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.

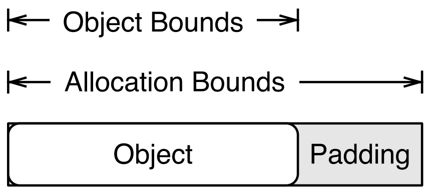

Figure 1: Allocated memory is often padded to a particular alignment boundary, and hence can be larger than the requested object size. By checking *allocation bounds* rather than object bounds, we allow benign accesses to the padding, but can significantly reduce the cost of bounds lookups at runtime.

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.

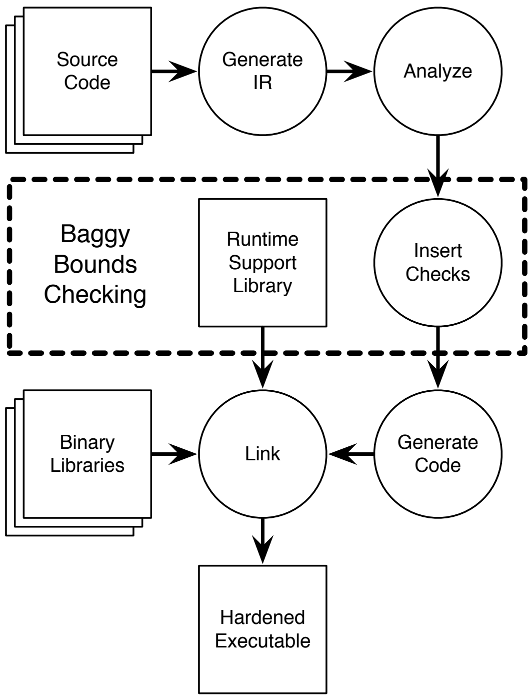

Figure 2: Overall system architecture, with our contribution highlighted within the dashed box.

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.

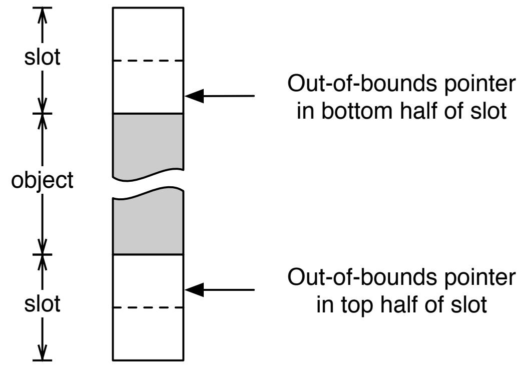
slot
object
slot
Out-of-bounds pointer in bottom half of slot
Out-of-bounds pointer in top half of slot

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.

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

pointer
arithmetic      char *p = buf[i];

             mov ebx, buf
             xor ebx, p
bounds       shr ebx, al
check        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;
    }
```

$$\downarrow$$

```
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

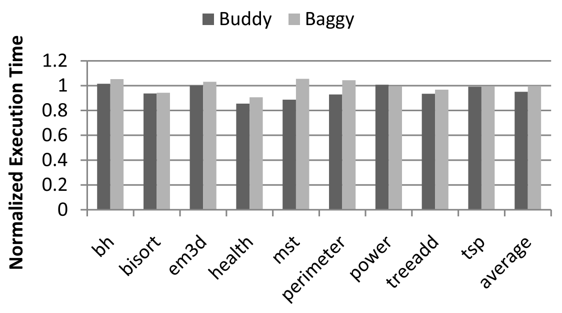

Figure 7: Execution time for the Olden benchmarks using the buddy allocator and our full system, normalized by the execution time using the standard system allocator without instrumentation.

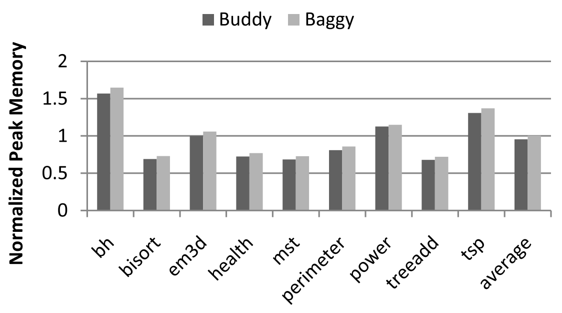

Figure 8: Peak memory use with the buddy allocator alone and with the full system for the Olden benchmarks, normalized by peak memory using the standard allocator without instrumentation.

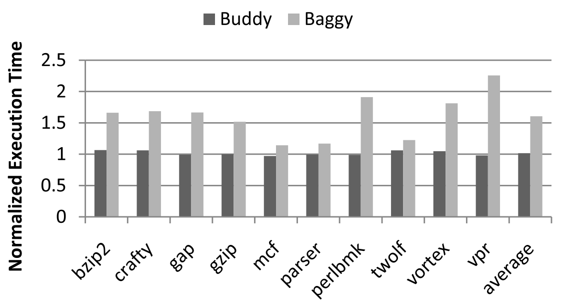

Figure 9: Execution time for SPECINT 2000 benchmarks using the buddy allocator and our full system, normalized by the execution time using the standard system allocator without instrumentation.

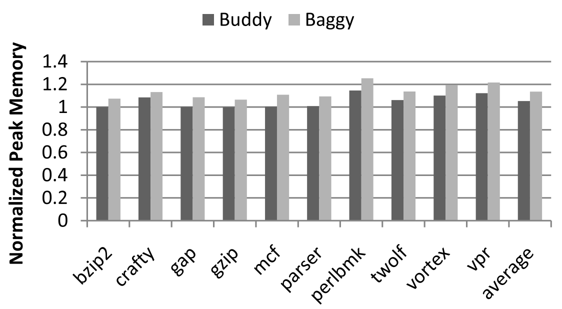

Figure 10: Peak memory use with the buddy allocator alone and with the full system for SPECINT 2000 benchmarks, normalized by peak memory using the standard allocator without instrumentation.

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

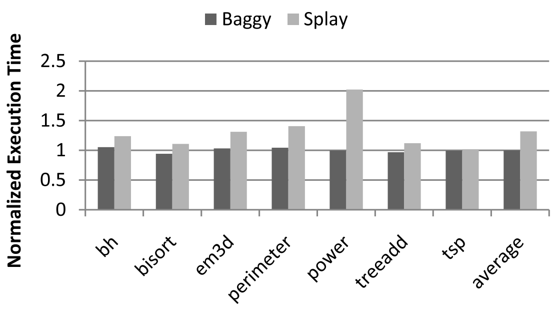

Figure 11: Execution time of baggy bounds checking versus using a splay tree for the Olden benchmark suite, normalized by the execution time using the standard system allocator without instrumentation. Benchmarks `mst` and `health` used too much memory and thrashed so their execution times are excluded.

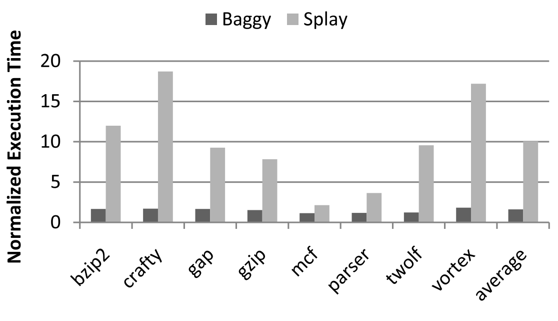

Figure 12: Execution time of baggy bounds checking versus using a splay tree for SPECINT 2000 benchmarks, normalized by the execution time using the standard system allocator without instrumentation.

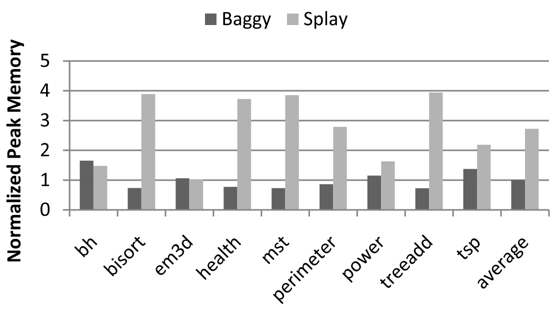

Figure 13: Peak memory use of baggy bounds checking versus using a splay tree for the Olden benchmark suite, normalized by peak memory using the standard allocator without instrumentation.

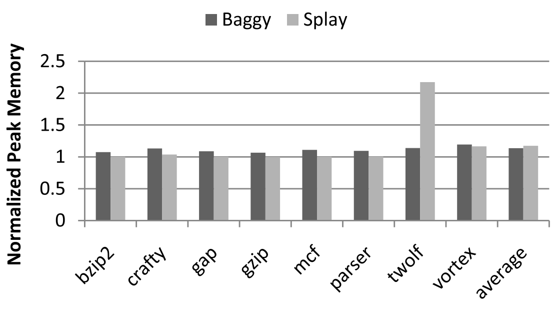

Figure 14: Peak memory use of baggy bounds checking versus using a splay tree for SPECINT 2000 benchmarks, normalized by peak memory using the standard allocator without instrumentation.

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

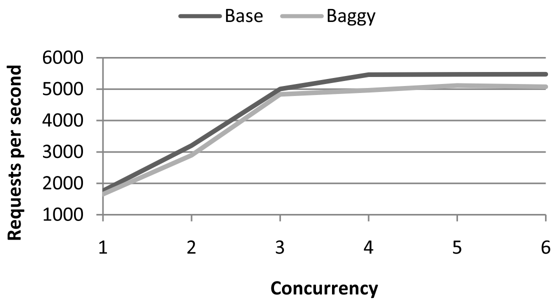

Figure 15: Throughput of Apache web server for varying numbers of concurrent requests.

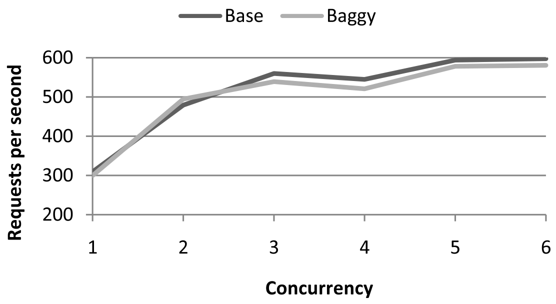

Figure 16: Throughput of NullHTTPD web server for varying numbers of concurrent requests.

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.

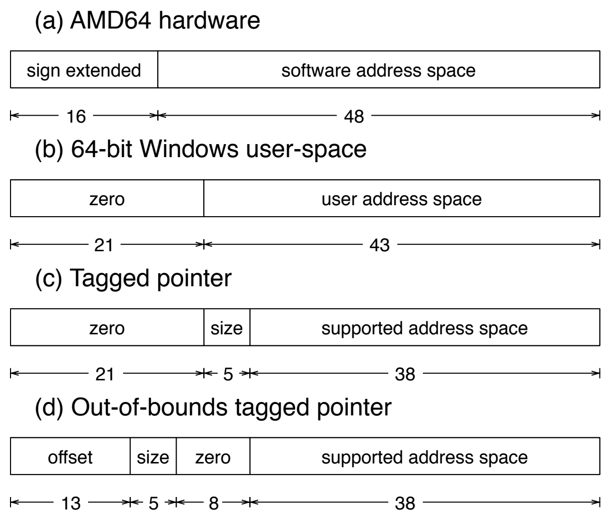

Figure 17: Use of pointer bits by AMD64 hardware, Windows applications, and baggy bounds tagged pointers.

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}).

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

pointer
arithmetic        char *p = buf[i];

              mov rbx, buf
bounds        xor rbx, p
check         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).

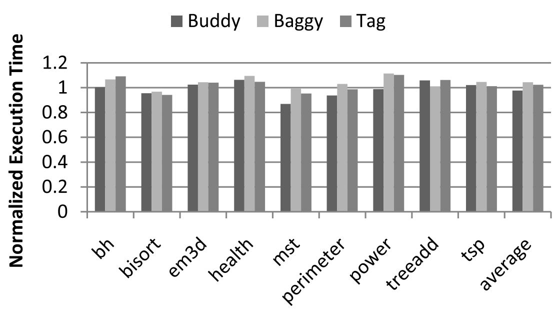

Figure 19: Normalized execution time on AMD64 with Olden benchmarks.

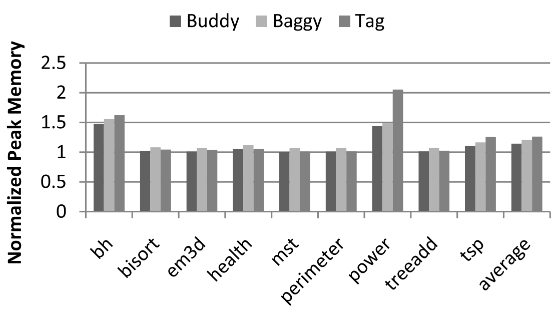

Figure 21: Normalized peak memory use on AMD64 with Olden benchmarks.

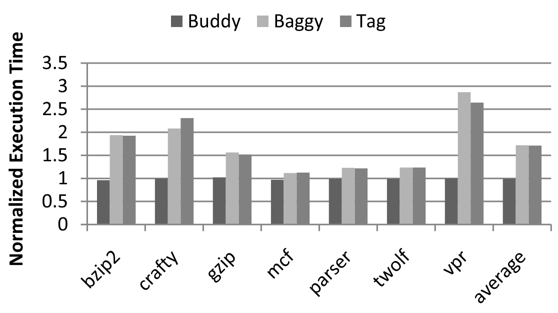

Figure 20: Normalized execution time on AMD64 with SPECINT 2000 benchmarks.

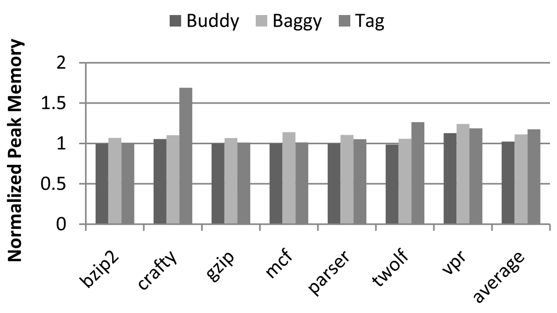

Figure 22: Normalized peak memory use on AMD64 with SPECINT 2000 benchmarks.

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 sub-objects 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.

Dynamic Test Generation To Find Integer Bugs in x86 Binary Linux Programs

David Molnar
UC Berkeley

Xue Cong Li
UC Berkeley

David A. Wagner
UC Berkeley

Abstract

Recently, integer bugs, including integer overflow, width conversion, and signed/unsigned conversion errors, have risen to become a common root cause for serious security vulnerabilities. We introduce new methods for discovering integer bugs using dynamic test generation on x86 binaries, and we describe key design choices in efficient symbolic execution of such programs. We implemented our methods in a prototype tool *SmartFuzz*, which we use to analyze Linux x86 binary executables. We also created a reporting service, `metafuzz.com`, to aid in triaging and reporting bugs found by SmartFuzz and the black-box fuzz testing tool `zzuf`. We report on experiments applying these tools to a range of software applications, including the `mplayer` media player, the `exiv2` image metadata library, and ImageMagick `convert`. We also report on our experience using SmartFuzz, `zzuf`, and `metafuzz.com` to perform testing at scale with the Amazon Elastic Compute Cloud (EC2). To date, the `metafuzz.com` site has recorded more than $2,614$ test runs, comprising $2,361,595$ test cases. Our experiments found approximately 77 total distinct bugs in 864 compute hours, costing us an average of $\$2.24$ per bug at current EC2 rates. We quantify the overlap in bugs found by the two tools, and we show that SmartFuzz finds bugs missed by `zzuf`, including one program where SmartFuzz finds bugs but `zzuf` does not.

1 Introduction

Integer overflow bugs recently became the second most common bug type in security advisories from OS vendors [10]. Unfortunately, traditional static and dynamic analysis techniques are poorly suited to detecting integer-related bugs. In this paper, we argue that *dynamic test generation* is better suited to finding such bugs, and we develop new methods for finding a broad class of integer bugs with this approach. We have implemented these methods in a new tool, *SmartFuzz,* that analyzes traces from commodity Linux x86 programs.

Integer bugs result from a mismatch between machine arithmetic and mathematical arithmetic. For example, machine arithmetic has bounded precision; if an expression has a value greater than the maximum integer that can be represented, the value wraps around to fit in machine precision. This can cause the value stored to be smaller than expected by the programmer. If, for example, a wrapped value is used as an argument to `malloc`, the result is an object that is smaller than expected, which can lead to a buffer overflow later if the programmer is not careful. This kind of bug is often known as an integer overflow bug. In Section 2 we describe two other classes of integer bugs: *width conversions*, in which converting from one type of machine integer to another causes unexpected changes in value, and *signed/unsigned conversions*, in which a value is treated as both a signed and an unsigned integer. These kinds of bugs are pervasive and can, in many cases, cause serious security vulnerabilities. Therefore, eliminating such bugs is important for improving software security.

While new code can partially or totally avoid integer bugs if it is constructed appropriately [19], it is also important to find and fix bugs in legacy code. Previous approaches to finding integer bugs in legacy code have focused on static analysis or runtime checks. Unfortunately, existing static analysis algorithms for finding integer bugs tend to generate many false positives, because it is difficult to statically reason about integer values with sufficient precision. Alternatively, one can insert runtime checks into the application to check for overflow or non-value-preserving width conversions, and raise an exception if they occur. One problem with this approach is that many overflows are benign and harmless. Throwing an exception in such cases prevents the application from functioning and thus causes false positives. Furthermore, occasionally the code intentionally relies upon overflow semantics; e.g., cryptographic code or fast hash functions. Such code is often falsely flagged by static analysis or runtime checks. In summary, both static analysis and runtime checking tend to suffer from either many false positives or many missed bugs.

In contrast, *dynamic test generation* is a promising approach for avoiding these shortcomings. Dynamic test generation, a technique introduced by Godefroid et al. and Engler et al. [13, 7], uses *symbolic execution* to generate new test cases that expose specifically targeted behaviors of the program. Symbolic execution works by collecting a set of constraints, called the *path condition*, that model the values computed by the program along a single path through the code. To determine whether there is any input that could cause the program to follow that path of execution and also violate a particular assertion, we can add to the path condition a constraint representing that the assertion is violated and feed the resulting set of constraints to a solver. If the solver finds any solution to the resulting constraints, we can synthesize a new test case that will trigger an assertion violation. In this way, symbolic execution can be used to discover test cases that cause the program to behave in a specific way.

Our main approach is to use symbolic execution to construct test cases that trigger arithmetic overflows, non-value-preserving width conversions, or dangerous signed/unsigned conversions. Then, we run the program on these test cases and use standard tools that check for buggy behavior to recognize bugs. We only report test cases that are verified to trigger incorrect behavior by the program. As a result, we have confidence that all test cases we report are real bugs and not false positives.

Others have previously reported on using dynamic test generation to find some kinds of security bugs [8, 15]. The contribution of this paper is to show how to extend those techniques to find integer-related bugs. We show that this approach is effective at finding many bugs, without the false positives endemic to prior work on static analysis and runtime checking.

The ability to eliminate false positives is important, because false positives are time-consuming to deal with. In slogan form: false positives in static analysis waste the programmer's time; false positives in runtime checking waste the end user's time; while false positives in dynamic test generation waste the tool's time. Because an hour of CPU time is much cheaper than an hour of a human's time, dynamic test generation is an attractive way to find and fix integer bugs.

We have implemented our approach to finding integer bugs in *SmartFuzz*, a tool for performing symbolic execution and dynamic test generation on Linux x86 applications. SmartFuzz works with binary executables directly, and does not require or use access to source code. Working with binaries has several advantages, most notably that we can generate tests directly from shipping binaries. In particular, we do not need to modify the build process for a program under test, which has been a pain point for static analysis tools [9]. Also, this allows us to perform whole-program analysis: we can find bugs that arise due to interactions between the application and libraries it uses, even if we don't have source code for those libraries. Of course, working with binary traces introduces special challenges, most notably the sheer size of the traces and the lack of type information that would be present in the source code. We discuss the challenges and design choices in Section 4.

In Section 5 we describe the techniques we use to generate test cases for integer bugs in dynamic test generation. We discovered that these techniques find many bugs, too many to track manually. To help us prioritize and manage these bug reports and streamline the process of reporting them to developers, we built *Metafuzz*, a web service for tracking test cases and bugs (Section 6). Metafuzz helps minimize the amount of human time required to find high-quality bugs and report them to developers, which is important because human time is the most expensive resource in a testing framework. Finally, Section 7 presents an empirical evaluation of our techniques and discusses our experience with these tools.

The contributions of this paper are the following:

- We design novel algorithms for finding signed/unsigned conversion vulnerabilities using symbolic execution. In particular, we develop a novel type inference approach that allows us to detect which values in an x86 binary trace are used as signed integers, unsigned integers, or both. We discuss challenges in scaling such an analysis to commodity Linux media playing software and our approach to these challenges.

- We extend the range of integer bugs that can be found with symbolic execution, including integer overflows, integer underflows, width conversions, and signed/unsigned conversions. No prior symbolic execution tool has included the ability to detect all of these kinds of integer vulnerabilities.

- We implement these methods in *SmartFuzz*, a tool for symbolic execution and dynamic test generation of x86 binaries on Linux. We describe key challenges in symbolic execution of commodity Linux software, and we explain design choices in SmartFuzz motivated by these challenges.

- We report on the bug finding performance of SmartFuzz and compare SmartFuzz to the `zzuf` black box fuzz testing tool. The `zzuf` tool is a simple, yet effective, *fuzz testing* program which randomly mutates a given seed file to find new test inputs, without any knowledge or feedback from the target program. We have tested a broad range of commodity Linux software, including the media players mplayer and ffmpeg, the ImageMagick convert tool, and the exiv2 TIFF metadata parsing library. This software comprises over one million lines of source code, and our test cases result in symbolic execution of traces that are millions of x86 instructions in length.

- We identify challenges with reporting bugs at scale, and introduce several techniques for addressing these challenges. For example, we present evidence that a simple stack hash is not sufficient for grouping test cases to avoid duplicate bug reports, and then we develop a *fuzzy stack hash* to solve these problems. Our experiments find approximately 77 total distinct bugs in 864 compute hours, giving us an average cost of \$2.24 per bug at current Amazon EC2 rates. We quantify the overlap in bugs found by the two tools, and we show that SmartFuzz finds bugs missed by zzuf, including one program where SmartFuzz finds bugs but zzuf does not.

Between June 2008 and November 2008, Metafuzz has processed over 2,614 test runs from both SmartFuzz and the zzuf black box fuzz testing tool [16], comprising 2,361,595 test cases. To our knowledge, this is the largest number of test runs and test cases yet reported for dynamic test generation techniques. We have released our code under the GPL version 2 and BSD licenses[1]. Our vision is a service that makes it easy and inexpensive for software projects to find integer bugs and other serious security relevant code defects using dynamic test generation techniques. Our work shows that such a service is possible for a large class of commodity Linux programs.

2 Integer Bugs

We now describe the three main classes of integer bugs we want to find: integer overflow/underflow, width conversions, and signed/unsigned conversion errors [2]. All three classes of bugs occur due to the mismatch between machine arithmetic and arithmetic over unbounded integers.

Overflow/Underflow. Integer overflow (and underflow) bugs occur when an arithmetic expression results in a value that is larger (or smaller) than can be represented by the machine type. The usual behavior in this case is to silently "wrap around," e.g. for a 32-bit type, reduce the value modulo 2^{32}. Consider the function badalloc in Figure 1. If the multiplication sz * n overflows, the allocated buffer may be smaller than expected, which can lead to a buffer overflow later.

Width Conversions. Converting a value of one integral type to a wider (or narrower) integral type which has a different range of values can introduce *width conversion* bugs. For instance, consider badcpy in Figure 1. If the first parameter is negative, the conversion from Int16 to UInt32 will trigger sign-extension, causing m to be very large and likely leading to a buffer overflow. Because memcpy's third argument is declared to have type size_t (which is an unsigned integer type), even if we passed n directly to memcpy the implicit conversion would still make this buggy. Width conversion bugs can also arise when converting a wider type to a narrower type.

```
char *badalloc(int sz, int n) {
  return (char *) malloc(sz * n);
}
void badcpy(Int16 n, char *p, char *q) {
  UInt32 m = n;
  memcpy(p, q, m);
}
void badcpy2(int n, char *p, char *q) {
  if (n > 800)
     return;
  memcpy(p, q, n);
}
```

Figure 1: Examples of three types of integer bugs.

Signed/Unsigned Conversion. Lastly, converting a signed integer type to an unsigned integer type of the same width (or vice versa) can introduce bugs, because this conversion can change a negative number to a large positive number (or vice versa). For example, consider badcpy2 in Figure 1. If the first parameter n is a negative integer, it will pass the bounds check, then be promoted to a large unsigned integer when passed to memcpy. memcpy will copy a large number of bytes, likely leading to a buffer overflow.

3 Related Work

An earlier version of SmartFuzz and the Metafuzz web site infrastructure described in this paper were used for previous work that compares dynamic test generation with black-box fuzz testing by different authors [1]. That previous work does not describe the SmartFuzz tool, its design choices, or the Metafuzz infrastructure in detail. Furthermore, this paper works from new data on the effectiveness of SmartFuzz, except for an anecdote in our "preliminary experiences" section. We are not aware of other work that directly compares dynamic test generation with black-box fuzz testing on a scale similar to ours.

The most closely related work on integer bugs is Godefroid et al. [15], who describe dynamic test generation with bug-seeking queries for integer overflow, underflow, and some narrowing conversion errors in the context of the SAGE tool. Our work looks at a wider

[1]http://www.sf.net/projects/catchconv

range of narrowing conversion errors, and we consider signed/unsigned conversion while their work does not. The EXE and KLEE tools also use integer overflow to prioritize different test cases in dynamic test generation, but they do not break out results on the number of bugs found due to this heuristic [8, 6]. The KLEE system also focuses on scaling dynamic test generation, but in a different way. While we focus on a few "large" programs in our results, KLEE focuses on high code coverage for over 450 smaller programs, as measured by trace size and source lines of code. These previous works also do not address the problem of type inference for integer types in binary traces.

IntScope is a static binary analysis tool for finding integer overflow bugs [28]. IntScope translates binaries to an intermediate representation, then it checks lazily for potentially harmful integer overflows by using symbolic execution for data that flows into "taint sinks" defined by the tool, such as memory allocation functions. SmartFuzz, in contrast, *eagerly* attempts to generate new test cases that cause an integer bug at the point in the program where such behavior could occur. This difference is due in part to the fact that IntScope reports errors to a programmer directly, while SmartFuzz filters test cases using a tool such as `memcheck`. As we argued in the Introduction, such a filter allows us to employ aggressive heuristics that may generate many test cases. Furthermore, while IntScope renders signed and unsigned comparisons in their intermediate representation by using hints from the x86 instruction set, they do not explicitly discuss how to use this information to perform type inference for signed and unsigned types, nor do they address the issue of scaling such inference to traces with millions of instructions. Finally, IntScope focuses only on integer overflow errors, while SmartFuzz covers underflow, narrowing conversion, and signed/unsigned conversion bugs in addition.

The dynamic test generation approach we use was introduced by Godefroid et al. [13] and independently by Cadar and Engler [7]. The SAGE system by Godefroid et al. works, as we do, on x86 binary programs and uses a generational search, but SAGE makes several different design choices we explain in Section 4. Lanzi et al. propose a design for dynamic test generation of x86 binaries that uses static analysis of loops to assist the solver, but their implementation is preliminary [17]. KLEE, in contrast, works with the intermediate representation generated by the Low-Level Virtual Machine target for `gcc` [6]. Larson and Austin applied symbolic range analysis to traces of programs to look for potential buffer overflow attacks, although they did not attempt to synthesize crashing inputs [18]. The BitBlaze [5] infrastructure of Song et al. also performs symbolic execution of x86 binaries, but their focus is on malware and signature generation, not on test generation.

Other approaches to integer bugs include static analysis and runtime detection. The Microsoft Prefast tool uses static analysis to warn about intraprocedural integer overflows [21]. Both Microsoft Visual C++ and `gcc` can add runtime checks to catch integer overflows in arguments to `malloc` and terminate a program. Brumley et al. provide rules for such runtime checks and show they can be implemented with low overhead on the x86 architecture by using jumps conditioned on the overflow bit in EFLAGS [4]. Both of these approaches fail to catch signed/unsigned conversion errors. Furthermore, both static analysis and runtime checking for overflow will flag code that is correct but relies on overflow semantics, while our approach only reports test cases in case of a crash or a Valgrind error report.

Blexim gives an introduction to integer bugs [3]. Fuzz testing has received a great deal of attention since its original introduction by Miller et al [22]. Notable public demonstrations of fuzzing's ability to find bugs include the Month of Browser Bugs and Month of Kernel Bugs [23, 20]. DeMott surveys recent work on fuzz testing, including the autodafe fuzzer, which uses `libgdb` to instrument functions of interest and adjust fuzz testing based on those functions' arguments [11, 27].

Our Metafuzz infrastructure also addresses issues not treated in previous work on test generation. First, we make *bug bucketing* a first-class problem and we introduce a *fuzzy stack hash* in response to developer feedback on bugs reported by Metafuzz. The SAGE paper reports bugs by stack hash, and KLEE reports on using the line of code as a bug bucketing heuristic, but we are not aware of other work that uses a fuzzy stack hash. Second, we report techniques for reducing the amount of human time required to process test cases generated by fuzzing and improve the quality of our error reports to developers; we are not aware of previous work on this topic. Such techniques are vitally important because human time is the most expensive part of a test infrastructure. Finally, Metafuzz uses on-demand computing with the Amazon Elastic Compute Cloud, and we explicitly quantify the cost of each bug found, which was not done in previous work.

4 Dynamic Test Generation

We describe the architecture of *SmartFuzz*, a tool for dynamic test generation of x86 binary programs on Linux. Dynamic test generation on x86 binaries—without access to source code—raises special challenges. We discuss these challenges and motivate our fundamental design choices.

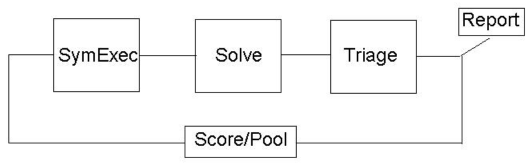

Figure 2: Dynamic test generation includes four stages: symbolic execution, solving to obtain new test cases, then triage to determine whether to report a bug or score the test case for addition to the pool of unexplored test cases.

4.1 Architecture

The SmartFuzz architecture is as follows: First, we add one or more test cases to a pool. Each test case in the pool receives a score given by the number of new basic blocks seen when running the target program on the test case. By "new" we mean that the basic block has not been observed while scoring any previous test case; we identify basic blocks by the instruction pointer of their entry point.

In each iteration of test generation, we choose a high-scoring test case, execute the program on that input, and use symbolic execution to generate a set of constraints that record how each intermediate value computed by the program relates to the inputs in the test case. SmartFuzz implements the symbolic execution and scoring components using the Valgrind binary analysis framework, and we use STP [12] to solve constraints.

For each symbolic branch, SmartFuzz adds a constraint that tries to force the program down a different path. We then query the constraint solver to see whether there exists any solution to the resulting set of constraints; if there is, the solution describes a new test case. We refer to these as *coverage queries* to the constraint solver.

SmartFuzz also injects constraints that are satisfied if a condition causing an error or potential error is satisfied (e.g., to force an arithmetic calculation to overflow). We then query the constraint solver; a solution describes a test case likely to cause an error. We refer to these as *bug-seeking queries* to the constraint solver. Bug-seeking queries come in different *types*, depending on the specific error they seek to exhibit in the program.

Both coverage and bug-seeking queries are explored in a *generational search* similar to the SAGE tool [14]. Each query from a symbolic trace is solved in turn, and new test cases created from successfully solved queries. A single symbolic execution therefore leads to many coverage and bug-seeking queries to the constraint solver, which may result in many new test cases.

We *triage* each new test case as it is generated, i.e. we determine if it exhibits a bug. If so, we report the bug; otherwise, we add the test case to the pool for scoring and possible symbolic execution. For triage, we use Valgrind `memcheck` on the target program with each test case, which is a tool that observes concrete execution looking for common programming errors [26]. We record any test case that causes the program to crash or triggers a `memcheck` warning.

We chose `memcheck` because it checks a variety of properties, including reads and writes to invalid memory locations, memory leaks, and use of uninitialized values. Re-implementing these analyses as part of the SmartFuzz symbolic execution tool would be wasteful and error-prone, as the `memcheck` tool has had the benefit of multiple years of use in large-scale projects such as Firefox and OpenOffice. The `memcheck` tool is known as a tool with a low false positive rate, as well, making it more likely that developers will pay attention to bugs reported by `memcheck`. Given a `memcheck` error report, developers do not even need to know that associated test case was created by SmartFuzz.

We do not attempt to classify the bugs we find as exploitable or not exploitable, because doing so by hand for the volume of test cases we generate is impractical. Many of the bugs found by `memcheck` are memory safety errors, which often lead to security vulnerabilities. Writes to invalid memory locations, in particular, are a red flag. Finally, to report bugs we use the Metafuzz framework described in Section 6.

4.2 Design Choices

Intermediate Representation. The sheer size and complexity of the x86 instruction set poses a challenge for analyzing x86 binaries. We decided to translate the underlying x86 code on-the-fly to an intermediate representation, then map the intermediate representation to symbolic formulas. Specifically, we used the Valgrind binary instrumentation tool to translate x86 instructions into VEX, the Valgrind intermediate representation [24]. The BitBlaze system works similarly, but with a different intermediate representation [5]. Details are available in an extended version of this paper[2].

Using an intermediate representation offers several advantages. First, it allows for a degree of platform independence: though we support only x86 in our current tool, the VEX library also supports the AMD64 and PowerPC instruction sets, with ARM support under active development. Adding support for these additional architectures requires only adding support for a small number of additional VEX instructions, not an entirely new instruction set from scratch. Second, the VEX library generates IR that satisfies the single static assignment property and

[2] `http://www.cs.berkeley.edu/~dmolnar/usenix09-full.pdf`

performs other optimizations, which makes the translation from IR to formulas more straightforward. Third, and most importantly, this choice allowed us to outsource the pain of dealing with the minutae of the x86 instruction set to the VEX library, which has had years of production use as part of the Valgrind memory checking tool. For instance, we don't need to explicitly model the EFLAGS register, as the VEX library translates it to boolean operations. The main shortcoming with the VEX IR is that a single x86 instruction may expand to five or more IR instructions, which results in long traces and correspondingly longer symbolic formulas.

Online Constraint Generation. SmartFuzz uses *online* constraint generation, in which constraints are generated while the program is running. In contrast, SAGE (another tool for dynamic test generation) uses *offline* constraint generation, where the program is first traced and then the trace is replayed to generate constraints [14]. Offline constraint generation has several advantages: it is not sensitive to concurrency or nondeterminism in system calls; tracing has lower runtime overhead than constraint generation, so can be applied to running systems in a realistic environment; and, this separation of concerns makes the system easier to develop and debug, not least because trace replay and constraint generation is reproducible and deterministic. In short, offline constraint generation has important software engineering advantages.

SmartFuzz uses online constraint generation primarily because, when the SmartFuzz project began, we were not aware of an available offline trace-and-replay framework with an intermediate representation comparable to VEX. Today, O'Callahan's `chronicle-recorder` could provide a starting point for a VEX-based offline constraint generation tool [25].

Memory Model. Other symbolic execution tools such as EXE and KLEE model memory as a set of symbolic arrays, with one array for each allocated memory object. We do not. Instead, for each load or store instruction, we first concretize the memory address before accessing the symbolic heap. In particular, we keep a map M from concrete memory addresses to symbolic values. If the program reads from concrete address `a`, we retrieve a symbolic value from $M(\mathtt{a})$. Even if we have recorded a symbolic expression a associated with this address, the symbolic address is ignored. Note that the value of `a` is known at constraint generation time and hence becomes (as far as the solver is concerned) a constant. Store instructions are handled similarly.

While this approach sacrifices precision, it scales better to large traces. We note that the SAGE tool adopts a similar memory model. In particular, concretizing addresses generates symbolic formulas that the constraint solver can solve much more efficiently, because the solver does not need to reason about aliasing of pointers.

Only Tainted Data is Symbolic. We track the taint status of every byte in memory. As an optimization, we do not store symbolic information for untainted memory locations, because by definition untainted data is not dependent upon the untrusted inputs that we are trying to vary. We have found that only a tiny fraction of the data processed along a single execution path is tainted. Consequently, this optimization greatly reduces the size of our constraint systems and reduces the memory overhead of symbolic execution.

Focus on Fuzzing Files. We decided to focus on single-threaded programs, such as media players, that read a file containing untrusted data. Thus, a test case is simply the contents of this file, and SmartFuzz can focus on generating candidate files. This simplifies the symbolic execution and test case generation infrastructure, because there are a limited number of system calls that read from this file, and we do not need to account for concurrent interactions between threads in the same program. We know of no fundamental barriers, however, to extending our approach to multi-threaded and network-facing programs.

Our implementation associates a symbolic input variable with each byte of the input file. As a result, SmartFuzz cannot generate test cases with more bytes than are present in the initial seed file.

Multiple Cooperating Analyses. Our tool is implemented as a series of independent cooperating analyses in the Valgrind instrumentation framework. Each analysis adds its own instrumentation to a basic block during translation and exports an interface to the other analyses. For example, the instrumentation for tracking taint flow, which determines the IR instructions to treat as symbolic, exports an interface that allows querying whether a specific memory location or temporary variable is symbolic. A second analysis then uses this interface to determine whether or not to output STP constraints for a given IR instruction.

The main advantage of this approach is that it makes it easy to add new features by adding a new analysis, then modifying our core constraint generation instrumentation. Also, this decomposition enabled us to extract our taint-tracking code and use it in a different project with minimal modifications, and we were able to implement the binary type inference analysis described in Section 5, replacing a different earlier version, without changing our other analyses.

Optimize in Postprocessing. Another design choice was to output constraints that are as "close" as possible to the intermediate representation, performing only limited optimizations on the fly. For example, we implement the "related constraint elimination," as introduced by tools such as EXE and SAGE [8, 14], as a post-processing step on constraints created by our tool. We then leave it up to the solver to perform common subexpression elimination, constant propagation, and other optimizations. The main benefit of this choice is that it simplifies our constraint generation. One drawback of this choice is that current solvers, including STP, are not yet capable of "remembering" optimizations from one query to the next, leading to redundant work on the part of the solver. The main drawback of this choice, however, is that while after optimization each individual query is small, the total symbolic trace containing all queries for a program can be several gigabytes. When running our tool on a 32-bit host machine, this can cause problems with maximum file size for a single file or maximum memory size in a single process.

5 Techniques for Finding Integer Bugs

We now describe the techniques we use for finding integer bugs.

Overflow/Underflow. For each arithmetic expression that could potentially overflow or underflow, we emit a constraint that is satisfied if the overflow or underflow occurs. If our solver can satisfy these constraints, the resulting input values will likely cause an underflow or overflow, potentially leading to unexpected behavior.

Width Conversions. For each conversion between integer types, we check whether it is possible for the source value to be outside the range of the target value by adding a constraint that's satisfied when this is the case and then applying the constraint solver. For conversions that may sign-extend, we use the constraint solver to search for a test case where the high bit of the source value is non-zero.

Signed/Unsigned Conversions. Our basic approach is to try to reconstruct, from the x86 instructions executed, signed/unsigned type information about all integral values. This information is present in the source code but not in the binary, so we describe an algorithm to infer this information automatically.

Consider four types for integer values: "Top," "Signed," "Unsigned," or "Bottom." Here, "Top" means the value has not been observed in the context of a signed or unsigned integer; "Signed" means that the value has been used as a signed integer; "Unsigned" means the value has been used as an unsigned integer; and "Bottom" means that the value has been used inconsistently as both a signed and unsigned integer. These types form a four-point lattice. Our goal is to find symbolic program values that have type "Bottom." These values are candidates for signed/unsigned conversion errors. We then attempt to synthesize an input that forces these values to be negative.

We associate every instance of every temporary variable in the Valgrind intermediate representation with a type. Every variable in the program starts with type Top. During execution we add *type constraints* to the type of each value. For x86 binaries, the sources of type constraints are signed and unsigned comparison operators: e.g., a signed comparison between two values causes both values to receive the "Signed" type constraint. We also add unsigned type constraints to values used as the length argument of `memcpy` function, which we can detect because we know the calling convention for x86 and we have debugging symbols for glibc. While the x86 instruction set has additional operations, such as `IMUL` that reveal type information about their operands, we do not consider these; this means only that we may incorrectly under-constrain the types of some values.

```
int main(int argc, char** argv) {
  char * p = malloc(800);
  char * q = malloc(800);
  int n;
  n = atol(argv[1]);
  if (n > 800)
      return;
  memcpy(p, q, n);
  return 0;
}
```

Figure 3: A simple test case for dynamic type inference and query generation. The signed comparison `n > 800` and unsigned `size_t` argument to `memcpy` assign the type "Bottom" to the value associated with `n`. When we solve for an input that makes `n` negative, we obtain a test case that reveals the error.

Any value that has received both a signed and unsigned type constraint receives the type Bottom. After adding a type constraint, we check to see if the type of a value has moved to Bottom. If so, we attempt to solve for an input which makes the value negative. We do this because negative values behave differently in signed and unsigned comparisons, and so they are likely to exhibit an error if one exists. All of this information is present in the trace without requiring access to the original program source code.

We discovered, however, that `gcc 4.1.2` inlines some calls to `memcpy` by transforming them to `rep movsb` instructions, even when the `-O` flag is not present. Furthermore, the Valgrind IR generated for the `rep movsb` instruction compares a decrementing counter variable to zero, instead of counting up and executing an unsigned comparison to the loop bound. As a result, on gcc 4.1.2 a call to `memcpy` does not cause its length argument to be marked as unsigned. To deal with this problem, we implemented a simple heuristic to detect the IR generated for `rep movsb` and emit the appropriate constraint. We verified that this heuristic works on a small test case similar to Figure 3, generating a test input that caused a segmentation fault.

A key problem is storing all of the information required to carry out type inference without exhausting available memory. Because a trace may have several million instructions, memory usage is key to scaling type inference to long traces. Furthermore, our algorithm requires us to keep track of the types of all values in the program, unlike constraint generation, which need concern itself only with tainted values. An earlier version of our analysis created a special "type variable" for each value, then maintained a map from IR locations to type variables. Each type variable then mapped to a type. We found that in addition to being hard to maintain, this analysis often led to a number of live type variables that scaled linearly with the number of executed IR instructions. The result was that our analysis ran out of memory when attempting to play media files in the `mplayer` media player.

To solve this problem, we developed a garbage-collected data structure for tracking type information. To reduce memory consumption, we use a union-find data structure to partition integer values into equivalence classes where all values in an equivalence class are required to have the same type. We maintain one type for each union-find equivalence class; in our implementation type information is associated with the representative node for that equivalence class. Assignments force the source and target values to have the same types, which is implemented by merging their equivalence classes. Updating the type for a value can be done by updating its representative node's type, with no need to explicitly update the types of all other variables in the equivalence class.

It turns out that this data structure is acyclic, due to the fact that VEX IR is in SSA form. Therefore, we use reference counting to garbage collect these nodes. In addition, we benefit from an additional property of the VEX IR: all values are either stored in memory, in registers, or in a temporary variable, and the lifetime of each temporary variable is implicitly limited to that of a single basic block. Therefore, we maintain a list of temporaries that are live in the current basic block; when we leave the basic block, the type information associated with all of those live temporaries can be deallocated. Consequently, the amount of memory needed for type inference at any point is proportional to the number of tainted (symbolic) variables that are live at that point—which is a significant improvement over the naive approach to type inference. The full version of this paper contains a more detailed specification of these algorithms[3].

[3]`http://www.cs.berkeley.edu/~dmolnar/usenix09-full.pdf`

6 Triage and Reporting at Scale

Both SmartFuzz and `zzuf` can produce hundreds to thousands of test cases for a single test run. We designed and built a web service, *Metafuzz*, to manage the volume of tests. We describe some problems we found while building Metafuzz and techniques to overcoming these problems. Finally, we describe the user experience with Metafuzz and bug reporting.

6.1 Problems and Techniques

The Metafuzz architecture is as follows: first, a Test Machine generates new test cases for a program and runs them locally. The Test Machine then determines which test cases exhibit bugs and sends these test cases to Metafuzz. The Metafuzz web site displays these test cases to the User, along with information about what kind of bug was found in which target program. The User can pick test cases of interest and download them for further investigation. We now describe some of the problems we faced when designing Metafuzz, and our techniques for handling them. Section 7 reports our experiences with using Metafuzz to manage test cases and report bugs.

Problem: Each test run generated many test cases, too many to examine by hand.

Technique: We used Valgrind's `memcheck` to automate the process of checking whether a particular test case causes the program to misbehave. `Memcheck` looks for memory leaks, use of uninitialized values, and memory safety errors such as writes to memory that was not allocated [26]. If `memcheck` reports an error, we save the test case. In addition, we looked for core dumps and non-zero program exit codes.

Problem: Even after filtering out the test cases that caused no errors, there were still many test cases that do cause errors.

Technique: The `metafuzz.com` front page is a HTML page listing all of the potential bug reports. showing all potential bug reports. Each test machine uploads information about test cases that trigger bugs to Metafuzz.

Problem: The machines used for testing had no long-term storage. Some of the test cases were too big to attach in e-mail or Bugzilla, making it difficult to share them with developers.

Technique: Test cases are uploaded directly to Metafuzz, providing each one with a stable URL. Each test case also includes the Valgrind output showing the Valgrind error, as well as the output of the program to `stdout` and `stderr`.

Problem: Some target projects change quickly. For example, we saw as many as four updates per day to the `mplayer` source code repository. Developers reject bug reports against "out of date" versions of the software.

Technique: We use the Amazon Elastic Compute Cloud (EC2) to automatically attempt to reproduce the bug

against the latest version of the target software. A button on the Metafuzz site spawns an Amazon EC2 instance that checks out the most recent version of the target software, builds it, and then attempts to reproduce the bug.
Problem: Software projects have specific reporting requirements that are tedious to implement by hand. For example, `mplayer` developers ask for a stack backtrace, disassembly, and register dump at the point of a crash.
Technique: Metafuzz automatically generates bug reports in the proper format from the failing test case. We added a button to the Metafuzz web site so that we can review the resulting bug report and then send it to the target software's bug tracker with a single click.
Problem: The same bug often manifests itself as many failing test cases. Reporting the same bug to developers many times wastes developer time.
Technique: We use the call stack to identify multiple instances of the same bug. Valgrind memcheck reports the call stack at each error site, as a sequence of instruction pointers. If debugging information is present, it also reports the associated filename and line number information in the source code.

Initially, we computed a *stack hash* as a hash of the sequence of instruction pointers in the backtrace. This has the benefit of not requiring debug information or symbols. Unfortunately, we found that a naive stack hash has several problems. First, it is sensitive to address space layout randomization (ASLR), because different runs of the same program may load the stack or dynamically linked libraries at different addresses, leading to different hash values for call stacks that are semantically the same. Second, even without ASLR, we found several cases where a single bug might be triggered at multiple call stacks that were similar but not identical. For example, a buggy function can be called in several different places in the code. Each call site then yields a different stack hash. Third, any slight change to the target software can change instruction pointers and thus cause the same bug to receive a different stack hash. While we do use the stack hash on the client to avoid uploading test cases for bugs that have been previously found, we found that we could not use stack hashes alone to determine if a bug report is novel or not.

To address these shortcomings, we developed a *fuzzy stack hash* that is forgiving of slight changes to the call stack. We use debug symbol information to identify the name of the function called, the line number in source code (excluding the last digit of the line number, to allow for slight changes in the code), and the name of the object file for each frame in the call stack. We then hash all of this information for the three functions at the top of the call stack.

The choice of the number of functions to hash determines the "fuzzyness" of the hash. At one extreme, we could hash all extant functions on the call stack. This would be similar to the classic stack hash and report many semantically same bugs in different buckets. On the other extreme, we could hash only the most recently called function. This fails in cases where two semantically different bugs both exhibit as a result of calling `memcpy` or some other utility function with bogus arguments. In this case, both call stacks would end with `memcpy` even though the bug is in the way the arguments are computed. We chose three functions as a trade-off between these extremes; we found this sufficient to stop further reports from the `mplayer` developers of duplicates in our initial experiences. Finding the best fuzzy stack hash is interesting future work; we note that the choice of bug bucketing technique may depend on the program under test.

While any fuzzy stack hash, including ours, may accidentally lump together two distinct bugs, we believe this is less serious than reporting duplicate bugs to developers. We added a post-processing step on the server that computes the fuzzy stack hash for test cases that have been uploaded to Metafuzz and uses it to coalesce duplicates into a single bug bucket.
Problem: Because Valgrind `memcheck` does not terminate the program after seeing an error, a single test case may give rise to dozens of Valgrind error reports. Two different test cases may share some Valgrind errors but not others.
Technique: First, we put a link on the Metafuzz site to a single test case for each bug bucket. Therefore, if two test cases share some Valgrind errors, we only use one test case for each of the errors in common. Second, when reporting bugs to developers, we highlight in the title the specific bugs on which to focus.

7 Results

7.1 Preliminary Experience

We used an earlier version of SmartFuzz and Metafuzz in a project carried out by a group of undergraduate students over the course of eight weeks in Summer 2008. When beginning the project, none of the students had any training in security or bug reporting. We provided a one-week course in software security. We introduced SmartFuzz, `zzuf`, and Metafuzz, then asked the students to generate test cases and report bugs to software developers. By the end of the eight weeks, the students generated over 1.2 million test cases, from which they reported over 90 bugs to software developers, principally to the `mplayer` project, of which 14 were fixed. For further details, we refer to their presentation [1].

7.2 Experiment Setup

Test Programs. Our target programs were `mplayer` version `SVN-r28403-4.1.2`, `ffmpeg` version

	mplayer	ffmpeg	exiv2	gzip	bzip2	convert
Coverage	2599	14535	1629	5906	12606	388
ConversionNot32	0	3787	0	0	0	0
Conversion32to8	1	26	740	2	10	116
Conversion32to16	0	16004	0	0	0	0
Conversion16Sto32	0	121	0	0	0	0
SignedOverflow	1544	37803	5941	24825	9109	49
SignedUnderflow	3	4003	48	1647	2840	0
UnsignedOverflow	1544	36945	4957	24825	9104	35
UnsignedUnderflow	0	0	0	0	0	0
MallocArg	0	24	0	0	0	0
SignedUnsigned	2568	21064	799	7883	17065	49

Figure 5: The number of each type of query for each test program after a single 24-hour run.

	Queries	Test Cases	Bugs
Coverage	588068	31121	19
ConversionNot32	4586	0	0
Conversion32to8	1915	1377	3
Conversion32to16	16073	67	4
Conversion16Sto32	206	0	0
SignedOverflow	167110	0	0
SignedUnderflow	20198	21	3
UnsignedOverflow	164155	9280	3
MallocArg	30	0	0
SignedUnsigned	125509	6949	5

Figure 6: The number of bugs found, by query type, over all test runs. The fourth column shows the number of distinct bugs found from test cases produced by the given type of query, as classified using our fuzzy stack hash.

SVN-r16903, exiv2 version SVN-r1735, gzip version 1.3.12, bzip2 version 1.0.5, and ImageMagick convert version 6.4.8 − 10, which are all widely used media and compression programs. Table 4 shows information on the size of each test program. Our test programs are large, both in terms of source lines of code and trace lengths. The percentage of the trace that is symbolic, however, is small.

Test Platform. Our experiments were run on the Amazon Elastic Compute Cloud (EC2), employing a "small" and a "large" instance image with SmartFuzz, zzuf, and all our test programs pre-installed. At this writing, an EC2 small instance has 1.7 GB of RAM and a single-core virtual CPU with performance roughly equivalent to a 1GHz 2007 Intel Xeon. An EC2 large instance has 7 GB of RAM and a dual-core virtual CPU, with each core having performance roughly equivalent to a 1 GHz Xeon.

We ran all mplayer runs and ffmpeg runs on EC2 large instances, and we ran all other test runs with EC2 small instances. We spot-checked each run to ensure that instances successfully held all working data in memory during symbolic execution and triage without swapping to disk, which would incur a significant performance penalty. For each target program we ran SmartFuzz and zzuf with three seed files, for 24 hours per program per seed file. Our experiments took 288 large machine-hours and 576 small machine-hours, which at current EC2 prices of \$0.10 per hour for small instances and \$0.40 per hour for large instances cost \$172.80.

Query Types. SmartFuzz queries our solver with the following types of queries: Coverage, ConversionNot32, Conversion32to8, Conversion32to16, SignedOverflow, UnsignedOverflow, SignedUnderflow, UnsignedUnderflow, MallocArg, and SignedUnsigned. Coverage queries refer to queries created as part of the generational search by flipping path conditions. The others are *bug-seeking queries* that attempt to synthesize inputs leading to specific kinds of bugs. Here MallocArg refers to a set of bug-seeking queries that attempt to force inputs to known memory allocation functions to be negative, yielding an implicit conversion to a large unsigned integer, or force the input to be small.

Experience Reporting to Developers. Our original strategy was to report all distinct bugs to developers and let them judge which ones to fix. The mplayer developers gave us feedback on this strategy. They wanted to focus on fixing the most serious bugs, so they preferred seeing reports only for out-of-bounds writes and double free errors. In contrast, they were not as interested in out-of-bound reads, even if the resulting read caused a segmentation fault. This helped us prioritize bugs for reporting.

7.3 Bug Statistics

Integer Bug-Seeking Queries Yield Bugs. Figure 6 reports the number of each type of query to the constraint solver over all test runs. For each type of query, we report the number of test files generated and the number of distinct bugs, as measured by our fuzzy stack hash. Some bugs may be revealed by multiple different kinds of queries, so there may be overlap between the bug counts in two different rows of Figure 6.

The table shows that our dynamic test generation methods for integer bugs succeed in finding bugs in our test programs. Furthermore, the queries for signed/unsigned bugs found the most distinct bugs out of all bug-seeking queries. This shows that our novel method for detecting signed/unsigned bugs (Section 5) is effective at finding bugs.

SmartFuzz Finds More Bugs Than zzuf, on mplayer. For mplayer, SmartFuzz generated 10,661 test cases over all test runs, while zzuf generated 11,297 test cases; SmartFuzz found 22 bugs while zzuf found 13. Therefore, in terms of number of bugs, SmartFuzz outperformed zzuf for testing mplayer. Another surprising result here is that SmartFuzz generated nearly as many test cases as zzuf, despite the additional overhead for

	SLOC	seedfile type and size	Branches	x86 instrs	IRStmts	asserts	queries
mplayer	723468	MP3 (159000 bytes)	20647045	159500373	810829992	1960	36
ffmpeg	304990	AVI (980002 bytes)	4147710	19539096	115036155	4778690	462346
exiv2	57080	JPG (22844 bytes)	809806	6185985	32460806	81450	1006
gzip	140036	TAR.GZ (14763 bytes)	24782	161118	880386	95960	13309
bzip	26095	TAR.BZ2 (618620 bytes)	107396936	746219573	4185066021	1787053	314914
ImageMagick	300896	PNG (25385 bytes)	98993374	478474232	2802603384	583	81

Figure 4: The size of our test programs. We report the source lines of code for each test program and the size of one of our seed files, as measured by David A. Wheeler's `sloccount`. Then we run the test program on that seed file and report the total number of branches, x86 instructions, Valgrind IR statements, STP assert statements, and STP query statements for that run. We ran symbolic execution for a maximum of 12 hours, which was sufficient for all programs except `mplayer`, which terminated during symbolic execution.

symbolic execution and constraint solving. This shows the effect of the generational search and the choice of memory model; we leverage a single expensive symbolic execution and fast solver queries to generate many test cases. At the same time, we note that `zzuf` found a serious InvalidWrite bug, while SmartFuzz did not.

A previous version of our infrastructure had problems with test cases that caused the target program to run forever, causing the search to stall. Therefore, we introduced a timeout, so that after 300 CPU seconds, the target program is killed. We manually examined the output of `memcheck` from all killed programs to determine whether such test cases represent errors. For `gzip` we discovered that SmartFuzz created six such test cases, which account for the two out-of-bounds read (InvalidRead) errors we report; `zzuf` did not find any hanging test cases for `gzip`. We found no other hanging test cases in our other test runs.

Different Bugs Found by SmartFuzz and `zzuf`. We ran the same target programs with the same seed files using `zzuf`. Figure 7 shows bugs found by each type of fuzzer. With respect to each tool, SmartFuzz found 37 total distinct bugs and `zzuf` found 59 distinct bugs. We found some overlap between bugs as well: 19 bugs were found by both fuzz testing tools, for a total of 77 distinct bugs. This shows that while there is overlap between the two tools, SmartFuzz finds bugs that `zzuf` does not and vice versa. Therefore, it makes sense to try both tools when testing software.

Note that we did not find any bugs for `bzip2` with either fuzzer, so neither tool was effective on this program. This shows that fuzzing is not always effective at finding bugs, especially with a program that has already seen attention for security vulnerabilities. We also note that SmartFuzz found InvalidRead errors in `gzip` while `zzuf` found no bugs in this program. Therefore `gzip` is a case where SmartFuzz's directed testing is able to trigger a bug, but purely random testing is not.

Block Coverage. We measured the number of basic blocks in the program visited by the execution of the seed file, then measured how many new basic blocks were visited during the test run. We discovered `zzuf` added a higher percentage of new blocks than SmartFuzz in 13 of the test runs, while SmartFuzz added a higher percentage of new blocks in 4 of the test runs (the SmartFuzz `convert-2` test run terminated prematurely.) Table 8 shows the initial basic blocks, the number of blocks added, and the percentage added for each fuzzer. We see that the effectiveness of SmartFuzz varies by program; for `convert` it is particularly effective, finding many more new basic blocks than `zzuf`.

Contrasting SmartFuzz and zzuf Performance. Despite the limitations of random testing, the blackbox fuzz testing tool `zzuf` found bugs in four out of our six test programs. In three of our test programs, `zzuf` found more bugs than SmartFuzz. Furthermore, `zzuf` found the most serious InvalidWrite errors, while SmartFuzz did not. These results seem surprising, because SmartFuzz exercises directed testing based on program behavior while `zzuf` is a purely blackbox tool. We would expect that SmartFuzz should find all the bugs found by `zzuf`, given an unbounded amount of time to run both test generation methods.

In practice, however, the time which we run the methods is limited, and so the question is which bugs are discovered first by each method. We have identified possible reasons for this behavior, based on examining the test cases and Valgrind errors generated by both tools[4]. We now list and briefly explain these reasons.

Header parsing errors. The errors we observed are often in code that parses the header of a file format. For example, we noticed bugs in functions of `mplayer` that parse MP3 headers. These errors can be triggered by simply placing "wrong" values in data fields of header files. As a result, these errors do not require complicated and unlikely predicates to be true before reaching buggy code, and so the errors can be reached without needing the full power of dynamic test generation. Similarly, code cov-

[4]We have placed representative test cases from each method at `http://www.metafuzz.com/example-testcases.tgz`

	mplayer		ffmpeg		exiv2		gzip		convert	
SyscallParam	4	3	2	3	0	0	0	0	0	0
UninitCondition	13	1	1	8	0	0	0	0	3	8
UninitValue	0	3	0	3	0	0	0	0	0	2
Overlap	0	0	0	1	0	0	0	0	0	0
Leak_DefinitelyLost	2	2	2	4	0	0	0	0	0	0
Leak_PossiblyLost	2	1	0	2	0	1	0	0	0	0
InvalidRead	1	2	0	4	4	6	2	0	1	1
InvalidWrite	0	1	0	3	0	0	0	0	0	0
Total	22	13	5	28	4	7	2	0	4	11
Cost per bug	\$1.30	\$2.16	\$5.76	\$1.03	\$1.80	\$1.03	\$3.60	NA	\$1.20	\$0.65

Figure 7: The number of bugs, after fuzzy stack hashing, found by SmartFuzz (the number on the left in each column) and `zzuf` (the number on the right). We also report the cost per bug, assuming \$0.10 per small compute-hour, \$0.40 per large compute-hour, and 3 runs of 24 hours each per target for each tool.

erage may be affected by the style of program used for testing.

Difference in number of bytes changed. The two different methods change a vastly different number of bytes from the original seed file to create each new test case. SmartFuzz changes exactly those bytes that must change to force program execution down a single new path or to violate a single property. Below we show that in our programs there are only a small number of constraints on the input to solve, so a SmartFuzz test case differs from its seed file by only a small number of bytes. In contrast, `zzuf` changes a fixed fraction of the input bytes to random other bytes; in our experiments, we left this at the default value of 0.004.

The large number of changes in a single fuzzed test case means that for header parsing or other code that looks at small chunks of the file independently, a single `zzuf` test case will exercise many different code paths with fuzzed chunks of the input file. Each path which originates from parsing a fuzzed chunk of the input file is a potential bug. Furthermore, because Valgrind `memcheck` does not necessarily terminate the program's execution when a memory safety error occurs, one such file may yield multiple bugs and corresponding bug buckets.

In contrast, the SmartFuzz test cases usually change only one chunk and so explore only one "fuzzed" path through such code. Our code coverage metric, unfortunately, is not precise enough to capture this difference because we measured block coverage instead of path coverage. This means once a set of code blocks has been covered, a testing method receives no further credit for additional paths through those same code blocks.

Small number of generations reached by SmartFuzz. Our 24 hour experiments with SmartFuzz tended to reach a small number of generations. While previous work on whitebox fuzz testing shows that most bugs are found in the early generations [14], this feeds into the previous issue because the number of differences between the fuzzed file and the original file is proportional to the generation number of the file. We are exploring longer-running SmartFuzz experiments of a week or more to address this issue.

Loop behavior in SmartFuzz. Finally, SmartFuzz deals with loops by looking at the unrolled loop in the dynamic program trace, then attempting to generate test cases for each symbolic `if` statement in the unrolled loop. This strategy is not likely to create new test cases that cause the loop to execute for vastly more iterations than seen in the trace. By contrast, the `zzuf` case may get lucky by assigning a random and large value to a byte sequence in the file that controls the loop behavior. On the other hand, when we looked at the `gzip` bug found by SmartFuzz and not by `zzuf`, we discovered that it appears to be due to an infinite loop in the `inflate_dynamic` routine of `gzip`.

7.4 SmartFuzz Statistics

Integer Bug Queries Vary By Program. Table 5 shows the number of solver queries of each type for one of our 24-hour test runs. We see that the type of queries varies from one program to another. We also see that for `bzip2` and `mplayer`, queries generated by type inference for signed/unsigned errors account for a large fraction of all queries to the constraint solver. This results from our choice to eagerly generate new test cases early in the program; because there are many potential integer bugs in these two programs, our symbolic traces have many integer bug-seeking queries. Our design choice of using an independent tool such as `memcheck` to filter the resulting test cases means we can tolerate such a large number of queries because they require little human oversight.

Time Spent In Each Task Varies By Program. Figure 9 shows the percentage of time spent in symbolic execution, coverage, triage, and recording for each run of our experiment. We also report an "Other" category, which includes the time spent in the constraint solver. This shows us where we can obtain gains through further optimization. The amount of time spent in each task depends greatly on the seed file, as well as on the

	Initial basic blocks		Blocks added by tests		Ratio of prior two columns	
Test run	SmartFuzz	zzuf	SmartFuzz	zzuf	SmartFuzz	zzuf
mplayer-1	7819	7823	5509	326	70%	4%
mplayer-2	11375	11376	908	1395	7%	12%
mplayer-3	11093	11096	102	2472	0.9%	22%
ffmpeg-1	6470	6470	592	20036	9.14%	310%
ffmpeg-2	6427	6427	677	2210	10.53%	34.3%
ffmpeg-3	6112	611	97	538	1.58%	8.8%
convert-1	8028	8246	2187	20	27%	0.24%
convert-2	8040	8258	2392	6	29%	0.073%
convert-3	NA	10715	NA	1846	NA	17.2%
exiv2-1	9819	9816	2934	3560	29.9%	36.3%
exiv2-2	9811	9807	2783	3345	28.3%	34.1%
exiv2-3	9814	9810	2816	3561	28.7%	36.3%
gzip-1	2088	2088	252	334	12%	16%
gzip-2	2169	2169	259	275	11.9%	12.7%
gzip-3	2124	2124	266	316	12%	15%
bzip2-1	2779	2778	123	209	4.4%	7.5%
bzip2-2	2777	2778	125	237	4.5%	8.5%
bzip2-3	2823	2822	115	114	4.1%	4.0%

Figure 8: Coverage metrics: the initial number of basic blocks, before testing; the number of blocks added during testing; and the percentage of blocks added.

	Total	SymExec	Coverage	Triage	Record	Other
gzip-1	206522s	0.1%	0.06%	0.70%	17.6%	81.6%
gzip-2	208999s	0.81%	0.005%	0.70%	17.59%	80.89%
gzip-3	209128s	1.09%	0.0024%	0.68%	17.8%	80.4%
bzip2-1	208977s	0.28%	0.335%	1.47%	14.0%	83.915%
bzip2-2	208849s	0.185%	0.283%	1.25%	14.0%	84.32%
bzip2-3	162825s	25.55%	0.78%	31.09%	3.09%	39.5%
mplayer-1	131465s	14.2%	5.6%	22.95%	4.7%	52.57%
mplayer-2	131524s	15.65%	5.53%	22.95%	25.20%	30.66%
mplayer-3	49974s	77.31%	0.558%	1.467%	10.96%	9.7%
ffmpeg-1	73981s	2.565%	0.579%	4.67%	70.29%	21.89%
ffmpeg-2	131600s	36.138%	1.729%	9.75%	11.56%	40.8%
ffmpeg-3	24255s	96.31%	0.1278%	0.833%	0.878%	1.8429%
convert-1	14917s	70.17%	2.36%	24.13%	2.43%	0.91%
convert-2	97519s	66.91%	1.89%	28.14%	2.18%	0.89%
exiv2-1	49541s	3.62%	10.62%	71.29%	9.18%	5.28%
exiv2-2	69415s	3.85%	12.25%	65.64%	12.48%	5.78%
exiv2-3	154334s	1.15%	1.41%	3.50%	8.12%	85.81%

Figure 9: The percentage of time spent in each of the phases of SmartFuzz. The second column reports the total wall-clock time, in seconds, for the run; the remaining columns are a percentage of this total. The "other" column includes the time spent solving STP queries.

target program. For example, the first run of `mplayer`, which used a mp3 seedfile, spent 98.57% of total time in symbolic execution, and only 0.33% in coverage, and 0.72% in triage. In contrast, the second run of `mplayer`, which used a mp4 seedfile, spent only 14.77% of time in symbolic execution, but 10.23% of time in coverage, and 40.82% in triage. We see that the speed of symbolic execution and of triage is the major bottleneck for several of our test runs, however. This shows that future work should focus on improving these two areas.

7.5 Solver Statistics

Related Constraint Optimization Varies By Program. We measured the size of all queries to the constraint solver, both before and after applying the related constraint optimization described in Section 4. Figure 10 shows the average size for queries from each test program, taken over all queries in all test runs with that program. We see that while the optimization is effective in all cases, its average effectiveness varies greatly from one test program to another. This shows that different programs vary greatly in how many input bytes influence each query.

The Majority of Queries Are Fast. Figure 10 shows the empirical cumulative distribution function of STP solver times over all our test runs. For about 70% of the test cases, the solver takes at most one second. The maximum solver time was about 10.89 seconds. These results reflect our choice of memory model and the effectiveness of the related constraint optimization. Because of these, the queries to the solver consist only of operations over bitvectors (with no array constraints), and most of the sets of constraints sent to the solver are small, yielding fast solver performance.

8 Conclusion

We described new methods for finding integer bugs in dynamic test generation, and we implemented these methods in SmartFuzz, a new dynamic test generation tool. We then reported on our experiences building the web site `metafuzz.com` and using it to manage test case generation at scale. In particular, we found that SmartFuzz finds bugs not found by `zzuf` and vice versa, showing that a comprehensive testing strategy should use both

	average before	average after	ratio (before/after)
mplayer	292524	33092	8.84
ffmpeg	350846	83086	4.22
exiv2	81807	10696	7.65
gzip	348027	199336	1.75
bzip2	278980	162159	1.72
convert	2119	1057	2.00

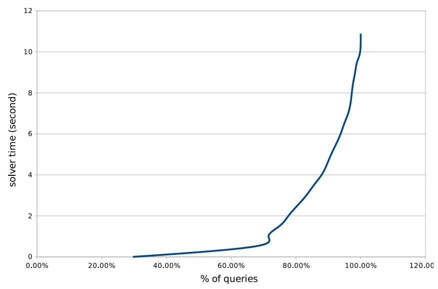

Figure 10: On the left, average query size before and after related constraint optimization for each test program. On the right, an empirical CDF of solver times.

white-box and black-box test generation tools.

Furthermore, we showed that our methods can find integer bugs without the false positives inherent to static analysis or runtime checking approaches, and we showed that our methods scale to commodity Linux media playing software. The Metafuzz web site is live, and we have released our code to allow others to use our work.

9 Acknowledgments

We thank Cristian Cadar, Daniel Dunbar, Dawson Engler, Patrice Godefoid, Michael Levin, and Paul Twohey for discussions about their respective systems and dynamic test generation. We thank Paul Twohey for helpful tips on the engineering of test machines. We thank Chris Karlof for reading a draft of our paper on short notice. We thank the SUPERB TRUST 2008 team for their work with Metafuzz and SmartFuzz during Summer 2008. We thank Li-Wen Hsu and Alex Fabrikant for their help with the metafuzz.com web site, and Sushant Shankar, Shiuan-Tzuo Shen, and Mark Winterrowd for their comments. We thank Erinn Clark, Charlie Miller, Prateek Saxena, Dawn Song, the Berkeley BitBlaze group, and the anonymous Oakland referees for feedback on earlier drafts of this work. This work was generously supported by funding from DARPA and by NSF grants CCF-0430585 and CCF-0424422.

References

[1] Aslani, M., Chung, N., Doherty, J., Stockman, N., and Quach, W. Comparison of blackbox and whitebox fuzzers in finding software bugs, November 2008. TRUST Retreat Presentation.

[2] blexim. Basic integer overflows. *Phrack 0x0b* (2002).

[3] blexim. Basic integer overflows. *Phrack 0x0b*, 0x3c (2002). `http://www.phrack.org/archives/60/p60-0x0a.txt`.

[4] Brumley, D., Chieh, T., Johnson, R., Lin, H., and Song, D. RICH : Automatically protecting against integer-based vulnerabilities. In *NDSS (Symp. on Network and Distributed System Security)* (2007).

[5] Brumley, D., Newsome, J., Song, D., Wang, H., and Jha, S. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006).

[6] Cadar, C., Dunbar, D., and Engler, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI 2008* (2008).

[7] Cadar, C., and Engler, D. EGT: Execution generated testing. In *SPIN* (2005).

[8] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R. EXE: Automatically Generating Inputs of Death. In *ACM CCS* (2006).

[9] Chen, K., and Wagner, D. Large-scale analysis of format string vulnerabilities in debian linux. In *PLAS - Programming Languages and Analysis for Security* (2007). `http://www.cs.berkeley.edu/~daw/papers/fmtstr-plas07.pdf`.

[10] Corporation, M. Vulnerability Type Distributions in CVE, May 2007. `http://cve.mitre.org/docs/vuln-trends/index.html`.

[11] DeMott, J. The evolving art of fuzzing. In *DEF CON 14* (2006). `http://www.appliedsec.com/files/The_Evolving_Art_of_Fuzzing.odp`.

[12] Ganesh, V., and Dill, D. STP: A decision procedure for bitvectors and arrays. CAV 2007, 2007. `http://theory.stanford.edu/~vganesh/stp.html`.

[13] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation)* (Chicago, June 2005), pp. 213–223.

[14] GODEFROID, P., LEVIN, M., AND MOLNAR, D. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)* (San Diego, February 2008). `http://research.microsoft.com/users/pg/public_psfiles/ndss2008.pdf`.

[15] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Active Property Checking. Tech. rep., Microsoft, 2007. MSR-TR-2007-91.

[16] HOCEVAR, S. zzuf, 2007. `http://caca.zoy.org/wiki/zzuf`.

[17] LANZI, A., MARTIGNONI, L., MONGA, M., AND PALEARI, R. A smart fuzzer for x86 executables. In *Software Engineering for Secure Systems, 2007. SESS '07: ICSE Workshops 2007* (2007). `http://idea.sec.dico.unimi.it/~roberto/pubs/sess07.pdf`.

[18] LARSON, E., AND AUSTIN, T. High Coverage Detection of Input-Related Security Faults. In *Proceedings of 12th USENIX Security Symposium* (Washington D.C., August 2003).

[19] LEBLANC, D. Safeint 3.0.11, 2008. `http://www.codeplex.com/SafeInt`.

[20] LMH. Month of kernel bugs, November 2006. `http://projects.info-pull.com/mokb/`.

[21] MICROSOFT CORPORATION. Prefast, 2008.

[22] MILLER, B. P., FREDRIKSEN, L., AND SO, B. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery 33*, 12 (1990), 32–44.

[23] MOORE, H. Month of browser bugs, July 2006. `http://browserfun.blogspot.com/`.

[24] NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI - Programming Language Design and Implementation* (2007).

[25] O'CALLAHAN, R. Chronicle-recorder, 2008. `http://code.google.com/p/chronicle-recorder/`.

[26] SEWARD, J., AND NETHERCOTE, N. Using valgrind to detect undefined memory errors with bit precision. In *Proceedings of the USENIX Annual Technical Conference* (2005). `http://www.valgrind.org/docs/memcheck2005.pdf`.

[27] VUAGNOUX, M. Autodafe: An act of software torture. In *22nd Chaos Communications Congress, Berlin, Germany* (2005). `autodafe.sourceforge.net`.

[28] WANG, T., WEI, T., LIN, Z., AND ZOU, W. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Network Distributed Security Symposium (NDSS)* (2009).

Memory Safety for Low-Level Software/Hardware Interactions

John Criswell
University of Illinois
criswell@uiuc.edu

Nicolas Geoffray
Université Pierre et Marie Curie
INRIA/Regal
nicolas.geoffray@lip6.fr

Vikram Adve
University of Illinois
vadve@uiuc.edu

Abstract

Systems that enforce memory safety for today's operating system kernels and other system software do not account for the behavior of low-level software/hardware interactions such as memory-mapped I/O, MMU configuration, and context switching. Bugs in such low-level interactions can lead to violations of the memory safety guarantees provided by a safe execution environment and can lead to exploitable vulnerabilities in system software. In this work, we present a set of program analysis and run-time instrumentation techniques that ensure that errors in these low-level operations do not violate the assumptions made by a safety checking system. Our design introduces a small set of abstractions and interfaces for manipulating processor state, kernel stacks, memory mapped I/O objects, MMU mappings, and self modifying code to achieve this goal, without moving resource allocation and management decisions out of the kernel. We have added these techniques to a compiler-based virtual machine called Secure Virtual Architecture (SVA), to which the standard Linux kernel has been ported previously. Our design changes to SVA required only an additional 100 lines of code to be changed in this kernel. Our experimental results show that our techniques prevent reported memory safety violations due to low-level Linux operations and that *these violations are not prevented by SVA without our techniques*. Moreover, the new techniques in this paper introduce very little overhead over and above the existing overheads of SVA. Taken together, these results indicate that it is clearly worthwhile to add these techniques to an existing memory safety system.

1 Introduction

Most modern system software, including commodity operating systems and virtual machine monitors, are vulnerable to a wide range of security attacks because they are written in unsafe languages like C and C++. In fact, there has been an increase in recent years of attack methods against the operating system (OS) kernel. There are reported vulnerabilities for nearly all commodity OS kernels (e.g., [2, 28, 43]). One recent project went so far as to present one OS kernel bug every day for a month for several different open source and commercial kernels [26] (several of these bugs are exploitable vulnerabilities). Preventing these kinds of attacks *requires protecting the core kernel and not just device drivers*: many of the vulnerabilities are in core kernel components [19, 40, 41, 43, 46].

To counter these threats, there is a growing body of work on using language and compiler techniques to enforce *memory safety* (defined in Section 2) for OS code. These include new OS designs based on safe languages [4, 18, 22, 33], compiler techniques to enforce memory safety for commodity OSs in unsafe languages [10], and instrumentation techniques to isolate a kernel from extensions such as device drivers [45, 47, 51]. We use the term "*safe execution environment*" (again defined in Section 2) to refer to the guarantees provided by a system that enforces memory safety for operating system code. Singularity, SPIN, JX, JavaOS, SafeDrive, and SVA are examples of systems that enforce a safe execution environment.

Unfortunately, all these memory safety techniques (even implementations of safe programming languages) make assumptions that are routinely violated by low-level interactions between an OS kernel and hardware. Such assumptions include a static, one-to-one mapping between virtual and physical memory, an idealized processor whose state is modified only via visible program instructions, I/O operations that cannot overwrite standard memory objects except input I/O targets, and a protected stack modifiable only via load/store operations to local variables. For example, when performing type checking on a method, a safe language like Java or Modula-3 or compiler techniques like those in SVA assume that pointer values are only defined via visible program operations. In a kernel, however, a *buggy* kernel operation might overwrite program state while it is off-processor and that state might later be swapped in between the definition and the use of the pointer value, a *buggy* MMU mapping might remap the underlying physical memory to a different virtual page holding data of a different type, or a *buggy* I/O operation might bring corrupt pointer values into memory.

In fact, as described in Section 7.1, we have injected bugs into the Linux kernel ported to SVA that are capable of *disabling the safety checks that prevented 3 of the 4 exploits* in the experiments reported in the original SVA work [10]: the bugs modify the metadata used to track array bounds and thus allow buffer overruns to go undetected. Similar vulnerabilities can be introduced with other bugs in low-level operations. For example, there are reported MMU bugs [3, 39, 42] in previous versions of the Linux kernel that are logical errors in the MMU configuration and could lead to kernel exploits.

A particularly nasty and very recent example is an insidious error in the Linux 2.6 kernel (not a device driver) that led to severe (and sometimes permanent) corruption of the e1000e network card [9]. The kernel was overwriting I/O device memory with the x86 `cmpxchg` instruction, which led to corrupting the hardware. This bug was caused by a write through a dangling pointer to I/O device memory. This bug took weeks of debugging by multiple teams to isolate. A strong memory safety system should prevent or constrain such behavior, either of which would have prevented the bug.

All these problems can, in theory, be prevented by moving some of the kernel-hardware interactions into a virtual machine (VM) and providing a high-level interface for the OS to invoke those operations safely. If an OS is *co-designed* with a virtual machine implementing the underlying language, e.g., as in JX [18], then eliminating such operations from the kernel could be feasible. For commodity operating systems such as Linux, Mac OS X, and Windows, however, reorganizing the OS in such a way may be difficult or impossible, requiring, at a minimum, substantial changes to the OS design. For example, in the case of SVA, moving kernel-hardware interactions into the SVA VM would require extensive changes to any commodity system ported to SVA.

Virtual machine monitors (VMMs) such as VMWare or Xen [16] do not solve this problem. They provide sufficiently strong guarantees to enforce isolation and fair resource sharing between different OS instances (i.e., different "domains") but do not enforce memory safety *within* a single instance of an OS. For example, a VMM prevents one OS instance from modifying memory mappings for a different instance but does not protect an OS instance from a bug that maps multiple pages of its own to the same physical page, thus violating necessary assumptions used to enforce memory safety. In fact, a VMM would not solve any of the reported real-world problems listed above.

In this paper, we present a set of novel techniques to prevent low-level kernel-hardware interactions from violating memory safety in an OS executing in a safe execution environment. There are two key aspects to our approach: (1) we define carefully a set of abstractions (an API) between the kernel and the hardware that enables a lightweight run-time checker to protect hardware resources and their behaviors; and (2) we leverage the existing safety checking mechanisms of the safe execution environment to *optimize* the extra checks that are needed for this monitoring. Some examples of the key resources that are protected by our API include processor state in CPU registers; processor state saved in memory on context-switches, interrupts, or system calls; kernel stacks; memory-mapped I/O locations; and MMU configurations. Our design also permits limited versions of self-modifying code that should suffice for most kernel uses of the feature. Most importantly, our design provides these assurances while leaving essentially all the *logical control* over hardware behavior in the hands of the kernel, i.e., no policy decisions or complex mechanisms are taken out of the kernel. Although we focus on preserving memory safety for commodity operating systems, these principles would enable any OS to reduce the likelihood and severity of failures due to bugs in low-level software-hardware interactions.

We have incorporated these techniques in the SVA prototype and correspondingly modified the Linux 2.4.22 kernel previously ported to SVA [10]. Our new techniques required a significant redesign of SVA-OS, which is the API provided by SVA to a kernel for controlling hardware and using privileged hardware operations. The changes to the Linux kernel were generally simple changes to use the new SVA-OS API, even though the new API provides much more powerful protection for the entire kernel. We had to change only about 100 lines in the SVA kernel to conform to the new SVA-OS API.

We have evaluated the ability of our system to prevent kernel bugs due to kernel-hardware interactions, both with real reported bugs and injected bugs. Our system prevents two MMU bugs in Linux 2.4.22 for which exploit code is available. Both bugs crash the kernel when run under the original SVA. Moreover, as explained in Section 7.1, we would also prevent the e1000e bug in Linux 2.6 if that kernel is run on our system. Finally, the system successfully prevents all the low-level kernel-hardware interaction errors we have tried to inject.

We also evaluated the performance overheads for two servers and three desktop applications (two of which perform substantial I/O). Compared with the original SVA, the new techniques in this paper add very low or negligible overheads. Combined with the ability to prevent real-world exploits that would be missed otherwise, it clearly seems worthwhile to add these techniques to an existing memory safety system.

To summarize, the key contributions of this work are:

- We have presented novel mechanisms to ensure that low-level kernel-hardware interactions (e.g., context switching, thread creation, MMU changes, and I/O operations) do not violate assumptions used to enforce a safe execution environment.
- We have prototyped these techniques and shown that they can be used to enforce the assumptions made by a memory safety checker for a commodity kernel such as Linux. To our knowledge, no previous safety enforcement technique provides such guarantees to commodity system software.
- We have evaluated this system experimentally and shown that it is effective at preventing exploits in the above operations in Linux while incurring little overhead over and above the overhead of the underlying safe execution environment of SVA.

2 Breaking Memory Safety with Low-Level Kernel Operations

Informally, *a program is type-safe* if all operations in the program respect the types of their operands. For the purposes of this work, we say *a program is memory safe* if every memory access uses a previously initialized pointer variable; accesses the same object to which the pointer pointed initially;[1] and the object has not been deallocated. Memory safety is necessary for type safety (conversely, type safety implies memory safety) because dereferencing an uninitialized pointer, accessing the target object out of bounds, or dereferencing a dangling pointer to a freed object, can all cause accesses to unpredictable values and hence allow illegal operations on those values.

A safe programming language guarantees type safety and memory safety for all legal programs [34]; these guarantees also imply a *sound operational semantics* for programs in the language. Language implementations enforce these guarantees through a combination of compile-time type checking, automatic memory management (e.g., garbage collection or region-based memory management) to prevent dangling pointer references, and run-time checks such as array bounds checks and null pointer checks.

Four recent compiler-based systems for C, namely, CCured [30], SafeDrive [51], SAFECode [15], and SVA [10] enforce similar, but weaker, guarantees for C code. Their guarantees are weaker in two ways: (a) they provide type safety for only a subset of objects, and (b) three of the four systems — SafeDrive, SAFECode and SVA — permit dangling pointer references (use-after-free) to avoid the need for garbage collection. Unlike SafeDrive, however, SAFECode and SVA *guarantee* that dangling pointer references do not invalidate any of the other safety properties, i.e., partial type safety, memory safety, or a sound operational semantics [14, 15]. We refer to all these systems – safe languages or safety checking compilers – as providing a *safe execution environment*.

All of the above systems make some fundamental assumptions regarding the run-time environment in enforcing their safety guarantees. In particular, these systems assume that the code segment is static; control flow can only be altered through explicit branch instructions, call instructions, and visible signal handling; and that data is stored either in a flat, unchanging address space or in processor registers. Furthermore, data can only be read and written by direct loads and stores to memory or direct changes to processor registers.

Low-level system code routinely violates these assumptions. Operating system kernels, virtual machine monitors, language virtual machines such as a JVM or CLR, and user-level thread libraries often perform operations such as context switching, direct stack manipulation, memory mapped I/O, and MMU configuration, that violate these assumptions. More importantly, as explained in the rest of this section, perfectly *type-safe* code can violate many of these assumptions (through logical errors), i.e., such errors will not be prevented by the language in the first place. This is unacceptable for safe language implementations and, at least, undesirable for system software because these violations can compromise safety and soundness and thus permit the vulnerabilities a safe language was designed to prevent, such as buffer overflows or the creation of illegal pointer values.

There are, in fact, a small number of root causes (or categories of root causes) of all these violations. This section enumerates these root causes, and the next section describes the design principles by which these root causes can be eliminated. We assume throughout this discussion that a safety checker (through some combination of static and run-time checking) enforces the language-level safety guarantees of a safe execution environment, described above, for the kernel.[2] This allows us to assume that the run-time checker itself is secure, and that static analysis can be used soundly on kernel code [15]. Our goal is to ensure the integrity of the *as-*

[1] Note that we permit a pointer to "leave" its target object and later return, as long as it is not accessed while it is out of bounds [32].

[2] This work focuses on enforcing memory safety for the kernel. The same techniques could be applied to protect user-space threads from these violations.

sumptions made by this safety checker. We refer to the extensions that enforce these assumptions as a *verifier*.

Briefly, the fundamental categories of violations are:

- corrupting processor state when held in registers or memory;
- corrupting stack values for kernel threads;
- corrupting memory mapped I/O locations;
- corrupting code pages in memory;
- other violations that can corrupt arbitrary memory locations, including those listed above.

Unlike the last category, the first four above are errors that are specific to individual categories of memory.

2.1 Corrupting Processor State

Corrupting processor state can corrupt both data and control flow. The verifier must first ensure that processor state cannot be corrupted while on the processor itself, i.e., preventing arbitrary changes to processor registers. In addition, however, standard kernels save processor state (i.e., data and control registers) in memory where it is accessible by standard (even type-safe) load and store instructions. Any (buggy) code that modifies this state before restoring the state to the processor can alter control flow (the program counter, stack pointer, return address register, or condition code registers) or data values. In safe systems that permit dangling pointer references, processor state can also be corrupted if the memory used to hold saved processor state (usually located on the heap [5]) is freed and reallocated for other purposes.

Note that there are cases where the kernel makes explicit, legal, changes to the interrupted state of user-space code. For example, during signal handler dispatch, the kernel modifies interrupted program state that has been saved to memory, including the interrupted program's program counter and stack pointer [5]. Also, returning from a signal handler requires undoing the modifications made by signal delivery. The verifier must be able to distinguish legal from illegal changes to saved state.

2.2 Corrupting Stack State

The kernel directly manages the stacks of both user and kernel threads; it allocates and deallocates memory to hold them, sets up initial stack frames for new threads and signal handlers, and switches between stacks during a context switch or interrupt/system call return.

Memory for the stack is obtained from some standard memory allocation. Several safety violations are possible through this allocated memory. First, the memory for the stack should only be used for stack frames created during normal function calls and not directly modified via arbitrary stores;[3] such stores could corrupt the stack frames and thus compromise safety. Second, the memory for the stack must not be deallocated and reused for other memory objects while the stack is still in use. Third, a context switch must switch to a stack and its corresponding saved processor state as a pair; a context switch should not load processor state with the wrong stack or with a stack that has been deallocated. Fourth, after a stack is deallocated, live pointers to local variables allocated on the stack must not be dereferenced (the exiting thread may have stored pointers to such objects into global variables or the heap where they are accessible by other threads).

2.3 Corrupting Memory-Mapped I/O

Most systems today use memory-mapped I/O for controlling I/O devices and either memory-mapped I/O or DMA for performing data transfers. Many hardware architectures treat regular memory and memory-mapped I/O device memory (hereafter called I/O memory) identically, allowing a single set of hardware instructions to access both. From a memory safety perspective, however, it is better to treat regular memory and I/O memory as disjoint types of memory that are accessed using distinct instructions. First, I/O memory is not semantically the same as regular memory in that a load may not return the value last stored into the location; program analysis algorithms (used to enforce and optimize memory safety [15]) are not sound when applied to such memory. Second, I/O memory creates side-effects that regular memory does not. While erroneously accessing I/O memory instead of regular memory may not be a memory safety violation per se, it is still an error with potentially dire consequences. For example, the e1000e bug [9] caused fatal damage to hardware when an instruction (`cmpxchg`) that was meant to write to memory erroneously accessed memory-mapped I/O registers, which has undefined behavior. Therefore, for soundness of regular memory safety and for protection against a serious class of programming errors, it is best to treat regular memory and I/O memory as disjoint.

2.4 Corrupting Code

Besides the general memory corruption violations described below, there are only two ways in which the contents of code pages can be (or appear to be) corrupted. One is through self-modifying code (SMC); the other is through incorrect program loading operations (for new code or loadable kernel modules).

[3] An exception is when Linux stores the process's task structure at the bottom of the stack.

Self-modifying code directly modifies the sequence of instructions executed by the program. This can modify program behavior in ways not predicted by the compiler and hence bypass any of its safety checking techniques. For these reasons, most type-safe languages prohibit self-modifying code (which is distinct from "self-extending" behaviors like dynamic class loading). However, modern kernels use limited forms of self-modifying code for operations like enabling and disabling instrumentation [9] or optimizing synchronization for a specific machine configuration [8]. To allow such optimizations, the verifier must define limited forms of self-modifying code that do not violate the assumptions of the safety checker.

Second, the verifier must ensure that any program loading operation is implemented correctly. For example, any such operation, including new code, self-modifying code, or self-extending code (e.g., loadable kernel modules) requires flushing the instruction cache. Otherwise, cached copies of the old instructions may be executed out of the I-cache (and processors with split instruction/data caches may even execute old instructions with fresh data). This may lead to arbitrary memory safety violations for the kernel or application code.

2.5 General Memory Corruption

Finally, there are three kinds of kernel functionality that can corrupt arbitrary memory pages: (1) MMU configuration; (2) page swapping; and (3) DMA. Note that errors in any of these actions are generally invisible to a safety checking compiler and can violate the assumptions made by the compiler, as follows.

First, the kernel can violate memory safety with direct operations on virtual memory. Fundamentally, most of these are caused by creating an incorrect virtual-to-physical page mapping. Such errors include modifying mappings in the range of kernel stack memory, mapping the same physical page into two virtual pages (unintentionally), and changing a virtual-to-physical mapping for a live virtual page. As before, any of these errors can occur even with a type-safe language.

A second source of errors is in page swapping. When a page of data is swapped in on a page fault, memory safety can be violated if the data swapped in is not identical to the data swapped out from that virtual page. For example, swapping in the wrong data can cause invalid data to appear in pointers that are stored in memory.

Finally, a third source of problems is DMA. DMA introduces two problems. First, a DMA configuration error, device driver error, or device firmware error can cause a DMA transfer to overwrite arbitrary physical memory, violating type-safety assumptions. Second, even a correct DMA transfer may bring in unknown data which cannot be used in a type-safe manner, unless special language support is added to enable that, e.g., to prevent such data being used as pointer values, as in the SPIN system [21].

3 Design Principles

We now describe the general design principles that a memory safe system can use to prevent the memory errors described in Section 2. As described earlier, we assume a safety checker already exists that creates a safe execution environment; the *verifier* is the set of extensions to the safety checker that enforces the underlying assumptions of the checker. Examples of safety checkers that could benefit directly from such extensions include SVA, SafeDrive, and XFI. We also assume that the kernel source code is available for modification.

Processor State: Preventing the corruption of processor state involves solving several issues. First, the verifier must ensure that the kernel does not make arbitrary changes to CPU registers. Most memory safe systems already do this by not providing instructions for such low-level modifications. Second, the verifier must ensure that processor state saved by a context switch, interrupt, trap, or system call is not accessed by memory load and store instructions. To do this, the verifier can allocate the memory used to store processor state within its own memory and allow the kernel to manipulate that state via special instructions that take an opaque handle (e.g., a unique integer) to identify which saved state buffer to use. For checkers like SVA and SafeDrive, the safety checker itself prevents the kernel from manufacturing and using pointers to these saved state buffers (e.g., via checks on accesses that use pointers cast from integers). Additionally, the verifier should ensure that the interface for context switching leaves the system in a known state, meaning that a context switch should either succeed completely or fail.

There are operations in which interrupted program state needs to be modified by the kernel (e.g., signal handler dispatch). The verifier must provide instructions for doing controlled modifications of interrupted program state; for example, it can provide an instruction to push function call frames on to an interrupted program's stack [11]. Such instructions must ensure that either their modifications cannot break memory safety or that they only modify the saved state of interrupted user-space programs (modifying user-space state cannot violate the kernel's memory safety).

Stack State: The memory for a kernel stack and for the processor state object (the in-memory representation of processor state) must be created in a single operation (instead of by separate operations), and the verifier should ensure that the kernel stack and processor state object are always used and deallocated together. To ease implementation, it may be desirable to move some low-level, error-prone stack and processor state object initialization code into the verifier. The verifier must also ensure that memory loads and stores do not modify the kernel stack (aside from accessing local variables) and that local variables stored on the stack can no longer be accessed when the kernel stack is destroyed.

Memory-mapped I/O: The verifier must require that all I/O object allocations be identifiable in the kernel code, (e.g., declared via a pseudo-allocator). It should also ensure that only special I/O read and write instructions can access I/O memory (these special instructions can still be translated into regular memory loads and stores for memory-mapped I/O machines) and that these special instructions cannot read or write regular memory objects. If the verifier uses type-safety analysis to optimize run-time checks, it should consider I/O objects (objects analogous to memory objects but that reside in memory-mapped I/O pages) to be *type-unsafe* as the device's firmware may use the I/O memory in a type-unsafe fashion. Since it is possible for a pointer to point to both I/O objects and memory objects, the verifier should place run-time checks on such pointers to ensure that they are accessing the correct type of object (memory or I/O), depending upon the operation in which the pointer is used.

Kernel Code: The verifier must not permit the kernel to modify its code segment. However, it can support a limited version of self-modifying code that is easy to implement and able to support the uses of self-modifying code found in commodity kernels. In our design, the kernel can specify regions of code that can be enabled and disabled. The verifier will be responsible for replacing native code with no-op instructions when the kernel requests that code be disabled and replacing the no-ops with the original code when the kernel requests the code to be re-enabled. When analyzing code that can be enabled and disabled, the verifier can use conservative analysis techniques to generate results that are correct regardless of whether the code is enabled or disabled. For example, our pointer analysis algorithm, like most other inter-procedural ones used in production compilers, computes a *may-points-to* result [24], which can be computed with the code enabled; it will still be correct, though perhaps conservative, if the code is disabled.

To ensure that the instruction cache is properly flushed, our design calls for the safety checker to handle all translation to native code. The safety checker already does this in JVMs, safe programming languages, and in the SVA system [10]. By performing all translation to native code, the verifier can ensure that all appropriate CPU caches are flushed when new code is loaded into the system.

General Memory Corruption: The verifier must implement several types of protection to handle the general memory corruption errors in Section 2.5.

MMU configuration: To prevent MMU misconfiguration errors, the verifier must be able to control access to hardware page tables or processor TLBs and vet changes to the MMU configuration before they are applied. Implementations can use para-virtualization techniques [16] to control the MMU. The verifier must prevent pages containing kernel memory objects from being made accessible to non-privileged code and ensure that pages containing kernel stack frames are not mapped to multiple virtual addresses (i.e., double mapped) or unmapped before the kernel stack is destroyed.[4] Verifiers optimizing memory access checks must also prohibit double mappings of pages containing *type known* objects; this will prevent data from being written into the page in a way that is not detected by compiler analysis techniques. Pages containing type-unknown memory objects can be mapped multiple times since run-time checks already ensure that the data within them does not violate any memory safety properties. The verifier must also ensure that MMU mappings do not violate any other analysis results upon which optimizations depend.

Page swapping: For page swapping, the kernel must notify the verifier before swapping a page out (if not, the verifier will detect the omission on a subsequent physical page remapping operation). The verifier can then record any metadata for the page as well as a checksum of the contents and use these when the page is swapped back in to verify that the page contents have not changed.

DMA: The verifier should prevent DMA transfers from overwriting critical memory such as the kernel's code segment, the verifier's code and data, kernel stacks (aside from local variables), and processor state objects. Implementation will require the use of IOMMU techniques like those in previous work [17, 36]. Additionally, if the verifier uses type information to optimize memory safety checks, it must consider the memory accessible via DMA as type-unsafe. This solution is strictly stronger than previous work (like that in SPIN [21]): it allows pointer values in input data whereas they do not (and they do not guarantee type safety for other input data).

Entry Points: To ensure control-flow integrity, the kernel should not be entered in the middle of a function. Therefore, the verifier must ensure that all interrupt, trap, and system call handlers registered by the kernel are the initial address of a valid function capable of servicing the interrupt, trap, or system call, respectively.

[4]We assume the kernel does not swap stack pages to disk, but the design can be extended easily to allow this.

4 Background: Secure Virtual Architecture

The Secure Virtual Architecture (SVA) system (Figure 1) places a compiler-based virtual machine between the processor and the traditional software stack [10, 11]. The virtual machine (VM) presents a virtual instruction set to the software stack and translates virtual instructions to the processor's native instruction set either statically (the default) or dynamically. The virtual instruction set is based on the LLVM code representation [23], which is designed to be low-level and language-independent, but still enables sophisticated compiler analysis and transformation techniques. This instruction set can be used for both user-space and kernel code [11].

SVA optionally provides strong safety guarantees for C/C++ programs compiled to its virtual instruction set, close to that of a safe language. The key guarantees are:

1. *Partial type safety*: Operations on a subset of data are type safe.
2. *Memory safety*: Loads and stores only access the object to which the dereferenced pointer initially pointed, and within the bounds of that object.
3. *Control flow integrity*: The kernel code only follows execution paths predicted by the compiler; this applies to both branches and function calls.
4. *Tolerating dangling pointers*: SVA does not detect uses of dangling pointers but *guarantees that they are harmless*, either via static analysis (for type-safe data) or by detecting violations through run-time checks (for non-type safe data).
5. *Sound operational semantics*: SVA defines a virtual instruction set with an operational semantics that is guaranteed not to be violated by the kernel code; sound program analysis or verification tools can be built on this semantics.

Briefly, SVA provides these safety guarantees as follows. First, it uses a pointer analysis called Data Structure Analysis (DSA) [24] to partition memory into logical partitions ("points to sets") and to check which partitions are always accessed or indexed with a single type. These partitions are called "type-known" (TK); the rest are "type-unknown" (TU). SVA then creates a run-time representation called a "metapool" for each partition. It maintains a lookup table in each metapool of memory objects and their bounds to support various run-time checks. Maintaining a table per metapool instead of a single global table greatly improves the performance of the run-time checks [14].

Compile-time analysis with DSA guarantees that all TK partitions are type-safe. Moreover, all uses of data

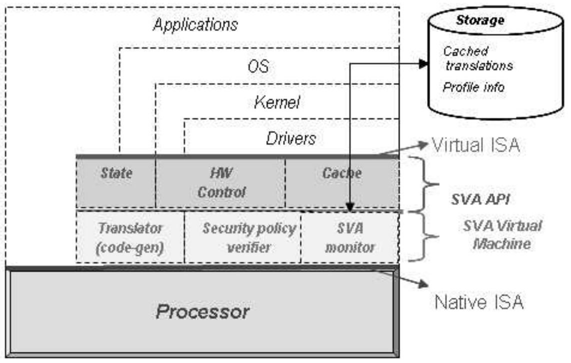

Figure 1: System Organization with SVA [10]

and function pointers loaded *out of* TK partitions are type safe. SVA simply has to ensure that dangling pointer references to TK metapools cannot create a type violation by enforcing two constraints: (a) objects in TK metapools are aligned identically; and (b) freed memory from such a metapool is never used for a different metapool until the former is destroyed. These constraints are enforced by modifying the kernel allocators manually during the process of porting the kernel to SVA; this means that the allocators are effectively trusted and not checked. To enforce these constraints for stack objects belonging to TK metapools, SVA automatically modifies the kernel code to allocate such objects on the heap. Together, these guarantee that a pointer to a freed object and a new object (including array elements) access values of identical type [15].

At run-time, the SVA VM (thereafter called VM) performs a number of additional checks and operations. All globals and allocated objects are registered in the metapool to which they belong (derived from the target partition of the return pointer). Loads and stores that use pointers loaded from TU metapools are checked by looking up the target address in the metapool lookup table. Note that this works whether or not the pointer value is a dangling pointer, and even for pointers "manufactured" by casting arbitrary integers. Similarly, it checks function pointers obtained from TU metapools to ensure that they only access one of the target functions of that pointer predicted by DSA. Run-time checks also ensure that pointers to TK objects that are loaded from TU memory objects are checked since a TU object may have an invalid value for the TK pointer. All array indexing operations for TK or TU metapools are checked in the lookup table, which records the bounds for each object [14][5].

Note that the VM relies on the safe execution environ-

[5]Note that we permit a pointer to "leave" its target object and later return, as long as it is not accessed while it is out of bounds [32].

ment to protect the VM code and data memory instead of using the MMU and incurring the cost of switching page tables on every VM invocation. Since the environment prevents access to unregistered data objects or outside the bounds of legal objects, we can simply monitor all run-time kernel object registrations and ensure that they do not reside in VM code or data pages.

A subset of the SVA instruction set, SVA-OS, provides instructions designed to support an operating system's special interaction with the hardware [10, 11]. These include instructions for loading from/storing to I/O memory, configuring the MMU, and manipulating program state. An important property is that a kernel ported to SVA using the SVA-OS instructions *contains no assembly code*; this simplifies the compiler's task of safety checking within SVA. Nevertheless, these instructions provide low-level hardware interactions that can generate all the problems described in Section 2 if used incorrectly; it is very difficult for the compiler to check their correct use in the original design. In particular, the VM does not perform any special checks for processor state objects, direct stack manipulation, memory mapped I/O locations, MMU configuration changes, or DMA operations. Also, it disallows self-modifying code.

For example, we tested two [39, 42] of the three reported low-level errors we found for Linux 2.4.22, the kernel version ported to SVA (we could not try the third [3] for reasons explained in Section 7.1). Although both are memory safety violations, *neither of them was detected or prevented by the original SVA*.

5 Design

Our design is an extension of the original Secure Virtual Architecture (SVA) described in Section 4. SVA provides strong memory safety guarantees for kernel code and an abstraction of the hardware that is both low-level (e.g., context switching, I/O, and MMU configuration policies are still implemented in the kernel), yet easy to analyze (because the SVA-OS instructions for interacting with hardware are slightly higher level than typical processor instructions). Below, we describe our extensions to provide memory safety in the face of errors in kernel-hardware interactions.

5.1 Context Switching

Previously, the SVA system performed context switching using the `sva_load_integer` and `sva_save_integer` instructions [10], which saved from and loaded into the processor the processor state (named Integer State). These instructions stored processor state in a kernel allocated memory buffer which could be later modified by memory-safe store instructions or freed by the kernel deallocator. Our new design calls a single instruction named `sva_swap_integer` (see Table 1) that saves the old processor state and loads the new state in a single operation.

This design has all of the necessary features to preserve memory safety when context switching. The `sva_swap_integer` instruction allocates the memory buffer to hold processor state within the VM's memory and returns an opaque integer identifier which can be used to re-load the state in a subsequent call to `sva_swap_integer`. Combined with SVA's original protections against manufactured pointers, this prevents the kernel from modifying or deallocating the saved processor state buffer. The design also ensures correct deallocation of the memory buffer used to hold processor state. The VM tracks which identifiers are mapped to allocated state buffers created by `sva_swap_integer`; these memory buffer/identifier pairs are kept alive until the state is placed back on the processor by another call to `sva_swap_integer`. Once state is placed back on the processor, the memory buffer is deallocated, and the identifier invalidated to prevent the kernel from trying to restore state from a deallocated state buffer.

Finally, `sva_swap_integer` will either succeed to context switch and return an identifier for the saved processor state, or it will fail, save no processor state, and continue execution of the currently running thread. This ensures that the kernel stack and the saved processor state are always synchronized.

5.2 Thread Management

A thread of execution consists of a stack and a saved processor state that can be used to either initiate or continue execution of the thread. Thread creation is therefore comprised of three operations: allocating memory for the new thread's stack, initializing the new stack, and creating an initial state that can be loaded on to the processor using `sva_swap_integer`.

The VM needs to know where kernel stacks are located in order to prevent them from being written by load and store instructions. We introduce a new SVA instruction, `sva_declare_stack`, which a kernel uses to declare that a memory object will be used as a stack. During pointer analysis, any pointers passed to `sva_declare_stack` and pointers that alias with such pointers are marked with a special *DeclaredStack* flag; this flag indicates that run-time checks are needed on stores via such pointers to ensure that they are not writing into a kernel stack. The compiler, on seeing an `sva_declare_stack` instruction, will also verify, statically (via pointer analysis) if possible but at run-time if necessary, that the memory object used for the new stack is either a global or heap object; this will prevent

Name	Description
`sva_swap_integer`	Saves the current processor state into an internal memory buffer, loads previously saved state referenced by its ID, and returns the ID of the new saved state.
`sva_declare_stack`	Declares that a memory object is to be used as a new stack.
`sva_release_stack`	Declares that a memory object is no longer used as a stack.
`sva_init_stack`	Initializes a new stack.

Table 1: SVA Instructions for Context Switching and Thread Creation.

stacks from being embedded within other stacks. After this check is done, `sva_declare_stack` will unregister the memory object from the set of valid memory objects that can be accessed via loads and stores and record the stack's size and location within the VM's internal data structures as a valid kernel stack.

To initialize a stack and the initial processor state that will use the memory as a stack, we introduce `sva_init_stack`; this instruction will initialize the stack and create a new saved Integer State which can be used in `sva_swap_integer` to start executing the new thread. The `sva_init_stack` instruction verifies (either statically or at run-time) that its argument has previously been declared as a stack using `sva_declare_stack`. When the new thread wakes up, it will find itself running within the function specified by the call to `sva_init_stack`; when this function returns, it will return to user-space at the same location as the original thread entered.

Deleting a thread is composed of two operations. First, the memory object containing the stack must be deallocated. Second, any Integer State associated with the stack that was saved on a context switch must be invalidated. When the kernel wishes to destroy a thread, it must call the `sva_release_stack` instruction; this will mark the stack memory as a regular memory object so that it can be freed and invalidates any saved Integer State associated with the stack.

When a kernel stack is deallocated, there may be pointers in global or heap objects that point to memory (i.e., local variables) allocated on that stack. SVA must ensure that dereferencing such pointers does not violate memory safety. Type-unsafe stack allocated objects are subject to load/store checks and are registered with the SVA virtual machine [10]. In order for the `sva_release_stack` instruction to invalidate such objects when stack memory is reclaimed, the VM records information on stack object allocations and associates this information with the metadata about the stack in which the object is allocated. In this way, when a stack is deallocated, any live objects still registered with the virtual machine are automatically invalidated as well; run-time checks will no longer consider these stack allocated objects to be valid objects. Type-known stack allocated objects can never be pointed to by global or heap objects; SVA already transforms such stack allocations into heap allocations [15, 10] to make dangling pointer dereferencing to type-known stack allocated objects safe [15].

5.3 Memory Mapped I/O

To ensure safe use of I/O memory, our system must be able to identify where I/O memory is located and when the kernel is legitimately accessing it.

Identifying the location of I/O memory is straightforward. In most systems, I/O memory is located at (or mapped into) known, constant locations within the system's address space, similar to global variables. In some systems, a memory-allocator-like function may remap physical page frames corresponding to I/O memory to a virtual memory address [5]. The insight is that I/O memory is grouped into objects just like regular memory; in some systems, such I/O objects are even allocated and freed like heap objects (e.g., Linux's `ioremap()` function [5]). To let the VM know where I/O memory is located, we must modify the kernel to use a pseudo-allocator that informs the VM of global I/O objects; we can also modify the VM to recognize I/O "allocators" like `ioremap()` just like it recognizes heap allocators like Linux's `kmalloc()` [5].

Given this information, the VM needs to determine which pointers may point to I/O memory. To do so, we modified the SVA points-to analysis algorithm [24] to mark the target (i.e., the "points-to set") of a pointer holding the return address of the I/O allocator with a special *I/O flag*. This also flags other pointers aliased to such a pointer because any two aliased pointers point to a common target [24].

We also modified the points-to analysis to mark I/O memory as *type-unknown*. Even if the kernel accesses I/O memory in a type-consistent fashion, the firmware on the I/O device may not. *Type-unknown* memory incurs additional run-time checks but allows kernel code to safely use pointer values in such memory as pointers.

We also extended SVA to record the size and virtual address location of every I/O object allocation and deallocation by instrumenting every call to the I/O allocator and deallocator functions. At run-time, the VM records these I/O objects in a per-metapool data structure that is disjoint from the structure used to record the bounds of regular memory objects. The VM also uses new run-time checks for checking I/O load and store instructions.

Since I/O pointers can be indexed like memory pointers (an I/O device may have an array of control registers), the bounds checking code must check both regular memory objects and I/O memory objects. Load and store checks on regular memory pointers *without the I/O flag* remain unchanged; they only consider memory objects. New run-time checks are needed on both memory and I/O loads and stores for pointers that have both the I/O flag and one or more of the memory flags (heap, stack, global) to ensure that they only access regular or I/O memory objects, respectively.

5.4 Safe DMA

We assume the use of an IOMMU for preventing DMA operations from overflowing object bounds or writing to the wrong memory address altogether [13]. The SVA virtual machine simply has to ensure that the I/O MMU is configured so that DMA operations cannot write to the virtual machine's internal memory, kernel code pages, pages which contain type-safe objects, and stack objects.

We mark all memory objects that may be used for DMA operations as type-unsafe, similar to I/O memory that is accessed directly. We assume that any pointer that is *stored into* I/O memory is a potential memory buffer for DMA operations. We require alias analysis to identify such stores; it simply has to check that the target address is in I/O memory and the store value is of pointer type. We then mark the points-to set of the store value pointer as *type-unknown*.

5.5 Virtual Memory

Our system must control the MMU and vet changes to its configuration to prevent safety violations and preserve compiler-inferred analysis results. Below, we describe the mechanism by which our system monitors and controls MMU configuration and then discuss how we use this mechanism to enforce several safety properties.

5.5.1 Controlling MMU Configuration

SVA provides different MMU interfaces for hardware TLB processors and software TLB processors [11]. For brevity, we describe only the hardware TLB interface and how our design uses it to control MMU configuration.

The SVA interface for hardware TLB systems (given in Table 2) is similar to those used in VMMs like Xen [16] and is based off the `paravirtops` interface [50] found in Linux 2.6. The page table is a 3-level page table, and there are instructions for changing mappings at each level. In this design, the OS first tells the VM which memory pages will be used for the page table (it must specify at what level the page will appear in the table); the VM then takes control of these pages by zeroing them (to prevent stale mappings from being used) and marking them read-only to prevent the OS from accessing them directly. The OS must then use special SVA instructions to update the translations stored in these page table pages; these instructions allow SVA to first inspect and modify translations before accepting them and placing them into the page table. The `sva_load_pagetable` instruction selects which page table is in active use and ensures that only page tables controlled by SVA are ever used by the processor. This interface, combined with SVA's control-flow integrity guarantees [10], ensure that SVA maintains control of all page mappings on the system.

5.5.2 Memory Safe MMU Configuration

For preventing memory safety violations involving the MMU, the VM needs to track two pieces of information. First, the VM must know the purpose of various ranges of the virtual address space; the kernel must provide the virtual address ranges of user-space memory, kernel data memory, and I/O object memory. This information will be used to prevent physical pages from being mapped into the wrong virtual addresses (e.g., a memory mapped I/O device being mapped into a virtual address used by a kernel memory object). A special instruction permits the kernel to communicate this information to the VM.

Second, the VM must know how physical pages are used, how many times they are mapped into the virtual address space, and whether any MMU mapping makes them accessible to unprivileged (i.e., user-space) code. To track this information, the VM associates with each physical page a set of flags and counters. The first set of flags are mutually exclusive and indicate the purpose of the page; a page can be marked as: `L1` (Level-1 page table page), `L2` (Level-2 page table page), `L3` (Level-3 page table page), `RW` (a standard kernel page holding memory objects), `IO` (a memory mapped I/O page), `stack` (kernel stack), `code` (kernel or SVA code), or `svamem` (SVA data memory). A second flag, the `TK` flag, specifies whether a physical page contains *type-known* data. The VM also keeps a count of the number of virtual pages mapped to the physical page and a count of the number of mappings that make the page accessible to user-space code.

The flags are checked and updated by the VM whenever the kernel requests a change to the page tables or performs relevant memory or I/O object allocation. Calls to the memory allocator are instrumented to set the RW and, if appropriate, the TK flag on pages backing the newly allocated memory object. On system boot, the VM sets the `IO` flag on physical pages known to be memory-mapped I/O locations. The `stack`

Name	Description
`sva_end_mem_init`	End of the virtual memory boot initialization. Flags all page table pages, and mark them read-only.
`sva_declare_l1_page`	Zeroes the page and flags it read-only and L1.
`sva_declare_l2_page`	Zeroes the page and flags it read-only and L2.
`sva_declare_l3_page`	Puts the default mappings in the page and flags it read-only and L3.
`sva_remove_l1_page`	Unflags the page read-only and L1.
`sva_remove_l2_page`	Unflags the page read-only and L2.
`sva_remove_l3_page`	Unflags the page read-only and L3.
`sva_update_l1_mapping`	Updates the mapping if the mapping belongs to an L1 page and the page is not already mapped for a type known pool, sva page, code page, or stack page.
`sva_update_l2_mapping`	Updates the mapping if the mapping belongs to an L2 page and the new mapping is for an L1 page.
`sva_update_l3_mapping`	Updates the mapping if the mapping belongs to an L3 page and the new mapping is for an L2 page.
`sva_load_pagetable`	Check that the physical page is an L3 page and loads it in the page table register.

Table 2: MMU Interface for a Hardware TLB Processor.

flag is set and cleared by `sva_declare_stack` and `sva_release_stack`, respectively. Changes to the page table via the instructions in Table 2 update the counters and the `L1`, `L2`, and`L3` flags.

The VM uses all of the above information to detect, at run-time, violations of the safety requirements in Section 3. Before inserting a new page mapping, the VM can detect whether the new mapping will create multiple mappings to physical memory containing *type-known* objects, map a page into the virtual address space of the VM or kernel code segment, unmap or double map a page that is part of a kernel stack, make a physical page containing kernel memory accessible to user-space code, or map memory-mapped I/O pages into a kernel memory object (or vice-versa). Note that SVA currently trusts the kernel memory allocators to (i) return different virtual addresses for every allocation, and (ii) not to move virtual pages from one metapool to another until the original metapool is destroyed.

5.6 Self-modifying Code

The new SVA system supports the restricted version of self-modifying code described in Section 3: OS kernels can disable and re-enable pre-declared pieces of code. SVA will use compile-time analysis carefully to ensure that replacing the code with no-op instructions will not invalidate the analysis results.

We define four new instructions to support self-modifying code. The first two instructions, `sva_begin_alt` and `sva_end_alt` enclose the code regions that may be modified at runtime. They must be properly nested and must be given a unique identifier. The instructions are not emitted in the native code. The two other instructions, `sva_disable_code` and `sva_enable_code` execute at runtime. They take the identifier given to the `sva_begin_alt` and `sva_end_alt` instructions. `sva_disable_code` saves the previous code and inserts no-ops in the code, and `sva_enable_code` restores the previous code.

With this approach, SVA can support most uses of self-modifying code in operating systems. For instance, it supports the `alternatives`[6] framework in Linux 2.6 [8] and Linux's `ftrace` tracing support [9] which disables calls to logging functions at run-time.

5.7 Interrupted State

On an interrupt, trap, or system call, the original SVA system saves processor state within the VM's internal memory and permits the kernel to use specialized instructions to modify the state via an opaque handle called the interrupt context [10, 11]. These instructions, which are slightly higher-level than assembly code, are used by the kernel to implement operations like signal handler dispatch and starting execution of user programs. Since systems such as Linux can be interrupted while running kernel code [5], these instructions can violate the kernel's memory safety if used incorrectly on interrupted kernel state. To address these issues, we introduce several changes to the original SVA design.

First, we noticed that all of the instructions that manipulate interrupted program state are either memory safe (e.g., the instruction that unwinds stack frames for kernel exception handling [11]) or only need to modify the interrupted state of user-space programs. Hence, all instructions that are not intrinsically memory safe will verify that they are modifying interrupted user-space program state. Second, the opaque handle to the interrupt context will be made implicit so that no run-time checks are needed to validate it when it is used. We have observed that the Linux kernel only operates upon the most recently created interrupt context; we do not see a need for other operating systems of similar design to do so, either. Without an explicit handle to the interrupt context's location in memory, no validation code is needed, and the kernel cannot create a pointer to the saved program state (except for explicit integer to pointer casts, uses of which will be caught by SVA's existing checks) [10].

[6]Linux 2.6, file include/asm-x86/alternative.h

5.8 Miscellaneous

To ensure control-flow integrity requirements, the VM assumes control of the hardware interrupt descriptor table; the OS kernel must use special instructions to associate a function with a particular interrupt, trap, or system call [11, 29]. Similar to indirect function call checks, SVA can use static analysis and run-time checks to ensure that only valid functions are registered as interrupt, trap, or system call handlers.

SVA provides two sets of atomic memory instructions: `sva_fetch_and_phi` where phi is one of several integer operations (e.g., add), and `sva_compare_and_swap` which performs an atomic compare and swap. The static and run-time checks that protect regular memory loads and stores also protect these operations.

6 Modifications to the Linux Kernel

We implemented our design by improving and extending the original SVA prototype and the SVA port of the Linux 2.4.22 kernel [10]. The previous section described how we modified the SVA-OS instructions. Below, we describe how we modified the Linux kernel to use these new instructions accordingly. We modified less than 100 lines from the original SVA kernel to port our kernel to the new SVA-OS API; the original port of the i386 Linux kernel to SVA modified 300 lines of architecture-independent code and 4,800 lines of architecture-dependent code [10].

6.1 Changes to Baseline SVA

The baseline SVA system in our evaluation (Section 7) is an improved version of the original SVA system [10] that is suitable for determining the extra overhead incurred by the run-time checks necessitated by the design in Section 5. First, we fixed several bugs in the optimization of run-time checks. Second, while the original SVA system does not analyze and protect the whole kernel, there is no fundamental reason why it cannot. Therefore, we chose to disable optimizations which apply only to incomplete kernel code for the experiments in Section 7. Third, the new baseline SVA system recognizes `ioremap()` as an allocator function even though it does not add run-time checks for I/O loads and stores. Fourth, we replaced most uses of the `__get_free_pages()` page allocator with `kmalloc()` in code which uses the page allocator like a standard memory allocator; this ensures that most kernel allocations are performed in kernel pools (i.e., `kmem_cache_ts`) which fulfill the requirements for allocators as described in the original SVA work [10].

We also modified the SVA Linux kernel to use the new SVA-OS instruction set as described below. This ensured that the only difference between our baseline SVA system and our SVA system with the low-level safety protections was the addition of the run-time checks necessary to ensure safety for context switching, thread management, MMU, and I/O memory safety.

6.2 Context Switching/Thread Creation

The modifications needed for context switching were straightforward. We simply modified the `switch_to` macro in Linux [5] to use the `sva_swap_integer` instruction to perform context switching.

Some minor kernel modifications were needed to use the new thread creation instructions. The original i386 Linux kernel allocates a single memory object which holds both a thread's task structure and the kernel stack for the thread [5], but this cannot be done on our system because `sva_declare_stack` requires that a stack consumes an entire memory object. For our prototype, we simply modified the Linux kernel to perform separate allocations for the kernel stack and the task structure.

6.3 I/O

As noted earlier, our implementation enhances the pointer analysis algorithm in SVA (DSA [24]) to mark pointers that may point to I/O objects. It does this by finding calls to the Linux `__ioremap()` function. To make implementation easier, we modified `ioremap()` and `ioremap_nocache()` in the Linux source to be macros that call `__ioremap()`.

Our test system's devices do not use global I/O memory objects, so we did not implement a pseudo allocator for identifying them. Also, we did not modify DSA to mark memory stored into I/O device memory as type-unknown. The difficulty is that Linux casts pointers into integers before writing them into I/O device memory. The DSA implementation does not have solid support for tracking pointers through integers i.e., it does not consider the case where an integer may, in fact, be pointing to a memory object. Implementing these changes to provide DMA protection is left as future work.

6.4 Virtual Memory

We implemented the new MMU instructions and run-time checks described in Section 5.5 and ported the SVA Linux kernel to use the new instructions. Linux already contains macros to allocate, modify and free page table pages. We modified these macros to use our new API (which is based on the `paravirtops` interface from Linux 2.6). We implemented all of the run-time checks except for those that ensure that I/O device memory isn't mapped into kernel memory objects. These checks require that the kernel allocate all I/O memory objects within a predefined range of the virtual address space, which our Linux kernel does not currently do.

7 Evaluation and Analysis

Our evaluation has two goals. First, we wanted to determine whether our design for low-level software/hardware interaction was effective at stopping security vulnerabilities in commodity OS kernels. Second, we wanted to determine how much overhead our design would add to an already existing memory-safety system.

7.1 Exploit Detection

We performed three experiments to verify that our system catches low-level hardware/software errors: First, we tried two different exploits on our system that were reported on Linux 2.4.22, the Linux version that is ported to SVA. The exploits occur in the MMU subsystem; both give an attacker root privileges. Second, we studied the e1000e bug [9]. We could not duplicate the bug because it occurs in Linux 2.6, but we explain why our design would have caught the bug if Linux 2.6 had been ported to SVA. Third, we inserted many low-level operation errors inside the kernel to evaluate whether our design prevents the safety violations identified in Section 2.

Linux 2.4.22 exploits. We have identified three reported errors for Linux 2.4.22 caused by low-level kernel-hardware interactions [3, 39, 42]. Our experiment is limited to these errors because we needed hardware/software interaction bugs that were in Linux 2.4.22. Of these, we could not reproduce one bug due to a lack of information in the bug report [3]. The other two errors occur in the `mremap` system call but are distinct errors.

The first exploit [42] is due to an overflow in a count of the number of times a page is mapped. The exploit code overflows the counter by calling `fork`, `mmap`, and `mremap` a large number of times. It then releases the page, giving it back to the kernel. However, the exploit code still has a reference to the page; therefore, if the page is reallocated for kernel use, the exploit code can read and modify kernel data. Our system catches this error because it disallows allocating kernel objects in a physical page mapped in user space.

The second exploit [39] occurs because of a missing error check in `mremap` which causes the kernel to place page table pages with valid page table entries into the page table cache. However, the kernel assumes that page table pages in the page table cache do not contain any entries. The exploit uses this vulnerability by calling `mmap, mremap` and `munmap` to release a page table page with page entries that contain executable memory. Then, on an `exec` system call, the linker, which executes with root privileges, allocates a page table page, which happens to be the previously released page. The end result is that the linker jumps to the exploit's executable memory and executes the exploit code with root privileges. The SVA VM prevents this exploit by always zeroing page table pages when they are placed in a page directory so that no new, unintended, memory mappings are created for existing objects.

The e1000e bug. The fundamental cause of the e1000e bug is a memory load/store (the x86 `cmpxchg` instruction) on a dangling pointer, which happens to point to an I/O object. The `cmpxchg` instruction has non-deterministic behavior on I/O device memory and may corrupt the hardware. The instruction was executed by the `ftrace` subsystem, which uses self-modifying code to trace the kernel execution. It took many weeks for skilled engineers to track the problem. With our new safety checks, SVA would have detected the bug at its first occurrence. The self-modifying code interface of SVA-OS only allows enabling and disabling of code; writes to what the kernel (incorrectly) thought was its code is not possible. SVA actually has a second line of defense if (hypothetically) the self-modifying code interface did not detect it: SVA would have prevented the I/O memory from being mapped into code pages, and thus prevented this corruption. (And, hypothetically again, if a dangling pointer to a data object had caused the bug, SVA would have detected any ordinary reads and writes trying to write to I/O memory locations.)

Kernel error injection. To inject errors, we added new system calls into the kernel; each system call triggers a specific kind of kernel/hardware interaction error that either corrupts memory or alters control flow. We inserted four different errors. The first error modifies the saved Integer State of a process so that an invalid Integer State is loaded when the process is scheduled. The second error creates a new MMU mapping of a page containing type-known kernel memory objects and modifies the contents of the page. The third error modifies the MMU mappings of pages in the stack range. The fourth error modifies the internal metadata of SVA to set incorrect bounds for all objects. This last error shows that with the original design, we can *disable the SVA memory safety checks that prevent Linux exploits*; in fact, it would not be difficult to do so with this bug alone for three of the four kernel exploits otherwise prevented by SVA [10].

All of the injected errors were caught by the new SVA implementation. With the previous implementation, these errors either crash the kernel or create undefined behavior. This gives us confidence about the correctness of our new design and implementation of SVA. Note that we only injected errors that our design addresses because we believe that our design is "complete" in terms of the possible errors due to kernel-hardware interactions. Nevertheless, the injection experiments are useful because they *validate that the design and implementation actually solve these problems.*

7.2 Performance

To determine the impact of the additional run-time checks on system performance, we ran several experiments with applications typically used on server and end-user systems. We ran tests on the original Linux 2.4.22 kernel (marked i386 in the figures and tables), the same kernel with the original SVA safety checks [10] (marked SVA), and the SVA kernel with our safety checks for low-level software/hardware interactions (marked SVA-OS).

It is important to note that an underlying memory safety system like SVA can incur significant run-time overhead for C code, especially for a commodity kernel like Linux that was not designed for enforcement of memory safety. Such a system is not the focus of this paper. Although we present our results relative to the original (unmodified) Linux/i386 system for clarity, we focus the discussion on the excess overheads introduced by SVA-OS beyond those of SVA since the new techniques in SVA-OS are the subject of the current work.

We ran these experiments on a dual-processor AMD Athlon 2100+ at 1,733 MHz with 1 GB of RAM and a 1 Gb/s network card. We configured the kernel as an SMP kernel but ran it in on a single processor since the SVA implementation is not yet SMP safe. Network experiments used a dedicated 1 Gb/s switch. We ran our experiments in single-user mode to prevent standard system services from adding noise to our performance numbers.

We used several benchmarks in our experiments: the thttpd Web server, the OpenSSH sshd encrypted file transfer service, and three local applications – bzip2 for file compression, the lame MP3 encoder, and a perl interpreter. These programs have a range of different demands on kernel operations. Finally, to understand why some programs incur overhead while others do not, we used a set of microbenchmarks including the HBench-OS microbenchmark suite [6] and two new tests we wrote for the poll and select system calls.

Application Performance First, we used ApacheBench to measure the file-transfer bandwidth of the thttpd web server [31] serving static HTML pages. We configured ApacheBench to make 5000 requests using 25 simultaneous connections. Figure 2 shows the results of both the original SVA kernel and the SVA kernel with the new run-time checks described in Section 5. Each bar is the average bandwidth of 3 runs of the experiment; the results are normalized to the

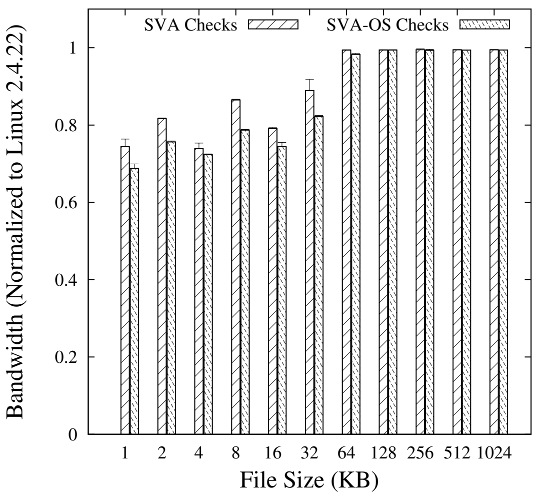

Figure 2: Web Server Bandwidth (Linux/i386 = 1.0)

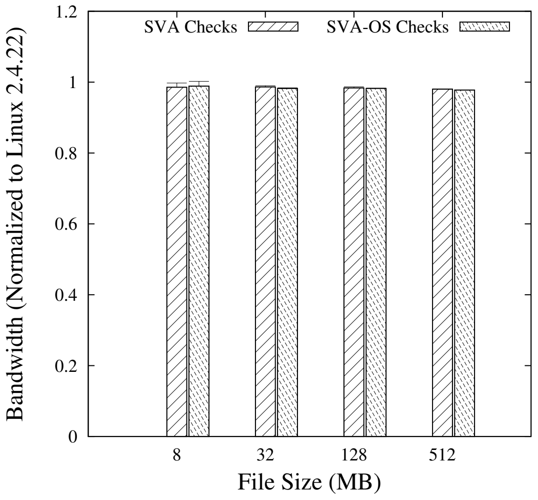

Figure 3: SSH Server Bandwidth (Linux/i386 = 1.0)

original i386 Linux kernel. For small files (1 KB - 32 KB) in which the original SVA system adds significant overhead, our new run-time checks incur a small amount of additional overhead (roughly a 9% decrease in bandwidth relative to the SVA kernel). However, for larger file sizes (64 KB or more), the SVA-OS checks add negligible overhead to the original SVA system.

We also measured the performance of `sshd`, a login server offering encrypted file transfer. For this test, we measured the bandwidth of transferring several large files from the server to our test client; the results are shown in Figure 3. For each file size, we first did a priming run to bring file system data into the kernel's buffer cache; subsequently, we transfered the file three times. Figure 3 shows the mean of the receive bandwidth of the three runs normalized to the mean receive bandwidth mea-

Benchmark	i386 (s)	SVA (s)	SVA-OS (s)	% Increase from i386 to SVA-OS	Description
bzip2	18.7 (0.47)	18.3 (0.47)	18.0 (0.00)	0.0%	Compressing 64 MB file
lame	133.3 (3.3)	132 (0.82)	126.0 (0.82)	-0.1%	Converting 206 MB WAV file to MP3
perl	22.3 (0.47)	22.3 (0.47)	22.3 (0.47)	0.0%	Interpreting scrabbl.pl from SPEC 2000

Table 3: Latency of Applications. Standard Deviation Shown in Parentheses.

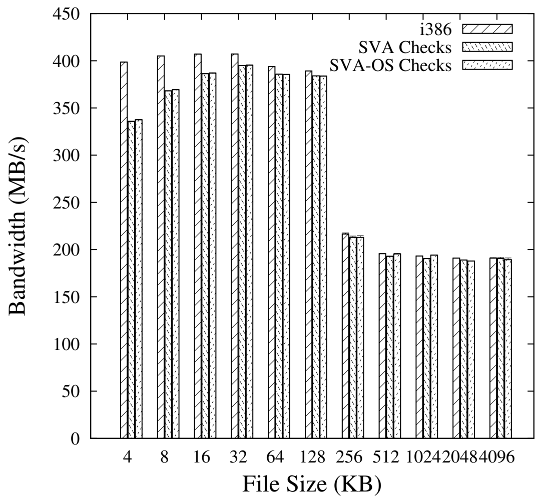

Figure 4: File System Bandwidth

Benchmark	i386 (s)	SVA (s)	SVA-OS (s)
bzip2	41	40	40
lame	203	202	202
perl	24	23	23

Table 4: Latency of Applications During Priming Run.

sured on the original i386 kernel; note that the units on the X-axis are MB. Our results indicate that there is no significant decrease in bandwidth due to the extra run-time checks added by the original SVA system or the new run-time checks presented in this paper. This outcome is far better than thttpd, most likely due to the large file sizes we transfered via `scp`. For large file sizes, the network becomes the bottleneck: transferring an 8 MB file takes 62.5 ms on a Gigabit network, but the overheads for basic system calls (shown in Table 5) show overheads of only tens of microseconds.

To see what effect our system would have on end-user application performance, we ran experiments on the client-side programs listed in Table 3. We tested bzip2 compressing a 64 MB file, the LAME MP3 encoder converting a 206 MB file from WAV to MP3 format, and the perl interpreter running the training input from the SPEC 2000 benchmark suite. For each test, we ran the program once to prime any caches within the operating system and then ran each program three times. Table 3 shows the average of the execution times of the three runs and the percent overhead that the applications experienced executing on the SVA-OS kernel relative to the original i386 Linux kernel. The results show that our system adds virtually no overhead for these applications, even though some of the programs (bzip2 and lame) perform substantial amounts of I/O. Table 4 shows the latency of the applications during their priming runs; our kernel shows no overhead even when the kernel must initiate I/O to retrieve data off the physical disk.

Microbenchmark Performance To better understand the different performance behaviors of the applications, we used microbenchmarks to measure the overhead our system introduces for primitive kernel operations. For these experiments, we configured HBench-OS to run each test 50 times.

Our results for basic system calls (Table 5) indicate that the original SVA system adds significant overhead (on the order of tens of microseconds) to individual system calls. However, the results also show that our new safety checks only add a small amount of additional overhead (25% or less) to the original SVA system.

We also tested the file system bandwidth, shown in Figure 4. The results show that the original SVA system reduces file system bandwidth by about 5-20% for small files but that the overhead for larger files is negligible. Again, however, the additional checks for low-level kernel operations add no overhead.

The microbenchmark results provide a partial explanation for the application performance results. The applications in Table 3 experience no overhead because they perform most of their processing in user-space; the overhead of the kernel does not affect them much. In contrast, the `sshd` and thttpd servers spend most of their time executing in the kernel (primarily in the poll(), select(), and write() system calls). For the system calls that we tested, our new safety checks add less than several microseconds of overhead (as shown in Table 5). For a small network transfer of 1 KB (which takes less than 8 μs on a Gigabit network), such an overhead can affect performance. However, for larger files sizes (e.g., an 8 MB transfer that takes 62.5 ms), this overhead becomes negligible. This effect shows up in our results for networked applications (thttpd and sshd): smaller file transfers see significant overhead, but past a certain file size, the overhead from the run-time safety checks becomes negligible.

Benchmark	i386 (μs)	SVA (μs)	SVA-OS (μs)	% Increase from SVA to SVA-OS	Description
getpid	0.16 (0.001)	0.37 (0.000)	0.37 (0.006)	0.0%	Latency of getpid() syscall
openclose	1.10 (0.009)	11.1 (0.027)	12.1 (0.076)	9.0%	Latency of opening and closing a file
write	0.25 (0.001)	1.87 (0.012)	1.86 (0.010)	-0.4%	Latency of writing a single byte to /dev/null
signal handler	1.59 (0.006)	6.88 (0.044)	8.49 (0.074)	23%	Latency of calling a signal handler
signal install	0.34 (0.001)	1.56 (0.019)	1.95 (0.007)	25%	Latency of installing a signal handler
pipe latency	2.74 (0.014)	30.5 (0.188)	35.9 (0.267)	18%	Latency of ping-ponging one byte message between two processes
poll	1.16 (0.043)	6.47 (0.080)	7.03 (0.014)	8.7%	Latency of polling both ends of a pipe for reading and writing. Data is always available for reading.
select	1.00 (0.019)	8.18 (0.133)	8.81 (0.020)	7.7%	Latency of testing both ends of a pipe for reading and writing. Data is always available for reading.

Table 5: Latency of Kernel Operations. Standard Deviation Shown in Parentheses.

8 Related Work

Previous work has explored several approaches to providing greater safety and reliability for operating system kernels. Some require complete OS re-design, e.g., capability-based operating systems [37, 38] and microkernels [1, 25]. Others use isolation (or "sandboxing") techniques, including device driver isolation within the OS [35, 44, 45, 51] or the hypervisor [17]. While effective at increasing system reliability, none of these approaches provide the memory safety guarantees provided by our system, e.g., none of these prevent corruption of memory mapped I/O devices, unsafe context switching, or improper configuration of the MMU by either kernel or device driver code. In fact, none of these approaches could protect against the Linux exploits or device corruption cases described in Section 7. In contrast, our system offers protection from all of these problems for both driver code and core kernel code.

The EROS [38] and Coyotos [37] systems provide a form of safe (dynamic) typing for abstractions, e.g., capabilities, at their higher-level OS ("node and page") layer. This type safety is preserved throughout the design, even across I/O operations. The lower-level layer, which implements these abstractions, is written in C/C++ and is theoretically vulnerable to memory safety errors but is designed carefully to minimize them. The design techniques used here are extremely valuable but difficult to retrofit to commodity systems.

Some OSs written in type-safe languages, including JX [18], SPIN [21], Singularity [22], and others [20] provide abstractions that guarantee that loads and stores to I/O devices do not access main memory, and main memory accesses do not access I/O device memory. However, these systems either place context switching and MMU management within the virtual machine run-time (JX) or provide no guarantee that errors in these operations cannot compromise the safety guarantees of the language in which they are written.

Another approach that could provide some of the guarantees of our work is to add annotations to the C language. For example, SafeDrive's annotation system [51] could be extended to provide our I/O memory protections and perhaps some of our other safety guarantees. Such an approach, however, would likely require changes to every driver and kernel module, whereas our approach only requires a one-time port to the SVA instruction set and very minor changes to machine-independent parts of the kernel.

The Devil project [27] defines a safe interface to hardware devices that enforces safety properties. Devil could ensure that writes to the device's memory did not access kernel memory, but not vice versa. Our SVA extensions also protect I/O memory from kernel memory and provide comprehensive protection for other low-level hardware interactions, such as MMU changes, context switching, and thread management.

Mondrix [49] provides isolation between memory spaces within a kernel using a word-granularity memory isolation scheme implemented in hardware [48]. Because Mondrix enables much more fine-grained isolation (with acceptable overhead) than the software supported isolation schemes discussed earlier, it may be able to prevent some or all of the memory-related exploits we discuss. Nevertheless, it cannot protect against other errors such as control flow violations or stack manipulation.

A number of systems provide Dynamic Information Flow Tracking or "taint tracking" to enforce a wide range of security policies, including memory safety, but most of these have only reported results for user-space applications. Raksha [12] employed fine-grain information flow policies, supported by special hardware, to prevent buffer overflow attacks on the Linux kernel by ensuring that injected values weren't dereferenced as pointers. Unlike our work, it does not protect against attacks that inject non-pointer data nor does it prevent use-after-free errors of kernel stacks and other state buffers used in low-level kernel/hardware interaction. Furthermore, this system does not work on commodity hardware.

The CacheKernel [7] partitions its functionality into an application-specific OS layer and a common "cache kernel" that handles context-switching, memory mappings, etc. The CacheKernel does not aim to provide memory safety, but its two layers are conceptually similar to the commodity OS and the virtual machine in our approach. A key design difference, however, is that our interface also attempts to make kernel code easier to analyze. For example, state manipulation for interrupted programs is no longer an arbitrary set of loads/stores to memory but a single instruction with a semantic meaning.

Our system employs techniques from VMMs. The API provided by SVA for configuring the MMU securely is similar to that presented by para-virtualized hypervisors [16, 50]. However, unlike VMMs, our use of these mechanisms is to provide fine-grain protection internal to a single domain, including isolation between user and kernel space and protection of type-safe main memory, saved processor state, and the kernel stack. For example, hypervisors would not be able to guard against [42], which our system does prevent, even though it is an MMU error. Also, a hypervisor that uses binary rewriting internally, e.g., for instrumenting itself, could be vulnerable to [9], just as the Linux kernel was. We believe VMMs could be a useful *target* for our work.

SecVisor [36] is a hypervisor that ensures that only approved code is executed in the processor's privileged mode. In contrast, our system does not ensure that kernel code meets a set of requirements other than being memory safe. Unlike SVA, SecVisor does not ensure that the approved kernel code is memory safe.

9 Conclusion

In this paper, we have presented new mechanisms to ensure that low-level kernel operations such as processor state manipulation, stack management, memory mapped I/O, MMU updates, and self-modifying code do not violate the assumptions made by memory safety checkers. We implemented our design in the Secure Virtual Architecture (SVA) system, a safe execution environment for commodity operating systems, and its corresponding port of Linux 2.4.22. Only around 100 lines of code were added or changed to the SVA-ported Linux kernel for the new techniques. To our knowledge, this is the first paper that (i) describes a design to prevent bugs in low-level kernel operations from compromising memory safe operating systems, including operating systems written in safe or unsafe languages; and (ii) implements and evaluates a system that guards against such errors.

Our experiments show that the additional runtime checks add little overhead to the original SVA prototype and were able to catch multiple real-world exploits that would otherwise bypass the memory safety guarantees provided by the original SVA system. Taken together, these results indicate that it is clearly worthwhile to add these techniques to an existing memory safety system.

Acknowledgments

We wish to thank our shepherd, Trent Jaeger, and the anonymous reviewers for their helpful and insightful feedback.

References

[1] ACCETTA, M., BARON, R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: A new kernel foundation for unix development. In *Proc. USENIX Annual Technical Conference* (Atlanta, GA, USA, July 1986), pp. 93–113.

[2] APPLE COMPUTER, INC. Apple Mac OS X kernel semop local stack-based buffer overflow vulnerability, April 2005. http://www.securityfocus.com/bid/13225.

[3] ARCANGELI, A. Linux kernel mremap local privilege escalation vulnerability, May 2006. http://www.securityfocus.com/bid/18177.

[4] BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E. G., BECKER, D., FIUCZYNSKI, M., CHAMBERS, C., AND EGGERS, S. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles* (Copper Mountain, CO, USA, 1995), pp. 267–284.

[5] BOVET, D. P., AND CESATI, M. *Understanding the LINUX Kernel*, 2^{nd} ed. O'Reilly, Sebastopol, CA, 2003.

[6] BROWN, A. *A Decompositional Approach to Computer System Performance*. PhD thesis, Harvard College, April 1997.

[7] CHERITON, D. R., AND DUDA, K. J. A caching model of operating system kernel functionality. In *Proc. USENIX Symp. on Op. Sys. Design and Impl.* (Monterey, CA, USA, November 1994), pp. 179–193.

[8] CORBET. SMP alternatives, December 2005. http://lwn.net/Articles/164121.

[9] CORBET, J. The source of the e1000e corruption bug, October 2008. http://lwn.net/Articles/304105.

[10] CRISWELL, J., LENHARTH, A., DHURJATI, D., AND ADVE, V. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles* (Stevenson, WA, USA, October 2007), pp. 351–366.

[11] CRISWELL, J., MONROE, B., AND ADVE, V. A virtual instruction set interface for operating system kernels. In *Workshop on the Interaction between Operating Systems and Computer Architecture* (Boston, MA, USA, June 2006), pp. 26–33.

[12] DALTON, M., KANNAN, H., AND KOZYRAKIS, C. Real-world buffer overflow protection for userspace & kernelspace. In *Proceedings of the USENIX Security Symposium* (San Jose, CA, USA, 2008), pp. 395–410.

[13] DEVICES, A. M. AMD64 architecture programmer's manual volume 2: System programming, September 2006.

[14] DHURJATI, D., AND ADVE, V. Backwards-compatible array bounds checking for C with very low overhead. In *Proc. of the Int'l Conf. on Software Engineering* (Shanghai, China, May 2006), pp. 162–171.

[15] DHURJATI, D., KOWSHIK, S., AND ADVE, V. SAFECode: Enforcing alias analysis for weakly typed languages. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (Ottawa, Canada, June 2006), pp. 144–157.

[16] DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., PRATT, I., WARFIELD, A., BARHAM, P., AND NEUGEBAUER, R. Xen and the art of virtualization. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles* (Bolton Landing, NY, USA, October 2003), pp. 164–177.

[17] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMS, M. Safe hardware access with the xen virtual machine monitor. In *Proceedings of the First Workshop on Operating System and Architectural Support for the on demand IT InfraStructure* (Boston, MA, USA, October 2004).

[18] GOLM, M., FELSER, M., WAWERSICH, C., AND KLEINODER, J. The JX Operating System. In *Proc. USENIX Annual Technical Conference* (Monterey, CA, USA, June 2002), pp. 45–58.

[19] GUNINSKI, G. Linux kernel multiple local vulnerabilities, 2005. http://www.securityfocus.com/bid/11956.

[20] HALLGREN, T., JONES, M. P., LESLIE, R., AND TOLMACH, A. A principled approach to operating system construction in haskell. In *Proc. ACM SIGPLAN Int'l Conf. on Functional Programming* (Tallin, Estonia, September 2005), pp. 116–128.

[21] HSIEH, W., FIUCZYNSKI, M., GARRETT, C., SAVAGE, S., BECKER, D., AND BERSHAD, B. Language support for extensible operating systems. In *Workshop on Compiler Support for System Software* (Arizona, USA, February 1996).

[22] HUNT, G. C., LARUS, J. R., ABADI, M., AIKEN, M., BARHAM, P., FHNDRICH, M., HODSON, C. H. O., LEVI, S., MURPHY, N., STEENSGAARD, B., TARDITI, D., WOBBER, T., AND ZILL, B. An overview of the Singularity project. Tech. Rep. MSR-TR-2005-135, Microsoft Research, October 2005.

[23] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. Conf. on Code Generation and Optimization* (San Jose, CA, USA, Mar 2004), pp. 75–88.

[24] LATTNER, C., LENHARTH, A. D., AND ADVE, V. S. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (San Diego, CA, USA, June 2007), pp. 278–289.

[25] LIEDTKE, J. On micro-kernel construction. *SIGOPS Oper. Syst. Rev. 29*, 5 (1995), 237–250.

[26] LMH. Month of kernel bugs (MoKB) archive, 2006. http://projects.info-pull.com/mokb/.

[27] MÉRILLON, F., RÉVEILLÈRE, L., CONSEL, C., MARLET, R., AND MULLER, G. Devil: an IDL for hardware programming. In *USENIX Symposium on Operating System Deisgn and Implementation* (San Diego, CA, USA, October 2000), pp. 17–30.

[28] MICROSYSTEMS, S. Sun solaris sysinfo system call kernel memory reading vulnerability, October 2003. http://www.securityfocus.com/bid/8831.

[29] MONROE, B. M. Measuring and improving the performance of Linux on a virtual instruction set architecture. Master's thesis, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Urbana, IL, Dec 2005.

[30] NECULA, G. C., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems* (2005).

[31] POSKANZE, J. thttpd - tiny/turbo/throttling http server, 2000. http://www.acme.com/software/thttpd.

[32] RUWASE, O., AND LAM, M. A practical dynamic buffer overflow detector. In *In Proceedings of the Network and Distributed System Security (NDSS) Symposium* (San Diego, CA, USA, 2004), pp. 159–169.

[33] SAULPAUGH, T., AND MIRHO, C. *Inside the JavaOS Operating System.* Addison-Wesley, Reading, MA, USA, 1999.

[34] SCOTT, M. L. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2001.

[35] SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing with disaster: Surviving misbehaved kernel extensions. In *USENIX Symposium on Operating System Deisgn and Implementation* (Seattle, WA, October 1996), pp. 213–227.

[36] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *SIGOPS Oper. Syst. Rev. 41*, 6 (2007), 335–350.

[37] SHAPIRO, J., DOERRIE, M. S., NORTHUP, E., SRIDHAR, S., AND MILLER, M. Towards a verified, general-purpose operating system kernel. In *1st NICTA Workshop on Operating System Verification* (Sydney, Australia, October 2004).

[38] SHAPIRO, J. S., AND ADAMS, J. Design evolution of the EROS single-level store. In *Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, June 2002), pp. 59–72.

[39] STARSETZ, P. Linux kernel do_mremap function vma limit local privilege escalation vulnerability, February 2004. http://www.securityfocus.com/bid/9686.

[40] STARZETZ, P. Linux kernel elf core dump local buffer overflow vulnerability. http://www.securityfocus.com/bid/13589.

[41] STARZETZ, P. Linux kernel IGMP multiple vulnerabilities, 2004. http://www.securityfocus.com/bid/11917.

[42] STARZETZ, P., AND PURCZYNSKI, W. Linux kernel do_mremap function boundary condition vulnerability, January 2004. http://www.securityfocus.com/bid/9356.

[43] STARZETZ, P., AND PURCZYNSKI, W. Linux kernel setsockopt MCAST_MSFILTER integer overflow vulnerability, 2004. http://www.securityfocus.com/bid/10179.

[44] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst 23*, 1 (2005), 77–110.

[45] ÚLFAR ERLINGSSON, ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating System Deisgn and Implementation* (Seattle, WA, USA, November 2006), pp. 75–88.

[46] VAN SPRUNDEL, I. Linux kernel bluetooth signed buffer index vulnerability. http://www.securityfocus.com/bid/12911.

[47] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review 27*, 5 (1993), 203–216.

[48] WITCHEL, E., CATES, J., AND ASANOVIC., K. Mondrian memory protection. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, CA, USA, October 2002), pp. 304–316.

[49] WITCHEL, E., RHEE, J., AND ASANOVIC, K. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles* (Brighton, UK, October 2005), pp. 31–44.

[50] WRIGHT, C. Para-virtualization interfaces, 2006. http://lwn.net/Articles/194340.

[51] ZHOU, F., CONDIT, J., ANDERSON, Z., BAGRAK, I., ENNALS, R., HARREN, M., NECULA, G., AND BREWER, E. Safedrive: Safe and recoverable extensions using language-based techniques. In *USENIX Symposium on Operating System Deisgn and Implementation* (Seattle, WA, USA, November 2006), pp. 45–60.