Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors

Periklis Akritidis,[⋆] *Manuel Costa,*[†] *Miguel Castro,*[†] *Steven Hand*[⋆]

[⋆]*Computer Laboratory*
University of Cambridge, UK
{pa280,smh22}@cl.cam.ac.uk

[†]*Microsoft Research*
Cambridge, UK
{manuelc,mcastro}@microsoft.com

Abstract

Attacks that exploit out-of-bounds errors in C and C++ programs are still prevalent despite many years of research on bounds checking. Previous backwards compatible bounds checking techniques, which can be applied to unmodified C and C++ programs, maintain a data structure with the bounds for each allocated object and perform lookups in this data structure to check if pointers remain within bounds. This data structure can grow large and the lookups are expensive.

In this paper we present a backwards compatible bounds checking technique that substantially reduces performance overhead. The key insight is to constrain the sizes of allocated memory regions and their alignment to enable efficient bounds lookups and hence efficient bounds checks at runtime. Our technique has low overhead in practice—only 8% throughput decrease for Apache—and is more than two times faster than the fastest previous technique and about five times faster—using less memory—than recording object bounds using a splay tree.

1 Introduction

Bounds checking C and C++ code protects against a wide range of common vulnerabilities. The challenge has been making bounds checking fast enough for production use and at the same time backwards compatible with binary libraries to allow incremental deployment. Solutions using *fat pointers* [24, 18] extend the pointer representation with bounds information. This enables efficient bounds checks but breaks backwards compatibility because increasing the pointer size changes the memory layout of data structures. Backwards compatible bounds checking techniques [19, 30, 36, 15] use a separate data structure to lookup bounds information. Initial attempts incurred a significant overhead [19, 30, 36] (typically 2x–10x) because looking up bounds is expensive and the data structure can grow large. More recent work [15] has applied sophisticated static pointer analysis to reduce the number of bounds lookups; this managed to reduce the runtime overhead on the Olden benchmarks to 12% on average.

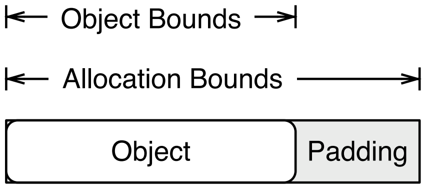


Figure 1: Allocated memory is often padded to a particular alignment boundary, and hence can be larger than the requested object size. By checking *allocation bounds* rather than object bounds, we allow benign accesses to the padding, but can significantly reduce the cost of bounds lookups at runtime.

In this paper we present *baggy bounds checking*, a backwards compatible bounds checking technique that reduces the cost of bounds checks. We achieve this by enforcing *allocation bounds* rather than precise object bounds, as shown in Figure 1. Since memory allocators pad object allocations to align the pointers they return, there is a class of benign out-of-bounds errors that violate the object bounds but fall within the allocation bounds. Previous work [4, 19, 2] has exploited this property in a variety of ways.

Here we apply it to efficient backwards compatible bounds checking. We use a binary buddy allocator to enable a compact representation of the allocation bounds: since all allocation sizes are powers of two, a single byte is sufficient to store the binary logarithm of the allocation size. Furthermore, there is no need to store additional information because the base address of an allocation with size s can be computed by clearing the $log_2(s)$ least significant bits of any pointer to the allocated region. This allows us to use a space and time efficient data structure for the bounds table. We use a contiguous array instead of a more expensive data structure (such as the splay trees used in previous work). It also provides us with an elegant way to deal with common cases of temporarily out-of-bounds pointers. We describe our design in more detail in Section 2.

We implemented *baggy bounds checking* as a compiler plug-in for the Microsoft Phoenix [22] code generation framework, along with additional run time components (Section 3). The plug-in inserts code to check bounds for all pointer arithmetic that cannot be statically proven safe, and to align and pad stack variables where necessary. The run time component includes a binary buddy allocator for heap allocations, and user-space virtual memory handlers for growing the bounds table on demand.

In Section 4 we evaluate the performance of our system using the Olden benchmark (to enable a direct comparison with Dhurjati and Adve [15]) and SPECINT 2000. We compare our space overhead with a version of our system that uses the splay tree implementation from [19, 30]. We also verify the efficacy of our system in preventing attacks using the test suite described in [34], and run a number of security critical COTS components to confirm its applicability.

Section 5 describes our design and implementation for 64-bit architectures. These architectures typically have “spare” bits within pointers, and we describe a scheme that uses these to encode bounds information directly in the pointer rather than using a separate lookup table. Our comparative evaluation shows that the performance benefit of using these spare bits to encode bounds may not in general justify the additional complexity; however using them just to encode information to recover the bounds for out-of-bounds pointers may be worthwhile.

Finally we survey related work (Section 6), discuss limitations and possible future work (Section 7) and conclude (Section 8).

2 Design

2.1 Baggy Bounds Checking

Our system shares the overall architecture of backwards compatible bounds checking systems for C/C++ (Figure 2). It converts source code to an intermediate representation (IR), finds potentially unsafe pointer arithmetic operations, and inserts checks to ensure their results are within bounds. Then, it links the generated code with our runtime library and binary libraries—compiled with or without checks—to create a hardened executable.

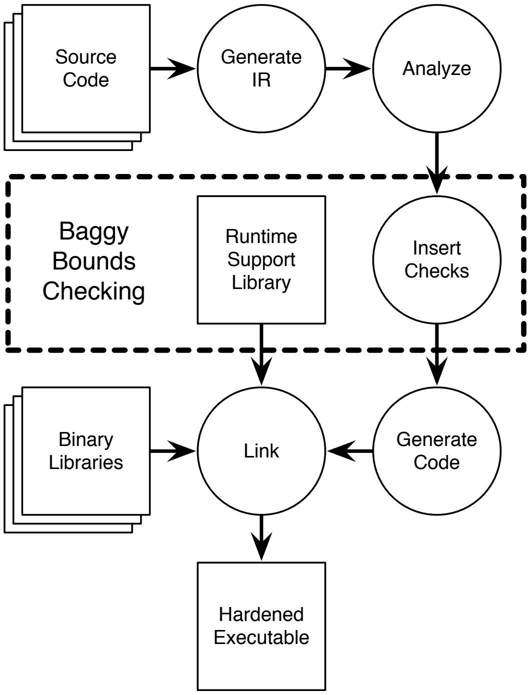


Figure 2: Overall system architecture, with our contribution highlighted within the dashed box.

We use the *referent object* approach for bounds checking introduced by Jones and Kelly [19]. Given an in-bounds pointer to an object, this approach ensures that any derived pointer points to the same object. It records bounds information for each object in a *bounds table*. This table is updated on allocation and deallocation of objects: this is done by the `malloc` family of functions for heap-based objects; on function entry and exit for stack-based objects; and on program startup for global objects.

The referent object approach performs bounds checks on pointer arithmetic. It uses the source pointer to lookup the bounds in the table, performs the operation, and checks if the destination pointer remains in bounds. If the destination pointer does not point to the same object, we mark it out-of-bounds to prevent any dereference (as in [30, 15]). However we permit its use in further pointer arithmetic, since it may ultimately result in an in-bounds pointer. The marking mechanism is described in detail in Section 2.4.

Baggy bounds checking uses a very compact representation for bounds information. Previous techniques recorded a pointer to the start of the object and its size in the bounds table, which requires at least eight bytes. We pad and align objects to powers of two and enforce allocation bounds instead of object bounds. This enables us to use a single byte to encode bounds information. We store the binary logarithm of the allocation size in the bounds table:

```
e = log2(size);
```

Given this information, we can recover the allocation size and a pointer to the start of the allocation with:

```
size = 1 << e;

base = p & ~(size-1);
```

To convert from an in-bounds pointer to the bounds for the object we require a *bounds table*. Previous solutions based on the referent object approach (such as [19, 30, 15]) have implemented the bounds table using a splay tree.

Baggy bounds, by contrast, implement the bounds table using a contiguous array. The table is small because each entry uses a single byte. Additionally, we partition memory into aligned *slots* with $slot\_size$ bytes. The bounds table has an entry for each slot rather than an entry per byte. So the space overhead of the table is $1/slot\_size$, and we can tune $slot\_size$ to balance memory waste between padding and table size. We align objects to slot boundaries to ensure that no two objects share a slot.

Accesses to the table are fast. To obtain a pointer to the entry corresponding to an address, we right-shift the address by the constant $log_2(slot\_size)$ and add the constant table base. We can use this pointer to retrieve the bounds information with a single memory access, instead of having to traverse and splay a splay tree (as in previous solutions).

Note that baggy bounds checking permits benign out-of-bounds accesses to the memory padding after an object. This does not compromise security because these accesses cannot write or read other objects. They cannot be exploited for typical attacks such as (*a*) overwriting a return address, function pointer or other security critical data; or (*b*) reading sensitive information from another object, such as a password.

We also defend against a less obvious attack where the program reads values from the padding area that were originally written to a deleted object that occupied the same memory. We prevent this attack by clearing the padding on memory allocation.

Pointer arithmetic operation:

```
p' = p + i
```

Explicit bounds check:

```
size = 1 << table[p>>slot_size]
base = p & ~(size-1)

p' >= base && p' - base < size
```

Optimized bounds check:

```
(p^p')>>table[p>>slot_size] == 0
```

Figure 3: Baggy bounds enables optimized bounds checks: we can verify that pointer p' derived from pointer p is within bounds by simply checking that p and p' have the same prefix with only the e least significant bits modified, where e is the binary logarithm of the allocation size.

2.2 Efficient Checks

In general, bounds checking the result p' of pointer arithmetic on p involves two comparisons: one against the lower bound and one against the upper bound, as shown in Figure 3.

We devised an optimized bounds check that does not even need to compute the lower and upper bounds. It uses the value of p and the value of the binary logarithm of the allocation size, e, retrieved from the bounds table. The constraints on allocation size and alignment ensure that p' is within the allocation bounds if it differs from p only in the e least significant bits. Therefore, it is sufficient to shift p^p' by e and check if the result is zero, as shown in Figure 3.

Furthermore, for pointers p' where sizeof(*p') > 1, we also need to check that (char *) p' + sizeof(*p') - 1 is within bounds to prevent a subsequent access to *p' from crossing the allocation bounds. Baggy bounds checking can avoid this extra check if p' points to a built-in type. Aligned accesses to these types cannot overlap an allocation boundary because their size is a power of two and is less than $slot\_size$. When checking pointers to structures that do not satisfy these constraints, we perform both checks.

2.3 Interoperability

Baggy bounds checking works even when instrumented code is linked against libraries that are not instrumented.

The library code works without change because it performs no checks but it is necessary to ensure that instrumented code works when accessing memory allocated in an uninstrumented library. This form of interoperability is important because some libraries are distributed in binary form.

We achieve interoperability by using the binary logarithm of the maximum allocation size as the default value for bounds table entries. Instrumented code overwrites the default value on allocations with the logarithm of the allocation size and restores the default value on deallocations. This ensures that table entries for objects allocated in uninstrumented libraries inherit the default value. Therefore, instrumented code can perform checks as normal when accessing memory allocated in a library, but checking is effectively disabled for these accesses. We could intercept heap allocations in library code at link time and use the buddy allocator to enable bounds checks on accesses to library-allocated memory, but this is not done in the current prototype.

2.4 Support for Out-Of-Bounds Pointers

A pointer may legally point outside the object bounds in C. Such pointers should not be dereferenced but can be compared and used in pointer arithmetic that can eventually result in a valid pointer that may be dereferenced by the program.

Out-of-bounds pointers present a challenge for the referent object approach because it relies on an in-bounds pointer to retrieve the object bounds. The C standard only allows out-of-bounds pointers to one element past the end of an array. Jones and Kelly [19] support these legal out-of-bounds pointers by padding objects with one byte. We did not use this technique because it interacts poorly with our constraints on allocation sizes: adding one byte to an allocation can double the allocated size in the common case where the requested allocation size is a power of two.

Many programs violate the C standard and generate illegal but harmless out-of-bounds pointers that they never dereference. Examples include faking a base one array by decrementing the pointer returned by `malloc` and other equally tasteless uses. CRED [30] improved on the Jones and Kelly bounds checker [19] by tracking such pointers using another auxiliary data structure. We did not use this approach because it adds overhead on deallocations of heap and local objects: when an object is deallocated the auxiliary data structure must be searched to remove entries tracking out-of-bounds pointers to the object. Additionally, entries in this auxiliary data structure may accumulate until their referent object is deallocated.

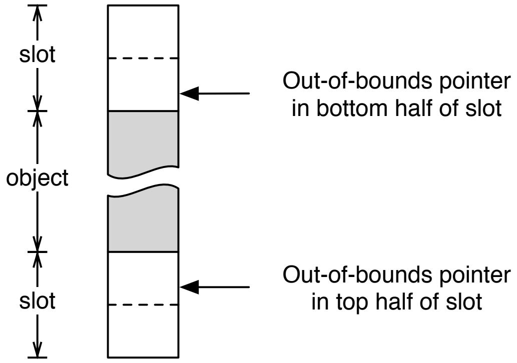

slot
object
slot
Out-of-bounds pointer in bottom half of slot
Out-of-bounds pointer in top half of slot

Figure 4: We can tell whether a pointer that is out-of-bounds by less than $slot\_size/2$ is below or above an allocation. This lets us correctly adjust it to get a pointer to the object by respectively adding or subtracting $slot\_size$.

We handle out-of-bounds pointers within $slot\_size/2$ bytes from the original object as follows. First, we mark out-of-bounds pointers to prevent them from being dereferenced (as in [15]). We use the memory protection hardware to prevent dereferences by setting the most significant bit in these pointers and by restricting the program to the lower half of the address space (this is often already the case for user-space programs). We can recover the original pointer by clearing the bit.

The next challenge is to recover a pointer to the referent object from the out-of-bounds pointer without resorting to an additional data structure. We can do this for the common case when out-of-bounds pointers are at most $slot\_size/2$ bytes before or after the allocation. Since the allocation bounds are aligned to slot boundaries, we can find if a marked pointer is below or above the allocation by checking whether it lies in the top or bottom half of a memory slot respectively, as illustrated in Figure 4. We can recover a pointer to the referent object by adding or subtracting $slot\_size$ bytes. This technique cannot handle pointers that go more than $slot\_size/2$ bytes outside the original object. In Section 5.2, we show how to take advantage of the spare bits in pointers on 64 bit architectures to increase this range, and in Section 7 we discuss how we could add support for arbitrary out-of-bounds pointers while avoiding some of the problems of previous solutions.

It is not necessary to instrument pointer dereferences. Similarly, there is no need to instrument pointer equality comparisons because the comparison will be correct whether the pointers are out-of-bounds or not. But we need to instrument inequality comparisons to support comparing an out-of-bounds pointer with an in-bounds one: the instrumentation must clear the high-order bit of the pointers before comparing them. We also instrument pointer differences in the same way.

Like previous bounds checking solutions [19, 30, 15], we do not support passing an out-of-bounds pointer to uninstrumented code. However, this case is rare. Previous work [30] did not encounter this case in several million lines of code.

2.5 Static Analysis

Bounds checking has relied heavily on static analysis to optimize performance [15]. Checks can be eliminated if it can be statically determined that a pointer is safe, i.e. always within bounds, or that a check is redundant due to a previous check. Furthermore, checks or just the bounds lookup can be hoisted out of loops. We have not implemented a sophisticated analysis and, instead, focused on making checks efficient.

Nevertheless, our prototype implements a simple intraprocedural analysis to detect safe pointer operations. We track allocation sizes and use the compiler's variable range analysis to eliminate checks that are statically shown to be within bounds. We also investigate an approach to hoist checks out of loops that is described in Section 3.

We also use static analysis to reduce the number of local variables that are padded and aligned. We only pad and align local variables that are indexed unsafely within the function, or whose address is taken, and therefore possibly leaked from the function. We call these variables *unsafe*.

3 Implementation

We used the Microsoft Phoenix [22] code generation framework to implement a prototype system for x86 machines running Microsoft Windows. The system consists of a plug-in to the Phoenix compiler and a runtime support library. In the rest of this section, we describe some implementation details.

3.1 Bounds Table

We chose a $slot\_size$ of 16 bytes to avoid penalizing small allocations. Therefore, we reserve $1/16^{th}$ of the address space for the bounds table. Since pages are allocated to the table on demand, this increases memory utilization by only 6.25%. We reserve the address space required for the bounds table on program startup and install a user space page fault handler to allocate missing table pages on demand. All the bytes in these pages are initialized by the handler to the value 31, which encompasses all the addressable memory in the x86 (an allocation size of 2^{31} at base address 0). This prevents out-of-bounds errors when instrumented code accesses memory allocated by uninstrumented code.

3.2 Padding and Aligning

We use a binary buddy allocator to satisfy the size and alignment constraints on heap allocations. Binary buddy allocators provide low external fragmentation but suffer from internal fragmentation because they round allocation sizes to powers of two. This shortcoming is put to good use in our system. Our buddy allocator implementation supports a minimum allocation size of 16 bytes, which matches our $slot\_size$ parameter, to ensure that no two objects share the same slot.

We instrument the program to use our version of `malloc`-style heap allocation functions based on the buddy allocator. These functions set the corresponding bounds table entries and zero the padding area after an object. For local variables, we align the stack frames of functions that contain unsafe local variables at runtime and we instrument the function entry to zero the padding and update the appropriate bounds table entries. We also instrument function exit to reset table entries to 31 for interoperability when uninstrumented code reuses stack memory. We align and pad static variables at compile time and their bounds table entries are initialized when the program starts up.

Unsafe function arguments are problematic because padding and aligning them would violate the calling convention. Instead, we copy them on function entry to appropriately aligned and padded local variables and we change all references to use the copies (except for uses of `va_list` that need the address of the last explicit argument to correctly extract subsequent arguments). This preserves the calling convention while enabling bounds checking for function arguments.

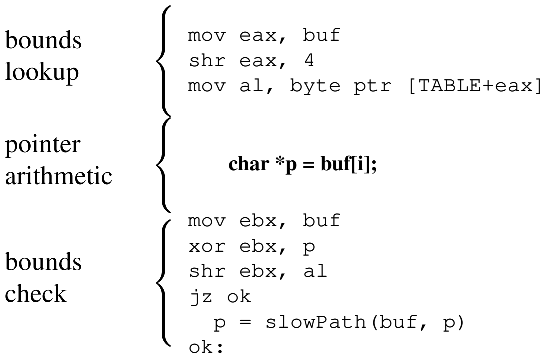
The Windows runtime cannot align stack objects to more than 8K nor static objects to more than 4K (configurable using the `/ALIGN` linker switch). We could replace these large stack and static allocations with heap allocations to remove this limitation but our current prototype sets the bounds table entries for these objects to 31.

Zeroing the padding after an object can increase space and time overhead for large padding areas. We avoid this overhead by relying on the operating system to zero allocated pages on demand. Then we track the subset of these pages that is modified and we zero padding areas in these pages on allocations. Similar issues are discussed in [9] and the standard allocator uses a similar technique for `calloc`. Our buddy allocator also uses this technique to avoid explicitly zeroing large memory areas allocated with `calloc`.

3.3 Checks

We add checks for each pointer arithmetic and array indexing operation but, following [15], we do not instrument accesses to scalar fields in structures and we do not check pointer dereferences. This facilitates a direct comparison with [15]. We could easily modify our implementation to perform these checks, for example, using the technique described in [14].

We optimize bounds checks for the common case of in-bounds pointers. To avoid checking if a pointer is marked out-of-bounds in the fast path, we set all the entries in the bounds table that correspond to out-of-bounds pointers to zero. Since out-of-bounds pointers have their most significant bit set, we implement this by mapping all the virtual memory pages in the top half of the bounds table to a shared zero page. This ensures that our slow path handler is invoked on any arithmetic operation involving a pointer marked out-of-bounds.



```
mov eax, buf
shr eax, 4
mov al, byte ptr [TABLE+eax]

    char *p = buf[i];

mov ebx, buf
xor ebx, p
shr ebx, al
jz ok
  p = slowPath(buf, p)
ok:
```

Figure 5: Code sequence inserted to check unsafe pointer arithmetic.

Figure 5 shows the `x86` code sequence that we insert before an example pointer arithmetic operation. First, the source pointer, `buf`, is right shifted to obtain the index of the bounds table entry for the corresponding slot. Then the logarithm of the allocation size e is loaded from the bounds table into register `al`. The result of the pointer arithmetic, `p`, is xored with the source pointer, `buf`, and right shifted by `al` to discard the bottom bits. If `buf` and `p` are both within the allocation bounds they can only differ in the $log_2 e$ least significant bits (as discussed before). So if the zero flag is set, `p` is within the allocation bounds. Otherwise, the `slowPath` function is called.

The `slowPath` function starts by checking if `buf` has been marked out-of-bounds. In this case, it obtains the referent object as described in 2.4, resets the most significant bit in `p`, and returns the result if it is within bounds. Otherwise, the result is out-of-bounds. If the result is out-of-bounds by more than half a slot, the function signals an error. Otherwise, it marks the result out-of-bounds and returns it. Any attempt to dereference the returned pointer will trigger an exception. To avoid disturbing register allocation in the fast path, the `slowPath` function uses a special calling convention that saves and restores all registers.

As discussed in Section 3.3, we must add `sizeof(*p)` to the result and perform a second check if the pointer is not a pointer to a built-in type. In this case, `buf` is a `char*`.

Similar to previous work, we provide bounds checking wrappers for Standard C Library functions such as `strcpy` and `memcpy` that operate on pointers. We replace during instrumentation calls to these functions with calls to their wrappers.

3.4 Optimizations

Typical optimizations used with bounds checking include eliminating redundant checks, hoisting checks out of loops, or hoisting just bounds table lookups out of loops. Optimization of inner loops can have a dramatic impact on performance. We experimented with hoisting bounds table lookups out of loops when all accesses inside a loop body are to the same object. Unfortunately, performance did not improve significantly, probably because our bounds lookups are inexpensive and hoisting can adversely effect register allocation.

Hoisting the whole check out of a loop is preferable when static analysis can determine symbolic bounds on the pointer values in the loop body. However, hoisting out the check is only possible if the analysis can determine that these bounds are guaranteed to be reached in every execution. Figure 6 shows an example where the loop bounds are easy to determine but the loop may terminate before reaching the upper bound. Hoisting out the check would trigger a false alarm in runs where the loop exits before violating the bounds.

We experimented with an approach that generates two versions of the loop code, one with checks and one without. We switch between the two versions on loop entry.

In the example of Figure 6, we lookup the bounds of p and if n does not exceed the size we run the unchecked version of the loop. Otherwise, we run the checked version.

```
    for (i = 0; i < n; i++) {
        if (p[i] == 0) break;
        ASSERT(IN_BOUNDS(p, &p[i]));
        p[i] = 0;
    }
```

↓

```
if (IN_BOUNDS(p, &p[n-1])) {
    for (i = 0; i < n; i++) {
        if (p[i] == 0) break;
        p[i] = 0;
    }
} else {
    for (i = 0; i < n; i++) {
        if (p[i] == 0) break;
        ASSERT(IN_BOUNDS(p, &p[i]));
        p[i] = 0;
    }
}
```

Figure 6: The compiler's range analysis can determine that the range of variable i is at most $0 \ldots n-1$. However, the loop may exit before i reaches $n-1$. To prevent erroneously raising an error, we fall back to an instrumented version of the loop if the hoisted check fails.

4 Experimental Evaluation

In this section we evaluate the performance of our system using CPU intensive benchmarks, its effectiveness in preventing attacks using a buffer overflow suite, and its usability by building and measuring the performance of real world security critical code.

4.1 Performance

We evaluate the time and peak memory overhead of our system using the Olden benchmarks and SPECINT 2000. We chose these benchmarks in part to allow a comparison against results reported for some other solutions [15, 36, 23]. In addition, to enable a more detailed comparison with splay-tree-based approaches—including measuring their space overhead—we implemented a variant of our approach which uses the splay tree code from previous systems [19, 30]. This implementation uses the standard allocator and is lacking support for illegal out-of-bounds pointers, but is otherwise identical to our system. We compiled all benchmarks with the Phoenix compiler using `/O2` optimization level and ran them on a 2.33 GHz Intel Core 2 Duo processor with 2 GB of RAM.

From SPECINT 2000 we excluded `eon` since it uses C++ which we do not yet support. For our splay-tree-based implementation only we did not run `vpr` due to its lack of support for illegal out-of-bounds pointers. We also could not run `gcc` because of code that subtracted a pointer from a NULL pointer and subtracted the result from NULL again to recover the pointer. Running this would require more comprehensive support for out-of-bounds pointers (such as that described in [30], as we propose in Section 7).

We made the following modifications to some of the benchmarks: First, we modified `parser` from SPECINT 2000 to fix an overflow that triggered a bound error when using the splay tree. It did not trigger an error with baggy bounds checking because in our runs the overflow was entirely contained in the allocation, but should it overlap another object during a run, the baggy checking would detect it. The unchecked program also survived our runs because the object was small enough for the overflow to be contained even in the padding added by the standard allocator.

Then, we had to modify `perlbmk` by changing two lines to prevent an out-of-bounds arithmetic whose result is never used and `gap` by changing 5 lines to avoid an out-of-bounds pointer. Both cases can be handled by the extension described in Section 5, but are not covered by the small out-of-bounds range supported by our 32-bit implementation and the splay-tree-based implementation.

Finally, we modified `mst` from Olden to disable a custom allocator that allocates 32 Kbyte chunks of memory at a time that are then broken down to 12 byte objects. This increases protection at the cost of memory allocation overhead and removes an unfair advantage for the splay tree whose time and space overheads are minimized when the tree contains just a few nodes, as well as baggy space overhead that benefits from the power of two allocation. This issue, shared with other systems offering protection at the memory block level [19, 30, 36, 15, 2], illustrates a frequent situation in C programs that may require tweaking memory allocation routines in the source code to take full advantage of checking. In this case merely changing a macro definition was sufficient.

We first ran the benchmarks replacing the standard allocator with our buddy system allocator to isolate its effects on performance, and then we ran them using our full system. For the Olden benchmarks, Figure 7 shows the execution time and Figure 8 the peak memory usage.

In Figure 7 we observe that some benchmarks in the Olden suite (`mst`, `health`) run significantly faster with

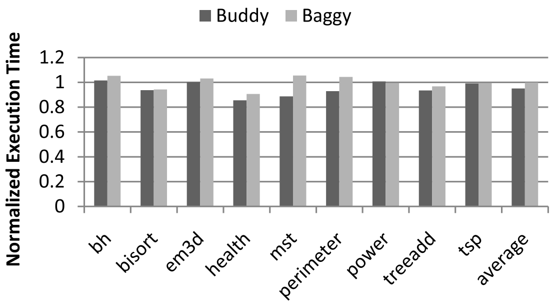


Figure 7: Execution time for the Olden benchmarks using the buddy allocator and our full system, normalized by the execution time using the standard system allocator without instrumentation.

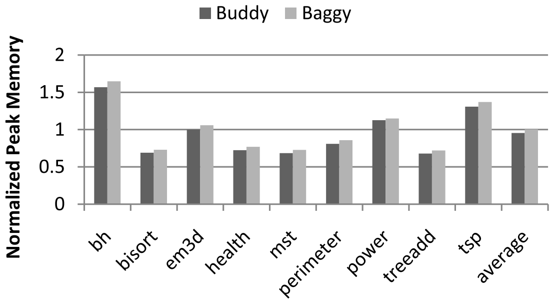


Figure 8: Peak memory use with the buddy allocator alone and with the full system for the Olden benchmarks, normalized by peak memory using the standard allocator without instrumentation.

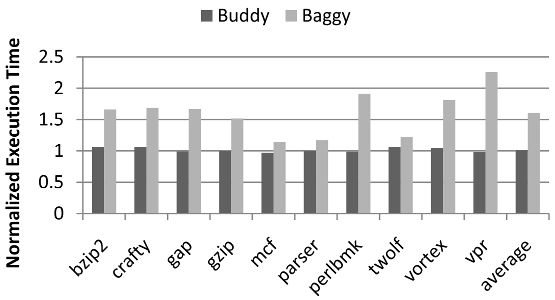


Figure 9: Execution time for SPECINT 2000 benchmarks using the buddy allocator and our full system, normalized by the execution time using the standard system allocator without instrumentation.

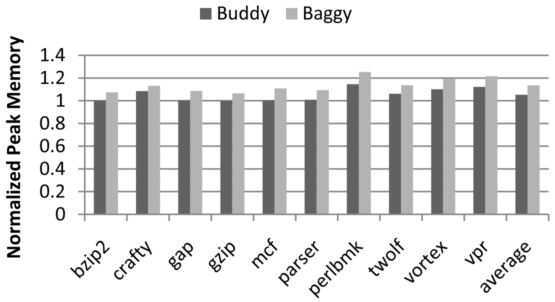


Figure 10: Peak memory use with the buddy allocator alone and with the full system for SPECINT 2000 benchmarks, normalized by peak memory using the standard allocator without instrumentation.

the buddy allocator than with the standard one. These benchmarks are memory intensive and any memory savings reflect on the running time. In Figure 8 we can see that the buddy system uses less memory for these than the standard allocator. This is because these benchmarks contain numerous small allocations for which the padding to satisfy alignment requirements and the per-allocation metadata used by the standard allocator exceed the internal fragmentation of the buddy system.

This means that the average time overhead of the full system across the entire Olden suite is actually zero, because the positive effects of using the buddy allocator mask the costs of checks. The time overhead of the checks alone as measured against the buddy allocator as a baseline is 6%. The overhead of the fastest previous bounds checking system [15] on the same benchmarks and same protection (modulo allocation vs. object bounds) is 12%, but their system also benefits from the technique of pool allocation which can also be used independently. Based on the breakdown of results reported in [15], their overhead measured against the pool allocation is 15%, and it seems more reasonable to compare these two numbers, as both the buddy allocator and pool allocation can be in principle applied independently on either system.

Next we measured the system using the SPECINT 2000 benchmarks. Figures 9 and 10 show the time and space overheads for SPECINT 2000 benchmarks.

We observe that the use of the buddy system has little effect on performance in average. The average runtime overhead of the full system with the benchmarks from SPECINT 2000 is 60%. `vpr` has the highest overhead of 127% because its frequent use of illegal pointers to fake base-one arrays invokes our slow path. We observed that adjusting the allocator to pad each allocation with 8 bytes from below, decreases the time overhead to 53% with only 5% added to the memory usage, although in general we are not interested in tuning the benchmarks like this. Interestingly, the overhead for `mcf` is a mere 16% compared to the 185% in [36] but the overhead of `gzip` is 55% compared to 15% in [36]. Such differences in performance are due to different levels of protection such as checking structure field indexing and checking dereferences, the effectiveness of different static analysis implementations in optimizing away checks, and the

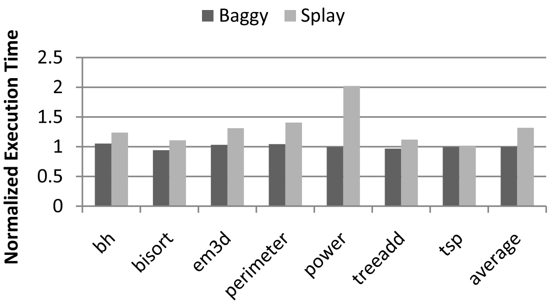


Figure 11: Execution time of baggy bounds checking versus using a splay tree for the Olden benchmark suite, normalized by the execution time using the standard system allocator without instrumentation. Benchmarks `mst` and `health` used too much memory and thrashed so their execution times are excluded.

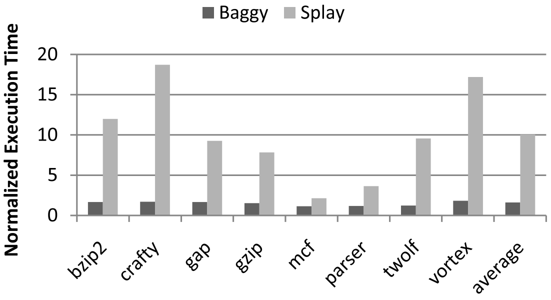


Figure 12: Execution time of baggy bounds checking versus using a splay tree for SPECINT 2000 benchmarks, normalized by the execution time using the standard system allocator without instrumentation.

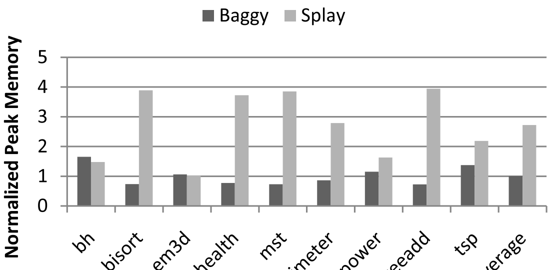


Figure 13: Peak memory use of baggy bounds checking versus using a splay tree for the Olden benchmark suite, normalized by peak memory using the standard allocator without instrumentation.

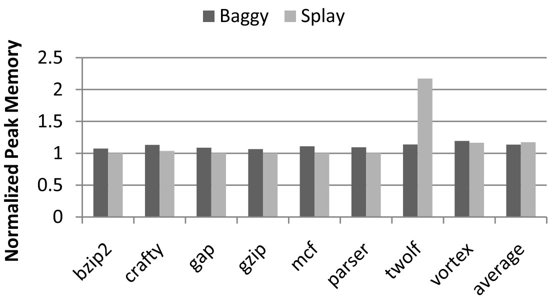


Figure 14: Peak memory use of baggy bounds checking versus using a splay tree for SPECINT 2000 benchmarks, normalized by peak memory using the standard allocator without instrumentation.

different compilers used.

To isolate these effects, we also measured our system using the standard memory allocator and the splay tree implementation from previous systems [19, 30]. Figure 11 shows the time overhead for baggy bounds versus using a splay tree for the Olden benchmarks. The splay tree runs out of physical memory for the last two Olden benchmarks (`mst`, `health`) and slows down to a crawl, so we exclude them from the average of 30% for the splay tree. Figure 12 compares the time overhead against using a splay tree for the SPECINT 2000 benchmarks. The overhead of the splay tree exceeds 100% for all benchmarks, with an average of 900% compared to the average of 60% for baggy bounds checking.

Perhaps the most interesting result of our evaluation was space overhead. Previous solutions [19, 30, 15] do not report on the memory overheads of using splay trees, so we measured the memory overhead of our system using splay trees and compared it with the memory overhead of baggy bounds. Figure 13 shows that our system had negligible memory overhead for Olden, as opposed to the splay tree version's 170% overhead. Clearly Olden's numerous small allocations stress the splay tree by forcing it to allocate an entry for each.

Indeed, we see in Figure 14 that its space overhead for most SPECINT 2000 benchmarks is very low. Nevertheless, the overhead of 15% for baggy bounds is less than the 20% average of the splay tree. Furthermore, the potential worst case of double the memory was not encountered for baggy bounds in any of our experiments, while the splay tree did exhibit greater than 100% overhead for one benchmark (`twolf`).

The memory overhead is also low, as expected, compared to approaches that track meta data for each pointer. Xu *et al.* [36] report 331% for Olden, and Nagarakatte *et al.* [23] report an average of 87% using a hash-table (and 64% using a contiguous array) over Olden and a subset of SPECINT and SPECFP, but more than about 260% (or about 170% using the array) for the pointer intensive Olden benchmarks alone. These systems suffer memory overheads per pointer in order to provide optional temporal protection [36] and sub-object protection [23] and

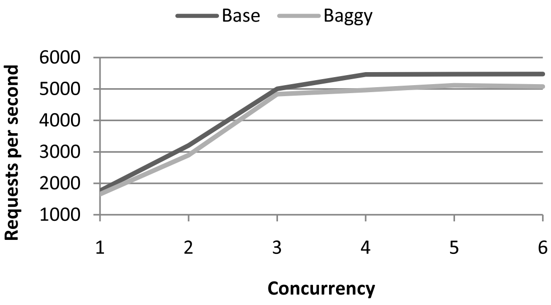


Figure 15: Throughput of Apache web server for varying numbers of concurrent requests.

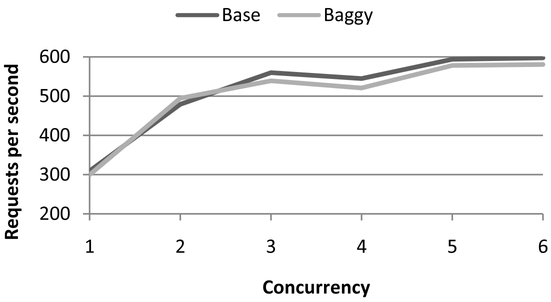


Figure 16: Throughput of NullHTTPD web server for varying numbers of concurrent requests.

it is interesting to contrast with them although they are not directly comparable.

4.2 Effectiveness

We evaluated the effectiveness of our system in preventing buffer overflows using the benchmark suite from [34]. The attacks required tuning to have any chance of success, because our system changes the stack frame layout and copies unsafe function arguments to local variables, but the benchmarks use the address of the first function argument to find the location of the return address they aim to overwrite.

Baggy bounds checking prevented 17 out of 18 buffer overflows in the suite. It failed, however, to prevent the overflow of an array inside a structure from overwriting a pointer inside the same structure. This limitation is also shared with other systems that detect memory errors at the level of memory blocks [19, 30, 36, 15].

4.3 Security Critical COTS Applications

Finally, to verify the usability of our approach, we built and measured a few additional larger and security critical COTS applications. Table 1 lists the total number of lines compiled in our experiments.

Program	KSLOC
openssl-0.9.8k	397
Apache-2.2.11	474
nullhttpd-0.5.1	2
libpng-1.2.5	36
SPECINT 2000	309
Olden	6
Total	1224

Table 1: Source lines of code in programs successfully built and run with baggy bounds.

We built the OpenSSL toolkit version 0.9.8k [28] comprised of about 400 KSLOC, and executed its test suite measuring 10% time and 11% memory overhead.

Then we built and measured two web servers, Apache [31] and NullHTTPD [27]. Running NullHTTPD revealed three bounds violations similar to, and including, the one reported in [8]. We used the Apache benchmark utility with the keep-alive option to compare the throughput over a LAN connection of the instrumented and uninstrumented versions of both web servers. We managed to saturate the CPU by using the keep-alive option of the benchmarking utility to reuse connections for subsequent requests. We issued repeated requests for the servers' default pages and varied the number of concurrent clients until the throughput of the uninstrumented version leveled off (Figures 15 and 16). We verified that the server's CPU was saturated at this point, and measured a throughput decrease of 8% for Apache and 3% for NullHTTPD.

Finally, we built `libpng`, a notoriously vulnerability prone library that is widely used. We successfully ran its test program for 1000 PNG files between 1–2K found on a desktop machine, and measured an average runtime overhead of 4% and a peak memory overhead of 3.5%.

5 64-bit Architectures

In this section we verify and investigate ways to optimize our approach on 64 bit architectures. The key observation is that pointers in 64 bit architectures have spare bits to use. In Figure 17 (a) and (b) we see that current models of AMD64 processors use 48 out of 64 bits in pointers, and Windows further limit this to 43 bits for user space programs. Thus 21 bits in the pointer representation are not used. Next we describe two uses for these spare bits, and present a performance evaluation on AMD64.

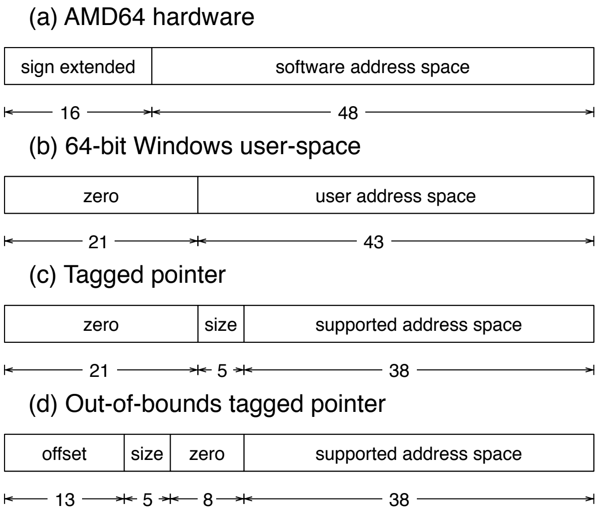


Figure 17: Use of pointer bits by AMD64 hardware, Windows applications, and baggy bounds tagged pointers.

5.1 Size Tagging

Since baggy bounds occupy less than a byte, they can fit in a 64 bit pointer's spare bits, removing the need for a separate data structure. These *tagged pointers* are similar to fat pointers in changing the pointer representation but have several advantages.

First, tagged pointers retain the size of regular pointers, avoiding fat pointers' register and memory waste. Moreover, their memory stores and loads are atomic, unlike fat pointers that break code relying on this. Finally, they preserve the memory layout of structures, overcoming the main drawback of fat pointers that breaks their interoperability with uninstrumented code.

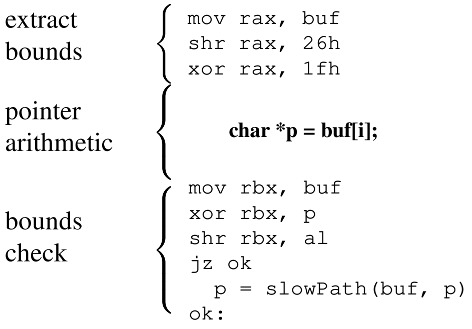
For interoperability, we must also enable instrumented code to use pointers from uninstrumented code and vice versa. We achieve the former by interpreting the default zero value found in unused pointer bits as maximal bounds, so checks on pointers missing bounds succeed. The other direction is harder because we must avoid raising a hardware exception when uninstrumented code dereferences a tagged pointer.

We solved this using the paging hardware to map all addresses that differ only in their tag bits to the same memory. This way, unmodified binary libraries can use tagged pointers, and instrumented code avoids the cost of clearing the tag too.

As shown in Figure 17(c), we use 5 bits to encode the size, allowing objects up to 2^{32} bytes. In order to use the paging hardware, these 5 bits have to come from the 43 bits supported by the operating system, thus leaving 38 bits of address space for programs.

With 5 address bits used for the bounds, we need to map 32 different address regions to the same memory. We implemented this entirely in user space using the `CreateFileMapping` and `MapViewOfFileEx` Windows API functions to replace the process image, stack, and heap with a file backed by the system paging file and mapped at 32 different locations in the process address space.

We use the 5 bits effectively ignored by the hardware to store the size of memory allocations. For heap allocations, our `malloc`-style functions set the tags for pointers they return. For locals and globals, we instrument the address taking operator "`&`" to properly tag the resulting pointer. We store the bit complement of the size logarithm enabling interoperability with untagged pointers by interpreting their zero bit pattern as all bits set (representing a maximal allocation of 2^{32}).



```
mov rax, buf
shr rax, 26h
xor rax, 1fh

char *p = buf[i];

mov rbx, buf
xor rbx, p
shr rbx, al
jz ok
  p = slowPath(buf, p)
ok:
```

Figure 18: AMD64 code sequence inserted to check unsafe arithmetic with tagged pointers.

With the bounds encoded in pointers, there is no need for a memory lookup to check pointer arithmetic. Figure 18 shows the AMD64 code sequence for checking pointer arithmetic using a tagged pointer. First, we extract the encoded bounds from the source pointer by right shifting a copy to bring the tag to the bottom 8 bits of the register and xoring them with the value `0x1f` to recover the size logarithm by inverting the bottom 5 bits. Then we check that the result of the arithmetic is within bounds by xoring the source and result pointers, shifting the result by the tag stored in `al`, and checking for zero.

Similar to the table-based implementation of Section 3, out-of-bounds pointers trigger a bounds error to simplify the common case. To cause this, we zero the bits that were used to hold the size and save them using 5 more bits in the pointer, as shown in Figure 17(d).

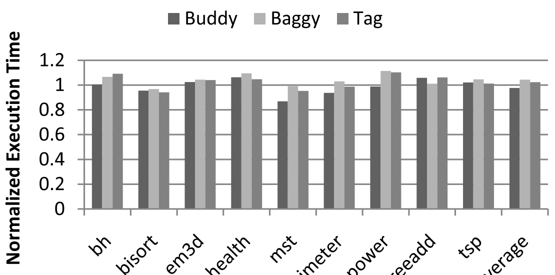


Figure 19: Normalized execution time on AMD64 with Olden benchmarks.

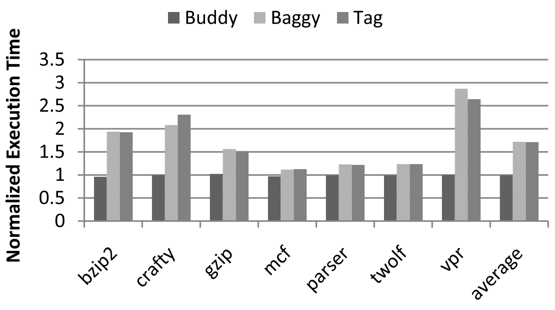


Figure 20: Normalized execution time on AMD64 with SPECINT 2000 benchmarks.

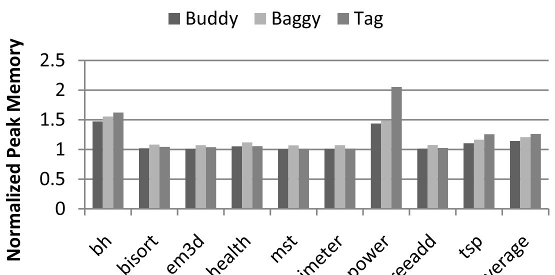


Figure 21: Normalized peak memory use on AMD64 with Olden benchmarks.

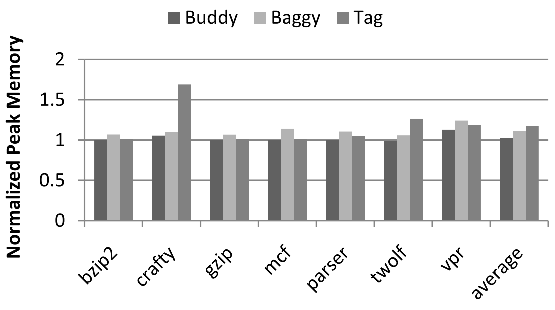


Figure 22: Normalized peak memory use on AMD64 with SPECINT 2000 benchmarks.

5.2 Out-Of-Bounds Offset

The spare bits can also store an offset that allows us to adjust an out-of-bounds pointer to recover the address of its referent object. We can use 13 bits for this offset, as shown in Figure 17(d). These bits can count slot or even allocation size multiples, increasing the supported out-of-bounds range to at least 2^{16} bytes above or below an allocation.

This technique does not depend on size tagging and can be used with a table instead. When looking up a pointer in the table, however, the top bits have to be masked off.

5.3 Evaluation

We evaluated baggy bounds checking on AMD64 using the subset of benchmarks from Section 4.1 that run unmodified on 64 bits. We measured the system using a contiguous array against the system using tagged pointers (Baggy and Tag in the figure legends respectively). We also measured the overhead using the buddy allocator only.

The multiple memory mappings complicated measuring memory use because Windows counts shared memory multiple times in peak memory reports. To overcome this, we measured memory use without actually tagging the pointers, to avoid touching more than one address for the same memory, but with the memory mappings in place to account for at least the top level memory management overheads.

Figures 19 and 20 show the time overhead. The average using a table on 64-bits is 4% for Olden and 72% for SPECINT 2000—close to the 32-bit results of Section 3. Figures 21 and 22 show the space overhead. The average using a table is 21% for Olden and 11% for SPECINT 2000. Olden's space overhead is higher than the 32-bit version; unlike the 32-bit case, the buddy allocator contributes to this overhead by 14% on average.

Tagged pointers are 1–2% faster on average than the table, and use about 5% less memory for most benchmarks, except a few ones such as `power` and `crafty`. These exceptions are because our prototype does not map pages to different addresses on demand, but instead maps 32 30-bit regions of virtual address space on program startup. Hence the fixed overhead is notable for these benchmarks because their absolute memory usage is low.

While we successfully implemented mapping multiple views entirely in user-space, a robust implementation would probably require kernel mode support. We feel that the gains are too small to justify the complexity. However, using the spare bits to store an out-of-bounds offset is a good solution for tracking out-of-bounds pointers when using the referent object approach of Jones and Kelly [19].

6 Related Work

Many techniques have been proposed to detect memory errors in C programs. Static analysis techniques, e.g., [33, 21, 7], can detect defects before software ships and they do not introduce runtime overhead, but they can miss defects and raise false alarms.

Since static techniques do not remove all defects, they have been complemented with dynamic techniques. Debugging tools such as Purify [17] and Annelid [25] can find memory errors during testing. While these tools can be used without source code, they typically slowdown applications by a factor of 10 or more. Some dynamic techniques detect specific errors such as stack overflows [13, 16, 32] or format string exploits [12]; they have low overhead but they cannot detect all spatial memory errors. Techniques such as control-flow integrity [20, 1] or taint tracking (e.g. [10, 26, 11, 35]) detect broad classes of errors, but they do not provide general protection from spatial memory errors.

Some systems provide probabilistic protection from memory errors [5]. In particular, DieHard [4] increases heap allocation sizes by a random amount to make more out-of-bounds errors benign at a low performance cost. Our system also increases the allocation size but enforces the allocation bounds to prevent errors and also protects stack-allocated objects in addition to heap-allocated ones.

Several systems prevent all spatial memory errors in C programs. Systems such as SafeC [3], CCured [24], Cyclone [18], and the technique in Xu *et al.* [36] associate bounds information with each pointer. CCured [24] and Cyclone [18] are memory safe dialects of C. They extend the pointer representation with bounds information, i.e., they use a fat pointer representation, but this changes memory layout and breaks binary compatibility. Moreover, they require a significant effort to port applications to the safe dialects. For example, CCured required changing 1287 out of 6000 lines of code for the Olden benchmarks [15], and an average of 10% of the lines of code have to be changed when porting programs from C to Cyclone [34]. CCured has 28% average runtime overhead for the Olden benchmarks, which is significantly higher than the baggy bounds overhead. Xu *et al.* [36] track pointers to detect spatial errors as well as temporal errors with additional overhead, thus their space overhead is proportional to the number of pointers. The average time overhead for spatial protection on the benchmarks we overlap is 73% versus 16% for baggy bounds with a space overhead of 273% versus 4%.

Other systems map any memory address within an allocated object to the bounds information for the object. Jones and Kelly [19] developed a backwards compatible bounds checking solution that uses a splay tree to map addresses to bounds. The splay tree is updated on allocation and deallocation, and operations on pointers are instrumented to lookup the bounds using an in-bounds pointer. The advantage over previous approaches using fat pointers is interoperability with code that was compiled without instrumentation. They increase the allocation size to support legal out-of-bounds pointers one byte beyond the object size. Baggy bounds checking offers similar interoperability with less time and space overhead, which we evaluated by using their implementation of splay trees with our system. CRED [30] improves on the solution of Jones and Kelly by adding support for tracking out-of-bounds pointers and making sure that they are never dereferenced unless they are brought within bounds again. Real programs often violate the C standard and contain such out-of-bounds pointers that may be saved to data structures. The performance overhead for programs that do not have out-of-bounds pointers is similar to Jones and Kelly if the same level of runtime checking is used, but the authors recommend only checking strings to lower the overhead to acceptable levels. For programs that do contain such out-of-bounds pointers the cost of tracking them includes scanning a hash-table on every dereference to remove entries for out-of-bounds pointers. Our solution is more efficient, and we propose ways to track common cases of out-of-bounds pointers that avoid using an additional data structure.

The fastest previous technique for bounds checking by Dhurjati *et al.* [15] is more than two times slower than our prototype. It uses inter-procedural data structure analysis to partition allocations into pools statically and uses a separate splay tree for each pool. They can avoid inserting some objects in the splay tree when the analysis finds that a pool is size-homogeneous. This should significantly reduce the memory usage of the splay tree compared to previous solutions, but unfortunately they do not report memory overheads. This work also optimizes the handling of out-of-bounds pointers in CRED [30] by relying on hardware memory protection to detect the dereference of out-of-bounds pointers.

The latest proposal, SoftBound [23], tracks bounds for each pointer to achieve sub-object protection. Sub-object protection, however, may introduce compatibility problems with code using pointer arithmetic to traverse structures. SoftBound maintains interoperability by storing bounds in a hash table or a large contiguous array. Storing bounds for each pointer can lead to a worst case memory footprint as high as 300% for the hash-table version or 200% for the contiguous array. The average space overhead across Olden and a subset of SPECINT and SPECFP is 87% using a hash-table and 64% for the contiguous array, and the average runtime overhead for checking both reads and writes is 93% for the hash table and 67% for the contiguous array. Our average space overhead over Olden and SPECINT is 7.5% with an average time overhead of 32%.

Other approaches associate different kinds of metadata with memory regions to enforce safety properties. The technique in [37] detects some invalid pointers dereferences by marking all writable memory regions and preventing writes to non-writable memory; it reports an average runtime overhead of 97%. DFI [8] computes reaching definitions statically and enforces them at runtime. DFI has an average overhead of 104% on the SPEC benchmarks. WIT [2] computes the approximate set of objects written by each instruction and dynamically prevents writes to objects not in the set. WIT does not protect from invalid reads, and is subject to the precision of a points-to analysis when detecting some out-of-bounds errors. On the other hand, WIT can detect accesses to deallocated/unallocated objects and some accesses through dangling pointers to re-allocated objects in different analysis sets. WIT is six times faster than *baggy bounds checking* for SPECINT 2000, so it is also an attractive point in the error coverage/performance design space.

7 Limitations and Future Work

Our system shares some limitations with other solutions based on the referent object approach. Arithmetic on integers holding addresses is unchecked, casting an integer that holds an out-of-bounds address back to a pointer or passing an out-of-bounds pointer to unchecked code will break the program, and custom memory allocators reduce protection.

Our system does not address temporal memory safety violations (accesses through “dangling pointers” to re-allocated memory). Conservative garbage collection for C [6] is one way to address these but introduces its own compatibility issues and unpredictable overheads.

Our approach cannot protect from memory errors in subobjects such as structure fields. To offer such protection, a system must track the bounds of each pointer [23] and risk false alarms for some legal programs that use pointers to navigate across structure fields.

In Section 4 we found two programs using out-of-bounds pointers beyond the $slot\_size/2$ bytes supported on 32-bits and one beyond the 2^{16} bytes supported on 64-bits. Unfortunately the real applications built in Section 4.3 were limited to software we could readily port to the Windows toolchain; wide use will likely encounter occasional problems with out-of-bounds pointers, especially on 32-bit systems. We plan to extended our system to support all out-of-bounds pointers using the data structure from [30], but take advantage of the more efficient mechanisms we described for the common cases. To solve the delayed deallocation problem discussed in Section 6 and deallocate entries as soon as the out-of-bounds pointer is deallocated, we can track out-of-bounds pointers using the pointer's address instead of the pointer's referent object's address. (Similar to the approach [23] takes for all pointers.) To optimize scanning this data structure on every deallocation we can use an array with an entry for every few memory pages. A single memory read from this array on deallocation (e.g. on function exit) is sufficient to confirm the data structure has no entries for a memory range. This is the common case since most out-of-bounds pointers are handled by the other mechanisms we described in this paper.

Our prototype uses a simple intra-procedural analysis to find safe operations and does not eliminate redundant checks. We expect that integrating state of the art analyses to reduce the number of checks will further improve performance.

Finally, our approach tolerates harmless bound violations making it less suitable for debugging than slower techniques that can uncover these errors. On the other hand, being faster makes it more suitable for production runs, and tolerating faults in production runs may be desired [29].

8 Conclusions

Attacks that exploit out-of-bounds errors in C and C++ continue to be a serious security problem. We presented *baggy bounds checking*, a backwards-compatible bounds checking technique that implements efficient bounds checks. It improves the performance of bounds checks by checking allocation bounds instead of object bounds and by using a binary buddy allocator to constrain the size and alignment of allocations to powers of 2. These constraints enable a concise representation for allocation bounds and let *baggy bounds checking* store this information in an array that can be looked up and maintained efficiently. Our experiments show that replacing a splay tree, which was used to store bounds information in previous systems, by our array reduces time overhead by an order of magnitude without increasing space overhead.

We believe *baggy bounds checking* can be used in practice to harden security-critical applications because it has low overhead, it works on unmodified C and C++ programs, and it preserves binary compatibility with uninstrumented libraries. For example, we were able to compile the Apache Web server with *baggy bounds checking* and the throughput of the hardened version of the server decreases by only 8% relative to an unistrumented version.

Acknowledgments

We thank our shepherd R. Sekar, the anonymous reviewers, and the members of the Networks and Operating Systems group at Cambridge University for comments that helped improve this paper. We also thank Dinakar Dhurjati and Vikram Adve for their communication.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[4] Emery D. Berger and Benjamin G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2006.

[5] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[6] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. In *Software Practice & Experience*, 1988.

[7] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. In *Software Practice & Experience*, 2000.

[8] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[9] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: reducing data lifetime through secure deallocation. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[10] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Can we contain Internet worms? In *Proceedings of the Third Workshop on Hot Topics in Networks (HotNets-III)*, 2004.

[11] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the 20th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2005.

[12] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[13] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[14] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: a safe execution environment for commodity operating systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[15] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

[16] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/main.html`.

[17] Reed Hasting and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[18] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track of the USENIX Annual Conference*, 2002.

[19] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging (AADEBUG)*, 1997.

[20] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[21] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[22] Microsoft. Phoenix compiler framework. `http://connect.microsoft.com/Phoenix`.

[23] Santosh Nagarakatte, Jianzhou Zhao, Milo Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[24] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.

[25] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. In *Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, 2004.

[26] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS)*, 2005.

[27] NullLogic. Null HTTPd. `http://nullwebmail.sourceforge.net/httpd`.

[28] OpenSSL Toolkit. `http://www.openssl.org`.

[29] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[30] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Network and Distributed System Security Symposium (NDSS)*, 2004.

[31] The Apache Software Foundation. The Apache HTTP Server Project. `http://httpd.apache.org`.

[32] Vendicator. StackShield. `http://www.angelfire.com/sk/stackshield`.

[33] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 7th Network and Distributed System Security Symposium (NDSS)*, 2000.

[34] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS)*, 2003.

[35] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[36] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE)*, 2004.

[37] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2003.