View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Royal Holloway - Pure

Stack Bounds Protection with Low Fat Pointers

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

Lorenzo Cavallaro‡
Information Security Group
Royal Holloway University of London
lorenzo.cavallaro@rhul.ac.uk

***Abstract*—Object bounds overflow errors are a common source of security vulnerabilities. In principle, bounds check instrumentation eliminates the problem, but this introduces high overheads and is further hampered by limited compatibility against un-instrumented code. On 64-bit systems, low-fat pointers are a recent scheme for implementing efficient and compatible bounds checking by transparently encoding meta information within the native pointer representation itself. However, low-fat pointers are traditionally used for heap objects only, where the allocator has sufficient control over object location necessary for the encoding. This is a problem for stack allocation, where there exist strong constraints regarding the location of stack objects that is apparently incompatible with the low-fat pointer approach. To address this problem, we present an extension of low-fat pointers to stack objects by using a collection of techniques, such as pointer mirroring and memory aliasing, thereby allowing stack objects to enjoy bounds error protection from instrumented code. Our extension is compatible with common special uses of the stack, such as `alloca`, `setjmp` and `longjmp`, exceptions, and multi-threading, which rely on direct manipulation of the stack pointer. Our experiments show that we successfully extend the advantages of the low-fat pointer encoding to stack objects. The end result is a competitive bounds checking instrumentation for the stack and heap with low memory and runtime overheads, and high compatibility with un-instrumented legacy code.**

I. Introduction

System code and applications with high-performance requirements are usually written in low-level languages such as `C` and `C++`. These programming languages do not provide any protection against memory errors (e.g., buffer overflows), and this is a well known source of security vulnerabilities and exploits. Although memory errors have been well researched with numerous proposed solutions [24], [26], the threat nevertheless persists. For example, a search for CVEs with "buffer overflow" on the NVD (National Vulnerability Database) returns at least 972 entries for the past three years alone. The 2014 Heartbleed [12] bug, which was perhaps one of the most serious and widespread vulnerabilities of recent times, was also a buffer overflow.

Given the well understood nature of buffer overflow and the numerous proposed solutions, it is reasonable to ask why protection mechanisms that prevent buffer overflow are not in widespread use. There are a number of barriers to adoption that have been identified [24], including:

- **Performance**: Does the solution slow the program down too much? Does the solution use too much memory?
- **Software Compatibility**: Does the solution work with existing code bases without additional modification?
- **Binary Compatibility**: Does the solution change the *Application Binary Interface* (ABI) meaning that binary modules (e.g., system or proprietary libraries) need to be re-compiled?

Low runtime performance overheads are important (and in general the lower the better), however the impact is application dependent. For example, the AddressSanitizer [22]-hardened Tor browser [25] is an example application where security is prioritized over performance. Memory overheads are also important, especially in server contexts where available system memory may be at a premium. Memory overheads can translate into time overheads under low memory conditions. Software compatibility means that programs which use a variety of compiler and language specific features continue to work, e.g., undefined behavior (out-of-bounds pointers, integer overflows, etc.), abnormal control-flow (`longjmp`, `C++` exceptions, etc.), and inline assembly. Binary compatibility is a necessity in the real world given that source code is not always available for libraries. Solutions that change the ABI are automatically incompatible with such applications.

Recently, a bounds overflow protection mechanism [10] for `C/C++` was proposed that achieves good runtime performance and software compatibility, along with excellent memory performance and binary compatibility. The protection mechanism is based on *low-fat pointers* that transparently encodes bounds meta information (i.e., an object's base address and size) into the native machine representation of a pointer itself. This meta information can be later retrieved to check all read and write accesses are within the bounds of the corresponding object, a.k.a. *bounds checking*. Since low-fat pointers are also regular machine pointers, good runtime performance and excellent memory performance with binary compatibility is achieved. The low-fat pointer approach is feasible on 64-bit systems with sufficient pointer bit-width, such as the `x86_64`.

† This research was partially supported by a grant from the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

‡ This research was partially supported by the UK EPSRC research grant EP/L022710/1.

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-1891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23287

The main challenge for low-fat pointers is that sufficient control over the memory allocator is required in order to enforce the pointer encoding. This is feasible for the heap allocator (e.g., `malloc`), but is more difficult for stack allocations where the placement of objects in memory is highly restricted. This is a major deficiency since many buffer/object overflow bugs still affect the stack [1].

In this paper, we show how to extend low-fat pointers to stack objects, thereby enjoying the same bounds overflow protection as heap objects. To achieve this, we introduce a *low-fat stack allocator*, that replaces the default stack allocation scheme automatically inserted by the compiler, but ensures that pointers to allocated stack objects satisfy the location and alignment requirements of the low-fat pointer encoding. The low-fat stack allocator uses a variety of techniques introduced in this paper, such as fast allocation size over-approximation, dynamic stack object alignment, stack object pointer mirroring (for location), and memory aliasing optimization to keep memory overheads low. Our solution specially focuses on the key practical barriers against adoption, such as overheads and software/binary compatibility. Good software compatibility is achieved by transparently supporting `C` and `C++` features, which make direct use of the stack, e.g., `setjmp/longjmp`, `C++` exceptions. Binary compatibility is achieved because: (a) the representation of pointers and the ABI is unchanged (low-fat pointer are regular machine pointers); and (b) un-instrumented code can be freely mixed with instrumented code that supports bounds checking (e.g., can link un-instrumented libraries).

We have implemented a version of the low-fat heap+stack allocators for `C` and `C++` on the Linux `x86_64` system architecture. We experimentally evaluate the implementation against the SPEC 2006 benchmarks [23], the Apache web server, the Wilander [28] and RIPE [29] benchmarks, as well as several recent CVEs concerning stack object overflows. We show that runtime overheads are competitive: a 54% overhead for memory read+write instrumentation on SPEC 2006 (while protecting both heap and stack), and a 17% overhead by instrumenting memory writes only. Memory overheads are as low as 3%. Our performance figures are significantly better than AddressSanitizer [22] memory error sanitizer and comparable to the original work of [10] that protects heap objects only. The Wilander, RIPE and CVEs benchmarks show that our method is effective at protecting against stack object bounds overflows for both artificial and real-world test cases.

II. Background

Object bounds errors (e.g., buffer overflows) are a perennial problem for low-level programming languages such as `C` and `C++`. There exists a significant body of previous literature regarding proposed solutions and mitigations, including [3], [4], [9], [10], [11], [14], [15], [19], [20], [22], [30] amongst others. In this section we give a brief overview of the different approaches.

The threat model for bounds errors is well known, so we only provide a summary below. An attacker can induce an *Out-of-Bounds* (OOB) memory write to corrupt other objects in memory. This forms the basis of several kinds of attacks:

- *Control Flow Hijacking* attacks, where the attacker overwrites a *code pointer* (e.g. a return address on the stack) to gain control over the execution of the program.
- *Data Flow* attacks [6], [13], where a *data value* is overwritten causing the program to misbehave in a way beneficial to the attacker, e.g., privilege escalation.

OOB reads can lead to:

- *Information Leakage* attacks, such as the exploitation of the Heartbleed bug [12]. This can also be seen as a subclass of data flow attacks.

One effective method for preventing OOB-errors is bounds checking instrumentation. The basic idea is as follows: given a pointer p associated with an object O with base address ($base$) and size ($size$), then p is out-of-bounds with respect to O if the following test (`isOOB`) succeeds:

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

Object bounds errors can be prevented by instrumenting every memory read or write involving p as follows (instrumentation code is shaded):

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

Here function `error()` reports the bounds error and aborts the program—preventing any control/data or information leakage attacks. The object's size and base is otherwise known as the *bounds meta information*.

A. Traditional Bounds Checking Methods

Over the years many different bounds check instrumentation systems have been proposed. Most differ on the underlying implementation technology, which we summarize below.

Systems such as Safe-C [4], CCured [20] and Cyclone [15] use "fat pointers" which fuse pointer values and associated bounds meta information into one unified object, e.g.:

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

The program is transformed such that fat pointer objects replace native machine pointers. Bounds meta information can then be read directly from the fat pointers, e.g., (`p.base`) and (`p.size`), and this information can be used for bounds check instrumentation. That said, there are several significant disadvantages to this approach, namely: problems arising from changes to memory layout, high performance overheads, increased memory overheads, and near zero binary compatibility. Fat pointers change the underlying ABI, meaning that libraries (including system libraries) must be recompiled or marshalled.

An alternative to fat pointers is to store (some representation of) the bounds meta information in a *shadow space* or *shadow memory* that is separate from the main memory of the program. Some form of shadow memory is used in memory safety systems such as SoftBound [19], PAriCheck [30], Baggy Bounds Checking [3], [9], MudFlap [11] and Intel MPX [14]. The basic idea is to map objects/pointers in the main memory to associated meta information stored in the shadow memory. AddressSanitizer [22] uses a slightly different approach in that shadow memory is used to track *poisoned red-zones* placed around objects. Here, an overflow into a red-zone is detected as an error. That said, overflows that skip red-zones into other valid objects may not be detected.

Shadow memory schemes tend to have better binary compatibility since the layout of objects in main memory has not changed. That said, binary compatibility is not perfect. For example, linking against un-instrumented code that allocates memory without updating the corresponding shadow memory may lead to inconsistencies that cause program misbehavior. Another problem is that shadow memory consumes additional resources leading to higher memory overheads.

Another difference between bounds instrumentation systems is the scope of the OOB-error protection. For example, PAriCheck [30] and Baggy Bounds Checking [3], protect array bounds errors only. In contrast, systems such as AddressSanitizer [22] and SoftBound [19] also protect against overflows arising from implicit pointer arithmetic from field access (e.g., `p->val`). Furthermore, most bounds instrumentation systems, such as [3], [10], [30], protect against *allocation bounds only*—meaning that only overflows beyond the allocation size of the object (including any padding added by the allocator) are detected. Overflows into padding are normally considered benign (such an overflow cannot overwrite code pointers nor data values and is generally not exploitable). These systems also do not detect sub-object bounds overflows. However, the issue is complicated by the fact that many sub-object bounds overflows are intentional, e.g., initializing an object with `memset`.

B. Other Protection Mechanisms

Due to the cost and compatibility issues of bounds checking many other different protection mechanisms have been proposed. These alternatives generally attempt to disrupt attacks or protect control flow rather than prevent memory errors. We give a brief overview of some of the more prominent solutions.

ASLR [21] randomizes memory layout and DEP [27] prevents data execution. These methods aim to frustrate attacks. ASLR is widely deployed, but implementations tend to be incomplete for performance/compatibility reasons.

Shadow stacks [8] split the stack into two parts: a "shadow" stack for storing sensitive data such as code pointers (e.g., the return address) and the main stack for storing everything else. Another variant [31] splits the stack into multiple parts based on object type. Security is derived from the physical separation of the main and shadow stacks. *Traditional shadow stacks* store code pointers contiguously, and *parallel shadow stacks* double stack memory to maintain a one-to-one correspondence between main and shadow locations. Performance overheads range from $\sim$10% for traditional [8], $\sim$4% for parallel [8], and $<$1% for multi-stacks [31].

Control Flow Integrity (CFI) [2] ensures that all indirect jumps target some statically determined set of valid locations. The performance overhead of CFI is $\sim$16% [2]. *Data Flow Integrity* (DFI) [5] is analogous to CFI but for data-flow attacks. The overhead of DFI is higher at $\sim$104% [5].

Code Pointer Integrity (CPI) [16] and the relaxation *Code Pointer Separation* (CPS) use program analysis and instrumentation to isolate code pointers into a separate region of memory. The protection is similar to that of shadow stacks, but also covers code pointers in globals and heap objects (e.g., `C++` virtual method tables). SafeStack [16] is a stack-specific instance of CPS. Because code pointers are typically sparse, performance overheads are very good: $\sim$8.4% for CPI, $\sim$1.9% for CPS, and $<$0.1% for SafeStack [16].

One problem with shadow-stacks, CFI, CPI, CPS, and SafeStack schemes is that they only target control flow attacks (with varying degrees of completeness). Turing completeness of non-control data attacks means that arbitrary code execution can happen even without a control-flow attack [13]. Conversely, DFI only targets data flow attacks but not control flow. Bounds checking can protect against control flow, data flow and information leakage attacks, all at once.

III. Low-fat Pointers

Low-fat pointers [10], [17] are a recent method for tracking bounds meta information that takes advantage of 64-bit systems with sufficient pointer bit-width. The basic idea is to encode bounds meta information *within the representation of machine pointer itself*. This is in contrast to fat pointer or shadow space methods that store bounds meta information explicitly in separate memory. Many possible low-fat pointer encodings are possible:

Example 1 (A Hypothetical Low-Fat Pointer Encoding): The following is a simple low-fat pointer encoding:

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

Here (`p.ptr`) is a native machine pointer to an object that also stores the object size (`p.size`) in the upper 10 bits. The base pointer of `p` can be stored implicitly by ensuring that all objects are aligned to multiples of the object size. The base can therefore be retrieved by rounding (`p.ptr`) down to the nearest object size multiple. Variable `p` is a low-fat pointer since (a) `p` is a regular machine pointer (via cast `p.ptr`), and (b) `p` can be used to extract bounds meta information, e.g., the object size via (`p.size`) and the object base via rounding. ■

The hypothetical encoding from Example 1 is difficult to implement in practice as it imposes strong constraints on the program's virtual address space layout. Other low-fat pointer encodings have been proposed, including [17] (for specialized hardware) and a more flexible encoding [10] for standard architectures such as the `x86_64`.

A. Flexible Low-Fat Pointers

In this paper we use the more flexible encoding from [10] which is summarized below. The flexible encoding was initially designed for heap allocation only, i.e., low-fat replacements for the `malloc`-family of functions (`malloc`, `realloc`, `memalign`, etc.). The main contribution of this paper is to extend this encoding to stack allocation in addition to the heap.

The basic idea of the flexible low-fat encoding from [10] is to divide the program's virtual address space into several regions of equal size (as specified by a `REGION_SIZE` parameter). Regions are contiguous and can be very large, e.g., the implementation in this paper assumes `REGION_SIZE`$=32GB$. Regions are further classified into two main types: *low-fat* regions and *non-low-fat* regions (or *non-fat* regions for short). Low-fat regions are reserved for the low-fat heap allocator, and non-fat regions cover everything else (including text, global,

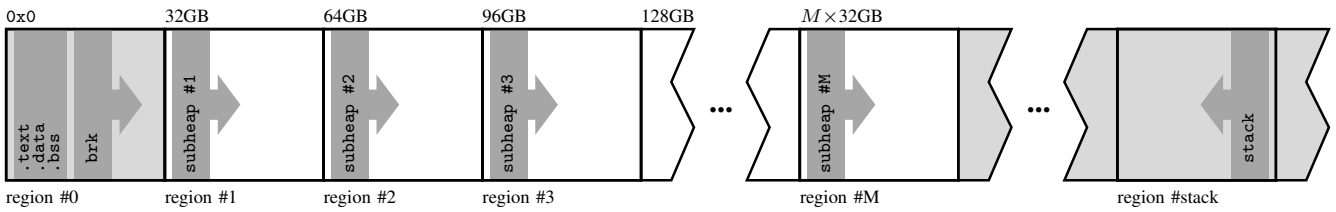

Fig. 1. Program virtual address space layout illustration.

stack and file mappings, etc.). Figure 1 illustrates the low-fat scheme used in [10] and extended in this paper, as follows:

- The stack is placed in the non-fat region labelled #stack.
- Globals and code (bss, data, text) are placed in the non-fat region #0.
- The heap is split into several *sub-heaps* that are placed in low-fat regions #1 to #M.

Each of the low-fat regions #1-#M is associated with a specific allocation size (or $allocSz$ for short) that is determined by a *size configuration* (or Sizes) parameter. The size configuration represents the sequence of allocation sizes supported by the low-fat heap allocator. For example, the size configuration:

$$\mathsf{Sizes} = \langle 16, 32, 48, 64, 80, 96, .. \rangle$$

specifies that the low-fat allocator supports allocation sizes of 16bytes (region #1), 32bytes (region #2), 48bytes (region #3), etc. During allocation, the requested object size (i.e., the $size$ parameter to malloc) is rounded-up (over-approximated) to the nearest allocation size ($allocSz$) that fits. For example, consider an object O of type char[50] with an object size of $size$=sizeof(O)=50. Assuming the above size configuration, $size$ will be rounded up to an allocation size of $allocSz$=64 by adding 14bytes of *padding*. To be consistent with the C standard, at least one byte of padding is always added (so $allocSz > size$) to ensure the pointer-to-the-end-of-an-object is never considered OOB.

The large region size ($32GB$) ensures that it is unlikely any given region will be exhausted for a typical program. If a full region does occur, this can be handled by returning ENOMEM or by reverting to libc malloc as a fail-safe, depending on the configuration. Unused memory, including padding on large (>1 page) objects, is marked as NORESERVE[1] meaning that it will not consume physical memory (RAM/swap) resources.

By design, the memory layout described in Figure 1 is compatible with the traditional layout of a standard Linux process. That is, the standard memory segments (text, stack, etc.) are all placed in their standard locations.

1) Low-fat Conditions: In order to implement low-fat pointers, the heap allocator allocates objects O (of allocation size $allocSz$) according to the following conditions:

- **Region**: The object O is allocated from the sub-heap in region #$\mathcal{I}$, where Sizes[$\mathcal{I}$] = $allocSz$; and
- **Alignment**: The object O is $allocSz$-aligned.

An object O that satisfies these conditions is known as a *low-fat object*. Both low-fat conditions are relatively easy to satisfy for heap allocation, where the allocator has significant control over the placement and alignment of objects. However, satisfying these conditions is more challenging for stack allocation, as will be discussed later in this paper.

2) Reconstructing Meta-information: A pointer p that points to (possibly the interior of) a low-fat object is a low-fat pointer. We can reconstruct the *size* and *base* bounds meta information from p using the following operations:

$$size(p) = \texttt{TABLE}[p \texttt{ / REGION\_SIZE}]\texttt{.size}$$
$$base(p) = (p \texttt{ / } size(p)) \texttt{ * } size(p)$$

Here (/) and (*) are 64-bit integer division and multiplication respectively, and TABLE is a pre-defined lookup table[2] that maps the region index, computed by (p / REGION_SIZE), to the corresponding allocation size from the size configuration (Sizes). The size of the object can then be retrieved by a TABLE lookup. The base of the object is calculated by rounding p down to the nearest multiple of $size(p)$; taking advantage of the low-fat alignment condition. The $base(p)$ operation can be optimized using bit-masks if Sizes are all powers-of-two or fixed point division otherwise. See [10] for more details.

Not all pointers are low-fat (e.g., code, globals, file mappings). To handle non-fat pointers, we define:

$$\texttt{TABLE}[\mathcal{I}]\texttt{.size} = \texttt{SIZE\_MAX} \qquad \mathcal{I} \notin 1..M$$

for all non-fat regions #$\mathcal{I}$. Thus, if q is a non-fat pointer, then $size(q)$ = SIZE_MAX (largest possible size) and $base(q)$ = 0 = NULL. By design, non-fat pointers have the widest possible bounds, and thus will "pass" any bounds check. This property of non-fat pointers is essential for binary compatibility.

B. Bounds Checking with Low-Fat Pointers

With low-fat pointers we can reconstruct bounds meta information and use this for dynamic bounds check instrumentation that enforces spatial memory safety. To associate bounds meta information with pointers, we transform the program using the *meta information propagation schema* from [10]. The transformation is applied automatically using an LLVM compiler infrastructure [18] pass. The basic idea is summarized as follows: for a given function f, the $size(p)$ and $base(p)$ primitives are used to construct meta information for all input pointers p. Here an input pointer p is any pointer value that is a function argument (f(int *p) {..}), return value from another call (p = g(..)), read from memory (p = *q), or cast from an integer (p = (int *)i). Next, meta information is propagated using the following rule: if a pointer q is derived from pointer p through pointer arithmetic (e.g., $p = q + 100$), then q inherits the same meta information as p. Note that,

[1] See the mmap manpage.

[2] TABLE is read-only and small (one entry per $32GB$ region).

```
1  void memcpy(void *dst, void *src, int n)
2  {
3    void *dst_base = base(dst);
4    size_t dst_size = size(dst);
5    void *src_base = base(src);
6    size_t src_size = size(src);
7    for (int i = 0; i < n; i++) {
8      void *dst_tmp = dst + i;
9      void *src_tmp = src + i;
10     if (isOOB(dst_tmp, dst_base, dst_size))
11       error();
12     if (isOOB(src_tmp, src_base, src_size))
13       error();
14     *dst_tmp = *src_tmp;
15   }
16 }
```

Fig. 2. Instrumented version of simple `memcpy`.

contrary to some related work (e.g., [3], [30]) we also consider field access (e.g., $q = \&p\texttt{->}val$) to be pointer arithmetic.

Example 2 (Instrumented Memory Copy): The following simplified `memcpy` is to show meta information propagation.

```
void memcpy(void *dst, void *src, int n) {
    for (int i = 0; i < n; i++) dst[i] = src[i];
}
```

The instrumented form is shown in Figure 2 with the instrumentation highlighted. Here, `dst` and `src` are input pointers, and `dst_tmp` and `src_tmp` are derived pointers from pointer arithmetic. The bounds meta information for the input pointers is calculated on lines 3-6. The memory operation on the derived pointers (line 14) is protected by the bounds checks on lines 10-13. ■

By design, the code in Figure 2 is compatible with both low-fat and non-fat pointers. However, only low-fat pointers enjoy bounds protection. For non-fat pointers, the $size$ and $base$ operations return "wide bounds" meaning that the `isOOB` checks on lines 10-13 never fail.

1) Handling OOB Pointers: Bounds meta information propagation handles the case where the input pointer is within bounds, and a subsequent derived pointer is out-of-bounds. However, it does not handle the case where the input pointer itself is already out-of-bounds. To catch such cases [10] enforces a global invariant that all escaped pointers must be within bounds. Here, an escaping pointer p is any pointer that is passed to another function as an argument (`f(p)`), returned from the current function (`return p`), written to memory (`*r = p`), or cast to an integer value (`i = (int)p`). OOB-pointers are prevented from escaping by inserting a bounds check before the relevant operation, i.e.:

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

C. Compatibility

The flexible low-fat pointer encoding and bounds instrumentation summarized in Sections III-A and III-B achieves good software compatibility and excellent binary compatibility.

Binary compatibility is a consequence of the fact that low-fat pointers are also regular machine pointers, and can therefore be passed to and from non-instrumented code without any special handling or marshalling. Likewise, non-fat pointers generated by non-instrumented code are automatically compatible with instrumented code, albeit without bounds protection. Binary compatibility is very important in practice [24], since it is not always possible to recompile code, such as system or external proprietary libraries.

The low-fat pointer encoding also achieves a high degree of software compatibility. The main source of software incompatibility arises from code that intentionally creates OOB-pointers (technically undefined behavior under the `C` standard). That said, many intentional OOB-pointers are compatible, provided the (1) OOB-pointer is not accessed (read or write) and (2) does not escape according to the rules in Section III-B1. Exceptions include some uncommon programming idioms, such as: using $base-1$ as a sentinel, accessing arrays via offsets (e.g. using $(a-k)[k]$ to access $a[0]$), and copying objects using pointer difference. Such idioms are relatively rare in practice—with only 24 functions out of the SPEC 2006 benchmark ($\sim$1.1 million lines of `C/C++` code) being affected [10]. The rest of the SPEC 2006 code is compatible without any additional modification.

Low-fat pointers are not compatible with all machine architectures. Specifically, low-fat pointer encodings assume that the native pointer bit-width is sufficient to encode bounds meta information. In practice this means only 64-bit architectures (such as the `x86_64`) are supported. We do not consider this to be a significant limitation as 64-bit architectures are commonplace. For the rest of the paper, we assume the underlying target architecture is the `x86_64`. The paper is written targeting `x86_64` assembler and Linux where needed to clarify details.

IV. Low-fat Stack Allocation

The flexible low-fat pointer encoding presented in Section III was initially designed for heap allocation only. This means that stack allocated objects are non-fat by default and thus are not protected against bounds errors. In this section, we design a new stack allocator, called the *low-fat stack allocator*, that extends the coverage of the flexible low-fat pointer encoding to also include stack objects. The low-fat stack allocator aims to retain all of the desirable properties of the default stack allocator, such as efficiency and automatic stack object deallocation (for both normal and abnormal control flow). The latter excludes replacing stack allocation with heap allocation, such as that with CCured's *heapified* stack objects [20]. Furthermore we wish to retain the existing flexible low-fat pointer encoding so that stack and heap allocation are compatible.

The low-fat stack allocator must allocate objects according to the alignment and region conditions from Section III-A1. Satisfying these conditions is not a problem for the heap allocator (which has significant freedom regarding the location and alignment of objects). However, stack allocation is much more restricted:

1) Stack allocation works by modifying the *stack pointer* which can only ever point-to/allocate-from a single region of memory. Under the reasonable assumption that stack objects are not of uniform size, it is necessary to allocate objects from different memory regions.
2) The stack pointer is only guaranteed to be aligned to the default *system stack alignment*, typically 8 or 16 bytes. Low-fat objects are required to be aligned by allocation size. Furthermore, for *variable sized objects*, such as variable length arrays (VLAs), the stack alignment must be calculated dynamically for a given object size.

Problem 1) will be addressed by splitting the stack across several regions—one for each potential stack object allocation size. The result of a stack allocation will be a pointer to local stack memory contained within the appropriate low-fat region. The challenge for such a split is twofold: (a) maintaining a small allocation state that is compatible with the default stack allocator (e.g., used by un-instrumented code), and (b) avoiding excess memory usage caused by fragmenting stack objects over multiple memory regions. We shall present solutions to both of these challenges. Problem 2) will be addressed by suitably aligning all low-fat stack allocated objects.

For the rest of this section we shall review the default stack allocator, discuss some assumptions, and then present the low-fat stack allocation schema.

A. Default Stack Allocation

On most systems, stack objects are (de)allocated by adjusting the stack pointer. For example, on the `x86_64` (where the stack grows down towards low memory addresses), the allocation of an S-byte object is traditionally handled by decrementing the stack pointer register (`%rsp`) by S bytes. Likewise the object can be deallocated by incrementing `%rsp` by S bytes—restoring the old value of the stack pointer. Compilers emit code that automatically deallocates all stack objects on function exit and (in the case of optimizing compilers) when stack objects go out-of-scope under the `C/C++` scoping rules. From the point of view of the programmer, stack allocation has several advantages, namely:

- Efficiency ((de)allocation is a few instructions);
- Automatic lifetime management and deallocation on function exit, including normal return or abnormal function exit such as `C++` exceptions or `longjmp`.
- Small allocation state (single word stack pointer).

Stack objects are allocated from *stack memory*, which is created by the operating system before the program starts. Since stack memory is outside the low-fat heap regions #1-#M (see Figure 1), stack objects are non-fat by default.

B. Assumptions

In order to simplify the low-fat stack allocator we make several assumptions that are listed below. The assumptions include: a generalized stack allocation template, the implementation details of the default stack allocator, and a restricted set of allocation sizes supported by the low-fat stack allocator.

1) Stack Allocation Template: Stack allocation has many forms, as illustrated by the following pseudo-code:

```
void f(int len) {
  int x;             // Local variable
  int buf1[50];      // Local array (fixed-length)
  float buf2[len];   // Local array (variable-length)
  char *buf3 = alloca(len); // Explicit allocation
  ... }
```

Stack space for `&x`/`buf1`/`buf2` will be implicitly allocated by code generated by the compiler. Stack space can also be explicitly allocated using the `alloca` standard library call,[3] where `alloca`($size$) allocates $size$ bytes of space from the stack frame of the caller. Here `buf3` is allocated using this method.

Without loss of generality, we shall assume that all stack allocations are of the explicit `alloca`-form:

`void *`ptr `= alloca(`$size$`);` (STACKALLOC)

Here input $size$ is the object size and output ptr is the resulting pointer to the stack allocated object. All other forms of stack allocation are rewritten into the (STACKALLOC) form, e.g.:

```
int x                  &x=alloca(sizeof(int))
int buf1[50]      →    buf1=alloca(50*sizeof(int))
float buf2[len]        buf2=alloca(len*sizeof(float))
```

The LLVM compiler infrastructure [18] already does such a transformation internally.

2) Default Stack Allocator Implementation: The default stack allocator implementation is compiler, hardware and operating-system specific. For this paper, we assume the default stack allocator as implemented by the LLVM compiler infrastructure [18] for the Linux `x86_64` target. That is, stack (de)allocation is implemented by (de-)incrementing the stack pointer that is stored in the `%rsp` register. The low-fat stack allocation method presented in this paper can be adapted to other compilers and targets.

3) A Special Size Configuration for Stack Allocation: The size configuration (Sizes) specifies the set of all allocation sizes supported by the low-fat heap allocator. For low-fat stack allocation, we shall assume a special stack-specific size configuration (StkSizes) that comprises the powers-of-two subset of Sizes. Here StkSizes can be defined as follows:

$$\mathsf{StkSizes} = \mathsf{Sizes} \cap \{16, 32, 64, 128, ..\}$$

We assume Sizes contains a reasonable StkSizes subset. Appendix A gives the configuration used by our implementation.

The motivation for StkSizes is to speed up low-fat stack allocation (and fast allocation is important for the stack). Specifically, powers-of-two sizes allow for efficient stack object allocation size over-approximation (discussed in Section IV-C) and allocation size alignment (discussed in Section IV-D). The main disadvantage is that the more coarse-gained size configuration leads to higher memory over-approximation overheads for stack allocated objects. We note that this does not affect heap allocated objects which continue to use the original fine-grained size configuration (Sizes).

We shall now introduce the low-fat stack allocation schema that replaces default stack allocation represented by the template (STACKALLOC) defined above. The schema is divided

[3] See the `alloca` man page.

into three parts: allocation size over-approximation (Section IV-C), allocation size alignment (Section IV-D), and stack object pointer mirroring (Section IV-E).

C. Allocation Size Over-approximation

During allocation, the object size is over-approximated (a.k.a. rounded up) to the nearest size from StkSizes that fits. For stack allocation, it is important for this operation to be fast. Our basic approach is to map object sizes to allocation sizes using a lookup table. This is a two-step process:

1) First the object size ($size$) is mapped to an index ($idx \in 0..64$) using a suitable logarithmic index function we name $index(size)$. One possible definition is:[4]
$$idx = index(size) = \lceil \log_2(size + 1) \rceil$$
2) Next we use (idx) to lookup the corresponding allocation size from a pre-defined lookup table we name `SIZES`.

Our approach takes advantage of the fact that StkSizes contains only power-of-two sizes, meaning that the base-2 logarithm of each allocation size maps to a unique index. The schema for allocation size over-approximation is therefore as follows:

```
size_t idx = index(size);
size_t allocSz = SIZES[idx];
```

Here $size$ is the input object size and $allocSz$ is the resulting allocation size. The `SIZES` table is pre-computed as follows:

$$fits(allocSz, i) = \forall size, index(size) = i : size \leq allocSz$$
$$\texttt{SIZES}[i] = \mathbf{min}\ \{allocSz \in \text{StkSizes} \mid fits(allocSz, i)\}$$

Essentially `SIZES`$[i]$ is the "best-fit" allocation size with respect to all possible object sizes ($size$) where i=$index(size)$.

A suitable choice for the index function ($index$) depends on what can be efficiently implemented for a given underlying system architecture. For the `x86_64`, a good choice is the leading zero count function (clz) that returns the number of leading zero bits in a 64-bit word, defined as follows:

$$clz(size) = 64 - \lceil \log_2(size + 1) \rceil$$

For example, $clz(50)$=$clz(\texttt{0x32})$=58. The leading zero count function is available as an LLVM [18] compiler built-in function (`clzll`) which compiles down a single *leading zero count* instruction (`lzcnt`) on modern versions of the `x86_64` architecture. On older `x86_64` CPUs, that do not support the `lzcnt` instruction, *bitscan forward* (`bsf`) is a good substitute.[5]

Example 3 (Over-approximation): Assuming that leading zero count is used as the index function, then the `SIZES` lookup table is defined as follows:

$$\ldots \quad \begin{array}{ll} \texttt{SIZES}[57] = 128 & \texttt{SIZES}[61] = 16 \\ \texttt{SIZES}[58] = 64 & \texttt{SIZES}[62] = 16 \\ \texttt{SIZES}[59] = 32 & \texttt{SIZES}[63] = 16 \\ \texttt{SIZES}[60] = 16 & \texttt{SIZES}[64] = 16 \end{array}$$

Suppose the program intends to allocate a stack object of size 50bytes, i.e. $size$=50, then

$$index(size) = clz(50) = 58$$
$$allocSz = \texttt{SIZES}[58] = 64$$

Note that the indexes idx∈60..64 correspond to object sizes of $size$∈0..15bytes. These map to the minimum allocation size of 16bytes. ■

D. Allocation Size Alignment

Low-fat objects are required to be aligned to a multiple of the allocation size as specified by the alignment condition of Section III-A1. To support dynamic alignment, we use a pre-computed `MASKS` lookup table (analogous to the `SIZES` table defined above). The `MASKS` table is defined as follows:

$$\texttt{MASKS}[i] = \texttt{UINT64\_MAX} \ll \log_2(\texttt{SIZES}[i])$$

Given $allocSz = \texttt{SIZES}[i]$, a pointer p can be $allocSz$-aligned by the bit-mask operation (p & `MASKS`$[i]$). This is possible because stack allocation sizes are always powers-of-two. The schema for aligned stack allocation is therefore:

```
    ⋮
%rsp = %rsp - allocSz;
uint64_t mask = MASKS[idx];
%rsp = %rsp & mask;
void *ptr = %rsp;
```

Here (idx) and ($allocSz$) are calculated according to the schema from Section IV-C which is represented by the vertical ellipses. Aligned stack allocation proceeds as follows:

1) Stack space is reserved by decrementing the stack pointer `%rsp` by the allocation size of $allocSz$ bytes.
2) Next the stack pointer `%rsp` is $allocSz$-aligned by masking `%rsp` with $mask$=`MASKS`$[idx]$.
3) Finally, pointer ptr is set to the now $allocSz$-aligned stack pointer `%rsp`. The allocated object spans addresses $ptr..ptr + size$, where $size$ is the original object size.

In LLVM, the stack pointer register `%rsp` can be indirectly manipulated using the `stack_save`/`stack_restore` builtins.

Example 4 (Alignment): We continue Example 3. Suppose that the stack pointer is `%rsp` = `0x7fff00001110` and given:

$$index(50) = clz(50) = 58 \qquad allocSz = \texttt{SIZES}[58] = 64$$

then aligned stack allocation proceeds as follows:

$$\begin{aligned} \texttt{\%rsp} &:= \texttt{0x7fff00001110} - 64 = \texttt{0x7fff000010d0} \\ \texttt{\%rsp} &:= \texttt{\%rsp \& MASKS}[58] \\ &= \texttt{0x7fff000010d0 \& (UINT64\_MAX << 6)} \\ &= \texttt{0x7fff000010c0} = ptr \end{aligned}$$

By design, the output ptr is $allocSz$-aligned, i.e.

$$\texttt{0x7fff000010c0} \bmod 64 = 0 \qquad ■$$

E. Stack Object Pointer Mirroring

The default stack allocator allocates objects from a non-fat region (region #stack in Figure 1) meaning that stack objects

[4]The increment ($size$+1) ensures ($allocSz$>$size$), see Section III-A.
[5]The main disadvantage of `bsf` is that it is undefined for $size = 0$.

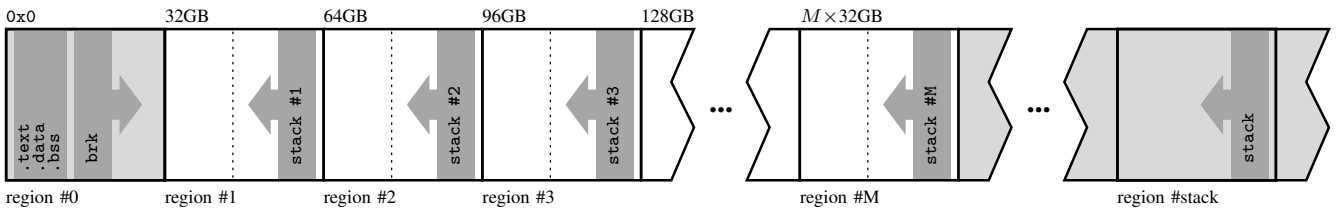

Fig. 3. Modified Program virtual address space layout illustration.

are non-fat by default. For the low-fat stack allocator, the region condition (Section III-A1) requires that objects are allocated from the low-fat region corresponding to the allocation size. To achieve this, this section introduces the notion of *stack object pointer mirroring* that maps non-fat stack object pointers to an equivalent low-fat pointer to the corresponding region. We show that pointer mirroring preserves the key properties of the default stack allocator, namely: automatic deallocation, a small allocator state and compatibility with existing code.

The basic idea of pointer mirroring is to split the main program stack into several *region-local virtual stacks*, one for each possible allocation size from (StkSizes). Here each region-local virtual stack $\#\mathcal{I}$ is wholly contained within the low-fat region $\#\mathcal{I}$. When space for an object (of allocation size $allocSz$) is allocated on the main program stack, the corresponding space for an object (of size $allocSz$) will be implicitly allocated on virtual stack $\#\mathcal{I}$. The pointer to the original space on the main stack is thus "mirrored" by the pointer to the space on virtual stack $\#\mathcal{I}$. The mirrored pointer is low-fat since it satisfies the alignment and region conditions, and will be afforded OOB-error protection in instrumented code. The main stack (in region #stack) is not completely replaced by the virtual stacks and is retained for several reasons:

- Function prologue/epilogues;
- Register spills and other non-fat stack allocations that do not need OOB-error protection; and
- Compatibility with non-instrumented code.

Essentially, each virtual stack $\#\mathcal{I}$ is analogous to the main stack, but is specialized for objects of the specific allocation size corresponding to low-fat region $\#\mathcal{I}$. One problem is how to maintain the virtual stack state analogous to the main stack state, a.k.a. the stack pointer (`%rsp`). Explicitly maintaining separate virtual stack pointers conflicts with our design goal of keeping the overall low-fat stack allocator state small. For example, a function return or `longjmp` would require restoring all virtual stack pointers, rather than one (`%rsp`) in the default case. To solve this problem we represent each virtual stack pointer implicitly as a linear function of the main stack pointer (`%rsp`), similar to parallel shadow stacks [8]. In the case of function return and `longjmp`, restoring the main stack pointer (`%rsp`) to an old value also implicitly restores all virtual stack pointers.

Initially, we assume the naïve approach where the main stack and virtual stacks are distinct physical regions of memory. This is potentially wasteful: as the overall stack memory usage is, in the worst case, multiplied by the total number of virtual stacks. Later in Section VI we shall introduce a *memory aliasing* optimization that maps all stacks to the same physical memory—thereby reducing memory overheads.

1) Stack Memory Layout: To implement virtual stacks, the program's virtual address space layout of Figure 1 is modified to that shown in Figure 3. The main difference is that each region $\#\mathcal{I}$ is now partitioned into disjoint stack and heap sub-regions, as represented by the dashed lines bisecting each low-fat region. Each virtual stack $\#\mathcal{I}$ is wholly contained within the stack sub-region $\#\mathcal{I}$. The exact partition between the heap and stack sub-regions is left as a design choice. Our implementation reserves $4\,GB$ for the stack sub-region[6] and $28\,GB$ for the heap sub-region (for a $32\,GB$ total region size). Each virtual stack $\#\mathcal{I}$ is configured to satisfy the following key properties:

1) *Same size*: The main/virtual stacks are the same size; and
2) *Same offset*: The base address of virtual stack $\#\mathcal{I}$ relative to region $\#\mathcal{I}$ is equal to the base address of the main stack relative to region #stack, thus giving the relationship:

$$base_{\text{stack } \#\mathcal{I}} - base_{\text{region } \#\mathcal{I}} = base_{\text{stack}} - base_{\text{region \#stack}}$$

Here $base_{object}$ represents the base address of $object$.

The purpose of these properties is to establish the following linear relationship between a pointer (ptr) to the main stack and "mirrored pointers" ($ptr_{\#\mathcal{I}}$) to each virtual stack $\#\mathcal{I}$:

$$ptr_{\#\mathcal{I}} = ptr + (\mathcal{I} - 4095) \times \texttt{REGION\_SIZE} \quad \text{(MIRRORING)}$$

Here the constant 4095 is region index of region #stack. The virtual stack pointer $\#\mathcal{I}$ for each $\mathcal{I}$ is defined to be the mirrored main stack pointer (`%rsp`).

2) Implementation: Assuming the modified layout described above, pointer mirroring is implemented using a precomputed `OFFSETS` lookup table analogous to the `SIZES` and `MASKS` tables defined previously. The `OFFSETS` table contains the pointer difference between the main stack and the virtual stack corresponding to each allocation size. Pointer mirroring can then be achieved by the addition of the appropriate entry from the `OFFSETS` table to a main stack object pointer.

The `OFFSETS` table is indexed by the $index(size)$ function, and is defined as follows:

$$\texttt{OFFSETS}[i] = (\mathcal{I} - 4095) \times \texttt{REGION\_SIZE}$$

Here $\mathcal{I}$ is the region index corresponding to $\texttt{SIZES}[i]$ and constant 4095 is region index of region #stack.

A pointer (ptr) to a main stack allocated object is mirrored to the appropriate virtual stack using the following schema:

[6]Enough to support multiple stacks for multi-threaded programs.

```
    ⋮
int64_t offset = OFFSETS[idx];
ptr = ptr + offset;
```

Note that since the region size (REGION_SIZE) and stack allocation sizes (StkSizes) are both powers-of-two, pointer mirroring will preserve the allocation size alignment of the transformed pointer. The resulting mirrored pointer therefore satisfies both the low-fat alignment and region conditions, and thus will enjoy OOB-error protection in instrumented code.

Example 5 (Pointer Mirroring): We continue Example 4:

$$allocSz = 64 \qquad idx = 58 \qquad ptr = \texttt{0x7fff000010c0}$$

The final step is to mirror (ptr) to the appropriate virtual stack. Assuming the size configuration (Sizes) from Appendix A, then $allocSz = 64$ corresponds to region #4, thus:

$$\begin{aligned}\texttt{OFFSETS}[58] &= (4{-}4095){\times}\texttt{REGION\_SIZE} = -140565689663488\\ ptr &:= ptr + \texttt{OFFSETS}[58]\\ &= \texttt{0x7fff000010c0} + (-140565689663488)\\ &= \texttt{0x27000010c0}\end{aligned}$$

The resulting mirrored pointer $ptr = \texttt{0x27000010c0}$ satisfies the region and alignment conditions of low-fat pointers:

- *region condition*: $(\texttt{0x27000010c0 / REGION\_SIZE}) = 4$
- *alignment condition*: $\texttt{0x27000010c0} \bmod 64 = 0$ ■

Object pointer mirroring is similar to shadow stacks [8], [31], which also splits the main stack into distinct regions of memory. The main difference is how security guarantees are enforced: we rely on the low-fat pointer encoding and bounds checking instrumentation, whereas shadow stacks mitigate control flow attacks by physically separating memory. In the case of pointer mirroring, it is not required that the memory be physically separated, as will be explained further in Section VI.

F. Low-fat Stack Allocation

Finally, we combine allocation size over-approximation (Section IV-C), allocation size alignment (Section IV-D) and stack object pointer mirroring (Section IV-E) to derive the complete low-fat stack allocation schema as shown in Figure 5. The low-fat stack allocator itself is implemented as an LLVM compiler pass that replaces default stack allocation (STACKALLOC)—as represented by the LLVM intermediate representation `alloca` instruction—with an instantiation of the pseudo-code shown in Figure 5. As discussed above, the resulting pointer (ptr) satisfies both the alignment and region conditions, and thus will enjoy OOB-error protection in instrumented code.

Example 6 (Low-Fat Stack Allocaton): The complete low-fat stack allocation sequence for Examples 3, 4 and 5 is shown in Figure 4. The allocation sequence proceeds as follows:

(a) The initial state with %rsp=0x7fff00001110. The initial virtual stack pointer #4 is implicitly the mirrored pointer: $\texttt{\%rsp} + (4-4095){\times}\texttt{REGION\_SIZE} = \texttt{0x2700001110}$.
(b) The stack pointer %rsp is decremented by $allocSz = 64$ bytes (Example 3);

```
/* Compute allocation size (Section IV-C). */
size_t idx = index(size);
size_t allocSz = SIZES[idx];

/* Allocate aligned object (Section IV-D). */
%rsp = %rsp - allocSz;
uint64_t mask = MASKS[idx];
%rsp = %rsp & mask;
void *ptr = %rsp;

/* Mirror allocated pointer (Section IV-E). */
int64_t offset = OFFSETS[idx];
ptr = ptr + offset;
```

Fig. 5. The complete low-fat stack allocation schema.

(c) The stack pointer %rsp is aligned to an $allocSz$-boundary (Example 4);
(d) The final state with %rsp=0x7fff000010c0;
(e) The final state of virtual stack #4, where (OBJECT) is allocated at virtual stack pointer #4 (Example 5): $\texttt{\%rsp} + (4-4095){\times}\texttt{REGION\_SIZE} = \texttt{0x27000010c0}$.

Low-fat stack deallocation is identical to that of the default stack allocator, and is achieved by restoring the stack pointer %rsp to its original value 0x7fff00001110. This implicitly restores the virtual stack pointer #4 to its original value of 0x2700001110, thereby deallocating (OBJECT). ■

We shall now discuss compatibility issues and optimizations of the low-fat stack allocator.

V. COMPATIBILITY

One of the main advantages of low-fat pointers is that they maintain very high compatibility with existing non-instrumented binary code. This is because (a) low-fat pointers are regular machine pointers, so the binary interface (ABI) remains unchanged, and (b) there is no shadow memory that needs to be maintained, which is especially problematic for non-instrumented code. Although binary compatibility is a security trade-off (non-instrumented code is not afforded OOB-error protection), in practice it is not always possible to re-compile and re-instrument everything, such as with closed-source or system libraries. Solutions like [7] work on binaries, but provide stack-only protections and generally introduce overheads higher than the one reported in our experiments. The lack of binary compatibility inhibits the adoption of a solution in the real world [24]. It is therefore essential that the low-fat stack allocator exhibits a high-level of compatibility similar to that of the heap allocator.

In this section we examine compatibility issues concerning the low-fat stack allocator, including compatibility with the default allocator (still used by uninstrumented code), multi-threading, and the compiled code.

A. Comparison with the Default Stack Allocator

1) Allocator State: Both the default and low-fat stack allocators use the same allocator state, i.e., the stack pointer that is stored in the %rsp machine register. The low-fat stack allocator also maintains several virtual stacks. However, the virtual stack pointers are derived from a linear function (mirroring) of the

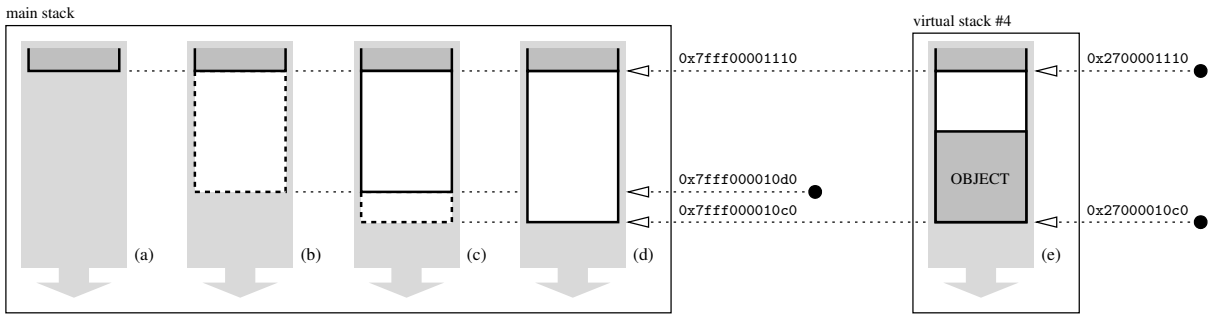

Fig. 4. Example low-fat stack allocation sequence. States (a), (b), (c) and (d) apply to the main stack, and (e) to virtual stack #4.

main stack pointer `%rsp` value, and thus do not need to be represented/stored explicitly.

Since both the default and low-fat stack allocators use the same allocation state, no special treatment is required when transitioning to and from legacy (non-instrumented) code. The legacy code may continue to allocate objects on the (main) stack as normal. However, pointers to these objects will be non-fat and not enjoy OOB-error protection.

2) Deallocation: Both default and low-fat stack deallocation works by restoring the main stack pointer (`%rsp`) to some previous value. In the case of the low-fat stack allocator, this also implicitly restores the virtual stack pointers for all virtual stacks, thereby deallocating any relevant low-fat stack objects. This means that the low-fat stack allocator is automatically compatible with all of the kinds of standard stack deallocation, including stack frame clean-up on normal function exit, abnormal function exit (such as `longjmp` and `C++` exceptions), and even direct stack pointer manipulation (e.g., via inline assembly). Furthermore, `C++` stack unwinding (that automatically runs destructors for stack objects on the event of an exception) is not affected by the low-fat stack allocator.

3) Alternative Stacks and Multi-threading: A program may use more than one stack—the most common case being multi-threaded programs where a new stack is created for each thread. The default stack allocator is compatible with any stack provided the corresponding memory is accessible. However, the low-fat stack allocator imposes the additional requirement that (a) each stack be allocated from region `#stack` (to be compatible with the `OFFSETS` table), and (b) be "mirrored" to a set of virtual stacks as illustrated in Figure 3.

For common cases, such as multi-threading, our prototype implementation intercepts relevant standard library function calls, such as `pthread_create`, to create low-fat allocator compatible stacks. This achieves a high degree of compatibility with most programs. However, it does not guarantee full compatibility with more esoteric programs that create custom stacks using other methods, e.g., direct allocation of stack memory and manipulation of the stack pointer (`%rsp`).

4) Speed: The default stack allocator is very fast, with (de)allocation typically implemented in a few low-latency instructions. The low-fat stack allocator, as represented by the schema in Figure 5, is more complicated. In lieu of the experimental results in Section VII, we first compare the

```
(a)  sub %rax, %rsp
     and $-8, %rsp
     mov %rsp, %rbx

     lzcnt %rax, %rax
     sub SIZES(,%rax,8), %rsp
(b)  and MASKS(,%rax,8), %rsp
     mov %rsp, %rbx
     add OFFSETS(,%rax,8), %rbx

(c)  sub $56, %rsp
     mov %rsp, %rbx

     sub $64, %rsp
     and $-64, %rsp
(d)  mov %rsp, %rbx
     add OFFSETS+464(%rip), %rbx
```

Fig. 6. Comparison of stack allocator compiled code, assuming `%rax`=$size$ (if applicable) and `%rbx`=ptr: (a) default (variable length) (b) low-fat (variable length) (c) default (fixed length=50, rounded up to length=56 to preserve alignment) (d) low-fat (fixed length=50, rounded up to length=64).

`x86_64` compiled version[7] of the low-fat stack allocator, as shown in Figure 6(b), with that of the default stack allocator, as shown in Figure 6(a). Here we assume that the input $size$ parameter is stored in the general purpose register `%rax`, and the resulting allocated ptr is to be stored in register `%rbx`. The default stack allocation (Figure 6(a)) consists of three instructions:

- Decrementing $size$ (`%rax`) bytes from the stack pointer `%rsp`;
- Aligning the stack pointer to the default stack alignment (here is assumed to be 8); and
- Saving the new stack pointer value into ptr (`%rbx`).

By comparison, low-fat stack allocation consists of five instructions, including:

- a *leading zero count* `lzcnt` instruction that implements the $index$ function introduced in Section IV-C; and
- *table lookups* for `SIZES`, `MASKS` and `OFFSETS` and the corresponding over-approximation/alignment/mirroring operations.

The insertion of extra instructions and memory reads introduces some runtime overhead compared with the default stack

[7]Here we use AT&T style syntax for assembly code: `op` src dst.

allocator. However, the impact of the memory reads is partially mitigated by the fact that the lookup tables are small (all three fit into a single page) and are read-only, and thus are cache friendly. The overall performance impact of the low-fat stack allocator will be experimentally evaluated in Section VII.

VI. Optimizations

The low-fat stack allocator introduces both time and space overheads compared with the default stack allocation scheme. In this section we consider three optimizations, namely *fixed-sized* objects, *non-escaping* pointers, and *memory aliasing*, that help reduce the overhead of the low-fat stack allocator. Of these optimizations, memory aliasing is particularly important with respect to the overall feasibility of our approach.

A. Fixed-size Stack Objects

It is common for stack allocated objects to have a known fixed size at compile time. We can exploit this to optimize low-fat stack allocation by using *constant propagation* to simplify the compiled code. This optimization can be described by the following example:

Example 7 (Fixed-Size Stack Objects): Assuming the fixed object size of $size$=50, the following values can be evaluated at compile time:

$$idx = 58 \qquad allocSz = 64$$
$$mask = \texttt{0xffffffffffffffc0} = -64$$

Under these assumptions, the schema from Figure 5 can be compiled into the more optimal code shown in Figure 6(d). The optimized allocation has eliminated the `lzcnt` (clz) index calculation and the `SIZES/MASKS` table lookups from Figure 6(b). The resulting code uses two instructions over the default allocator of Figure 6(c). Note that the `OFFSETS` table lookup is not removed.[8] This is because, unlike allocation sizes and masks, offsets are too big to store as `x86_64` immediate values (constants are limited to 32-bit values under `x86_64`). ■

B. Non-Escaping Stack Object Pointers

A pointer to a stack object *escapes* if it can be passed (directly or indirectly) to another function invocation. Stack object pointers can be passed directly as function parameters or indirectly by being stored in memory.

If a stack object pointer (1) does not escape, and (2) all access to the object can be statically determined to be within bounds, then the object can be allocated using the default stack allocator. This eliminates all low-fat stack allocator overheads for that object. *Escape analysis* (already provided by the LLVM compiler infrastructure) can be used to determine if pointers to stack objects escape or not.

C. Stack Memory Aliasing

In order to satisfy the low-fat region condition, objects are allocated from the virtual stack corresponding to the allocation size. Thus far, it has not been specified how the main and virtual stack memory is arranged. The naïve approach is to keep each stack in separate regions of physical memory—a similar approach to parallel shadow stacks [8].

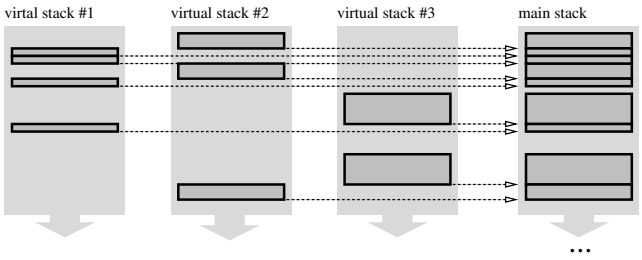

Fig. 7. Low-fat stack (3 different object sizes).

The problem with the naïve approach is that it can lead to high memory overheads. In the worst-case, physical stack memory usage will be multiplied by N+1, where N is the number of virtual stacks. The overhead can be partially mitigated by using `NORESERVE` memory—meaning that virtual pages that are never touched (e.g. if the program does not allocate stack objects of a particular size) will not consume physical memory resources. Nevertheless the potential for significant stack memory overheads remain.

We address this problem by using the *memory aliasing* optimization, where each virtual stack is backed by the same physical memory as the main stack. This optimization exploits the virtual memory implementation of modern CPUs, such as the `x86_64`, where a single page of physical memory can be mapped multiple times to different pages of virtual memory at different addresses. Each virtual addresses can then be used to access and modify the underlying physical memory equally. Where applicable, memory aliasing reduces memory overheads—i.e., although logically multiple virtual pages exist, in reality only a single page of physical memory is actually consumed.

In the context of the low-fat stack allocator, memory aliasing takes advantage of the fact that allocated stack objects are *pairwise disjoint* with respect to other objects from different virtual stacks. That is, when an object is allocated on virtual stack #$\mathcal{I}$, the corresponding memory remains unallocated on the main stack and all virtual stacks #$\mathcal{J}$ for $\mathcal{J}\neq\mathcal{I}$, as partly illustrated by Figure 4. All stack allocated objects can therefore be mapped to the same physical memory without collision. This can be illustrated with a simple example:

Example 8 (Stack Memory Aliasing): Consider the simplified virtual stack layout as shown in Figure 7. Here we assume there are three low-fat regions and virtual stacks #1, #2, #3. Each stack object is allocated from the virtual stack corresponding to the allocation size, with small objects from virtual stack #1, medium from #2, and large from #3. All objects are pairwise disjoint and can be projected back onto the main program stack. The main and virtual stacks can use memory aliasing to share the same physical memory without objects overlapping. ■

Memory aliasing can be implemented using POSIX shared memory objects[9] and the Linux `shm_open` API. The basic idea

[8] The operand (`OFFSETS+464(%rip)`) represents the `OFFSETS`[58] lookup generated by the compiler, where $58 \times$ `sizeof(ssize_t)` $= 464$.

[9] Not to be confused with *shared objects*. See the `shm_overview` manpage.

is to create a shared memory object and map it multiple times, once for each of the main and virtual stacks. This effectively replaces $N+1$ stacks with a single physical stack.

Finally we note that allocation size over approximation and allocation size alignment also introduce memory overheads unrelated to memory aliasing. These overheads are unavoidable but are less significant. We defer further discussion to the experimental evaluation in Section VII.

1) Implementation Issues: The `shm_open` call unavoidably creates a temporary name on the filesystem that can be used by other programs to access the object. This name is immediately deleted but a race condition remains. To solve this, our implementation ensures that the object is opened exactly once (i.e., by the program) using the Linux `fcntl` lease API.

If memory aliasing is used, care must be taken with handling the `fork` family of system calls. The problem arises from the Linux `fork` semantics concerning shared memory mappings, such as the aliased stack. Ordinarily, the stack is a private mapping, meaning that `fork` will create a local copy of the stack for the child process that is backed by different physical memory. In contrast, shared mappings are directly inherited by the child. This means that the parent and child will compete for the same physical stack memory, which inevitably results in memory corruption. To solve this problem, our implementation intercepts the `fork` system call and copies the stack manually. Note that this problem does not exist for operating systems that do not support `fork`-like operations, such as Windows.

VII. Experiments

In this section we evaluate the performance and effectiveness of an implementation of the low-fat stack allocator. Our implementation, called LowFat, extends the low-fat heap allocator described in [10]; and is configured using the parameters from Appendix A. The low-fat stack allocator is implemented as an LLVM compiler infrastructure [18] pass that replaces default stack allocation (as represented by the LLVM intermediate representation `alloca` instruction) according to the schema shown in Figure 5. The compiler pass implements all of the optimizations described in Section VI. The runtime component is implemented as a library that is linked against the resulting executable.

All experiments were run on a `x86_64` GNU/Linux system with an Intel Xeon E5-2660 v3 CPU (clocked at 2.60GHz) with $64GB$ of RAM. We use the `clang-3.5` compiler for both the instrumented and un-instrumented tests.

A. Performance

To measure the performance of the LowFat implementation we use the SPEC 2006 benchmark suite [23]. For these experiments we focus on the combined performance of the heap and stack OOB-error protection, as this is the intended use case. The results are shown in Figure 8(a) for timings and 8(b) for memory usage. We compare the LowFat implementation with:

- Orig: The un-instrumented program that uses the default heap (i.e., stdlib `malloc`) and stack allocators; and
- ASAN a.k.a. *AddressSanitizer* [22]: a popular and robust programming tool for detecting memory bugs based on poisoned red-zones.

For a fairer comparison we disable AddressSanitizer's instrumentation for globals, alloc/free mismatch detection, and leak detection. We choose to compare against AddressSanitizer for several reasons, namely: prominence (actively used by large projects such as Google Chrome [22] and Mozilla Firefox), stability, accessibility (already "built-in" to `clang` [18]), and works "out-of-the-box" (after a patch[10] is applied) on the SPEC 2006 benchmark suite. The tested version of AddressSanitizer does have one notable limitation in that it only instruments fixed-sized stack objects. LowFat instruments both fixed and variable-sized stack objects which may translate to a small performance disadvantage in our tests.

Our LowFat implementation detects all 24 OOB-pointer violations summarized in Section III-C. For the sake of performance testing, we disable instrumentation for the corresponding functions for both LowFat and AddressSanitizer. The only known unintentional SPEC 2006 stack bounds overflow relates to an off-by-one error in the `h264ref` benchmark. Under LowFat this overflow affects allocation padding only—and does not affect other objects. Again for the sake of testing, this error is patched for both LowFat and AddressSanitizer. Finally, out of all the SPEC 2006 benchmarks, only `perlbench` uses the `fork` system call. This is handled using the method described in Section VI-C1.

1) Timings: The timings for the SPEC 2006 benchmarks are shown in Figure 8(a). The original timings and the best instrumented timings are highlighted in **bold**. Here (Total) represents the sum of each column and (Avg. Ratio) the average ratio per benchmark relative to the (Orig) baseline. In addition to the (base) LowFat implementation, we also test the following variants:

- $+alias$: Enables memory aliasing (Section VI-C);
- $+pow2$: Restrict both heap and stack allocations to power-of-two sizes for faster bounds checking (see [10] Section 5.2); and
- $+w.o.$: Instrument memory write operations only.

Optimizations are cumulative left-to-right. Note that both the fixed-object-size optimization (Section VI-A) and the no-escape optimization (Section VI-B) are enabled by default for all LowFat variants. The $+w.o.$ variant reduces instrumentation (no reads) and thus exchanges security for speed. The rationale is that most control and data flow attacks require an OOB write, so the $+w.o.$ variant still provides sufficient defense. That said, the $+w.o.$ variant will not protect against information leakage attacks, such as Heartbleed [12].

The base LowFat implementation exhibits a 62% (63% avg. ratio) overhead compared with the un-instrumented benchmarks (Orig). This is reduced to a 58% overhead when $+alias$ is enabled. The $+alias$ variant benefits from better stack object locality in physical memory; translating in to a $\sim$4% overall reduction in overhead.

The $+pow2$ optimization restricts the size configuration (Sizes) to be powers-of-two for both the heap and stack. This

[10] https://github.com/google/sanitizers/blob/master/address-sanitizer/spec/spec2006-asan.patch

	Orig	LowFat				ASAN	
Bench.	base	base	$+alias$	$+pow2$	$+w.o.$	base	$+w.o.$
perlbench	**310**	494	485	464	**393**	1032	921
bzip2	**479**	829	799	793	**589**	866	633
gcc	**288**	585	580	541	**497**	656	571
mcf	**244**	296	308	299	**252**	401	276
gobmk	**452**	585	571	543	**512**	820	594
hmmer	**423**	1070	1070	987	538	819	**473**
sjeng	**480**	554	548	550	**514**	884	618
libquantum	**321**	364	347	361	**331**	392	353
h264ref	**537**	1070	1072	1003	**611**	1236	734
omnetpp	**306**	480	393	490	**392**	647	578
astar	**393**	642	621	592	**433**	636	449
xalancbmk	**204**	306	290	277	**185**	435	372
milc	**529**	718	659	674	**477**	616	504
namd	**356**	565	568	552	390	563	**376**
dealII	**275**	561	549	499	**349**	621	457
soplex	**217**	312	301	304	**229**	337	279
povray	**142**	299	297	277	**172**	408	277
lbm	**341**	408	402	401	**339**	361	**339**
sphinx3	**482**	870	853	804	693	903	**549**
Total	**6779**	162%	158%	154%	**116%**	186%	138%
Avg. Ratio	**100%**	163%	158%	154%	**117%**	192%	145%

(a) Timings (s)

	Orig	LowFat			ASAN
Bench.	base	base	$+alias$	$+pow2$	base
perlbench	680	650	632	735	2461
bzip2	872	883	869	869	917
gcc	908	928	897	897	3030
mcf	1718	1724	1718	1718	1956
gobmk	31	60	34	34	449
hmmer	28	42	29	29	643
sjeng	180	213	182	182	206
libquantum	100	109	100	100	415
h264ref	67	91	69	74	427
omnetpp	175	189	171	223	943
astar	335	362	348	570	1138
xalancbmk	432	522	511	646	1817
milc	697	712	698	698	1025
namd	49	67	51	51	127
dealII	815	863	843	1047	2214
soplex	443	638	621	621	1001
povray	7	27	9	9	400
lbm	420	427	421	421	496
sphinx3	46	69	48	48	591
Total	8003	107%	103%	112%	253%
Avg. Ratio	100%	135%	106%	115%	797%

(b) Memory usage (MB)

Fig. 8. SPEC2006 benchmark timings and memory usage.

allows for faster bounds check instrumentation that uses bit-masks in place of fixed-point division for calculating the base address of objects. This optimization reduces the overhead to 54% at the cost of higher memory usage (see below). Finally, the $+w.o$ variant significantly reduces the amount of instrumentation leading to an overall 16% (avg. ratio 17%) overhead. This is low enough to be used in production code for some applications.

AddressSanitizer exhibits higher overheads, with 86% (avg. ratio 92%) for base and 38% (avg. ratio 45%) for the $+w.o.$ variant. The LowFat $+w.o.$ variant is faster in 16 out of 19 benchmarks with one benchmark (`lbm`) tied. We note that, for some benchmarks, the overhead of AddressSanitizer is particularly high, e.g., almost 3x slower for `perlbench`.

2) Memory Usage: One of the main advantages of low-fat pointers is that there is no need to explicitly store bounds meta information, meaning that memory overheads are low compared with the un-instrumented code (Orig). That said, the low-fat stack allocator introduces several new sources of memory overheads, namely:

- Extra space for virtual stacks (assuming that the $+alias$ optimization is disabled);
- Allocation size over-approximation (Section IV-C);
- Allocation size alignment (Section IV-D).

The extra overheads are balanced by the fact that stack memory tends to be small (Linux default is $8MB$) and stack objects are typically short-lived. In this section we experimentally evaluate the memory overheads of the low-fat stack allocator, including the overall program memory usage and the precise memory usage for stack memory only.

The results for the overall memory usage are shown in Figure 8(b). For these tests we measure the peak resident set size (RSS), the same method used in [10]. The base LowFat implementation introduces a 7% (35% avg. ratio) memory overhead compared with the un-instrumented baseline. This is further reduced to a 3% (6% avg. ratio) overhead when $+alias$ is enabled. Finally, we see that $+pow2$ trades speed for higher memory overheads, with 12% (15% avg. ratio) overall.

In contrast to LowFat, AddressSanitizer exhibits a very high 153% (avg. ratio 697%) memory overhead overall. This is because AddressSanitizer uses poisoned red-zones which are memory intensive.

In addition to the "big picture" results in Figure 8(b), we also measured the precise stack memory usage overheads for each benchmark. For this experiment, we initialize each page of stack memory with a random nonce during program initialization. Next we measure the number of pages that were altered at program exit. Assuming that memory aliasing optimization was disabled, it was determined that the stack memory usage for the SPEC 2006 benchmarks is 5.20x over the (Orig) baseline. The worst affected benchmark was `xalancbmk`, with an 8.01x increase in stack memory usage. With the $+alias$ optimization enabled, the overall physical stack memory usage is reduced to 1.95x, which is comparable with parallel shadow stacks [8]. For many programs, the space used by the stack is only a small fraction of the total memory usage. Therefore the overall memory overhead for LowFat in Figure 8(b) remains low.

B. Web Server

To test the performance for I/O bound applications, we compiled the Apache HTTP server ($\sim$275K lines of code) using LowFat and compared it with an un-instrumented version (Orig). For this test we transfer a $2GB$ file with the daemon connected to the local host. Two types of tests were performed: one where the file was cached in RAM, and the other where the disk cache was purged. Each test was run for a total of 50 times and results averaged.

Bench.	Orig	LOWFAT
`httpd-2.4.23` (cached)	1.06	1.07
`httpd-2.4.23` (purged)	24.0	24.4

Fig. 9. Apache HTTP server benchmark timings.

The results (in seconds) are shown in Figure 9. In both cases, the overhead of LOWFAT was very low (<2%). In applications such as Apache, which is not primarily CPU-bound, we expect that the overheads of LOWFAT bounds checking to be small.

C. Effectiveness

To test the effectiveness of the low-fat stack allocator we use the Wilander [28] and RIPE [29] benchmarks in addition to some recent CVEs. The results are shown in Figure 10. Here, (sLOC) is the number of source lines of code, (#Test) is the number of test cases, (Abort) is the number of allocation bound overflows that were detected (causing LOWFAT to abort execution), and (Pad) is the number of overflows into padding introduced by allocation size over-approximation. The latter is considered benign for the application of program hardening, i.e., an overflow in padding cannot corrupt code pointers or data values stored in other objects.

The Wilander benchmarks [28] consist of 12 stack-based overflows, all of which are detected by LOWFAT.

The RIPE benchmarks consists of several test cases that combine a bounds overflow error followed by a control flow hijack attack. Some issues were encountered when testing the RIPE benchmarks, namely:

1) RIPE requires 32-bit whereas LOWFAT requires 64-bit; and
2) The RIPE attacks are fragile and break when ported.

To solve these issues we only port (to 64-bit) the RIPE tests such that execution only up to the buffer overflow error is preserved. The rationale is: if the memory error is detected then any subsequent control flow hijack attack will be prevented. The RIPE benchmarks contain 10 unique bounds errors: one direct array overflow (`homebrew`), and 9 errors induced by passing invalid parameters to stdlib functions such as `sprintf`, `sscanf`, etc. For this experiment we recompile the relevant `glibc` functions with LOWFAT instrumentation enabled.[11] We test all RIPE parameter combinations that (1) are stack object overflows, (2) are not reported as "impossible" by the RIPE test framework, and (3) are not sub-object overflows. This yields a total of 70 tests, all of which are detected as OOB-errors by LOWFAT (Abort).

Finally we test several recent (2016 at the time of writing) CVEs relating to stack buffer overflows that are listed in Figure 10. The purpose of these tests is to show that LOWFAT is applicable to real world bugs in addition to artificial tests. The CVEs originate from `glibc` (standard `C` library), `pcre2` (Perl Compatible Regular Expressions) and `php` (a server-side scripting language). As above, for the `glibc` test, we recompile only the function(s) relevant to the error, and this is reflected in the sLOC column from Figure 10. For all five CVEs tested, two are detected (Abort) and three overflow into padding (Pad).

[11] Currently it is not possible to recompile `glibc` in its entirety. This is because `glibc` requires `gcc` whereas LOWFAT is implemented using `clang`.

Bench.	sLOC	#Test	Abort	Pad
Wilander [28]	0.4K	12	12	0
RIPE [29] + `glibc`	6.7K	70	70	0
`CVE-2016-1234` (`glibc-2.19`)	1.4K	1	1	0
`CVE-2016-3191` (`pcre2-10.20`)	73.6K	1	0	1
`CVE-2016-6297` (`php-7.0.3`)	759.5K	1	0	1
`CVE-2016-6289` (`php-7.0.3`)	〃	1	1	0
`CVE-2016-2554` (`php-5.5.31`)	781.6K	1	0	1

Fig. 10. Effectiveness against various benchmarks and CVEs.

D. Comparison with Other Systems

Our system extends [10] with low-fat pointers for stack objects in addition to the heap. Our 58% overhead result for $+alias$ corresponds to the 67% overhead result for $+fdiv$ from [10] Figure 4(a). Despite protecting both the heap and stack our version is faster overall—the result of an improved implementation with better optimization. For the $+w.o.$ variant we are slightly slower, with 17% overhead versus 13%. The overall memory performance is similar.

Our results are also competitive with other bounds instrumentation systems. PAriCheck [30] and BaggyBounds [3] report 49% and 60% overheads respectively for the SPEC 2000 benchmarks. Note that this paper uses the SPEC 2006 benchmarks so the results are not directly comparable. Furthermore, unlike LOWFAT, both PAriCheck and BaggyBounds do not instrument field access, resulting in less bounds checking. SoftBound [19] reports a similar 67% performance overhead for SPEC 2000, but with a significantly higher 64% memory overhead. For SPEC 2006, CPI reports that they could only compile four benchmarks with SoftBound and the time overheads range between 60–249% [16]. This also highlights the importance of compatibility issues. Alternatives to bounds checking, such as shadow-stacks, CPI, CPS, SafeStack [8], [16], tend to have lower performance overheads, ranging from 0.1%–10% depending on the solution. That said, none of these solutions prevent memory errors. Rather, such solutions aim to mitigate any subsequent control flow attack. The LOWFAT $+w.o.$ variant with a 17% overhead can protect against control flow attacks in addition to other kinds of attacks, such as data flow [6]. This, combined with high compatibility, makes LOWFAT a competitive solution.

VIII. CONCLUSION

Object bounds errors are a common source of security vulnerabilities and bounds check instrumentation with low-fat pointers (with low overheads and high compatibility) is an attractive solution. However, low-fat pointers require sufficient control over object allocation, and as such, previous work was limited to heap objects only. In this paper, we have shown how to extended low-fat pointers to stack objects by using a combination of techniques, including: fast allocation size over-approximation, dynamic stack object alignment, stack object pointer mirroring, and the memory aliasing optimization. We show that the new low-fat stack allocator is compatible with existing software and binary code. Our experiments show that the overall performance and memory impact of stack object

protection is minimal over the previously published results [10] for heap only. Furthermore, for protecting memory writes only, the overhead drops to 17%, which is low enough for enabling real-world deployments while still preventing many attacks.

With low-fat pointers extended to both heap and stack objects, the remaining class yet to be covered is globals. In principle, low-fat global objects could be realized by further splitting each low-fat region to also include a *global sub-region*, in addition to the heap and stack sub-regions. Global objects are then placed in the appropriate sub-region (based on allocation size) by the linker. Modifying the static and dynamic linkers to be "low-fat aware" is left as future work.

REFERENCES

[1] "CVE-2016-1234, CVE-2016-3191, CVE-2016-6297, CVE-2016-6289."

[2] M. Abadi, M. Budiu, Z. Erlingsson, and J. Ligatti, "Control-Flow Integrity," in *Computer and Communication Security*. ACM, 2005.

[3] 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 Symposium*. USENIX, 2009.

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

[5] M. Castro, M. Costa, and T. Harris, "Securing Software by Enforcing Data-flow Integrity," in *USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 2006.

[6] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer, "Non-control-data Attacks are Realistic Threats," in *USENIX Security Symposium*. USENIX, 2005.

[7] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries," in *Network and Distributed System Security Symposium*. The Internet Society, 2015.

[8] T. Dang, P. Maniatis, and D. Wagner, "The Performance Cost of Shadow Stacks and Stack Canaries," in *ACM Symposium on Information, Computer and Communications Security*. ACM, 2015.

[9] B. Ding, Y. He, Y. Wu, A. Miller, and J. Criswell, "Baggy Bounds with Accurate Checking," in *International Symposium on Software Reliability Engineering Workshops*. IEEE Computer Society, 2012.

[10] G. Duck and R. Yap, "Heap Bounds Protection with Low Fat Pointers," in *Compiler Construction*. ACM, 2016.

[11] F. Eigler, "Mudflap: Pointer Use Checking for C/C++," in *GCC Developer's Summit*, 2003.

[12] Heartbleed bug, `http://heartbleed.com`, 2016.

[13] H. Hu, S. Shinde, S. Adrian, Z. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2016.

[14] Intel Corporation, "Intel 64 and IA-32 Architectures Software Developers Manual," 2016.

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

[16] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer Integrity," in *USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 2014.

[17] 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.

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

[19] 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.

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

[21] PaX, "Address Space Layout Randomization," http://pax.grsecurity.net/docs/aslr.txt.

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

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

[24] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013.

[25] Tor, "The Tor Project," https://www.torproject.org/.

[26] 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.

[27] A. Ven, "New Security Enhancements in Red Hat Enterprise Linux," http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf.

[28] J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," in *Network and Distributed System Security Symposium*. The Internet Society, 2003.

[29] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime Intrusion Prevention Evaluator," in *Annual Computer Security Applications Conference*. ACM, 2011.

[30] 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.

[31] Y. Younan, D. Pozza, F. Piessens, and W. Joosen, "Extended Protection Against Stack Smashing Attacks Without Performance Loss," in *Annual Computer Security Applications Conference*. IEEE Computer Society, 2006.

APPENDIX

A. Parameters

This paper uses the following low-fat allocation parameters:

$$\texttt{REGION\_SIZE} = 32GB$$

$$M = |\mathsf{Sizes}| = 61$$

$$\mathsf{Sizes} = \begin{array}{l}\langle 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 192, 224, 256,\\ 272, 320, 384, 448, 512, 528, 640, 768, 896, 1024, 1040,\\ 1280, 1536, 1792, 2048, 2064, 2560, 3072, 3584, 4096,\\ 4112, 5120, 6144, 7168, 8192, 8208, 10240, 12288,\\ 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB,\\ 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB,\\ 256MB, 512MB, 1GB, 2GB, 4GB, 8GB\rangle\end{array}$$

$$\mathsf{StkSizes} = \begin{array}{l}\langle 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192,\\ 16KB, 32KB, 64KB, 128KB, 256KB, 512KB,\\ 1MB, 2MB, 4MB, 8MB, 16MB,\\ 32MB, 64MB, 128MB, 256MB, 512MB\\ 1GB, 2GB, 4GB, 8GB\rangle\end{array}$$