arguments are above the stack pointer,[2] the return address is at the stack word pointed to by the stack pointer, and local variables used by the function will be placed below the stack pointer. If we know the size and number of arguments, we can create a segment-type X for the stack frame, which is bounded at one end at the address of most extreme function argument, and is unbounded at the other end. This segment-type can be stored in a stack of segment-types.

2. The values within the segment should be shadowed with n when the segment-type is created, as they are uninitialised and should not be used as pointers. The exception is the function arguments, whose shadow values have already been initialised and should be left untouched.

3. The only pointer to a new stack segment is the stack pointer; its shadow is set to $p(X)$, where X is the segment-type just created.

4. When the function is exited, the segment-type can be removed from the segment-type stack, and marked as deallocated.[3]

These segments allow us to detect two kinds of errors. First, any access using the stack pointer (or a register assigned the stack pointer's value, such as the frame pointer) that exceeds the end of the stack frame will be detected.

Second, and more usefully, any dangling pointers to old stack frames will be caught. Such dangling pointers can occur if the address of a stack variable is saved, or returned from the function. Even better, it will detect any use of dangling pointers in multithreaded programs that have multiple threads sharing stacks. Such bugs can be extremely difficult to trace down.

Knowing the size and number of function arguments requires access to the debug information. If the debug information is missing, an approximation can be used: instead of a segment that is bounded at one end, use an unbounded segment. The first error above, violations of the stack's edge, will not be detected. However, the second error, that of dangling pointer use, will be. This is because any access to a freed segment triggers an error, regardless of whether that access is in range. This approximation could also be used for functions like `printf()` that have a variable number of arguments.

As with heap blocks and static memory, we could do tracking of individual variables within stack frames, but it would be very fiddly.

2.5 Operations

Every data-manipulating instruction executed by the program must be shadowed by an operation to manipulate the relevant metadata. These shadow operations are described in the following sections.

2.5.1 *Copying Values*

For machine instructions that merely copy existing values around (e.g. memory-register, register-memory, and register-register copies) the metadata must be correspondingly copied. Note that $p(X)$ metavalues must be copied by reference, so that if the segment pointed to, X, is freed, all $p(X)$ types that point to it see its change in status.

2.5.2 *New Static Values*

Machine instructions that mention constants effectively introduce new values. The type of each static value is found in the same way that the types of values in newly loaded static memory are given, as was described in Section 2.4.2; this will be n, u, or the appropriate pointer-type.

2.5.3 *New Dynamic Values*

Each machine operation that produces a new value has a corresponding operation to produce the metadata for that value. This includes arithmetic operations, logic operations, and address computations. Figure 1 shows the metadata computation for several common binary operations; the type of the first operand is shown in the leftmost column, and the type of the second operand is shown in the top row.

$+$	n	u	$p(Y)$
n	n	u	$p(Y)$
u	u	u	u
$p(X)$	$p(X)$	u	n^*

(a) Add

$\times$	n	u	$p(Y)$
n	n	n	n
u	n	n	n
$p(X)$	n	n	n^*

(b) Multiply

&	n	u	$p(Y)$
n	n	u	$p(Y)$
u	u	u	u
$p(X)$	$p(X)$	u	n^*

(c) Bitwise-and

^	n	u	$p(Y)$
n	u	u	u
u	u	u	u
$p(X)$	u	u	u

(d) Bitwise-xor

Figure 1: Basic operations

Let us start with addition, shown in Figure 1(a). Adding two non-pointers results in a non-pointer. Adding a non-pointer to a pointer results in a pointer of the same segment; this is crucial for handling pointer arithmetic and array indexing. Adding two pointers together produces a non-pointer, and an error is issued; while not actually incorrect, it is such a dubious operation that we consider it worth flagging. The '*' on that entry in the table indicates an error message is issued. Finally, if either of the arguments are unknown, the result is unknown. Thus unknown values tend to "taint" known values, which could lead to a large loss of accuracy quite quickly. However, before the metadata operation takes place, we perform the real operation and check its result. As we do for static values, we use a range test, and if the result is clearly a non-pointer we give it the type n. This is very important to prevent unknown-ness from spreading too much.

Multiplication, shown in Figure 1(b), is simpler; the result is always a non-pointer, and an error is issued if two pointers are multiplied. At first we issued errors if a pointer was multiplied by a non-pointer, but in practice this occasionally happens legitimately. Similarly, division sometimes occurs

[2] We assume the stack is growing towards lower addresses.
[3] This assumes function entries and exits are paired nicely, and can be identified. Section 3.5 discusses what happens in practice.

on pointers, e.g. when putting pointers through a hash function. Several times we had to remove warnings on pointer arithmetic operations we had assumed were ridiculous, because real programs occasionally do them. Generally, this should not be a problem because the result is always marked as n, and any subsequent use of the result to access memory will be flagged as an error.

Bitwise-and, shown in Figure 1(c), is more subtle. If a non-pointer is bitwise-and'd with a pointer, the result can be a non-pointer or a pointer, depending on the non-pointer value. For example, if the non-pointer has value `0xfffffff0`, the operation is probably finding some kind of base value, and the result is a pointer. If the non-pointer has value `0x000000ff`, the operation is probably finding some kind of offset, and the result is a non-pointer. We deal with these possibilities by assuming the result is a pointer, but also doing the range test on the result and converting it to n if necessary. The resulting shadow operation is thus the same as that for addition.

For bitwise-xor, shown in Figure 1(d), we do not try anything tricky; we simply use a range test to choose either u or n. This is because there are not any sensible ways to *transform* a pointer with bitwise-xor. However, there are two cases where bitwise-xor could be used in a non-transformative way. First, the following C code swaps two pointers using bitwise-xor.

```
p1 ^= p2;
p2 ^= p1;
p1 ^= p2;
```

Second, in some implementations of doubly-linked lists, the forward and backward pointers for a node are bitwise-xor'd together, to save space.[4] In both these cases, the pointer information will be lost, and the recovered pointers will end up with the type u, and thus not be checked.

Most other operations are straightforward. Increment and decrement are treated like addition of a non-ptr to a value, except no range test is performed. Address computations (on x86, using the `lea` instruction) are treated like additions. Shift/rotate operations give a n or u result, depending on the result value—we do not simply set the result to n just in case a pointer is rotated one way, and then back to the original value. Negation and bitwise-not give an n result.

2.5.4 Subtraction

We have not mentioned subtraction yet. It is somewhat similar to addition, and is shown in Figure 2. Subtracting two non-pointers gives a non-pointer; subtracting a non-pointer from a pointer gives a pointer; subtracting a pointer from a non-pointer is considered an error.

$-$	n	u	$p(Y)$
n	n	u	n^*
u	u	u	u
$p(X)$	$p(X)$	u	$n/?$

Figure 2: Subtraction

[4]The pointers can be recovered by using the bitwise-xor'd values from the adjacent nodes; their pointers can be recovered from their adjacent nodes, and so on, all the way back to the first or last node, which holds one pointer bitwise-xor'd with `NULL`.

The big complication is that subtracting one pointer from another is legitimate, and the result is a non-pointer. If the two pointers involved in the subtraction point to the same segment, there is no problem. However consider this C code:

```
char p1[10];
char p2[10];
int  diff = p2 - p1;
p1[diff] = 0;
```

This uses the pointer `p1` to access the array pointed to by `p2`, which is a different segment. ANSI-C actually forbids such inter-array pointer subtraction, but in practice it occurs in real C code. Also, it may be valid in other programming languages, and if our technique is to be language-independent, we must handle it.

The problem is that the addition of a non-pointer with a pointer can result in a pointer to a *different segment*. The most accurate way to handle this is to generalise the type n to $n(X,Y)$, which is like n for all operations except addition and subtraction, in which case we require that $p(X)+n(X,Y)$ gives $p(Y)$, $p(X)-n(Y,X)$ gives $p(Y)$, and so on. Also, pointer differences should be added transitively, so that $n(X,Y)+n(Y,Z)$ gives $n(X,Z)$.

However, this is a complex solution for a problem that does not occur very often. We describe how the implementation handles this case more simply in Section 3.7.

2.5.5 Propagation of Unknown

Note that it is tempting to be less rigorous with preserving u types. For example, one might try making the result of $u+p(X)$ be $p(X)$. After all, if the u operand is really a non-pointer, the result is appropriate, and if the u operand is really a pointer, the result will undoubtedly be well outside the range of $p(X)$, so any memory accesses through it will be erroneous, and should be caught. By comparison, strict u-preservation will cause us to miss this error.

At first, we tried doing this, being as aggressive as possible, and assuming unknowns are pointers when possible. However, in practice it causes false positives, usually from obscure sequences of instructions that would fool the shadow operations into assigning the wrong segment-type to a pointer, e.g. assigning a heap segment-type to a stack pointer.

Each time we saw such a false positive, we reduced the aggressiveness; after it happened several times, we decided that trying to second-guess like this was not the best way to proceed. Instead, we switched to being more conservative with u values, and using range tests throughout.

There is no way of handling these shadow operations that is clearly the best. Instead, we just have to try alternatives and find one that strikes a good balance between finding real bounds-errors and avoiding false positives.

2.5.6 Other Operations

Machine instructions that do not produce or copy values, such as jumps, do not require any instrumentation.

2.6 Comments

The technique is very heavyweight, since it relies on (a) instrumenting every instruction in the program that moves an existing value, or produces a new value, and (b) shadowing every word in registers and memory with a shadow value.

3. PRACTICE

This section describes our prototype implementation of our technique.

3.1 Implementation Framework

As mentioned in Section 2.6, the technique is very heavyweight. It was very much designed to be implemented using Valgrind [9], a system for building supervision tools on x86/Linux, which supports exactly this sort of pervasive instrumentation and value tracking. The tool, which we call Annelid, is written as a plug-in for Valgrind, implemented as about 3,500 lines of C (including blanks and comments).

The instrumentation is entirely dynamic. The Valgrind core gains control of the supervised program at start-up, and none of the supervised program's original code is run. Instead, Valgrind translates the original code on demand, one basic block at a time, into a RISC-like intermediate language called UCode, which is expressed in virtual registers. The UCode is passed to the plug-in, which adds the necessary instrumentation, and then passes it back to the Valgrind core. The core reallocates registers, converts the UCode back to x86 machine code, caches it for future reuse, and runs it. Dynamically loaded libraries are handled exactly like any other code. For more details about how Valgrind works, please see [9].

3.2 Metadata Representation

The four types of metadata are represented as follows.

- A segment-type X is represented by a dynamically allocated *segment structure* containing its base address, size, a pointer to an "execution context" (a stack-trace from the time the segment is allocated, which is updated again when the segment is freed), and a tag indicating which part of memory the segment is in (heap, stack, or static) and its status (in-use or freed). Each structure is 16 bytes. Execution contexts are stored separately, in such a way that no single context is stored more than once, because repeated contexts are very common. Each execution context contains n pointers, where n is the stack-trace depth chosen by the user (default depth is four).
- A non-pointer-type n is represented by a small constant `NONPTR`.
- An unknown-type u is represented by a small constant `UNKNOWN`.
- A pointer-type $p(X)$ is represented by a pointer to the segment structure representing the segment-type X. Pointer-types can be easily distinguished from non-pointer-types and unknown-types because the segment structures never have addresses so small as `NONPTR` and `UNKNOWN`.

Each register and word of memory is shadowed,[5] and holds `NONPTR`, `UNKNOWN`, or a pointer to a segment structure. Memory shadowing is done in 64KB chunks. A shadow memory table stores 65,536 pointers—one for every 64KB chunk of memory. Initially every pointer in the table points to a distinguished shadow chunk (every word of which is set to `NONPTR`) which non-buggy programs will never read from. Each store is accompanied by a shadow store; when a chunk is written to for the first time, a new shadow chunk is allocated for it, and the appropriate pointer in the shadow memory table is updated. Shadow memory thus slightly more than doubles memory usage.

Sub-word writes to registers and memory will destroy any pointer information in those words; the type for that word becomes either n or u, depending on a range test. Thus, any byte-by-byte copying of pointers to or from memory will cause that information to be lost. (Fortunately, `glibc`'s implementation of `memcpy()` does word-by-word copying.)

Similarly, metadata is not stored across word boundaries in memory. With respect to metadata, an unaligned word-sized write is handled as two sub-word writes; therefore any pointer information will be lost, and the two aligned memory words partially written to will be set to `NONPTR` or `UNKNOWN`, depending on range tests. Fortunately, compilers are loathe to do unaligned writes, so this does not come up very often.

3.3 Segment Management

3.3.1 Storing Segments

Segment structures are stored in a skip list [12], which gives amortised $\log n$ insertion, lookup, and deletion, and is much simpler to implement than a balanced binary tree.

3.3.2 Freeing Segments

When a memory block, such as a heap block, is freed, the segment structure for that block is retained, but marked as being freed. In this way, we can detect any accesses to freed blocks via dangling pointers.

However, we need to eventually free segment structures representing freed segments, otherwise our tool will have a space leak. There are two ways to do this. The first is to use garbage collection to determine which segment structures are still live. Unfortunately, the rootset is extremely large, consisting of all the shadow registers and all of shadow memory.[6] Therefore, collection pauses could be significant.

The second possibility is to use segment recycling. We would start by allocating each new segment structure as necessary, and then placing it in a queue when its segment is freed. Once the queue reaches a certain size (say 1000 segments), we can start recycling the oldest structure segments. If the queue size drops to the threshold value, we would re-start allocating new segments until it grew bigger again.

Thus we would maintain a window of preserved freed-segments. It is possible that a segment structure could be recycled, and then memory could be accessed through a danging pointer that points to the freed segment. In this case, the error message produced will mention the wrong segment. Or, if we are exceedingly unlucky, the pointer will be within the range of the new segment and the error will be missed. The chance of this last event happening can be reduced by ensuring old freed segments are only recycled into new segments with non-overlapping ranges.

In practice, since accesses through dangling pointers are not that common, with a suitably large threshold on the

[5] Shadow type information for words in memory are stored in an equally-sized piece of shadow memory. Similarly, each register has a corresponding shadow register.

[6] This is a simplified garbage collection, as the traversals will only be one level deep, since segment structures cannot cannot point to other segment structures.

freed-queue this should happen extremely rarely. Also, recycled segments could be marked, and error messages arising from them could include a disclaimer that there is a small chance that the range given is wrong.

Alternatively, since the $p(X)$ representation is a pointer to a segment structure, which is aligned, there are two bits available in the pointer which could be used to store a small generation number. If the generation number in the pointer doesn't match the generation number in the segment structure itself, a warning can be issued.

One other characteristic of recycling is worth mentioning. If the program being checked leaks heap blocks, the corresponding segments will also be leaked and lost by our tool. This would not happen with garbage collection.

So the trade-off between the two approaches is basically that garbage collection could introduce pauses, whereas recycling has a small chance of causing incorrect error messages. Currently our prototype uses recycling, which was easier to implement.

3.4 Static Segments

Section 2.4.2 described how being too aggressive in identifying static array pointers can lead to false positives, e.g. when dealing with array accesses like `a[i-1]`. In our implementation, by default we use aggressive static pointer identification, but a command-line option can be used to revert to conservative identification.

3.5 Stack Segments

Section 2.4.3 discussed how stack segments could be handled by our technique. There would be no problem if we knew exactly when stack frames were built and torn down; more precisely, if we knew when functions were entered and exited. Unfortunately, in practice this is quite difficult. This is because functions are not always entered using the `call` instruction, and they are not always exited using the `ret` instruction; some programs do unusual things with jumps to achieve the same effects. However, unusually enough, tail-recursion will not really cause problems, as long as the function returns normally once it completes. The tail-calls will not be identified, and it will be as if every recursive function invocation is part of a single function invocation.

It might not be a problem if some entry/exit pairs were not detected; then multiple stack frames could be treated by our tool as a single frame. This could cause some errors to be missed, but the basic technique would still work. However, even detecting matching function entries with exits is difficult. The main reason is the use of `longjmp()`, which allows a program to effectively exit any number of function calls with a single jump. The obvious way to store stack segments is in a stack data structure. With `longjmp()`, it becomes necessary to sometimes pop (and mark as freed) multiple segments from this segment stack. If calls to `longjmp()` could be reliably spotted, this would not be a problem. However, GCC has a built-in non-function version of `longjmp()` that cannot be reliably spotted. Since we cannot even tell when a `longjmp()` has occurred, we do not know when to pop multiple frames. If we miss the destruction of any stack frames, we will fail to mark their segment-types as freed, and so our tool will not spot any erroneous accesses to them via dangling pointers.

Stack-switching also causes big problems. If a program being checked switches stacks, our tool should switch segment stacks accordingly. But detecting stack switches by only looking at the dynamic instruction stream is difficult, since it is often hard to distinguish a stack switch from a large stack allocation.

We have tried various heuristics in an attempt to overcome these problems. For example, after a `longjmp()` occurs, the stack pointer will have passed several old frames in a single step. This evidence can be used to determine that these frames are now dead. However, we have not managed to find any heuristics robust enough to deal with all the difficulties. As a result, our tool currently does not track stack segments at all. This is a shame, and we hope to resolve these problems in the future.

3.6 Range Tests

As mentioned in Section 2.5, we can convert many u result types to n if they are definitely not pointers. In our implementation, this test succeeds if a result value is less than `0x01000000`, or greater than `0xff000000`.

This is slightly risky, as a (strange) program could use `mmap()` to map a file or create a memory segment below `0x01000000`. Since Valgrind has complete control over memory allocation, we could ensure this never happens. Alternatively, we could track the lowest and highest addressable addresses, and declare any value outside this range as a non-pointer (with a suitable safety margin). In this case, for most programs on typical x86/Linux systems, non-pointers would be values below `0x8048000` (which is where the executable is loaded) and above `0xc0000000` (which is reserved by the kernel).

3.7 Pointer Differences

Section 2.5.4 suggested handling pointer differences between different segments precisely by generalising the n type to a $n(X, Y)$ type. We attempted this approach, but it was a pain. It made the addition and subtraction operations more complex; also, pointer differences are often scaled, so $n(X, Y)$ types would have to be propagated on multiplication, division, and shifting. Finally, having to free $n(X, Y)$ structures complicated segment structure storage.

A much easier solution was to introduce a new type b (short for "bottom"). Any value of type b is not checked when used as a pointer for a load or store, and the result of any type operations involving b as an operand is b. Values of type b are never changed to type n via a range test.

This is a blunt solution, but one that is not needed very often—recall that intra-segment pointer differences, which are much more common, are handled accurately.

3.8 System Calls

Valgrind does not trace into the OS kernel. However, since system calls are well defined, it knows which arguments are pointers, and what memory ranges will be read and/or written by each system call. Valgrind tools thus know which memory ranges will be written and/or read. Our prototype does normal range checks on each of these, and so can find bounds errors in system call arguments.

Most system calls return an integer error code or zero; for these we set the return type to n. Some system calls can produce pointers on success, notably `mmap()` and `mremap()`. Our current approach is to give these results the value u. We originally tried tracking segments returned by `mmap()` like other segments, but abandoned this because they are

```c
#include <stdlib.h>

int static_array[10];

int main(void)
{
   int  i;
   int* heap_array = malloc(10 * sizeof(int));

   for (i = 0; i <= 10; i++) {
      heap_array  [i] = 0;    // overrun: i==10
      static_array[i] = 0;    // overrun: i==10
   }
   free(heap_array);
   heap_array[0] = 0;         // block is freed
}
```

Figure 3: Sample Program: `bad.c`

difficult to deal with, since they are often resized, and can be truncated or split by other calls to `mmap()` that overlap the range. We decided this would not be a great loss, as programs tend to use `mmap()` in straightforward ways, and we suspect overruns of mapped segments are extremely rare.

3.9 Custom Allocators

Our prototype handles the standard allocators called via `malloc()`, `new`, `new[]`, `free()`, `delete`, `delete[]`. Custom allocators can be handled with a small amount of effort, by inserting *client requests* into the program being checked. These are macros that pass information to our tool about the size and location of allocated and deallocated blocks.

3.10 Leniency

Some common programming practices cause bounds to be exceeded. Most notably, `glibc` has heavily optimised versions of functions like `memcpy()`, which read arrays one word at a time. On 32-bit x86 machines, these functions can read up to three bytes past the end of an array. In practice, this does not cause problems. Therefore, by default we allow aligned, word-sized reads to exceed bounds by up to three bytes, although there is a command-line option to turn on stricter checking that flags these as errors.

3.11 Examples

This section shows some example errors given by our prototype. Figure 3 shows a short C program, `bad.c`. This contrived program shows three common errors: two array overruns, and an access to a freed heap block.

Figure 4 shows the output produced by our prototype. The error messages are sent to standard error by default, but can be redirected to any other file descriptor, file, or socket.

Each line is prefixed with the running program's process ID. Each error report consists of a description of the error, the location of the error, a description of the segment(s) involved, and the location where the segment was allocated or freed (whichever happened most recently). The functions `malloc()` and `free()` are identified as being in the file `vg_replace_malloc.c` because that is the file that contains our tool's implementations of these functions, which override the standard ones.

The program was compiled with `-g` to include debugging information. If it had not been, the code locations would have been less precise, identifying only the code addresses and file names, not actual line numbers. Also, the second error involving the static array would not have been found, as we discussed in Section 2.4.2.

3.12 Performance

This section presents some basic performance figures for our prototype. All experiments were performed on an 1400 MHz AMD Athlon with 1GB of RAM, running Red Hat Linux 9.0, kernel version 2.4.19. The test programs are a subset of the SPEC2000 suite. All were tested with the "test" (smallest) inputs.

Table 1 shows the performance of our prototype. Column 1 gives the benchmark name, column 2 gives its normal running time in seconds, and column 3 gives the slow-down factor. Programs above the line are integer programs, those below are floating point programs.

Program	Time (s)	Slow-down
bzip2	10.9	32.2
crafty	3.5	67.0
gap	1.0	39.5
gcc	1.5	48.0
gzip	1.9	40.9
mcf	0.4	18.1
parser	3.7	31.8
twolf	0.3	35.4
vortex	6.6	80.7
ammp	19.3	29.0
art	28.5	14.7
equake	2.2	35.5
mesa	2.5	52.5
median		35.5

Table 1: Slow-down factor of prototype

As mentioned, this technique is heavyweight. Therefore, the overhead is high and programs run much slower than normal. However, the slow-down experienced is not dissimilar to that with all thorough memory checking tools that instrument a program at link-time or run-time. Also, we have not yet tried optimising our prototype very much.

4. SHORTCOMINGS

Like all error-checking techniques, ours is far from perfect. How well it does depends on the circumstances. Happily, it exhibits "graceful degradation"; as the situation becomes less favourable, more and more $p(X)$ metavalues will be lost and seen instead as u. Thus it will detect fewer errors, but will not give more false positives.

4.1 Optimal Case

In the best case, the program will have all debug information and symbol tables present. In that case, even if implemented optimally, our technique will have problems in the following cases.

- Certain operations, such as swapping pointers with the bitwise-xor trick, cause $p(X)$ metavalues to be "downgraded" to u; erroneous accesses using those pointers will not be caught. One could imagine beefing up the

```
==16884== Invalid write of size 4
==16884==    at 0x8048398: main (bad.c:11)
==16884==  Address 0x40D1C040 is 0 bytes after the expected range,
==16884==   the 40-byte heap block allocated
==16884==    at 0x400216E7: malloc (vg_replace_malloc.c:161)
==16884==    by 0x8048375: main (bad.c:8)
==16884==
==16884== Invalid write of size 4
==16884==    at 0x80483A2: main (bad.c:12)
==16884==  Address 0x80495E8 is 0 bytes after the expected range,
==16884==   a 40-byte static array in the main executable
==16884==
==16884== Invalid write of size 4
==16884==    at 0x80483C5: main (bad.c:15)
==16884==  Address 0x40D1C018 is 0 bytes inside the once-expected range,
==16884==   the 40-byte heap block freed
==16884==    at 0x40021CE9: free (vg_replace_malloc.c:187)
==16884==    by 0x80483BE: main (bad.c:14)
```

Figure 4: Sample Output

type system to handle such cases, but the cost/benefit ratio would be very high.

- Directly out-of-bounds accesses to static and stack data objects (e.g. accessing `a[10]` in an array of ten elements) cannot be detected if they happen to fall within a nearby data object. Also, pointers not to the start of arrays must either be ignored, missing potential errors, or handled, causing potential false positives.

4.2 Implementation

Our implementation suffers from a few more shortcomings, mostly because the optimal case is too complex to implement.

- Pointer-types are lost if pointers are written unaligned to memory.
- Likewise, pointer-types are lost if pointers are copied byte-by-byte between registers and/or memory words. (Fortunately, as mentioned in Section 3.2, `glibc`'s implementation of `memcpy()` does word-by-word copying. If it didn't, we could just override it with our own that did not use byte-by-byte copying.)
- The use of b for inter-segment pointer differences will cause some errors to be missed.
- The implementation uses debugging information about types in only simple ways. For example, it does not try at all to break up heap blocks into sub-segments. Also, the only static data objects it constructs segment structures for are arrays (not for structs or basic data types).
- Stack segments are currently not handled at all, although we hope to rectify this soon.

4.3 No Debugging Information

If debugging information is not present, no static checking can be performed, as we cannot recognise pointers to static data objects.[7] Also, if we were checking stack frames, we would have to fall back to using unlimited-size segments, as discussed in Section 2.4.3.

[7] We could check the bounds of entire static data segments, since that information is known without debugging information.

4.4 No Symbols

If a program has had its symbol tables stripped, error checking might degrade further. This is because, if our implementation did stack checking, it would rely on symbols being present for detecting function entry and exit points.[8]

4.5 Avoiding Shortcomings

A lot of the shortcomings arise because the information available in a program binary at run-time is less than that present in the original program. Debugging information and symbol tables help represent some of this information, but there are still some things that our tool would like to know about.

The obvious way to improve the situation is to involve the compiler producing the programs; a lot of checking can be done purely dynamically, but some things clearly require static help.

First, a compiler could promise, in certain circumstances, to avoid generating code that causes problems for our tool. For example, it could ensure that all pointer reads and writes are aligned; it could ensure where possible that array accesses are done via the array's base pointer, rather than pre-computing offsets.

Second, a compiler could embed extra information into programs for our tool to use. In our implementation, it could use client requests (mentioned in Section 3.9 for handling custom allocators) to do this. This information might be simple, for example, indicating that a particular memory access is intended to be to a particular segment; or signalling when a `longjmp()` occurs. Or it could be more complex, for example, indicating that all memory accesses within a particular function or module need not be checked—this might be suitable if our tool was used in conjunction with a static

[8] The other obvious alternative—detecting entry from the dynamic instruction stream—is in practice extremely difficult. Some compilers can produce "debugging grade" code which includes hooks that tell tools when functions are entered and exited. This would make things much simpler, but no Linux compilers we know of provide this.

bounds checker (see Section 5.2 for a description of some).

Third, users could embed information themselves manually, although this option is only really practical for rare events, such as stack switches.

Finally, some bounds errors that our tool cannot find should arguably be found by a compiler. In particular, directly out-of-bounds accesses to static and stack arrays (e.g. accessing element `a[10]` of a ten-element array) could easily be found by the compiler.

Our technique has some very nice characteristics, but it clearly also has some significant shortcomings. A hybrid static/dynamic techniques may be more effective; this is an area worthy of future work.

5. RELATED WORK

This section describes several tools that find bounds errors for C and C++ programs, and compares them to our technique. No single technique is best; each has its strengths and weaknesses, and they complement each other.

5.1 Redzones

The most common kinds of bounds-checking tools dynamically check accesses to objects on the heap. This approach is common because heap bounds-checking is easy to do. The simplest approach is to replace the standard versions of `malloc()`, `new`, and `new[]` to produce heap blocks with a few bytes of padding at their ends (*redzones*). These redzones are filled with a distinctive values, and should never be accessed by a program. When the heap block is freed with `free()`, `delete` or `delete[]`, the redzones are checked, and if they have been written to, a warning is issued. The documentation for mpatrol [13] lists many tools that use this technique.

This technique is very simple, but it has many shortcomings.

1. It only detects small overruns/underruns, within the redzones—larger overruns or completely wild accesses could access the middle of another heap block, or non-heap memory.

2. It only detect writes that exceed bounds, not reads.

3. It only reports errors when a heap block is freed, which causes two problems: first, not all heap blocks will necessarily be freed, and second, this gives no information about where the overrun/underrun occurred. Alternatively, calls to a heap-checking function can be inserted, but that requires source code modification, will cause pauses while the entire heap is checked, and still does not give precise information about when an error occurs.

4. Accesses to freed heap blocks via dangling pointers are not detected, unless they happen to hit another block's redzone (even then, identifying the problem will be difficult).

5. It does not work with heap blocks allocated with custom allocators (although the technique can be built into custom allocators).

6. It only works with heap blocks—stack and static blocks are pre-allocated by the compiler, and so redzones cannot (without great difficulty) be used for them.

This technique has too many problems to be considered further. All these problems are avoided by techniques that track pointer bounds, such as ours.

Some of these shortcomings are overcome by Electric Fence [11], another `malloc()` replacement that uses entire virtual pages as redzones. These pages are marked as inaccessible, so that any overruns/underruns cause the program to abort immediately, whereupon the offending instruction can be found using a debugger. This avoids problems 2 and 3 above, and mitigates problem 1 (because the redzones are so big). However, it increases virtual memory usage massively, making it impractical for use with large programs.

A better approach is used by the Valgrind tool Memcheck [9], and Purify [5]. They too replace `malloc()` *et al* with versions that produce redzones, but they also maintain addressability metadata about each byte of memory, and check this metadata before all loads and stores. Because the redzones are marked as inaccessible, all heap overruns/underruns within the redzones are spotted immediately, avoiding problems 2 and 3 above. If the freeing of heap blocks is delayed, this can mitigate problem 4. These tools also provide hooks that a custom allocator can use to tell them when new memory is allocated, alleviating problem 5.

Purify is also capable of inserting redzones around static variables in a pre-link step, in certain circumstances, as explained in the Purify manual:

> Purify inserts guard zones into the data section only if all data references are to known data variables. If Purify finds a data reference that is relative to the start of the data section as opposed to a known data variable, Purify is unable to determine which variable the reference involves. In this case, Purify inserts guard zones at the beginning and end of the data section only, not between data variables.

Similarly, the Solaris implementation of Purify can also insert redzones at the base of each new stack frame, and so detect overruns into the parent frame. We do not know the details of how Purify does this, but we suspect that the way the SPARC stack is handled makes it much easier to do than on x86.

Redzones and addressability tracking works very well, which accounts for the widespread use of Memcheck and Purify. However, the remaining shortcomings—1 and particularly 6 (even with Purify's partial solution)—are important enough that tools tracking pointer bounds are worth having.

5.2 Fat Pointers

A different technique is to augment each normal pointer in a program with bounds metadata—typically the minimum and maximum address it can be used to access—giving a *fat pointer*. All accesses through fat pointers are checked, which gives very thorough checking, and avoids all the problems of the tools described in Section 5.1. But there are other significant disadvantages.

In earlier implementations, every pointer was replaced with a struct containing the pointer, plus the bounds metadata (e.g. [1]). This approach has several major problems.

1. It requires compiler support, or a pre-compilation transformation step.

2. All code must be recompiled to use fat pointers, including libraries. In practice, this can be an enormous hassle.

   Alternatively, parts of the program can be left unrecompiled, so long as interface code is produced that converts fat pointers to normal pointers and vice versa when moving between the two kinds of code. Producing this code requires a lot of work, as there are many libraries used by normal programs. If this work is done, two kinds of errors can still be missed. First, pointers produced by the library code may lack the bounds metadata and thus not be checked when they are used in the "fat" code. Second, library code will not check the bounds data of fat pointers when performing accesses.

3. Changing the size of a fundamental data type will break any code that relies on the size of pointers, for example, code that casts pointers to integers or vice versa, or C code that does not have accurate function prototypes.

4. Support may be required in not only the compiler, but also the linker (some pointer bounds cannot be known by the compiler), and possibly debuggers (if the fat pointers are to be treated transparently).

Jones and Kelly describe a better implementation in [6]. Each pointer's metadata is stored separately from the pointer itself, so it preserves backward compatibility with existing programs, avoiding problem 3, and reducing problem 4.[9] Patil and Fischer [10] also store metadata separately, in order to perform the checking operations on a second processor. We believe that if library code was handled better, this technique would be much more widely used.

Both these approaches require compiler support, and the checking is only done within modules that have been recompiled by the compiler, and on pointers that were created within these recompiled modules.

Our technique has the same basic idea of tracking a pointer's bounds, but the implementation is entirely different, as it works on already-compiled code, and naturally covers the entire program, including libraries. It also works with programs written in any language; this is useful for systems written in a mix of C or C++ and another language. Our technique is less accurate, however, and the overhead is much greater.

5.3 Static Checking

CCured [8] is a tool for making C programs type-safe, implemented as a source code transformer. It does a sophisticated static analysis of C source code, then adds bounds metadata, and inserts dynamic bounds-checks, for any pointers for which it cannot prove correctness statically. It suffers from problems 1–3 described in Section 5.2. These additional checks slow performance; published figures range from 10–150%. Also, on larger programs, one "has to hold CCured's hand a bit" for it to work; getting such programs to work can take several days' work [4]. Again, non-coverage of library code is a problem.

Compuware's BoundsChecker Professional tool [3] inserts memory checks into source code at compile-time. It seems to be similar to CCured, but without the clever static analysis to avoid unnecessary checks, and so can suffer much larger slow-downs.

[9]In practice, very small numbers of source code changes are needed.

By comparison, our technique is entirely dynamic, and does not require any recompilation or source code modification. It does find fewer errors, though.

5.4 Runtime Type Checking

We know of two systems that perform run-time type checking. Hobbes [2] maintains run-time type metadata about every value in a program. It gives warnings on run-time type violations, e.g. if two pointers are added. It can also detect bounds errors if they lead to run-time type errors. Implemented via an x86-binary interpreter, its slow-down factor was in the range 50–175. RTC [7] is similar, but only works for C programs as it uses a source-to-source transformation to add the checks. Slow-downs when using it are in the range 6–130. These tools use heap block padding to find some overrun errors, but as we saw in Section 5.1, our technique is more precise for finding bounds errors as it associates an explicit range with each pointer.

6. FUTURE WORK AND CONCLUSION

We have described a technique for dynamically checking pointer usage, which finds many but not all bounds errors. It effectively converts program pointers into fat pointers, without requiring programs to be modified, recompiled, or relinked, and works with programs written in any language. It also checks library code just as well as non-library code, without any extra effort. We have a prototype implementation of the technique which performs many of the possible checks.

Further improvements to the prototype would be desirable, particular to make it faster, and to include bounds checking on the stack. In the long-term, we hope hybrid static/dynamic techniques may be able to produce new, better bounds-checking tools that overcome the weaknesses of our technique, and other existing ones.

We have also found that there is a tendency for a large number of a program's values to decay to type u, even in fairly simple programs. A clear direction for future work is to identify why type information is being lost, and how to avoid it.

7. ACKNOWLEDGMENTS

Many thanks to: Julian Seward, for creating Valgrind, and for many useful discussions, and to Alan Mycroft and the anonymous reviewers for their comments about this paper. The first author gratefully acknowledges the financial support of Trinity College, Cambridge.

8. REFERENCES

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94)*, pages 290–301, Orlando, Florida, USA, June 1994.

[2] M. Burrows, S. N. Freund, and J. L. Wiener. Run-time type checking for binary programs. In *Proceedings of CC 2003*, pages 90–105, Warsaw, Poland, Apr. 2003.

[3] Compuware Corporation. Boundschecker. `http://www.compuware.com/products/devpartner/bounds.htm`.

[4] George Necula *et al.* *CCured Documentation*, Sept. 2003. `http://manju.cs.berkeley.edu/ccured/`.

[5] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, San Francisco, California, USA, Jan. 1992.

[6] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automated Debugging (AADEBUG'97)*, pages 13–26, Linköping, Sweden, May 1997.

[7] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. In *Proceedings of FASE 2001*, Genoa, Italy, Apr. 2001.

[8] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of POPL 2002*, pages 128–139, London, United Kingdom, Jan. 2002.

[9] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proceedings of RV'03*, Boulder, Colorado, USA, July 2003.

[10] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software—Practice and Experience*, 27(1):87–110, Jan. 1997.

[11] B. Perens. Electric fence. `ftp://ftp.perens.com/pub/ElectricFence/`.

[12] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[13] G. S. Roy. *mpatrol: A Library for controlling and tracing dynamic memory allocations*, Jan. 2002. `http://www.cbmamiga.demon.co.uk/mpatrol/`.