Heap Bounds Protection with Low Fat Pointers *

Gregory J. Duck Roland H. C. Yap
Department of Computer Science
National University of Singapore, Singapore
{gregory,ryap}@comp.nus.edu.sg

Abstract

Heap buffer overflow (underflow) errors are a common source of security vulnerabilities. One prevention mechanism is to add object bounds meta-information and to instrument the program with explicit bounds checks for all memory access. The so-called "fat pointers" approach is one method for maintaining and propagating the meta-information where native machine pointers are replaced with "fat" objects that explicitly store object bounds. Another approach is "low fat pointers", which encodes meta-information within a native pointer itself, eliminating space overheads and also code compatibility issues. This paper presents a new low-fat pointer encoding that is fully compatible with existing libraries (e.g. precompiled libraries unaware of the encoding) and standard hardware (e.g. x86_64). We show that our approach has very low memory overhead, and competitive with existing state-of-the-art bounds instrumentation solutions.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; K.6.5 [*Management of Computing and Information Systems*]: Security and Protection

Keywords memory safety, buffer overflows, low-fat pointers

1. Introduction

Low level languages such as C and C++ allow directly manipulation of pointers and thus arbitrary access to memory. While this is needed in such languages for efficiency and to provide direct access to memory, it allows for memory bugs. Despite at least three decades of research with numerous proposed solutions [15, 16], memory errors continue to plague us today and it is listed as among the top software vulnerabilities, i.e. in the CWE/SANS top 25 errors list. For instance, searching in the NIST National Vulnerability Database for "memory corruption" turns up 1,454 reported vulnerabilities between 2012 to 2015 alone. The survey by Szekeres et al. [15] suggests that it continues to be an arms race between offense and defense.

* This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

CC'16, March 17–18, 2016, Barcelona, Spain
ACM. 978-1-4503-4241-4/16/03...
http://dx.doi.org/10.1145/2892208.2892212

Memory safety can be divided into spatial safety and temporal safety. In this paper, we focus on spatial safety. Bounds checking based approaches which check whether a pointer access lies within its object bounds, thus, enforcing spatial memory safety is known to be among the best techniques for ensuring memory integrity [15], yet, adoption tends to be poor. The survey by Szekeres et al. [15] identifies three main factors which influence adoption: (i) accuracy (false positives (wrongly reported error) and false negatives (missed error)) ; (ii) cost (performance and space overheads); and (iii) compatibility which is further divided into:

Source compatibility: The source code does not need modification to take advantage of protection.

Binary compatibility: Ability to link with unmodified libraries. This also includes backward compatibility to support legacy libraries.

Modularity support: Support for separate compilation. In particular, dynamic link libraries (DLLs) are required in modern operating systems.

While many buffer overflow defenses give good accuracy for spatial memory safety, the cost can be significant and the amount of compatibility varies with the method. The conclusion from [15] is that performance and compatibility are the main barriers to adoption for memory error protection techniques.

In this paper, we focus on buffer overflow defenses for dynamic memory in the heap for C programs. Our objectives are to maximize compatibility (binary and modularity)[1] and provide good performance (low performance and memory overheads) making the technique more viable for practical adoption. One approach to get good accuracy is to modify the representation of pointers to associate the pointer with information on the bounds of the underlying object. This is also called "fat pointers" since the pointer can be thought of as being enlarged with meta-information. However, fat pointers can interfere with compatibility since the representation of pointers and storage is affected. Furthermore, accessing the meta-information can be costly and the space overheads can be significant. An approach to address the compatibility and space overhead drawbacks is the idea of low-fat pointers [7] which encodes the meta-information into the pointer itself. Low-fat pointers were originally proposed for hardware implementation. In this paper, we propose a new low-fat pointer scheme suitable for software implementation on 64-bit architectures which provides bounds protection for heap objects aimed at getting good performance, maximizing compatibility and good accuracy. The performance overhead is low and space overhead is very low – on the SPEC 2006 benchmarks. Our low-fat pointer representation is transparent, thus, the pointer value can be directly used for dereferencing. This provides both efficiency and compatibility (binary and modularity).

[1] Most buffer overflow techniques already achieve source compatibility.

In summary our main contributions are:

Low-Fat Pointers: We present a novel low-fat pointer encoding that can support arbitrarily sized objects on the heap and is optimized for standard 64-bit hardware (`x86_64`). Our low-fat pointers can be efficiently mapped to bounds meta-information (size + base) of the underlying object, and this information is used to instrument programs with explicit bounds checking. We propose several optimizations to our basic approach.

Compatibility: Low-fat pointers are also native machine pointers and thus can be passed to, and used by, software components that are not low-fat aware (e.g. pre-compiled external libraries). The reverse direction is also supported, i.e. non-low-fat pointers are compatible with code which has been instrumented without performance penalty. The types, data layout, and binary function interface of instrumented programs *does not change*, meaning that our approach achieves full binary compatibility and modularity support.

Compiler Tool: We have implemented a low-fat pointer runtime system and a bounds instrumentation compiler pass using `clang`/LLVM [8] for the GNU/Linux + `x86_64` platform.

Low Cost: We demonstrate that our approach has low performance overheads for a bounds instrumentation schema (56% for reads+writes, 13% for writes-only) on the SPEC 2006 benchmark suite [14]. This compares very favorably with existing solutions such as AddressSanitizer [13]. We also show that our approach has low memory overheads.

2. Background

Detection or prevention of spatial memory errors is widely recognized as a critical problem for low-level programming languages such as `C` and `C++`. As such there is a significant body of existing literature on proposed solutions, including [1–4, 6, 10, 11, 13, 17] amongst others. In this section we provide a brief overview of the different approaches. We also introduce our own approach, as well as explain how it compares to other proposed solutions.

Most bounds instrumentation systems follow the same basic schema. Given a pointer p and an object O with base address $base$ and size (in bytes) $size$, then p is *Out-Of-Bounds* (OOB) with respect to O if

$$(p < base) \;||\; (p > base+size-\texttt{sizeof}(*p)) \qquad \text{(IsOOB)}$$

By defining a macro $\texttt{isOOB}(p, base, size) = \text{(IsOOB)}$ we can *instrument* a program (e.g. via automatic compiler transformation) by inserting *bounds checks* before every memory read or write operation, i.e.:

```
if (isOOB(p, base, size))
    error();
v = *p;  or  *p = v;
```

Here the instrumentation code has been highlighted, and the role of function `error()` is to report the bounds error and abort the program. The $(size, base)$ pair is the *bounds meta-information* associated to object O. The main difference between different bounds instrumentation systems lies in the technical and implementation details, i.e. how to determine which object O is associated to pointer `p`? Or how to determine the bounds meta-information (i.e. $size$ and $base$) of object O? Several solutions have been proposed, and are traditionally classified into two main approaches: *object-based* that associate bounds meta-information with *objects*, and *pointer-based* approaches, that associate meta-information with each *pointer*. Each approach has (dis)advantages as we summarize below.

Object-Based Approaches. Object-based approaches work by associating meta-information with each *object*. Examples of systems that use the object-based approach include AddressSanitizer [13], BaggyBounds [1], MudFlap [4], etc.

A prominent implementation of an object-based approach is AddressSanitizer [13]. AddressSanitizer works by maintaining a separate *shadow memory* that tracks the state of each object (allocated, free, etc.). Furthermore, each object is buffered by a 128-byte *poisoned red zone* for the purpose of detecting OOB-errors. AddressSanitizer achieves good software compatibility, but the shadow memory and red-zones result in significant memory overheads. Another system, BaggyBounds checking [1] stores size information is a separate *bounds table*, which is more space efficient, but restricts object sizes to powers-of-two (2^B for some B).

One disadvantage of object-based approaches is *completeness* – i.e. are all OOB-errors guaranteed to be detected by the system? For example, a bounds error of $*(p+129)$ may bypass an AddressSanitizer red-zone and not be detected, although such errors are less common in practice.

Pointer-Based Approaches. Pointer-based approaches work by associating meta-information to pointers rather than objects.

The most direct pointer-based approach is "fat-pointers" which encapsulates the meta-information directly inside the pointer itself. The basic idea is to transform the program (e.g. via a compiler pass) replacing native pointers with a "fat" object that stores bounds meta-information explicitly, e.g.

```
struct { void *ptr; void *base; size_t size; }
```

The fat-pointer objects are passed-by-copy in place of native pointers, at the cost of time and space overheads. Fat-pointers are used by Safe-C [2], CCured [11] and Cyclone [6] amongst others. Since fat-pointers replace native pointers, this generally breaks both binary compatibility and modularity support.

An alternative to fat-pointers is to pass meta-information via separate channels. This approach taken by systems such as PAriCheck [17] and SoftBound [9, 10]. For example, SoftBound propagates meta-information using a *shadow stack* (for function calls) [9] and *shadow memory space* (for memory), thereby improving compatibility compared to fat pointers. Although SoftBound provides for compatibility, libraries that manipulate pointers and expose such pointers to the program need either recompilation or wrappers to update the meta-information [9].

Pointer-based approaches tend to be complete, e.g. the OOB-error $q{=}(p{+}129)$ will be detected, since the new pointer q inherits the same bounds meta-information from p.

2.1 The Low-Fat Pointer Approach

In this paper, we focus on the idea of so-called "low-fat" pointers [7]. Low-fat pointers are conceptually similar to fat pointers in that bounds meta-information is attached to each pointer used by the program. However unlike fat pointers, which use "fat" objects, low-fat pointers aim to encode the meta-information into a native pointer directly – thus "trimming the fat".

Example 1 (A Hypothetical Low-fat Pointer Encoding)**.** As an example of a simple low-fat pointer encoding, consider the following type declaration:

```
union { void *ptr;
        struct {uintptr_t size:10;  // MSB
                uintptr_t unused:54; } meta;} p;
```

Here the `size` bit-field stores the allocation size which can be retrieved via `p.meta.size`. This representation is low-fat since $\texttt{sizeof}(\texttt{p}) = \texttt{sizeof}(\texttt{void}*)$. Furthermore the bit-wise representation must be the same as a native pointer (i.e. `p.ptr` can be dereferenced directly), even accounting for the value stored in

`size`. Encoding of object *base* can be achieved by ensuring that all objects are *size-aligned*, thus $base = \texttt{p} - (\texttt{p} \,\%\, \texttt{p.meta.size})$. The enforcement all these conditions can still be challenging to implement in practice needing to fit operating systems constraints while providing binary compatibility. □

In this paper, we present a more sophisticated encoding (Section 3) that stores the size information indirectly; rather than explicitly as Example 1.

The fact that low-fat pointers are regular native pointers has several advantages. Accessing a low-fat pointer is no different than accessing any other native pointer. Unlike traditional fat pointers, low-fat pointers can be passed to and from instrumentation-aware code without any special conversion or marshalling. Furthermore low-fat pointers do not change the data layout of objects used by the program. In contrast, other bounds checking solutions, such as [10, 13], can only preserve data layout by storing meta information in a separate *shadow heap* or *shadow stack*; consuming additional memory resources. Since our low-fat neither requires marshalling or shadow space, our low-fat pointers achieve full compatibility (binary and modular) with low additional memory overheads. These are the key advantages of our low-fat approach.

Low-fat pointers do have some trade-offs. Firstly, low-fat pointers are only feasible on architectures with sufficient pointer bitwidth. In practice, this means 64-bit (including 48-bit effective) architectures such as the `x86_64`. We note that 64-bit systems are nowadays quite common, so this is not a significant limitation. Another trade-off is that our low-fat pointer representation encodes *allocation bounds* rather than precise object bounds (a bounds overflow violating object but not allocation bounds may not be detected). As future work, it may be possible to adapt the method from [3] to extend the protection to object rather than bounds (at the cost of decreased performance). One final trade-off is the type of pointers that can be protected. The low-fat pointer approach implies control over the pointer value for each object of interest. This is possible for heap allocation, where the memory allocator has full control, but difficult for stack allocations – where pointer values are constrained to be within the current stack frame. In this paper, we focus on heap protection since: (a) this is the most natural fit for the low-fat pointer approach, and (b) heap bounds errors still represent significant security problem in their own right; hence a specialized solution is justifiable. Extending our results to stack pointers may be possible by (1) object migration (stack to heap), and/or (2) the virtual configuration of [1] (Section 5.1). We leave this as future work.

Using low-fat pointers for bounds checking requires some careful design. For example, given $q{=}(p{+}129)$, then p's bounds meta-information should be used when checking a dereference of pointer q. To achieve this we propose a form of meta-information propagation similar to that of SoftBound [10] but optimized to preserve binary compatibility.

Finally, as far as we know, no existing method can maximize all objectives (performance, compatibility, accuracy) and each method has its own trade-offs. Our low-fat pointer implementation aims to provide full binary compatibility, complete heap allocation bounds protection, low space overhead and good performance. We contrast with the other trade-offs of some well known methods which have good performance: (i) AddressSanitizer: good binary compatibility, high space overhead, incomplete bounds protection for heap and non-heap objects; (ii) BaggyBounds: good binary compatibility, moderate space overhead, allocation bounds protection for stack and heap objects; and (iii) SoftBound: partial binary compatibility, high space overhead, complete bounds protection for heap and non-heap objects.

2.2 Related Work (Low-Fat Pointers)

In addition to the bounds instrumentation systems outlined above, there are some existing proposals that specifically relate to low-fat pointers, including: a hardware-based solution of [7] and the "size tagging" variant of BaggyBounds checking introduced in [1] (Section 5.1).[2]

The proposal of [7] encodes bounds meta-information as various bit-fields inside a regular machine pointer, in a similar way to Example 1. For example, in the *aligned encoding* variant, pointer bits 57–63 encode a B bit-field, and the (allocation) size of the corresponding object is defined as 2^B. The authors also present a more complicated encoding for handling non-power-of-two sized objects. Their focus is the design of specialized hardware to efficiently handle checks on pointer operations, such as access and pointer arithmetic, etc. In contrast, we present a pure software-based encoding that is optimized for standard generic hardware such as the `x86_64`.

The "size tagging" variant of BaggyBounds checking [1] can also be viewed as a low-fat pointer encoding. Here the size is encoded as a bit-field B (a.k.a. the "tag") in a similar fashion to the proposal from [7]. One disadvantage of BaggyBounds is the power-of-two sized object restriction, which can lead to increased memory overheads. Our proposal is more flexible and can be configured to allocate arbitrary sized objects. Furthermore our encoding is *extensible* in that it can support other kinds of meta-information that are useful for optimization.

3. Runtime Environment

Our low-fat pointer implementation is based on a *low-fat memory allocator* that uses a special *memory layout* that exploits the large virtual address space to encode low-fat pointers. The low-fat allocator is designed to be a drop-in replacement for the standard `libc` `malloc` and related family of functions (`realloc`, `memalign`, etc.). In this section we give an overview of the allocator, as well as how the bounds meta-information can be reconstructed from low-fat pointers to heap objects.

3.1 Low-Fat Pointer Memory Allocator

The *low-fat pointer memory allocator* (or *low-fat memory allocator* for short) relies on a special region-based *memory layout* that is summarized in Figure 1. Here, the program's virtual address space is evenly divided into several different large *regions* of equal size. Our description assumes a *region size* of 4GB, although other sizes may be reasonable. The regions are organized according to the following layout:

- Special region #0 contains the standard program *text* and *data* segments;
- Regions #1–#M, for some pre-defined maximum M, contain sub-heaps for the low-fat memory allocator; and
- Special region #stack contains the program *stack*.

The special region #0 spans the first $4GB$ of memory (addresses `0x0`–`0xffffffff`), and contains all of the standard Linux process memory segments, including the `text`, `data`, and `bss` segments, in their usual position. As with normal processes, the size of the data segment is extensible using the `sbrk()` and `mmap()` system calls, and is therefore compatible with `libc`'s `malloc`.[3] Furthermore, a special #stack region spans addresses `0x7fff00000000`–`0x800000000000` and contains the program's stack. Importantly, the size, position, and layout of the program's standard memory

[2] The work of [1] does not use the terminology "low-fat pointer".

[3] The low-fat memory allocator can co-exist with the standard `libc` `malloc` allocator.

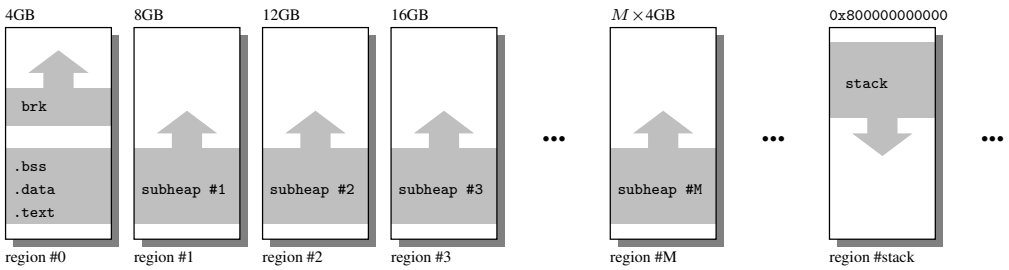

Figure 1. Program memory layout illustration.

segments (text/data/stack) is the same as any other standard `x86_64` Linux process, minimizing potential compatibility problems.

Regions #1–#M are reserved for the special low-fat memory allocator. Unlike traditional implementations of `malloc` that use a single contiguous heap (`sbrk`'ed memory), the low-fat memory allocator instead allocates from M *sub-heaps*, where each region contains exactly one corresponding sub-heap. Each region has a fixed size of 4GB, and a base address that is a multiple of 4GB. For example, region #1 has the base address of `0x100000000` (4GB), region #2 has `0x200000000` (2×4GB), etc. Each sub-heap is confined to the corresponding region, meaning that a maximum of 4GB can be allocated out from any given region.

In order to implement low-fat pointers, the low-fat allocator enforces the following critical requirements:

- (*Region*) all objects of a given size must be allocated from the *same memory region*; and
- (*Alignment*) all objects must be allocation *size-aligned*.

The aim of the (Region) and (Alignment) requirements is to establish a direct relationship between the value of a pointer and the corresponding object size and base pointer of the object. For example, if p points to an object contained within region #i, then by the (Region) requirement, the object size for p is the corresponding object size associated to region #i. Likewise, the (Alignment) requirement ensures that the base pointer of the object for p is p rounded down to the nearest size-aligned boundary, similar to Example 1. A more detailed discussion of the size and base calculation will be presented in Section 3.2.

Regions #1–#M occupy virtual address space that would otherwise be unused by most standard programs, again minimizing compatibility problems. Even accounting for large M values, significant portions of the virtual address space remain empty between regions $\#M$ and $\#stack$. This empty address space can be used by the program for future mappings (e.g. `mmap` or loading of dynamic libraries).

3.1.1 Size Configuration

The low-fat allocator restricts the possible *allocation sizes* to a fixed finite set.[4] Arbitrary object sizes are rounded up to the nearest supported allocation size if necessary. We define the *size configuration* Sizes to be the sequence (indexed from 1) of all possible allocation sizes supported by the low-fat allocator:

$$\mathsf{Sizes} = \langle size_1, size_2, \cdots, size_M \rangle$$

Here the sequence is arranged such that region #i corresponds to $size_i$. We also define $M = |\mathsf{Sizes}|$ to be the *maximum region number*.

The exact size configuration (Sizes) is flexible, and can be left to the implementation.[5] Our prototype implementation uses the following size configuration by default:

$$\mathsf{Sizes} = \langle 16B, 32B, 48B, 64B, .., 8KB, 16KB, 32KB, 64KB, .., 1GB \rangle \quad \text{(SIZES)}$$

such that $size_1 = 16B$ is the smallest allocation size, and $size_M = 1GB$ is the maximum supported allocation size. The maximum region number is therefore $M = 530$ heap regions (excluding the special non-heap region #0). This configuration is designed to minimize memory overheads as discussed below.

3.1.2 Low-Level Considerations

Memory regions #1–#M are created during program initialization using the `mmap` system call with the `MAP_NORESERVE` flag. This prevents the operating system from creating competing mappings for the same region of virtual address space. Furthermore, the `MAP_NORESERVE` flag instructs the operating system not to reserve physical memory resources (i.e. RAM or swap) for any of the regions when the mapping is created. Instead, physical memory is reserved "on demand" when the corresponding virtual addresses are allocated and actually used by the program.

The exact memory allocation algorithm used within each region is left as an implementation detail. Our prototype implementation uses a simple combination of a *free-list* (for previously deallocated objects) or allocates from *fresh space* within the region should the free-list be empty. Initially the fresh space is inaccessible with memory protection `PROT_NONE`. The allocator ensures that fresh objects are made accessible before being returned to the program. Other memory allocation algorithms are compatible possible, provided they respect the size and alignment requirements. Our implementation is also thread-safe and applicable to parallel programs.

The size and number of memory regions is also configurable. Our prototype implementation assumes a region size of $4GB$, ensuring that the memory region layout is compatible with the standard program text, data and stack segments, as illustrated in Figure 1. Other region sizes may also be reasonable. Technically, regions need not be contiguous nor the same size, however, these assumptions help simplify the implementation.

The size configuration given by (SIZES) is designed to minimize memory overheads introduced by allocation size rounding. For small ($< 8KB$) allocations, the allocation size is rounded up to the nearest multiple of 16 bytes. For large ($\geq 8KB$) allocations, the allocation size is rounded up to the nearest power-of-two size, up to a maximum allocation size of $1GB$. Small objects waste a maximum of $15B$ of memory after rounding, and large objects waste a maximum of $4KB - 1B$ bytes after rounding, assuming a page size of $4KB$. Note that for large multi-paged objects, the physical memory overhead is always bounded by the page size, as any unused pages at the end of the object will remain unmapped and therefore do not consume physical memory resources. Furthermore, the allocator ensures that unused pages are inaccessible (i.e. have memory protection `PROT_NONE`). The unused pages serve as "guards"

[4] This is also common practice for memory allocation algorithms.

[5] If the sizes needed by a program are known, they can be used to configure Sizes.

and provide additional protection against overflows in addition to bounds instrumentation.

The allocator does not store any explicit metadata information between objects, i.e. objects are "tightly packed" with no gaps. This makes it trivial to satisfy the size-alignment requirement, by ensuring that the boundaries between objects are always size-aligned.

3.1.3 Incompatible Allocations

One issue is how to deal with allocation requests that cannot be serviced by the low-fat memory allocator. For example, very large ($> 1GB$) allocations, or if the corresponding memory region has run out of free space (e.g. $\geq 4GB$ of allocations of the same size). No matter the configuration of the low-fat allocator (i.e. region size, allocation sizes, etc.), such limits can never be removed completely, and can always be reached by a sufficiently demanding program.

As a fall-back, our implementation reverts to using the `libc malloc` should the low-fat allocator be unable to service a given allocation request. This ensures that all allocation requests will always be serviced. As will be discussed later, the `libc malloc` is compatible with (and can co-exist with) the low-fat memory allocator. Pointers returned by `libc malloc` will not be "low-fat", and therefore will not afforded the same protections against object bounds overflows.

3.2 Reconstructing Meta-information

The aim of a low-fat pointer representation is make it possible to efficiently calculate bounds meta-information (size and base) based on the pointer's value. In this section we describe the implementation in more detail.

3.2.1 Calculating Index

A useful operation is to map pointers to the corresponding *memory region index* for which the object is contained.

Assuming a region size of $2^{32} = 4GB$ and the memory layout of Section 3.1, then a pointer p can be mapped to the memory region index using the $index$ operation defined as follows:

$$index(p) = p \texttt{ >> } 32 \qquad \text{(INDEX)}$$

The $index$ operation compiles down into a single shift right (logical) instruction.

3.2.2 Calculating Size

The $size$ operation is defined as $size(p) = \mathsf{Sizes}[index(p)]$, where Sizes is the size configuration defined in Section 3.1.

To efficiently implement the $size$ operation, the runtime system stores the Sizes sequence in a *metadata table* `TABLE` as follows:

$$\texttt{TABLE}[i].\texttt{size} = \mathsf{Sizes}[i] \qquad \text{for all } i \in 1..M$$

$$size(p) = \texttt{TABLE}[index(p)].\texttt{size} \qquad \text{(SIZE)}$$

Here M is the max region size, and `TABLE` is currently undefined for other regions.

In our implementation, the metadata table is stored in an `mmap`'ed file that is loaded to a fixed location during program initialization. The metadata table is both constant (read-only with `PROT_READ`) and small (2 pages for size-only meta-information). Later, we extend `TABLE` to include additional information.

For example, suppose that $p =$ `0x180000000` is a pointer to a object allocated by the low-fat allocator. We see that $index(p) = 1$. Assuming the size configuration Sizes defined in Section 3.1, then

$$size(p) = \texttt{TABLE}[1].\texttt{size} = \mathsf{Sizes}[1] = 16B$$

Therefore p points to an object of size 16 bytes.

The $size$ operation compiles down into a single memory lookup operation (from `TABLE`). Note that the `TABLE` itself is small (8-byte size per $4GB$ region) and read-only. These factors are favorable regarding CPU cache behavior.

Our approach is also different from Example 1 and related work [1, 7] that encode the size directly into the bit representation of the pointer. Instead, our approach is *indirect*, i.e. the size is stored in a separate metadata table `TABLE`. Our approach is both flexible (arbitrary Sizes configurations are supported) and extensible (we insert additional meta-information later).

3.2.3 Calculating Base

Pointers can point to the *interior* rather than the *base* of objects, e.g. if p=`malloc`(100) then q=$(p$+$50)$ is an interior pointer. For bounds checking, we need to map (possibly) interior pointers to the corresponding *base pointer* for the object.

The low-fat allocator ensures that all heap objects are allocation-size aligned. We can exploit this alignment to map an interior pointer p to base pointers as follows:

$$base(p) = (p \texttt{ / } size(p)) \texttt{ * } size(p) \qquad \text{(BASE)}$$

Here (`/`) and (`*`) are standard `C` 64-bit unsigned integer division and multiplication respectively. The combination of integer division and multiplication has the affect of rounding p down to the nearest $size(p)$-aligned boundary, which is the base address.

For example, suppose that $p =$ `0x180000005` points to a heap allocated object. Then $index(p) = 1$ and $size(p) = 16$. We can calculate the base pointer as follows:

$$base(p) = (p \texttt{ / } 16) \texttt{ * } 16 = \texttt{0x18000000}$$

The $base$ operation compiles down to a two-instruction sequence: a division followed by a multiplication instruction. An alternative definition for (BASE) that uses 64-bit integer modulus (`%`) instead of division is $base(p) = p$ `-` $(p$ `%` $size(p))$. Both division and modulus are relatively expensive CPU operations. In Section 5 we shall present an optimized definition.

Using alignment to calculate the base address is used by other systems, such as BaggyBounds [1]. However, unlike [1], allocation sizes need not be powers-of-two.

3.3 Handling Non-fat Pointers

So far our method for reconstructing bounds meta-information assumes the pointer is low-fat, i.e. compliant with the memory layout and alignment requirements specified in Section 3.1. However, not all pointers used by a given program will be low-fat – for example, pointers to global variables, the stack, or were created via calls to `mmap` or `sbrk` memory; or from calls to the standard `libc`'s `malloc`; or even the `NULL` pointer. Furthermore, whether a pointer is low-fat or not can be *ambiguous* at compile-time. For example, a pointer argument `p` to `f` may or may not be low-fat:

```
extern int f(int *p) { return *p; }
```

For brevity, we refer to non-low-fat pointers as *non-fat pointers*.

Our aim is to extend the meta-information reconstruction to handle non-fat pointers. Unlike low-fat pointers, there is no requirement for non-fat pointers to be bounds checked. That said, for efficiency reasons it is important not to treat non-fat pointers as a special case, i.e. by introducing separate paths for low-fat vs. non-fat in the instrumented code. Instead, our approach is to extend the definitions of $size$ and $base$ to return "wide bounds" for all non-fat pointers, thereby, ensuring any subsequent bounds check will never fail and result in a spurious error.

First, we extend the definition of $size$ by adding extra entries to the metadata table `TABLE` (Section 3.2) for regions corresponding

```
p = NULL;
p_size = UINT64_MAX;
p_base = 0;
```

(a) p is the NULL pointer

```
p = malloc(n);
p_size = n;
p_base = p;
```

(b) p allocated from heap

```
p = q + c;
p_size = q_size;
p_base = q_base;
```

(c) p created by pointer arithmetic

```
p = (type *)q;
p_size = q_size;
p_base = q_base;
```

(d) p cast from a pointer

```
p = (type *)i;
p_size = size(p);
p_base = base(p);
```

(e) p cast from an integer

```
p = *q;
p_size = size(p);
p_base = base(p);
```

(f) p loaded from memory

```
p = f(···);
p_size = size(p);
p_base = base(p);
```

(g) p returned by a function

```
basic_block: p = Φ q1, ···, qn;
p_size = Φ q_size1, ···, q_sizen;
p_base = Φ q_base1, ···, q_basen;
```

(h) p created by a Φ-node (PHI-node)

```
type f(type *p) { ···
p_size = size(p);
p_base = base(p);
```

(i) p is a function argument

```
type p[];
p_size = UINT64_MAX;
p_base = 0;
```

(j) p is a global or stack pointer

Figure 2. Schema for size and base meta-information propagation.

to non-fat pointers as follows:

$$\texttt{TABLE}[i].\texttt{size} = \texttt{UINT64\_MAX} \qquad \text{for all } i \notin 1..M$$
$$size(p) = \texttt{TABLE}[index(p)].\texttt{size}$$

That is, if p is non-fat, then $index(p) \notin 1..M$, i.e. not in the index range $1..M$ is covered by the low-fat-memory allocator. For such pointers p, the $size(p)$ operation will therefore evaluate to UINT64_MAX, i.e. the maximum possible unsigned 64-bit number.

Conveniently, we see that

$$base(p) = (p \texttt{ / UINT64\_MAX}) \texttt{ * UINT64\_MAX} = 0$$

for all p using Definition (BASE). Thus, given an non-fat pointer p, the base address will always evaluate to zero (the null address).

Under the extended definitions, we have $size(p) = \texttt{UINT64\_MAX}$ and $base(p) = 0$ for all non-fat pointers p. For example, consider the pointer $p = \texttt{0x400000}$ (the start of the text segment in a standard Linux process). We see that $index(p) = 0 \notin 1..M$ and therefore p is non-fat. Furthermore:

$$size(p) = \texttt{TABLE}[0].\texttt{size} = \texttt{UINT64\_MAX}$$
$$base(p) = (p \texttt{ / UINT64\_MAX}) \texttt{ * UINT64\_MAX} = 0$$

Accessing the pointer p can never generate a spurious error since:

$$p \geq 0 \wedge p \leq 0 + \texttt{UINT64\_MAX}$$

Extending the metadata table to cover the entire 48-bit x86_64 address space requires 65536 entries. Most entries correspond to non-fat pointers and contain the same data. Our implementation maps these duplicate entries to the same physical page and thus minimizes physical memory usage (3 pages for size-only meta-information).

3.3.1 Binary Compatibility

The extended definitions make bounds checking compatible with *all* program pointers, whether they be low-fat or non-fat. However, only low-fat pointers will be explicitly protected against bounds errors (consistent with the aims of this paper).

The extended definitions are essential for binary compatibility, as not all software components (e.g. external libraries) are low-fat-pointer aware. Since low-fat pointers are regular machine pointers, they are automatically compatible with the majority of external software components. In the reverse direction, non-fat pointers created by external software components are compatible with the bounds checking instrumentation in the sense they can never generate spurious errors. Importantly, this achieves full binary compatibility.[6]

Another consequence is that the low-fat-allocator is compatible with, and can co-exist with, the standard libc malloc. Software libraries that use malloc can continue to do so. Furthermore the low-fat allocator can use malloc as a "fall back" should any allocation request be unserviceable, e.g. if a memory region becomes full.

4. Compiling Bounds Checking

Utilizing the runtime environment of Section 3, a compiler transformation modifies the program in two ways: (1) *metadata propagation* aims to propagate bounds meta-information (size and base) for each pointer, and (2) *bounds instrumentation* inserts protection against out-of-bounds memory access.

4.1 Meta-Information Propagation

In order to check the bounds for pointer p dereference, the *size* and *base* meta-information of the object pointed to by p must be known. We apply a simple program transformation that associates each pointer variable p (in the program code) with an explicit p_size (of type size_t) and p_base (of type void *) pair of variables that store the *size* and *base* respectively. This meta-information propagates according to the schema shown in Figure 2. Here, each schema rule comprises a program statement or declaration that creates a new pointer p. The program is then modified to include

[6] Binary compatibility is critical when the binary component itself is using and creating pointers which are passed also to instrumented code.

the corresponding statements (highlighted) for meta-information propagation using variables `p_size` and `p_base`.

The schema considers three main special cases:

1. p is unambiguously *not a heap pointer*;
2. p is *created by a heap allocation*; or
3. p is *derived from another pointer* q.

For everything else the schema resorts to calculating the bounds meta-information with explicit calls to $size$ and $base$.

Figure 2(a)(j) handle the special case where `p` is unambiguously not a heap pointer, i.e. when `p` is the `NULL` pointer, *stack allocated*, or a pointer to a *global variable*. Such pointers inherit the "wide bounds" of `p_base` $=$ `NULL` $= 0$ and `p_size` $=$ `UINT64_MAX`.

Figure 2(b) handles the special case when `p` is the result of a heap allocation (assuming the low-fat allocator). Here `p_base` $=$ `p` and `p_size` $= n$.

Figure 2(c)(d) handles the special case where `p` is derived from another pointer `q`, namely *pointer arithmetic* and *pointer casts*. In these cases the derived pointer `p` inherits the size and base meta information of the parent pointer `q`. Note that pointer arithmetic also includes the *address-of* (`&`) operator. For example `&p[n]` is equivalent to `p` $+$ `n`, and `&p->val` is equivalent to `p` $+$ *offset* where field `val` is *offset* bytes from the start of the object. Both *array access* and *field access* are treated as pointer arithmetic followed by a memory access operation, e.g. (`p[3] = 4`) is equivalent to (`*(p+3) = 4`).

Figure 2(h) handles the case where `p` is constructed by a Φ-node joining two or more basic blocks. Φ-nodes are implicit instructions inserted by the compiler in order to transform the program into *Single Static Assignment* (SSA) form. In this case, `p_size` and `p_base` are Φ-nodes of the corresponding bounds meta-information for each origin pointer q_i.

Figure 2(e)(f)(g)(i) handles all other cases. Namely, where `p` is cast from an integer, loaded from memory, returned from a function call or passed into the current function as a parameter. Here the schema calculates the size and base information explicitly using the $size(p)$ and $base(p)$ operations defined in Section 3.2.

Metadata propagation does not need to be inserted for all pointers. The propagation can be omitted for pointers that are never dereferenced or are otherwise never used in a bounds check. Furthermore, the propagation code can be optimized by using *variable substitution* rather than inserting explicit assignments for rules Figure 2(b)(c)(d). For example, for rule Figure 2(c), we can avoid adding superfluous by aliasing `p_base` $=$ `q_base` and `p_size` $=$ `q_size`. For our examples, we use explicit assignments for readability.

Example 2 (Metadata Propagation)**.** Consider the following function that calculates the length of a linked-list.

```
int list_length(Node *list) {
    int len = 0;
    while (list != NULL)
        { len++; list = list->next; }
    return len; }
```

The function takes as a parameter `list` with a pointer type (`Node *`). In the body of the loop, the program implicitly creates the pointer `&list->next`, i.e. the body of the loop is equivalent to:

```
len++; Node **next = &list->next; list = *next;
```

The instrumented version is shown in Figure 3. Since `list` is a parameter, the size and base are calculated explicitly using rule Figure 2(i) (lines 4–5). Likewise, after a new value for `list` is read from memory in the loop body (line 14), the size and base are recalculated using rule Figure 2(f) (lines 15–16). Pointer `next` inherits the size and base of `list` using rule Figure 2(c) (lines 10–11). For readability our example does not use SSA form, so rule Figure 2(h) for pointer `list` is left implicit. Lines 12–13 contain the bounds check which is explained in the next section. □

```
1  int list_length(Node *list)
2  {
3      int len = 0;
4      void *list_base = base(list);
5      size_t list_size = size(list);
6      while (list != NULL)
7      {
8          len++;
9          Node **next = &list->next;
10         void *next_base = list_base;
11         size_t next_size = list_size;
12         if (isOOB(next, next_base, next_size))
13             error();
14         list = *next;
15         list_base = base(list);
16         list_size = size(list);
17     }
18     return len;
19 }
```

Figure 3. Fully instrumented version of the list length function from Example 2.

Our meta-information propagation schema is related to that used by SoftBound [10] with some differences. For example, we do not add additional parameters to functions, e.g. both versions of `list_length` have exactly the same interface, i.e. the same number of parameters, same return value, and use the same types. This is also important for binary and modular compatibility. Another difference is that, for efficiency reasons, we only aim to preserve allocation bounds and not sub-object bounds. The latter is a possible extension for future work.

4.2 Bounds Checking

In existing literature, there are two main approaches to bounds check instrumentation:

1. instrument *memory load/store* operations (e.g. [1, 10, 13]); or
2. instrument *pointer arithmetic* operations (e.g. PAriCheck [17]).

The former prevents OOB-pointer dereference, and the latter prevents the creation of OOB-pointers altogether. Low-fat pointers can be used for either approach. However, for completeness[7] (i.e. guarantee all heap bounds errors will be detected), a combination of both approaches is required. For example, consider the following simple example:

```
q = p + 100; x = list_length(q);
```

Here `q` (of type `Node *`) is an OOB-pointer. Since the instrumented version of `list_length` (Figure 3) maintains the same binary interface as the non-instrumented version (Example 2), the bounds meta-information for the parameter `list` is reconstructed on function entry (lines 4–5) using the $size$ and $base$ operations from Section 3.2. However, if `list_length` is passed an already OOB-pointer, as is the case with `q` above, then lines 4–5 construct the meta-information for whatever object `q` happens to point to, rather than the original object pointed to by `p`. This means that the bounds error (line 14) will not be detected, i.e. the bounds instrumentation is *incomplete*. Here we say that the OOB-pointer `q` has been passed into a different *context*, i.e. from the callee to the function `list_length`.

For completeness, we propose a combination of both load/store and pointer-arithmetic bounds instrumentation described in this

[7] Completeness is with respect to heap pointers only

section. The purpose of the latter is to prevent OOB-pointers from being passed between contexts as with the example above.

4.2.1 Load/Store Instrumentation

The load/store instrumentation follows the basic schema outlined in Section 2. For this we insert a bounds check before each load/store operation as follows:

```
if (isOOB(p, p_base, p_size))
    error();
v = *p; or *p = v;
```

An example is shown in Figure 3, lines 12–13, for the dereferenced pointer `next`.

4.2.2 Pointer Arithmetic Instrumentation

To address incompleteness we additionally instrument all *pointer arithmetic* `p=q+k` in cases where the resulting pointer `p` can be passed to different context. These cases include:
(a) *casting* `p` to an integer (`i = (int)p`);
(b) *storing* `p` to memory (`*r = p`);
(c) *passing* `p` to a function (`f(p)`); and
(d) *returning* `p` from a function (`return p`);
Here cases (a)–(d) correspond to the schema rules in Figure 2(e)-(f)(g)(i) respectively. The bounds check is inserted in between the pointer arithmetic and the operation (a)–(d):

```
p = q + k;
if (isOOB(p, q_base, q_size))
    error();
i=(int)p; or *r=p; or f(p); or return p;
```

Note that the instrumentation only affects pointers that are the result of pointer arithmetic. Many common cases are not affected. For example, consider the code fragment:

```
q = p->next; x = list_length(q);
```

Here `q` is the result of a memory load rather than arithmetic, and therefore will not be instrumented before the function call. Note that `q` cannot be an OOB-pointer under rule (b) above, i.e. `q` would be bounds checked before it was stored in field `p->next`.

Some incompleteness may remain if the compiler cannot identify all forms of pointer arithmetic, e.g. if pointers are cast to and from integers by the programmer. This problem affects all bounds instrumentation systems and is not specific to our approach.

The pointer-arithmetic instrumentation is optional and can be omitted if completeness is not desired. Not all bounds instrumentation systems are complete, e.g. object-based approaches such as AddressSanitizer [13]. Pointer-arithmetic based instrumentation may lead to spurious bounds errors if the programmer intentionally creates OOB-pointers (although this technically violates the `C` standard). This is a known problem for pointer-arithmetic based instrumentation. However, unlike other systems, we only check the specific case where OOB-pointers are passed between contexts, rather than preventing OOB-pointers from being created altogether. The former is less common [1, 12], which helps software compatibility.

4.2.3 Standard Library Functions

In addition to instrumentation, we also replace the standard library calls `memcpy`, `memmove`, `memset` with efficient safe equivalents that perform bounds checking.

5. Optimizations

The basic low-fat pointer representation and bounds checking scheme is, thus far, not designed to be efficient. In this section, we shall present various optimizations designed to minimize runtime overheads to make the overall scheme practical. Some of the optimizations are specific to our approach, whereas others are more general and apply to other bounds checking systems (as will be noted).

5.1 Fast Division

To calculate the base pointer $base(p)$ for a given pointer p using Definition BASE requires a 64-bit division operation.[8] Division is a relatively expensive operation even on modern CPUs; a single `x86_64 div` instruction can have a latency of 32-123 cycles [5] compared to 1 cycle for `add` and 3–4 cycles for `mul`. Our experiments show that overhead introduced by division is significant.

A faster alternative is to replace expensive division (p `/` $size$) with a cheaper multiplication ($p{*}(1/size)$) by using *fixed-point arithmetic*. This approach is feasible size the set of allocation sizes (Sizes) is constant, and thus the set of *allocation size reciprocals* $\{1/size \mid size \in \mathsf{Sizes}\}$ can be pre-calculated and stored in a lookup table.

We formalize the implementation as follows: First the metadata table `TABLE` is extended to include an extra 64-bit `invSize` field that will store the pre-calculated reciprocal of the allocation size. This is defined as follows:

$$\begin{aligned} &invSizeInit(size) = \texttt{UINT64\_MAX} \,/\, size + epsilon \\ &\texttt{TABLE}[i].\texttt{invSize} = invSizeInit(\mathsf{Sizes}[i]) && \text{for all } i \in 1..M \\ &\texttt{TABLE}[i].\texttt{invSize} = 0 && \text{for all } i \notin 1..M \end{aligned}$$

Here, the $invSizeInit(size)$ function calculates an approximation of the fixed-point representation of $(1/size)$ assuming an 128-bit representation with a radix point of 64-bits. The $epsilon = 1$ is added for error control and will be explained below. For non-fat pointers, the corresponding value for `invSize` is set to zero.

Given a pointer p, the base pointer for p can now be calculated as $((p{*}(1/size)){*}size)$ using fixed-point arithmetic. Assuming our 128-bit fixed-point representation, the $base$ operation can be implemented as follows:

$$\begin{aligned} invSize(p) &= (\texttt{uint128\_t})\texttt{TABLE}[index(p)].\texttt{invSize} \\ base(p) &= (((\texttt{uint128\_t})p * invSize(p)) \texttt{ >> } 64) * size(p) \end{aligned}$$
(FASTBASE)

Note that the inner (`*`)-operation is a 128-bit multiplication. The purpose of the shift operation is to truncate the 128-bit fixed-point result back into a 64-bit integer representation of $(p{*}(1/size))$. This integer is then multiplied by the $size$ to derive the base pointer.

Note that the 128-bit multiplication and 64-bit shift can conveniently be implemented as a single `imul` instruction on the `x86_64` platform. Definition (FASTBASE) effectively replaces the 92-cycle division operation with a 4-cycle combined multiplication and memory load from `TABLE`.

Example 3 (Fast Base)**.** Consider the pointer $p =$ `0x380000045` which points to an object allocated from region #3. Assuming that region #3 is for objects of size 48 bytes, then

$$\begin{aligned} invSize(p) &= \texttt{UINT64\_MAX} \,/\, 48 + 1 = 384307168202282327 \\ base(p) &= ((\texttt{0x380000045} * 384307168202282327) \texttt{ >> } 64)) * 48 \\ &= (\texttt{577705354318230258054070283}5 \texttt{ >> } 64) * 48 \\ &= 313174700 * 48 = 15032385600 = \texttt{0x380000040} \end{aligned}$$

Therefore the base address of p is `0x380000040`, as expected. □

For non-fat pointers p, we see that $invSize(p) = 0$ and therefore $base(p) = 0$ as per the original definition.

[8] Alternatively a 64-bit *modulus* operation can be used. However, both division and modulus are implemented using the same `x86_64 div` instruction, thus have similar poor performance.

In addition to fixed-point arithmetic, we also experimented with an alternative encoding that uses *floating-point arithmetic*. For this, we change the type of the table's `invSize` field to be a floating point value and redefine:

$$\begin{aligned} \mathit{invSizeInit}(\mathit{size}) &= 1.0 \;/\; \mathit{size} \\ \mathit{invSize}(p) &= \texttt{TABLE}[\mathit{index}(p)].\texttt{invSize} \\ \mathit{base}(p) &= (\texttt{unint64\_t})(p * \mathit{invSize}(p)) * \mathit{size}(p) \end{aligned}$$

In our experience, this definition is inferior thanks to the extra conversion instructions required (i.e. pointer/integer to double and back again). This is not a problem for the fixed-point representation that exclusively uses integers.

5.1.1 Precision Errors

Fixed-point arithmetic can suffer from *precision errors*, meaning that (FASTBASE) can return the wrong result for some addresses. Such errors may lead to false negatives during bounds checking for large (>2MB) allocation sizes, which is clearly not acceptable. By using $\mathit{epsilon} = 1$, we can ensure that the precision errors only affect the *end* of objects. That is, given $p = \texttt{malloc}(\mathit{size})$, then Definition (FASTBASE) will return the correct result for addresses $p, .., p+\mathit{size}-\delta$ for some $\delta < \mathit{size}$.

Numerical analysis can be used to determine the maximum possible value for δ. However, we adopted a simpler approach: find the maximum δ using an exhaustive search over all possible heap addresses (i.e. within regions #1-#M). The exhaustive search yields the following information:

- $\delta = 0$ for $\mathit{size} \leq 2MB$; and
- $\delta \leq 4KB$ for $2MB < \mathit{size} \leq 1GB$ (max size)

To account for precision errors we therefore pad all large (>2MB) allocations by a value of $4KB$. This ensures that any address potentially affected by a precision error will reside within the padding and not within the object itself. For example, an allocation of size $4MB$ will be treated as an allocation size of $4MB + 4KB$, ensuring that all pointers within the range $p..p + 4MB$ (where p is the returned pointer) will not be affected by precision errors. This approach slightly increases the overall space overhead of the allocator by a maximum of $4KB/2MB = \sim 0.2\%$.

5.2 Power-of-Two Object Sizes

An alternative to the (FASTBASE) approach is to restrict object sizes to powers-of-two, namely by redefining:

$$\mathsf{Sizes} = \langle 16B, 32B, 64B, ..., 1GB \rangle$$

This allows for the *base* operation to be implemented as a simple bit-mask, eliminating the need for expensive division or fixed-point arithmetic. Assuming power-of-two object sizes and the standard region layout, we extend the metadata table `TABLE` to include a `mask` field and define:

$$\begin{aligned} \texttt{TABLE}[i].\texttt{mask} &= \texttt{UINT64\_MAX} \texttt{ << } (i+3) && \text{for all } i \in 1..M \\ \texttt{TABLE}[i].\texttt{mask} &= 0 && \text{for all } i \notin 1..M \\ \mathit{base}(p) &= p \texttt{ \& } \texttt{TABLE}[\mathit{index}(p)].\texttt{mask} && \text{(POW2BASE)} \end{aligned}$$

The metadata table accounts for non-low-fat pointers by setting the corresponding mask to zero.

Power-of-two sized objects represent a time versus space trade-off. Such objects are less granular leading to potentially higher space overheads for sub-page ($< 4KB$) sized objects. For multi-page objects, space wastage is less of a problem since unused pages will remain unmapped.

Definition (POW2BASE) can seemingly be improved further by calculating the bit-mask at runtime:

$$\mathit{base}(p) = p \texttt{ \& } (\texttt{UINT64\_MAX} \texttt{ << } \mathit{index}(p) + 3)$$

However, this new definition does not work for non-fat pointers.

Using power-of-two sized objects is a common optimization. It is used in systems such as BaggyBounds [1].

5.3 Other Optimizations

We now summarize some common optimizations for bounds checking instrumentation. These optimizations are not specific to our approach, and some have been implemented in other systems.

5.3.1 A Faster Check

An alternative definition of `isOOB` is:

$$\begin{gathered} \texttt{isOOB}(p, \mathit{pBase}, \mathit{pSize}) = \\ ((\texttt{uintptr\_t})p - (\texttt{uintptr\_t})\mathit{pBase} \geq \mathit{pSize}) \end{gathered}$$

This version uses *unsigned integer underflow* to simultaneously test the upper and lower object bounds using a single comparison. That is, for the case where $p < \mathit{pBase}$, the operation $p - \mathit{pBase}$ underflows, resulting in a large integer that is ($\geq \mathit{pSize}$). This is a common optimization (used by [1, 10, 13]).

5.3.2 Removing Redundant Checks

Some other common optimizations for bounds checkers include removing redundant bounds checks. For example, consider the statement (`*p = *p + 1`), which will be instrumented as follows:

```
if (isOOB(p, p_base, p_size)) error();
int tmp = *p;
if (isOOB(p, p_base, p_size)) error();
*p = tmp + 1;
```

The second bounds check is redundant and can be removed. Since the {`p`, `p_base`, `p_size`} values are unchanged, this optimization is automatic for modern compilers (e.g. `clang`). Similar optimizations exist in other systems, e.g. AddressSanitizer [13]. However, AddressSanitizer uses a different encoding so the optimization is not automatic.

5.3.3 Restricting Checks

Another common optimization is to restrict the bounds instrumentation to specific subsets of operations, such as *memory writes only*, at the expense of completeness. Such optimizations are implemented by most bounds checking systems.

6. Experiments

We have implemented a prototype low-fat pointer and bounds checking system (LowFatPtr) as two components: (1) a library implementing the runtime environment described in Section 3, and (2) an LLVM [8] plugin implementing the metadata and bounds check instrumentation described in Section 4. The system supports all of the optimizations described in Section 5. Each optimization can be enabled or disabled using a suitable compiler flag.

We have evaluated the performance of the system against the C/C++ subset of the SPEC 2006 benchmark suite [14]. We used a dual quad-core Intel Xeon E5540 (clocked at 2.53GHz) with 48GB of RAM as the test machine. We used the `clang-3.5` compiler, and all programs (both instrumented and uninstrumented) were compiled using the (`-O2`) compiler optimization level flag. For our experiments we test both timings and memory usage.

To evaluate the performance of system, we compare against non-instrumented binaries compiled using standard `libc malloc`. We also compare against binaries compiled with AddressSanitizer [13] enabled (by passing the `-fsanitize=address` flag to

	Orig	LowFatPtr + BoundsCheck				AddressSanitizer	
Bench.	base	base	$+fdiv$	$+pow2$	$+w.o.$	base	$+w.o.$
perlbench	**468**	1042	716	653	**532**	1572	1144
bzip2	**603**	1303	982	948	**704**	1049	726
gcc	**390**	1276	827	770	**663**	814	684
mcf	**339**	546	517	476	356	485	**349**
gobmk	**562**	888	701	667	**619**	1089	683
hmmer	**499**	1306	1178	1133	586	1139	**547**
sjeng	**604**	840	681	663	**639**	1292	757
libquantum	**484**	2058	756	659	520	663	**502**
h264ref	**724**	2575	1459	1312	**829**	1697	1030
omnetpp	**388**	903	537	529	**432**	*CE*	*CE*
astar	**520**	1432	824	790	**577**	858	597
xalancbmk	**304**	792	481	437	**270**	620	395
milc	**463**	889	709	685	**502**	735	574
namd	**477**	893	826	833	**517**	798	523
dealII	**372**	1399	796	716	**473**	757	532
soplex	**309**	865	505	464	**338**	495	370
povray	**263**	861	506	458	**312**	544	366
lbm	**342**	404	405	405	**361**	445	381
sphinx3	**640**	1642	1209	1141	699	1273	**697**
Total	**8751**	250%	167%	156%	**113%**	195%	130%

(a) Timings (s)

	Orig	LowFatPtr		ASAN
Bench.	base	base	$+pow2$	base
perlbench	679	633	732	2484
bzip2	871	868	868	946
gcc	906	894	892	3059
mcf	1717	1717	1717	1980
gobmk	29	29	29	466
hmmer	26	31	26	665
sjeng	180	180	180	230
libquantum	99	99	99	442
h264ref	66	66	72	448
omnetpp	173	166	221	*CE*
astar	335	348	569	1166
xalancbmk	429	496	618	1835
milc	696	704	704	1053
namd	47	49	49	149
dealII	812	839	1042	2257
soplex	442	614	614	1027
povray	5	6	6	424
lbm	419	420	420	514
sphinx3	46	46	46	616
Total	7977	103%	111%	253%

(b) Memory usage (MB)

Figure 4. Experimental results for the SPEC2006 benchmark suite.

`clang-3.5`). To help keep the comparison as fair as possible,[9] we configure AddressSanitizer to only protect against heap overflow errors by disabling stack instrumentation, global instrumentation, alloc/dealloc mismatch detection and leak detection. We choose to compare against AddressSanitizer for several reasons, namely: prominence (actively used in large projects such as Google Chrome [13] and Mozilla Firefox), stability, accessibility (already "built-in" to `clang` [8]), and works "out-of-the-box" on the SPEC 2006 benchmark suite.

6.1 Results

The experimental results are shown in Figures 4(a) and 4(b).

Our LowFatPtr implementation detected several 23 OOB violations in 4 of the 19 SPEC 2006 benchmark programs, including 3 functions from `perlbench`, 17 from `gcc`, 1 from `namd` and 2 from `soplex`. All of these violations were caused by the programmer intentionally creating OOB-pointers that are passed between contexts (technically undefined behavior under the `C` standard), and are detected by the *pointer arithmetic instrumentation* of Section 4.2. The OOB-pointers include: using $base-1$ as a sentinel (`perlbench`), accessing arrays via offsets (e.g. using $(a-k)[k]$ to access $a[0]$) (`gcc`), and copying objects via *pointer difference* (`soplex`). Overall we found that such problematic idioms are rare, with only a few functions affected. For the sake of testing, we disable instrumentation (via blacklisting) for these 23 functions for both LowFatPtr and AddressSanitizer.

Neither LowFatPtr nor AddressSanitizer detected a read/write outside of an heap object bounds. This result is in line with expectations – although some memory errors for SPEC 2006 are known [13], none correspond to spatial heap errors outside of allocation bounds.

For AddressSanitizer, the compilation of the `omnetpp` benchmark failed because of a known but as yet unfixed bug.[10] This is marked by *CE* (*Compiler Error*) in Figure 4. We exclude `omnetpp` when comparing results against AddressSanitizer.

6.1.1 Timings

The results (timings) are shown in Figure 4(a). Here, Orig is the uninstrumented program (using standard `libc malloc`), LowFatPtr+BoundsCheck is the program compiled with the low-fat allocator and with bounds check instrumentation, and is the program compiled using AddressSanitizer. The original timing and the best instrumented timing are highlighted in **bold**. For LowFatPtr, we enable various optimizations (where applicable), including:

- $+fdiv$ the *fast division* described in Section 5.1;
- $+pow2$ the *power-of-two sized objects* described in Section 5.2;
- $+w.o.$ enables *writes-only instrumentation* described in Section 5.3.

The optimizations, where applicable, are cumulative left-to-right. The *fast check* and *redundant check removal* (Section 5.3) are always enabled for all versions. Only the $+w.o.$ optimization is supported by AddressSanitizer.

The results highlight the importance of the $+fdiv$ optimization that replaces slow native division with a fast fix-point arithmetic alternative (Section 5.1). Without this optimization, our low-fat pointer representation would be too slow to be practical for many applications with 150% overall overhead. There is no disadvantage to the $+fdiv$ optimization and so it is enabled by default.

An alternative to $+fdiv$ is the $+pow2$ optimization that restricts object sizes to powers of two. The $+pow2$ optimization reduces the overall overhead by a further 11%. However, since the set of allocation sizes is less granular, the $+pow2$ has the potential to increase the overall memory usage of the program (see below). The *writes only* ($+w.o.$) optimization results in a significant reduction in overhead at the cost of making the bounds protection incomplete. With all optimizations enabled, the overall overhead is just 13%, which is low enough to be used in production code.

Our results compare favorably against AddressSanitizer – with an overall 95% slowdown for the base instrumentation (compared to 67% for $+fdiv$) and a 30% slowdown for the ($+w.o.$) optimization. For some benchmarks (e.g. `mcf`) AddressSanitizer is slightly better, whereas for others (e.g. `perlbench`, `sjeng`) AddressSani-

[9] By default AddressSanitizer protects against multiple types of memory error, including stack/global overflows, and some temporal memory errors such as double-free.

[10] LLVM Bug #19660.

tizer is significantly worse (up to two times slower). These results show that the low-fat pointer approach, with $+fdiv$, is viable and competitive with existing state-of-the-art bounds checking instrumentation systems.

6.1.2 Memory Usage

For these experiments, we measure the memory overheads of the LowFatPtr implementation. For this, we measure the *peak resident set size* (RSS), the same method as used in [13]. The rationale is measure memory actually accessed, rather than virtual address space which is reserved. We compare against the `stdlib` `malloc` and AddressSanitizer. We note that our test machine uses $48GB$ of RAM and therefore has low memory pressure for the SPEC 2006 benchmark suite.

The results (memory) are shown in Figure 4(b). Here we see that the average memory usage overhead of the base LowFatPtr implementation is minimal 3% compared to the significant 153% introduced by AddressSanitizer (ASAN). In the case of `gcc`, ASAN takes more than three times the memory of LowFatPtr. The low memory overhead is explained by several factors, namely:

- Bounds meta-information does not need to be explicitly stored (hence "low-fat" pointers);
- There are no "poisoned" red-zones appended to allocated objects. The exception is "guard pages" for large objects, however these do not consume physical memory resources; and
- The `TABLE` lookup-table is small (4 physical pages).[11]

The memory usage under the low-fat allocator is sometimes better than standard `malloc` (e.g. for `perlbench`, `omnetpp`, etc.) and sometimes worse (e.g. for `astar`, `soplex`, etc.). The reason is because each allocator has different types of memory overheads. For example, the low-fat allocator does not need to explicitly store the allocation size which saves space. On the other hand, standard `malloc` can pack different sized objects on to the same physical pages. As a result, the low-fat allocator tends to favor programs that allocate many same-sized objects, and may be worse otherwise. The "best" allocator is program/benchmark specific. One possible direction for future work is to use memory profiling to specialize a size configuration Sizes to a specific program, further reducing overheads.

In addition to the default size configuration, we additionally test the memory overheads of the $+pow2$ optimization. As expected, the less granular allocation sizes leads to an overall increase of the memory overheads, from 3% to 11%. Some benchmarks were affected more than others, depending on the allocation pattern of the tested program. The `astar` benchmark was worst affected, with a 64% increase in overall memory usage. The $+pow2$ optimization also uses a smaller size configuration Sizes, which can lead to reduced overheads on some benchmarks (`gcc`, `hmner`). Overall the $+pow2$ optimization is a classic time versus space trade-off, but may be worthwhile depending on the application.

Overall we see that the low-fat pointer approach has a minimal impact on memory usage, and is directly comparable to `stdlib` C's native `malloc`. In addition to saving memory resources, a low memory overhead can translate to faster execution times for systems with high memory pressure.

7. Conclusion

In this paper, we presented a novel low-fat pointer encoding for heap objects that is optimized for 64-bit architectures (`x86_64`). Such pointers can be efficiently mapped to the bounds meta-information of the corresponding object. We presented a compiler transform (implemented in LLVM) for bounds check instrumentation using our low-fat pointer approach. Experimental results show our scheme has good performance and low memory usage impact for the SPEC 2006 benchmark suite.

In addition to performance our low-fat pointer representation achieves full binary compatibility and modularity support. Our result is based on two main factors:

- low-fat pointers are regular pointers, and do not change the types (unlike "fat" pointers) nor the data layout used by the program; and
- the binary interface of functions does not change.

Furthermore, with careful design, non-fat pointers are compatible with the instrumented code albeit lacking bounds overflow protection. Binary compatibility is crucial in practice, where it is not uncommon for real-world code to rely on external dependencies (libraries) that are difficult or impossible (for closed-source) to recompile.

[11] Including the `invSize` (Section 5.1) or `mask` (Section 5.2) meta-information.

References

[1] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense Against Out-of-Bounds Errors. In *USENIX Security*. USENIX, 2009.

[2] T. Austin, S. Breach, and G. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Programming Language Design and Implementation*. ACM, 1994.

[3] B. Ding, Y. He, Y. Wu, A. Miller, and J. Criswell. Baggy Bounds with Accurate Checking. In *Software Reliability Engineering Workshops*, 2012.

[4] F. Eigler. Mudflap: Pointer Use Checking for C/C++. In *GCC Developerś Summit*, 2003.

[5] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2016.

[6] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *Annual Technical Conference*. USENIX, 2002.

[7] A. Kwon, U. Dhawan, J. Smith, T. Knight, and A. DeHon. Low-fat Pointers: Compact Encoding and Efficient Gate-level Implementation of Fat Pointers for Spatial Safety and Capability-based Security. In *Computer and Communications Security*. ACM, 2013.

[8] LLVM. `http://llvm.org`, 2016.

[9] S. Nagarakatte. *Practical Low-overhead Enforcement of Memory Safety for C Programs*. PhD thesis, 2012.

[10] S. Nagarakatte, Z. Santosh, M. Jianzhou, M. Milo, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Programming Language Design and Implementation*. ACM, 2009.

[11] G. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe Retrofitting of Legacy Software. *Transactions on Programming Languages and Systems*, 27(3), 2005.

[12] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Network and Distributed System Security*, 2004.

[13] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Annual Technical Conference*. USENIX, 2012.

[14] SPEC. `https://www.spec.org/cpu2006/`, 2016.

[15] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *Security and Privacy*, 2013.

[16] V. van der Veen, N. Dutt-Sharma, L. Cavallaro, and H. Bos. Memory Errors: The Past, the Present, and the Future. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2012.

[17] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Information, Computer and Communications Security*. ACM, 2010.