CGuard: Scalable and Precise Object Bounds Protection for C

Piyus Kedia
IIIT Delhi
India

Rahul Purandare
UNL
USA

Udit Agarwal*
UBC
Canada

Rishabh
GGSIPU
India

ABSTRACT

Spatial safety violations are the root cause of many security attacks and unexpected behavior of applications. Existing techniques to enforce spatial safety work broadly at either object or pointer granularity. Object-based approaches tend to incur high CPU overheads, whereas pointer-based approaches incur both high CPU and memory overheads. SGXBounds, an object-based approach, provides precise out-of-bounds protection for objects at a lower overhead compared to other tools with similar precision. However, a major drawback of this approach is that it cannot support address space larger than 32-bit.

In this paper, we present CGuard, a tool that provides precise object-bounds protection for C applications with comparable overheads to SGXBounds without restricting the application address space. CGuard stores the bounds information just before the base address of an object and encodes the relative offset of the base address in the spare bits of the virtual address available in x86_64 architecture. For an object that cannot fit in the spare bits, CGuard uses a custom memory layout that enables it to find the base address of the object in just one memory access. Our study revealed spatial safety violations in the `gcc` and `x264` benchmarks from the SPEC CPU2017 benchmark suite and the `string_match` benchmark from the Phoenix benchmark suite. The execution time overheads for the SPEC CPU2017 and Phoenix benchmark suites were 42% and 26% respectively, whereas the reduction in the throughput for the Apache webserver when the CPUs were fully saturated was 30%. These results indicate that CGuard can be highly effective while maintaining a reasonable degree of efficiency.

CCS CONCEPTS

• **Security and privacy** → **Systems security**.

KEYWORDS

Spatial safety, Buffer overflow

ACM Reference Format:
Piyus Kedia, Rahul Purandare, Udit Agarwal, and Rishabh. 2023. CGuard: Scalable and Precise Object Bounds Protection for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598137

1 INTRODUCTION

Spatial safety violations are the root cause of many security attacks [10, 11, 15, 17, 22, 31, 34, 38, 39]. Attackers can exploit spatial safety bugs to hijack an application's control flow or steal sensitive information (e.g., passwords). Beyond security issues, spatial safety is important to ensure expected application behavior. For example, unintentionally accessing an out-of-bounds location can cause unexpected behavior or program crashes that are hard to debug.

Spatial safety is just one aspect of reliability. Managed languages, such as Java and C#, offer better reliability by providing complete (spatial and temporal) memory and type safety. However, C does not guarantee any of these safeties by default. Despite the lack of memory safety, C and C++ are still preferred over managed languages for systems applications because managed languages are less efficient. Consequently, many performance-sensitive applications are still vulnerable to security exploits and are therefore not reliable. In this work, we propose a mechanism to enforce spatial safety for C applications.

Several techniques have been proposed to enforce spatial safety for C/C++ applications. At a high level, these techniques can be categorized into pointer-based [9, 16, 23, 29, 30, 40] and object-based [8, 19–21, 24, 26, 36, 42] approaches.

Pointer-based approaches track the bounds of sub-objects and can detect sub-object overflows. Even with hardware support [4, 18], these approaches incur high CPU and memory overheads because they need to store and update bounds information for every pointer. Oleksenko et al. [33] have reported around 75% CPU and 125% memory overheads for SPEC benchmarks for the Intel MPX [4] implementation of the ICC compiler.

In object-based approaches, spatial safety checks ensure that the memory access using a pointer is within the heap/stack/global allocation bounds. These approaches have low memory overheads because they do not need to store the bounds for every pointer. Finding the base or limit of an object using a pointer is challenging because the pointer can be an internal address of an object. Initial approaches [19, 24, 36] used a splay-tree-based lookup to check if the pointer points to a location within object bounds. For efficiency, later works [8, 20, 21, 42] enforced spatial safety at loose (or imprecise) allocation bounds rather than the actual allocation bounds. These works pad the actual allocation size to satisfy an alignment property and keep track of alignment instead of the actual allocation size. An important drawback of these approaches is that they allow applications to access the padded area that is not within the actual bounds of the objects. This would allow unintended behavior that may remain undetected.

Our goal is to enforce checking for the actual bounds of the object, thereby providing *precise* object-bounds protection. SGXBounds [26] is so far the most efficient technique (41% and 55% CPU overheads inside and outside the SGX enclaves [5, 28] for SPEC CPU2006) that provides precise object-bounds protection, but it restricts the

[*] A part of this work was done while the author was at IIIT Delhi

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598137

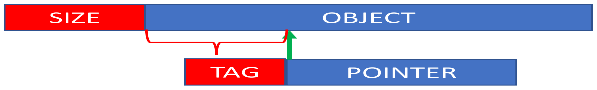


Figure 1: Object layout (top) and pointer layout (bottom) in a general case. Here, the pointer is pointing to some internal field in the object (shown by the arrow). The tag contains the relative address with respect to the base address of the object. The size is stored just before the base address.

application's usable address space to 32-bit on a 64-bit platform. SGXBounds uses the remaining 32 bits to store the upper bound of the object. This allows SGXBounds to compute the upper bound directly from the pointer itself without any expensive search. The fundamental weakness of this approach is that it cannot support larger address space because the upper bound cannot fit in the unused bits of the virtual address. As a consequence, the application's address space gets restricted to 32-bit.

We propose CGuard, a tool that provides object-bounds protection without restricting the virtual address space. Figure 1 shows the layout of an object and pointer in our scheme in a general case. CGuard stores the size of an object before the base address of the object and attaches a tag to every pointer to efficiently locate the base address. CGuard uses the spare 16-bits of a virtual address available in the x86_64 hardware to store the tag. In the tag, CGuard stores the relative offset of the pointer with respect to the base address of the object referred to by the pointer. To find the base address, CGuard simply subtracts the offset from the pointer value. For objects that cannot fit in the spare bits, CGuard uses a custom allocator that allows it to find the base of an object in just one memory access. A major challenge in our design is that, unlike SGXBounds, CGuard needs to update the offset in the tag on every pointer arithmetic. CGuard performs static analysis to reduce the number of tag updates. The mean CPU and memory overheads incurred by CGuard for SPEC CPU2017 are 42.1% and 1.1%, respectively.

Spatial safety mechanisms for managed languages are well understood. The size of an object is stored along with the object. Managed languages do not allow pointer arithmetic, enabling the mechanisms to discover the size of the object at all program points statically. On the other hand, C allows programmers to create interior pointers, store them in memory, pass them to other routines, and return them to a caller. This makes the static tracking of base pointers very hard. In our approach, the tag information needs to be updated only if a statically known potential interior pointer escapes the static scope. Thus our scheme allows programmers to control the overhead of spatial safety. If the usage of interior pointers is restricted to the static scope, our tagged pointers are equivalent to normal pointers, and the spatial safety handling mechanism is similar as in the case of a managed language.

AddressSanitizer [37] detects sequential overflows and underflows; and some *use-after-free* bugs at low overhead. The tool has already been integrated into GCC and LLVM compilers. It tracks the validity of stack, heap, and global objects using shadow memory. For every eight-byte of main memory, one byte of shadow memory is used to track its validity. The shadow memory is kept at a constant offset from the corresponding main memory, and thus checking the validity of a memory address before the access is very efficient. To detect overflows, AddressSantizer inserts extra memory blocks, *redzones*, around every object, marks them invalid in the shadow memory, and then tracks access to them. However, this scheme cannot detect out-of-bounds accesses that jump the redzones. AddressSanitizer detects use-after-bugs by putting the freed region into quarantine for some time. If access to a free object happens during quarantine, then it's a use-after-free bug. We compared CGuard also with AddressSanitizer because it performs better than SGXBounds in a normal unconstrained environment.

```
1.  int* bar(int *arr, int i, int **var,
2.           struct node *n) {
3.    int *newarr = *var;
4.    arr[i] = 200;
5.    newarr[i] = 40;
6.    if (newarr == arr + 1)
7.      n->field_i = 0;
8.    return &arr[i];
9.  }
10. void foo(int i, int **var) {
11.   int x[100];
12.   struct node n;
13.   *var = &x[6];
14.   *var = bar(&x[5], i, var, &n);
15. }
```

Figure 2: Code snippet to discuss the overview of CGuard.

In summary, we make the following contributions.

(1) An approach based on pointer tagging to provide object-bounds protection for C applications at low overheads without restricting the application address space.
(2) An optimization and its evaluation that eliminates the need for bounds checking for structure accesses that are used similar to objects in managed languages.
(3) A tool, CGuard, based on LLVM and its performance evaluation using real world benchmarks.
(4) Detection and reporting of bugs in the SPEC CPU2017 and Phoenix-2.0 benchmark suites.

2 DESIGN

2.1 Overview

CGuard inserts an *object-header* before the object's base address to store the object's size. To compute the object bounds, we need to find the base address of the object at runtime.

In the code snippet depicted in Figure 2, the argument arr in bar is an interior pointer. To compute the bounds of arr at line-4, we need to find the base address of arr. To locate the base address, we store the offset from the base address in the tag area of the pointer. Here, the tag area of argument arr contains value 20. Using this information, CGuard can compute the base address by simply subtracting offset (20) from the virtual address of arr. CGuard does not update the tag area for every pointer. For example, at line-4, after computing the address of arr[i] for the memory access, CGuard does not need to update the tag because it statically knows that arr and &arr[i] belong to the same array, and it can compute the base address from the argument arr at line-1. We call arr the *static-base* of &arr[i]. Similarly, the static-base of &newarr[i] at line-5 is newarr at line-3. CGuard statically analyzes the routine to identify the static-base for every pointer. Whenever a pointer escapes the static scope, it may become a static-base in other parts of the program. For example, at line-13, &x[6] leaves the static scope and becomes the static-base at line-3. Therefore, we update the tag before storing &x[6] in var at line-13. Similarly, CGuard updates the tag of &x[5] (at line-14) and &arr[i] (at line-8) before passing to and returning from the bar routine. CGuard does not need to update the tag while storing the return value of bar in var at line-14. This is because the return value of a function is a static-base, and it already has the correct offset in its tag area.

A problem with this approach is that the maximum offset gets restricted by the number of bits in the tag field (CGuard uses 15 bits to store the offset). For objects that cannot fit into 15 bits, CGuard uses a segmented heap. In this case, the base of the object is computed using the alignment property of the segmented heap. Another problem is that C does not distinguish between a pointer and an array. For example, argument n in bar at line-2 is a pointer to a structure element; however, CGuard needs to add bounds check at line-7 before accessing the structure field because it could be an array of structures. On the contrary, in managed languages, the type-system can distinguish between an object and an array of objects. Therefore, object accesses do not need to perform explicit bounds checks. To eliminate the need for these bound checks, we expect all static-bases to point to a memory area that is large enough to store at least one element of the corresponding array. We call this property the *size-invariant* property. In the above example, CGuard requires the argument n to point to a memory area that is at least "sizeof(struct node)" long. We found that for most benchmarks, this property holds. In our scheme, programs that do not satisfy this property may have to pay an additional performance penalty.

In our scheme, changing the pointer layout further complicates the pointer comparison and subtraction operations. Now, the same pointers may have different offsets in their tag areas depending on their static bases. For example, the equality checks at line-6 will fail because newarr and arr contain different offsets in their tag areas. To handle this correctly, CGuard resets the tags in the pointer operands during these operations. CGuard also resets the tag before every memory access. CGuard uses custom wrappers to invoke system library routines. These wrappers reset the tag field from the pointer arguments because the unmodified library does not understand CGuard's pointer layout. Finally, CGuard inserts dynamic checks before memory accesses to abort the program if the accesses are not within the object-bounds.

Figure 3 shows the architecture of CGuard. CGuard takes the intermediate representation (IR) of a program as input. The IR is in static single assignment (SSA) form. We incorporate our spatial safety logic in the IR to generate the checked IR. The checked IR is compiled to an executable. At load time, the executable is linked with our custom library that implements wrappers, custom library routines, and the custom allocator. In the rest of this section, we explain our scheme in detail.

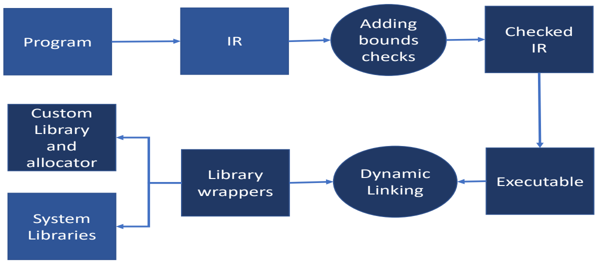


Figure 3: Architecture of CGuard.

2.2 Identifying Static-Base

Before every dereference of pointer x, the bounds can be computed using the offset field in the tag area of x. Ensuring the correct offset for every pointer definition is expensive because tags need to be updated on every pointer arithmetic. To reduce the number of updates, when pointer x is dereferenced, CGuard statically tries to find another pointer y that points to the same object as x and contains the correct offset. We call y the static base of x. The bounds of the object are computed using y. We explain below our approach to find the static-base for different kinds of definitions in the IR.
1) For pointer arithmetic and typecast operation x, we recursively trace back all arithmetic and typecast operations to obtain a pointer y that is not the result of pointer arithmetic or a typecast operation. In this case, the static-base of y is the static-base of x. Consider the following IR code.

```
x = bitcast ty1 ptr to ty2
y = getelementptr ty, ty* arr, i32 i
```

Here, the bitcast instruction generates a new definition x after casting ptr of type ty1 to ty2. In this case, the static-base of x is the same as the static-base of ptr. The operands of getelementptr instruction are arr and i. getelementptr generates a new definition y whose value is &arr[i]. In this case, the static-base of y is the same as the static-base of arr.
2) For an integer-to-pointer typecast x, if we can statically correlate x with a previous pointer-to-integer operation y, we infer the static-base of x as the static-base of y. If such a pointer-to-integer operation is not found, x is treated as the static-base of itself.
3) Pointers loaded from memory, the return value of a function call, function arguments, stack allocations, and global variables are also the static-bases of themselves.
4) The SSA representation contains *phi-nodes* to merge the definitions coming from multiple predecessor basic-blocks. In this case, we add a new phi-node that merges the static-bases of the definitions coming from these predecessors.

```
z_sb = phi <sb(x), pred1>, <sb(y), pred2>
z = phi <x, pred1>, <y, pred2>
```

In this example, z is a phi-node that merges definitions x and y coming from basic blocks pred1 and pred2. We add a new phi-node z_sb, the static-base of z, that merges the static-bases of x and y denoted using sb(x) and sb(y).
5) The IR contains the instruction select that emulates the ternary operator as shown below.

```
z_sb = select cond, sb(x), sb(y)
```

```
z = select cond, x, y
```

In this example, select takes condition cond and definitions x and y as input and creates a new definition z. At runtime, z will be equal to x or y depending on the value of cond. To find the static-base, we add an additional select instruction that takes cond, sb(x), and sb(y) as inputs and create a new definition z_sb, the static-base of z.

2.3 Tagged Pointer

We use the spare higher 16-bits of virtual address to store the tag. A tagged-pointer has the following structure.

```
typedef struct {
  unsigned long long address:48;
  unsigned long long invalid:1;
  unsigned long long offset:15;
} tag_t;
```

In the rest of the paper, we will refer to the tagged-pointer type using tag_t. The lower 48-bits of a pointer contain the actual address, represented using the address field in the tag_t. The invalid field is used to mark a pointer invalid (as discussed later in this section). The maximum offset that can be stored in the 15-bits offset field is MAX_OFFSET (0x7FFF). The allocation for a size larger than or equal to MAX_OFFSET is performed from the segment-based allocator (discussed in the next paragraph). The offset field in the static-base tagged pointer contains the offset relative to the actual base address of the object referred by the static-base. If the relative offset is equal to MAX_OFFSET, then the base address is computed using the alignment property of the segment-base allocator.

Segment-based allocator. The segment-based allocator maintains a list of segments that are shared across all threads. A segment is a 4GB (configurable at compile time) contiguous virtual address space. The starting address of a segment is aligned to 4GB. The segment is divided into fixed-size slots. Both the size and alignment of a slot are 2^k (a power of two). The value of k can vary across segments. The first few pages of the segment are used to store the metadata (e.g., a bitmap to track free slots). Initially, the virtual addresses are reserved for the entire segment. Physical pages are mapped only during the actual allocation. For every allocation, a slot is returned to the caller. Because a slot can be much larger than an actual allocation size, we only map the number of physical pages that are sufficient for the allocation size. The physical pages are reclaimed during the deallocation.

CGuard manages stack allocations of sizes greater than or equal to MAX_OFFSET using malloc and free. For these objects, CGuard replaces the calls to stack allocation API with calls to malloc and inserts free when the objects go out of scope. CGuard also inserts object-headers before stack and global allocations.

Updating the pointer tag. The tag is updated every time a pointer escapes the static scope as a result of it being passed to a function, stored in memory, or returned to a caller. We do not track a pointer if it escapes after being typecasted to an integer. Instead, we expect that the program casts it back to a pointer before the escape if the integer is out-of-bounds or modified due to some arithmetic operations on the integer. After the escape, the pointer may become a static-base in other parts of the program. For example, in our static-base identification logic (Section 2.2), a loaded value is identified as static-base. After a pointer is stored in memory, it can be loaded at different parts of the program and treated as a static-base. We update the tag before the escape to ensure that all the tagged static-bases always contain the correct offset. If the pointer is not within the bounds or does not satisfy the size-invariant property (Section 2.5), the invalid bit in the tag area is set. Accessing pointers with the invalid-bit set result in runtime exceptions preventing out-of-bounds memory accesses.

```
#define OBJ_HEADER_SIZE 8
#define SEGMENT_MASK ~(SEGMENT_SIZE - 1)

void *get_base(tag_t sb) {
  if (sb.offset < MAX_OFFSET)
    return (void*)(sb.address - sb.offset);
  if (sb.invalid) return NULL;
  if (is_global_var(sb.address))
    return get_base_allocator(sb.address);
  segment_t *s;
  void *ret;
  s = (segment_t*)(sb.address & SEGMENT_MASK);
  ret = (void*)(ptr.address & s->slot_mask);
  return ret + OBJ_HEADER_SIZE;
}
```

Figure 4: Routine used to obtain the base from an input tagged static-base (sb). If the offset in the tagged static-base is less than MAX_OFFSET, the base (lower bound) is computed after subtracting the offset from the address of the static-base pointer. Otherwise, the segment alignment property or the allocator API is used to obtain the base.

Handling out-of-bounds pointers. In the general case, CGuard allows memory access when an in-bounds pointer y derived from an out-of-bounds static-base x (due to pointer arithmetic) is dereferenced. If the offset field in the tag area of x is less than MAX_OFFSET, the actual base address can be computed by subtracting the offset and ignoring the invalid-bit. In the other case, CGuard cannot compute the actual base, and thus the dereference of y is not allowed.

2.4 Computing Bounds and Inserting Checks

CGuard computes the base address of a pointer definition using the tagged static-base. The base computation logic (get_base) is shown in Figure 4. If the offset is less than MAX_OFFSET, get_base subtracts the offset in the tag from the address of the static-base pointer. Otherwise, if the static-base is invalid (i.e., out-of-bounds), get_base cannot retrieve the actual base and returns NULL. If the offset is equal to the MAX_OFFSET and the address belongs to the range of global variables, it calls the allocator API get_base_allocator (discussed in Section 3), which does not rely on pointer tag to obtain the base. It also maintains a small cache to avoid calls to the allocator API, which works well in practice because there are only a few global variables of size greater than or equal to MAX_OFFSET across all of our benchmarks. Finally, for segment-based allocation, the base address of the object is computed using the alignment property of the segments. All slots in a segment are aligned to 2^k. The starting address of a slot is computed by resetting the lower k-bits of the pointer address. The first eight bytes of a segment contain slot_mask (~($2^k - 1$)). The starting address of the object slot is computed by 'anding' the pointer address and the slot_mask. The starting address of a slot is the object-header. The base address is computed by skipping the header.

If the static-base is an integer-to-pointer typecast, the offset field can be incorrect due to untracked integer arithmetic operations performed on the integer. To handle this case, if the static-base is an integer-to-pointer instruction or a phi or select node that depends on an integer-to-pointer instruction, CGuard backtracks all operations on the integer to check if it is involved in any arithmetic. If so, CGuard uses the allocator API to find the base. In case an integer which escapes the static scope with an incorrect tag can be accessed in the future, we rely on the application to typecast it into a pointer before letting it escape.

Bounds check. Our bounds check logic is shown below.

```
void bounds_check(void *base, void *ptr,
    void *ptrlimit, void *limit) {
  if (ptr < base || ptrlimit > limit) abort();
}
```

The arguments to the bounds_check routine are the pointer (ptr) (without tag) that is being accessed, the base address (base) of the object referred by ptr (e.g., obtained using get_base), the upper bound of the memory access (ptrlimit), and the upper bound of the object (limit). The upper bound of the object is computed by adding its size obtained from the object-header to its base address. If ptr does not lie between base and limit, the program is aborted.

2.5 Size-Invariant

CGuard enforces the size-invariant to eliminate checks when only the first element of an array or pointer to a structure element is accessed. This invariant requires all static-bases to point to a memory area that is large enough to store at least one element of the corresponding array. For example, if char *a is a static base, then a must point to a memory area that is at least one byte long; if the type of a is unsigned long long *, it must point to a memory area that is at least eight bytes long. If a pointer escapes the static scope, we invalidate the pointer if the size-invariant does not hold. This allows CGuard to remove bounds-check when only the first element of the array is accessed, since CGuard does not reset the invalid bit for these accesses. Accessing a pointer that does not satisfy the size-invariant property result into access violation.

In our experiments, we found that the size-invariant holds for the majority of the benchmarks (Section 4.4). For the benchmarks that violate the invariant, the problem can be addressed either by using a smaller type for the static-base and external typecasts whenever needed or by extra allocation. Consider the following code snippet:

```
struct info {
  int a, b, c, d;
};
int foo(struct info *i) {
  return i->b;
}
void bar() {
  int arr[2] = {1, 2};
  return foo((struct info*)arr);
}
```

In this snippet, the size-invariant requires bar to pass an object of size at least sizeof(struct info) to foo. Because the size-invariant does not hold, CGuard invalidates the parameter passed to foo. The hardware generates an access violation when foo tries to dereference the invalid pointer. In this case, the bounds check is performed in the signal handler as discussed in Section 2.6. However, signal handling is expensive. These cases can be efficiently handled using code refactoring. A way to fix this problem is to allocate at least sizeof(struct info) memory for the variable arr in the bar routine. This approach incurs memory overhead. An alternative is to rewrite the foo and bar routines as follows:

```
int foo(int *a) {
  struct info *i = (struct info*)a;
  return i->b;
}
void bar() {
  int arr[2] = {1, 2};
  return foo(arr);
}
```

In this case, since the argument type in foo is int*, bar does not invalidate the parameter passed to foo. CGuard adds dynamic checks in foo while dereferencing i because based on the size-invariant, it only knows that i is at least four bytes long. This approach does not incur any memory overhead but has a CPU overhead due to bounds checking. If types cannot be modified, an additional attribute can be used to disable or pick a different size for the size-invariant optimization for a given type. We plan to implement the type attribute in the future.

2.6 Recovery from Size-Invariant Errors

A legal memory access can cause an access violation if the size-invariant property is violated at runtime. We discuss our technique to recover from these errors using the example in Figure 5.

Let us say the caller of foo passes a single object of type struct smallTy, and consequently, the argument n gets invalidated because the size invariant property does not hold at the call site. However, access to n->v.e in foo is legal because it's within the bounds of the object (pad is outside the bounds).

At function entry, %rdi contains the argument n. At line-1, the address of n->v.e is computed. At line-3, CGuard resets the offset field in the pointer tag. At line-4, the actual dereference happens. Because the size invariant property does not hold, the hardware throws an exception at line-4. At this point, the signal handler in the userspace is called. To recover from fault, we perform a bound check in the signal handler for which we need to compute the base address. The base address can be computed using the fault address, tag bits, and the offset from the static-base. However, as we can see, the tag information is lost at this point because the %rdi register that was originally holding the tag has been overwritten at line-3. To obtain the tag bits, we have modified the compiler to ensure that the value of the potential fault address with the tag remains live

```
struct smallTy {
  int a, b, c, d, e;
};
struct largeTy {
  struct smallTy v;
  char pad[8];
};
int foo(struct largeTy *n) {
   return n->v.e;
}

Without recovery:
1. lea 0x10(%rdi), %rdi ;compute &n->v.e
2. mov $0x1FFFFFFFFFFFF, %r10
3. and %r10, %rdi  ;reset top 15-bits
4. mov (%rdi), %eax ;eax = n->v.e
5. ret

With recovery:
6. lea 0x10(%rdi), %rdi ;compute &n->v.e
7. mov $0x1FFFFFFFFFFFF, %r10
8. mov %rdi, %r11   ;saving the tag
9. and %r10, %rdi   ;reset top 15-bits
10.mov (%rdi), %eax ;eax = n->v.e
11.ret

Stub:
mov (%rdi),%eax  ;execute excepting instr
pop rdi          ;restore base register
ret
```

Figure 5: CGuard generates code labeled as "With recovery" (line:6-11) to recover from the size-invariant errors. In this case, CGuard does not add bounds-check before the memory access in foo. Instead, CGuard ensures that the pointer tag is live (line:8) across the memory access (line:10) to enable the emulation of the bounds-check in the signal handler, which is called if the size-invariant property is violated at runtime.

during the access violation. In the modified assembly, at line-8, the compiler saves the content of %rdi in the %r11 register, which is live during the memory access.

In addition, CGuard generates metadata that is used by the signal handler to emulate the bounds check. The metadata includes the base register and displacement of the potential excepting instruction (%rdi and 0), the register that contains the tag (%r11), the offset from the static-base (0x10), the size of the memory access (4), and the length of the excepting instruction. Using this information, CGuard performs the bounds check in the signal handler. If the bounds check succeeds, CGuard generates a stub corresponding to the excepting instruction. The first instruction in the stub is the excepting instruction. The stubs are cached and reused for future faults to the same instruction pointer. Before calling the stub, the signal handler saves the address of the next instruction (address of line-11) and the contents of the base register (%rdi) on the stack (i.e., the stack pointer before the exception). It then sets the instruction pointer to the starting address of the stub and resets the invalid bit in the base register (%rdi) before returning from the signal handler. After returning from the signal handler, the stub code is executed that executes the excepting instruction and restores the value of the base register (%rdi) before returning to the original code (line-11).

This approach can only help us recover from those accesses in which the offset field in the tag is less than MAX_OFFSET. We cannot retrieve the base address for large objects because the invalid bit is used for both size-invariant violation and an out-of-bounds address. The additional overheads for these changes are in the range 0-5% for the SPEC benchmarks.

2.7 Library Calls

We assume that system libraries are safe. Since library code cannot interpret our tagged pointers, we add wrappers around library calls to mediate between an instrumented application binary and unmodified system libraries, as shown in Figure 3. We trust most of the library functions to use pointer arguments safely. For some library routines, we insert bounds check to ensure spatial safety.

For many library calls, CGuard simply resets the tags in the pointer arguments before calling the target function. However, this is not always sufficient. For example, library functions may return an interior pointer, perform a callback to the application routine compiled using CGuard, and return their internal objects. In addition, the internal fields of an argument may contain tagged pointers. CGuard uses a custom implementation to handle these cases correctly.

2.8 Object Initialization and Memory Accesses

If an object is not initialized properly, the application may access any arbitrary memory location. To prevent such cases, we initialize the pointer fields in all allocations (including stack and global variables) with NULL. Furthermore, if a global variable is initialized with an interior pointer, we also update the corresponding tag in the initialization.

If memory access is guarded by a bounds check, we reset the pointer tag before the memory access; otherwise, we only reset the offset field to catch the invalid accesses using pointers that do not satisfy the size-invariant (Section 2.5).

For indirect calls with memory operands, we reset the pointer tag. If the address of a function leaves the static scope, we make it invalid. Marking the function addresses invalid disallows read/write on these addresses; however, the execution of invalid addresses is allowed using an indirect call. As a result, the application can execute any arbitrary virtual address using an indirect call. The existing mechanism for protecting indirect calls [7, 43] can be used alongside our scheme to enforce control flow integrity.

3 IMPLEMENTATION

We implemented CGuard[1] as a compiler pass in the LLVM-10.0.0 compiler. We used JEMALLOC-5.2.1 as our allocator. We extended the JEMALLOC allocator to allocate large objects from our segment-based allocator as discussed in Section 2. We discuss below our implementation and optimizations to reduce the CPU overheads.

Instrumentation. To insert the bounds check, we need to find the base first. If multiple memory accesses share the same static base, we try to insert a single call to our base finding routine that gets invoked at runtime. We implemented the base finding routine in the assembly. A function call may create register pressure, so we use a different calling convention for these calls that only uses the first argument (%rdi) and the return value (%rax) as caller-saved registers. In the fast path of our implementation, only %rdi and %rax are used. In the slow path, we save/restore other registers that are used. Once we calculate the base, the bounds check logic, as shown in Section 2.4, is directly instrumented in the LLVM IR.

Detecting illegal accesses. Notice that in addition to bounds checks, we also rely on the invalid bit in the tag area to detect illegal accesses. Before accessing memory, we reset the top 15 bits excluding the invalid bit. If the invalid bit is set, the hardware generates the `SIGSEGV` signal because the address is non-canonical. We register a signal handler for `SIGSEGV` using the `sigaction` system call. If an exception is generated due to illegal memory access, the operating system transfers the control to the signal handler in the user space. In the signal handler, we also implement our recovery mechanism for size-invariant errors.

Finding base. The `get-base-allocator` routine (Section 2), takes an internal address of an object and returns the base address of the object. For heap objects, the base address is computed using the allocator's internal data structures. For large-heap objects, the base address is computed using the alignment property of the segments.

To support the base finding for stack variables, we register stack objects with the allocator when they are created and deregister them when they are destroyed. Note that this is required only for the stack variables that escape the static scope or are typecasted to an integer. For global and static variables, at load time, the allocator walks global objects in different sections of the executable as specified in the executable format and stores the bases in sorted order to find the base using the binary search during the program execution.

Memory-mapped files and shared memory. We do not support memory-mapped files and some shared-memory APIs. However, we support `ANONYMOUS mmap` by allocating one extra page for storing the object-header. Notice that `mmap` always returns a page-aligned address.

Loop optimization. If i) a pointer is always accessed inside a loop, ii) the pointer address only depends on the induction variable and the values outside the loop, iii) the lower bound, upper bound, and the step count of the induction variable are known, iv) the loop executes at least once, and v) the loop condition is the only way to exit from the loop, then we move the bounds check outside the loop. The example in Figure 6 demonstrates our optimization.

Updating pointer tag. The tag update for escaping pointers (Section 2.3) requires a bounds-check involving memory access. However, the escaping pointers could be invalid (or uninitialized) such as the ones pointing to a freed object, and may trigger a violation, if accessed. Even though CGuard handles these unlikely situations, we omit the implementation details since we never encountered this case in any of our benchmarks. To recognize invalid pointers, CGuard initializes all pointers fields with NULL during allocation and marks all pointers initialized with a constant integer or a function address as invalid.

```
Before optimization:
if (j > 0) {
  for (i = 0; i < j; i++) {
    bounds_check(arr_base, &arr[i+k],
                 &arr[i+k+1], arr_limit);
    arr[i+k] = m;
  }
}
After optimization:
if (j > 0) {
  bounds_check(arr_base, &arr[k],
               &arr[k+j], arr_limit);
  assert(&arr[k+j] > &arr[k]);
  for (i = 0; i < j; i++)
    arr[i+k] = m;
}
```

Figure 6: `arr` is accessed inside the loop. Since the lower and upper bounds for the array accesses inside the loop are `arr[k]` and `arr[k+j]`, the bounds check is moved outside the loop.

4 EVALUATION

4.1 Experimental Setup and Benchmarks

We ran our experiments on a machine running Ubuntu-20.04.2 equipped with an 8-core 3.6 GHz Intel i9-9900k processor, 32GB RAM, 1-Gb Ethernet controller, and 512GB SSD drive for persistent storage. We disabled hyper-threading during our experiments. We measured CPU overheads using the SPEC CPU2017 [13] benchmarks. We used the reference input size for SPEC. For multicore performance, we used Phoenix-2.0 [41] and the Apache-2.4.46 webserver. For Apache, we also instrumented `apr-1.7.0` and `apr--util-1.6.1` for spatial safety checks. We configured Phoenix and Apache not to use memory-mapped files. In addition, we configured Apache to use *anonymous MMAP* for shared memory instead of *System V shared memory APIs*. Phoenix [41] reports that for the `kmeans`, `pca`, and `histogram` benchmarks, the `pthread` version is more scalable than the `map-reduce` version. We ran the `pthread` version for these benchmarks and the `map-reduce` version for the rest. We used the large input set in our evaluations. For `matrix-multiply`, `pca`, and `kmeans`, we used input sizes of 2000x2000, 3000x3000, and 200000 respectively to make them run for at least a second. These benchmarks have a very short execution time even for the large input set. For the security evaluation, we ran the BugBench [27] benchmark suite.

To measure the execution time, we took the median of five runs for every benchmark. To report the memory overhead, we used the "Maximum resident set size" reported by "/usr/bin/time -v" command. We used the geometric mean to compute the average overhead. For the server experiment, we ran the client on a different machine (with a 1-Gbps network card) and directly connected both the machines. For scalability experiments, we disabled CPU cores using the CPU hotplug feature in the Linux kernel. For native results, we used the unmodified version of the `LLVM-10.0.0` compiler and the `JEMALLOC-5.2.1` allocator that we have used for our implementation. In our configuration for the native run, we used "-fsanitize=address" flag to generate the results for AddressSanitizer [37]. We disabled the memory leak detection feature of AddressSantizer because we could not run some benchmarks due to memory leaks. We compiled all our benchmarks with the O3 optimization level. The `GeoMean` label in our graphs represents the geometric mean average.

4.2 Performance

Figure 7a shows the runtime overheads for SPEC benchmarks with and without size-invariant optimization, and the runtime overheads of AddressSanitizer. With all optimizations, the overheads are in the range of 1-245%. The geometric mean is 42.1%, as shown in the last column. `Perlbench` (denoted as `Plbench`) has the worst overhead of 245%, whereas `lbm` shows merely 1.2% overhead. SGXBounds reported 41% overheads inside the SGX enclaves [5, 28] and 55% overheads for outside the enclaves for SPEC CPU2006. Their average overhead also includes C++ benchmarks; therefore, direct comparison is not possible. Outside enclave, SGXBounds overheads for `lbm` and `mcf` are around -50% (better than native) and 30% compared to our overheads of 1.2% and 47.9% for these benchmarks. Inside enclave, SGXBounds reported around 5% overhead for `lbm` and 1% overhead for `mcf`. Interestingly, SGXBounds reported that `lbm` also performs better than the native version for the AddressSanitizer implementation in the LLVM compiler. They attributed the change in memory layout to this speedup. `Perlbench` and `gcc` are the two worst performing benchmarks in our experiments. They were not evaluated by SGXBounds because they require custom modifications in the source code. We also require custom changes for these benchmarks, as described in Section 4.4. The overheads of `gcc`, `mcf`, and `imagick` are 170.9%, 106%, and 68.3% without size-invariant optimization compared to 107.2%, 47.9%, and 34.1% overheads with the size-invariant optimization indicating its usefulness.

AddressSanitizer performs better than SGXBounds in an unconstrained environment [26]. Therefore, we also compared the performance of CGuard with AddressSanitizer. AddressSanitizer could run all SPEC CPU 2017 benchmarks with the geometric mean overhead of 68%. The overheads were in the range of 24%-201%. It performed better than CGuard for `Perlbench`, `mcf`, `xz` benchmarks. Notice that AddressSanitizer had high overheads for `Perlbench` (160%, lower than CGuard 245% overhead) and `gcc` (201%, higher than CGuard 107% overhead) benchmarks. SGXBounds could not run these benchmarks. Unlike SGXBounds, in our experiments `lbm` did not perform better than native using AddressSanitizer.

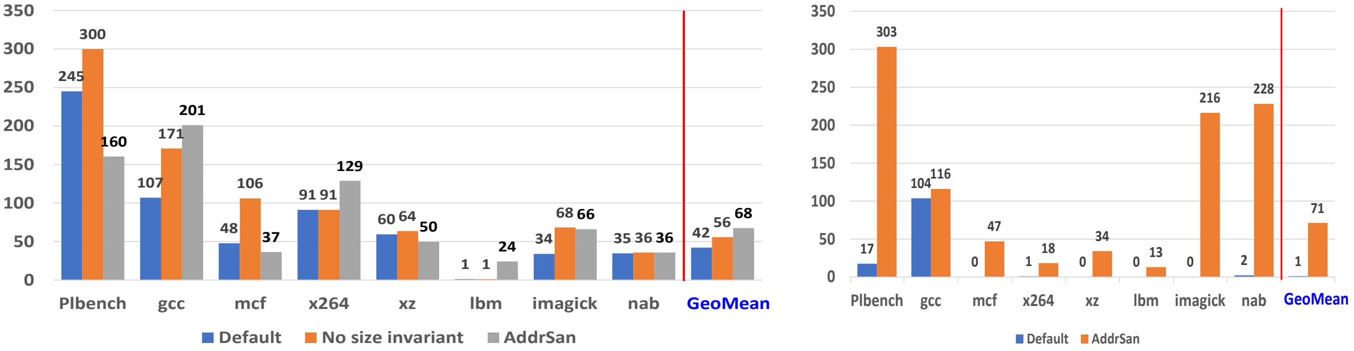
Figure 7b shows the memory overhead of CGuard and AddressSanitizer for the SPEC benchmarks. Our memory overhead for SPEC is 1.16% (Figure 7b), which is slightly higher than the 0.4% overhead reported by SGXBounds. `gcc` and `perlbench` are the worst-performing benchmarks with overheads of 104% and 17%, respectively. For `gcc`, our memory overhead is mainly due to the source code modifications related to the size-invariant (discussed in Section 4.4). To confirm this, we ran the native run with our custom allocator. The memory overhead in this case was 2%. We performed a similar experiment for `perlbench` and observed 16% overhead. This confirms that source code refactoring is not the reason for the memory overhead in `perlbench`. To validate that the overhead is not due to our segment-based allocation, we modified the original allocator to allocate eight additional bytes for every allocation. With the modified allocator, the overhead was the same as with our custom allocator. This indicates that the overhead is primarily due to the small objects for which the overhead of object headers is high. On the other hand, the memory overheads of AddressSanitizer were in the range of 13%-303%, with the geometric mean overhead of 71.2%. This is expected because AddressSanitizer uses shadow memory and puts freed objects into quarantine.

To summarize, it is difficult for us to directly compare with SGXBounds because they could not run `Perlbench` and `gcc`, which shows substantial overhead with our tool. On average, CGuard performs better than AddressSanitizer, which performs better than SGXBounds in an unconstrained environment.

Scalability. To test the scalability of our approach, we ran the Phoenix benchmark suite with 1, 2, 4, and 8 CPUs. Figure 8 shows the execution time overhead of CGuard and AddressSanitizer w.r.t. the native execution. Phoenix's average CPU and memory overheads using CGuard are 26.3% and 1.6% on a single core and 19.9% and 5.9% on eight cores respectively. As expected, our performance does not degrade significantly as the number of cores increases. However, we observed a sharp decrease in overheads with an increasing number of CPUs in the `histogram` and `linear-regression` benchmarks. This is because both of these benchmarks are not fully utilizing the CPUs on multiple cores, thus leaving scope for CGuard to steal some CPU cycles. The CPU utilization for `histogram` for the native run on 1, 2, 4, and 8 cores is 99%, 139%, 177%, and 205%, compared to 99%, 151%, 205%, and 251% CPU utilization for CGuard. A similar pattern is observed for the `linear-regression` benchmark, where the additional CPU overheads after disabling the size-invariant optimization were within the range of 10% except for the `kmeans` for which it is around 25%.

For the Phoenix benchmark suite, SGXBounds performs better than CGuard. For `kmeans`, SGXBounds reported around 60% overhead compared to 148% overhead in our approach. For the remaining benchmarks, the CPU overheads in SGXBounds were less than 10%. Notice that, unlike CGuard , SGXBounds does not need to update the tags on every pointer arithmetic. We discussed several optimizations in our design to address this issue. Still, for benchmarks in which the interior pointers are stored or passed to a function frequently, SGXBounds may perform better. However, unlike SGXBounds, our solution works even if the total memory consumption exceeds 4GB. CGuard outperformed AddressSantizer for all benchmarks. The CPU overheads of AddressSanitizer were in the range of 18%-631%, with the geometric mean overheads of 89.4% and 72.9% on single and eight cores, respectively.

We observed large variations in the memory overheads for the `kmeans` and `matrix multiply` benchmarks (Figure 9). The overheads vary between 47-108% for `kmeans` and 5-25% for `matrix--multiply`. The memory consumption of these benchmarks is very small: 10MB for `kmeans` and 53MB for `matrix-multiply`. We believe that the page table pages corresponding to our custom heap segments are adding a few extra MBs, which is prominent due to the small memory footprint. To validate our hypothesis, we ran these benchmarks with relatively large inputs, and the resulting overheads of `kmeans` and `matrix-multiply` were in the ranges 57-62% and 2-6%. To further validate that the high overheads in `kmeans` are



(a) % runtime overheads w.r.t. native execution for CGuard with size-invariant optimization, CGuard without size-invariant optimization, and AddressSanitizer.

(b) % memory overheads w.r.t. native execution for CGuard and AddressSanitizer.

Figure 7: Runtime and memory overhead of CGuard and AddressSanitizer for the SPEC benchmarks.

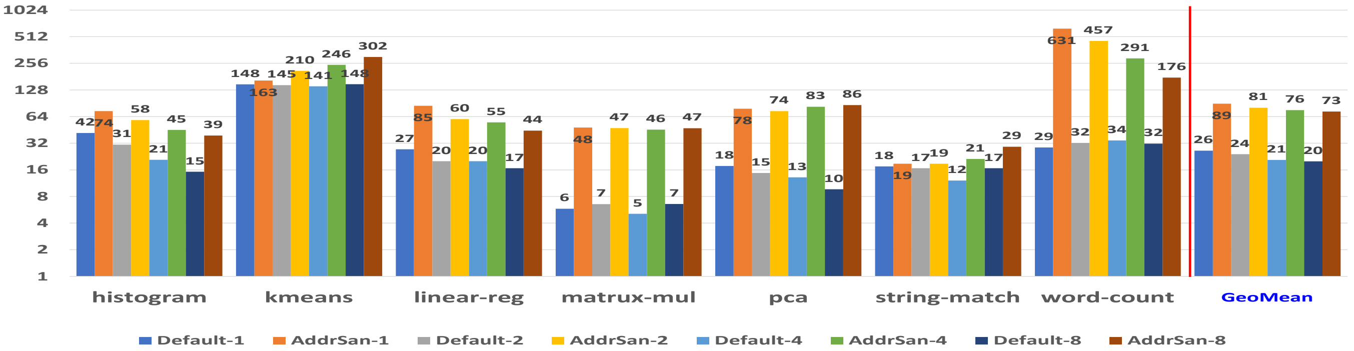


Figure 8: % runtime overheads w.r.t. native execution of CGuard and AddressSanitizer for Phoenix benchmarks running on 1,2,4, and 8 CPUs. Default-n corresponds to CGuard with n CPUs, AddrSan-n corresponds to AddressSanitizer with n CPUs.

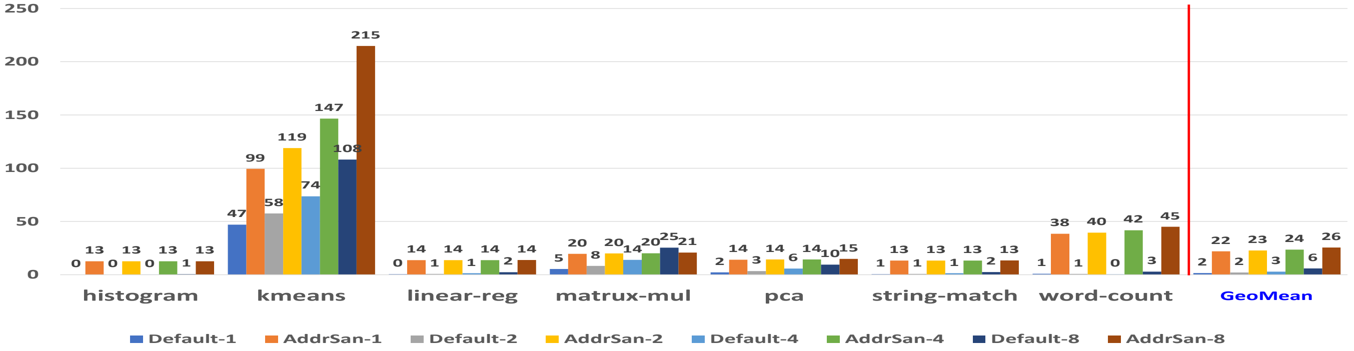


Figure 9: % memory overheads w.r.t. native execution of CGuard and AddressSanitizer for Phoenix benchmarks running on 1,2,4, and 8 CPUs. Default-n corresponds to CGuard with n CPUs, AddrSan-n corresponds to AddressSanitizer with n CPUs.

not due to our segment-based allocation, we ran the native version with a modified allocator that allocates eight extra bytes for each allocation. In this case, we found that the memory overheads for kmeans w.r.t. the native execution were in the range of 1-4%. This means that the memory overheads in kmeans are mainly due to the large number of small live objects. AddressSanitizer also showed significant variations in memory overheads (99%-215%) for different numbers of CPUs for kmeans. As expected, the geometric mean memory overhead of AddressSanitizer is higher than CGuard.

To further validate the usability of our tool for real applications, we ran the Apache webserver. Using a 1Gbps network card, we could not saturate all the cores even with concurrent requests. In the native run, the network card could only saturate three cores, so we ran our experiments with increasing number of cores. We ran the

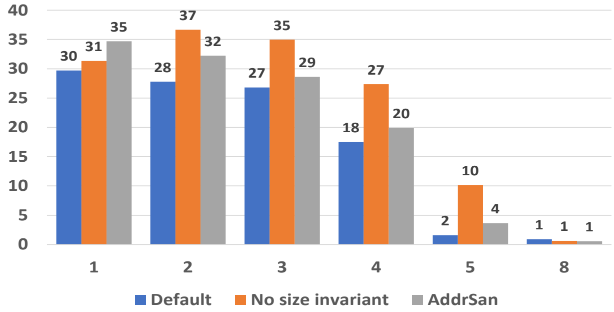


Figure 10: % reduction in the number of requests per seconds w.r.t. native execution (CGuard w/ size-invariant optimization, CGuard w/o size-invariant optimization, and AddressSanitizer) for the Apache webserver running on 1,2,3,4,5, and 8 CPUs.

Table 1: BugBench benchmarks, which have known spatial safety bugs, SPEC CPU2017 and Phoenix-2.0 benchmarks, and known vulnerabilities in Nginx and Memcached applications for which CGuard could detect spatial safety violations. The second column contains the filename:line-number pairs, at which the spatial safety violations were detected.

Benchmark	Access violation points
bc	bc.c:1425; util.c:270,577; storage.c:177
gzip	gzip.c:828
man	man.c:977,983,155; manfile.c:243
ncompress	compress42.c:896
ploymorph	polymorph.c:120,44,277,194,198,200, 231
gcc	reload1.c:1868
x264	biaridecod.c:297
string	string_match.c:158
CVE-2013-2028	ngx_recv.c:136
CVE-2011-4971	memcached.c:3534

ab tool on the client machine and enabled the KeepAlive feature in the requests. To find the right metric for the concurrency level, we tried different parameters until we observed either a reduction or no significant change in the throughput. During these experiments, we ran our instrumented server and used its default pages. We got different concurrency levels for a different number of cores.

Figure 10 plots the result for 1,2,3,4,5, and 8 cores. The first and second bars correspond to overheads with and without the size-invariant optimization. The third bar shows the overheads of AddressSantizer. We observed 29.7% overhead with the size-invariant optimization, 36.6% overhead without the size-invariant optimization, and 34.7% with AddressSanitizer when the CPUs were fully saturated (i.e., with less than four cores). Our numbers started improving when the cores were partially saturated in the native run. With eight cores, we observed only 0.9% overhead. The relative standard deviations were in the range of 0.25-1.47% across all runs.

4.3 Security

To test the effectiveness of CGuard, we ran the BugBench [27] benchmark suite, which contains a set of buggy applications some of which have spatial safety bugs. Table 1 shows all the program points at which CGuard detected out-of-bounds accesses for the BugBench, SPEC, and Phoenix benchmark suites. We found all the bugs reported in the BugBench code repository. In addition, CGuard also detected spatial safety violations in gcc and x264 benchmarks from the SPEC CPU2017 benchmark suite.

Table 2: Source code refactoring: Type a) changes related to size-invariant, b) automatic pointer comparison to int comparison generated by the frontend, and c) other changes.

Benchmark	Type	Source code modification	KLOC
Perlbench	a	hv.h:48; pad.c:2808; MD5.c:184; op.c:8401	291
	b	regexp.h:474, regcomp.c:13764	
	c	pp_pack.c:3038; av.c:159; perly.c:408; pp_hot.c:3175; regcomp.c:16274;	
gcc	a	tree-ssa-operands.c:130,133; tree-ssa-sccvn.c:1542,1580,1610; tree.c: 2102,863,865,958,1467,1584,3604, 9411; sbitmap.c:82; gimple.c:148; sparseset.c:38; rtl.c:199,341; reload1.c:915; cpp_symtab.c:173	971
	b	obstack.h:526,538	
	c	c-common.c:5296; dominance.c:1339; ggc-page.c:571; pointer_set.c:67	
Apache	a	event.c:1501	301
Phoenix	b	linear_regression.c:256	7
	c	atomic.h:81	

In gcc, the global variable hard_regno_nregs is accessed using a negative index. The check for the negative index is conducted after the variable access. Importantly, the AddressSanitizer implementation in LLVM could not detect this bug in gcc. SPEC CPU2017 contains an old version of gcc compiler. This bug is not present in the current gcc compiler.

In x264, global variables INIT_FLD_MAP_I and INIT_FLD_LAST_I are accessed at an index that is outside the bounds of these objects. These variables are passed at lines-90,91 in context_ini.c. In the string_match benchmark from the Phoenix, fdata_keys, which is allocated for size finfo_keys.st_size at line-259 in string_match.c is accessed in the loop. This loop has an incorrect bound check in the loop condition that allows the program to access an additional byte past the original allocation size. Our post-evaluation inspection revealed that the bug reported for perlbench [6] in SPEC CPU2006 has already been fixed in SPEC CPU2017. Therefore, CGuard did not report it.

CGuard also detected known spatial safety violations in nginx-1.4.0 (CVE-2013-2028)[2] and memcached-1.4.4 (CVE-2011 –4971)[3]. Nginx was tricked into receiving a message of arbitrary length, which can be controlled by the user in a stack-allocated array. The spatial safety checks in our library wrapper for recv caught this bug. In Memcached, a negative value is passed as a size parameter to the memmove routine triggering violation.

To summarize, CGuard could detect all the bugs reported by existing tools and found a new bug in the gcc benchmark from SPEC CPU2017.

4.4 Usability

For most benchmarks, we did not need to refactor source code. Table 2 provides a summary of our changes. At a broad level, we categorized the changes as follows: a) Related to the size-invariant, b) Related to a pointer comparison converted to an integer comparison by the frontend, and c) Other.

Most changes were related to the size-invariant, and these were the easiest to fix. We found that for most of these cases, CGuard threw an exception at the allocation point itself.

In some cases, the frontend generated an integer comparison instead of a pointer comparison. These conversions were typically done for "!" style comparison. We refactored the code so that the frontend generated a pointer comparison. In the future, we plan to extend the frontend to avoid the need for these changes.

In gcc, pointers are used as integers in comparisons, array indexes, and hash table keys. In all these cases, we changed the source code to reset the pointers' tags.

To summarize, most benchmarks did not require any refactoring. Even for large applications e.g. Apache, we needed refactoring at only one place. This indicates that our approach is feasible. We tried to run gcc, perlbench, and apache without the size-invariant modifications to test our size-invariant recovery mechanism. Perlbench and apache ran successfully without additional overheads because parts of code that require size-invariant modification are not on the hot path. However, gcc crashed because it employs a custom allocator that uses system allocator in the backend. As a result, most of the objects are large for which CGuard cannot retrieve the base address during the size-invariant violations.

5 LIMITATIONS AND FUTURE WORK

CGuard relies on a programmer to typecast an integer to a pointer if an integer with an inconsistent tag escapes the static scope and is accessible later. In our experiments, we found that this practice is generally followed (see Section 4.4). We also assume that the implicit integer-to-pointer typecasts are safe. In a rare case, if the size-invariant property is violated due to an implicit typecast some bugs may remain undetected.

At a more general level, CGuard assumes that the developer's intent is benign. It also assumes a weaker form of type safety (discussed in the previous paragraph) and temporal safety. Existing works have similar limitations. In SGXBounds[26] approach, if the limit of the tagged pointer is modified using an integer, the bounds check may incorrectly succeed or fail at runtime. For BaggyBounds [8], PAriCheck [42], and Low Fat Pointers [20, 21], an out-of-bound pointer can be created and accessed using integer arithmetic. These works also require source code refactoring. The primary reason behind such limitations is that it is hard to statically track arithmetic operations on an escaped integer that is also a pointer.

Our current implementation does not ensure safety for variable-length arguments. To correctly handle this case, we require the caller to pass the number of arguments for every function call.

In the future, we will investigate whether our work can be extended to support temporal safety. However, existing techniques [12] for temporal safety can be used alongside our approach with minor modifications for tagged pointers. We also plan to extend CGuard to support an OS kernel.

6 RELATED WORK

Jones and Kelly [24] proposed the idea of object-bounds protection. However, it did not allow the creation of an out-of-bounds pointer. CRED [36] improved on this work by supporting an in-bounds pointer derived from an out-of-bounds pointer. However, both these works suffered from CPU overheads due to the splay-tree-based implementation for bounds checking. Dhurjati and Adve [19] reduced CPU overheads by using per-pool splay-trees instead of a global splay-tree.

Baggy Bounds [8], PAriCheck [42], and Low Fat Pointers [20, 21] further reduce the CPU overheads by adding extra padding to objects that allow them to locate the base address without an expensive search. However, these works do not provide precise object-bounds protections because they allow the applications to access the padded area. These works have also used the pointer tagging approach. SGXBounds [26] provides precise object-bounds protection but restricts the application address space to 32-bit on a 64-bit platform. Delta Pointers [25] further reduces the CPU overheads of SGXBounds by only detecting overflows. Delta Pointers can support 48-bit address space provided the maximum object size is restricted to 32 KB. Both SGXBounds and Delta Pointers use pointer tagging, and they store the tag in the virtual address of the pointers, similar to us. CUP [14] uses a table to compute the bounds of an object at runtime. Because the table size can be huge, it limits the total number of objects to 2^{31} to restrict the table size and enable fast indexing. The table index is embedded in the address of an object.

Another line of work provides spatial safety for pointer-bounds. These approaches can detect sub-object overflow at the cost of high CPU and memory overheads because they need to store and update bounds for every pointer.

CCured [16, 30] statically categorized the pointers into SAFE, SEQ, and WILD. SAFE pointers are normal pointers and do not require any checks. SEQ and WILD pointers are fat-pointers that store the bounds information of pointers and objects and require runtime checks. Cyclone [23] uses fat-pointers and also provides programmers a variety of pointer qualifiers to control the runtime checks. SoftBound [29] stores per-pointer metadata in a disjoint address space for better compatibility. SafeC [9] and Xu et al. [40] also track bounds for every pointer and can also detect temporal safety bugs in addition to spatial safety bugs.

AddressSanitizer [37], Valgrind [32], and Purify [35] can only detect sequential buffer overflows and underflows, but they also have much wider goals.

7 CONCLUSION

We presented CGuard, a tool that provides precise object-bounds protection for C applications at low CPU and memory overheads without restricting the application address space on the x86_64 platform. CGuard requires applications to obey a weak form of type-safety. Our evaluation revealed that for most applications, this property holds. The changes needed for applications that did not satisfy the property were minor. CGuard was able to detect spatial safety violations in widely used benchmarks. In particular, it detected a bug in gcc that was not reported in any other works to the best of our knowledge. This evaluation demonstrates that our approach is effective and can scale to real applications.

REFERENCES

[1] Cguard. https://github.com/piyus/CGuard_proj.

[2] Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (cve-2013-2028). https://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html, 2013 (accessed Dec 2, 2021).

[3] Vulnerability details : Cve-2011-4971. https://www.cvedetails.com/cve/cve-2011-4971, 2013 (accessed Dec 2, 2021).

[4] Intel 64 and ia-32 architectures developer's manual: Vol. 1. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html, 2016 (accessed Dec 2, 2021).

[5] Intel 64 and ia-32 architectures software developer's manual: 3d. https://www.intel.in/content/www/in/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3d-part-4-manual.html, 2016 (accessed Dec 2, 2021).

[6] Addresssanitizerfoundbugs. https://github.com/google/sanitizers/wiki/AddressSanitizerFoundBugs#Spec_CPU_2006, 2018 (accessed Dec 2, 2021).

[7] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[8] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, volume 10, 2009.

[9] 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 1994 conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[10] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel {SGX}. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1213–1227, 2018.

[11] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[12] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

[13] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, 2018.

[14] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. Cup: Comprehensive user-space protection for c/c++. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 381–392, 2018.

[15] Stephen Checkoway, Ariel J Feldman, Brian Kantor, J Alex Halderman, Edward W Felten, and Hovav Shacham. Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. *EVT/WOTE*, 2009, 2009.

[16] Jeremy Condit, Matthew Harren, Scott McPeak, George C Necula, and Westley Weimer. Ccured in the real world. *ACM SIGPLAN Notices*, 38(5):232–244, 2003.

[17] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 952–963, 2015.

[18] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. *ACM SIGOPS Operating Systems Review*, 42(2):103–114, 2008.

[19] 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*, pages 162–171, 2006.

[20] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142, 2016.

[21] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *NDSS*, volume 17, pages 1–15, 2017.

[22] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In *2015 IEEE Symposium on Security and Privacy*, pages 781–796. IEEE, 2015.

[23] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[24] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *AADEBUG*, volume 97, pages 13–26. Citeseer, 1997.

[25] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[26] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221, 2017.

[27] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5, 2005.

[28] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[30] 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*, pages 128–139, 2002.

[31] Nergal. The advanced return-into-lib(c) exploits: Pax case study. In *Phrack Magazine, Volume 11, Issue 0x58*, 2001.

[32] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[33] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):1–30, 2018.

[34] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[35] Bob Joyce Reed Hastings. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*. Citeseer, 1991.

[36] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, volume 2004, pages 159–169, 2004.

[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 309–318, 2012.

[38] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[39] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, pages 1–8, 2009.

[40] 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 Twelfth International Symposium on Foundations of Software Engineering*, pages 117–126, 2004.

[41] Richard M Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 198–207. IEEE, 2009.

[42] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R Sekar, Frank Piessens, and Wouter Joosen. Paricheck: an efficient pointer arithmetic checker for c programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 145–156, 2010.

[43] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and Communications Security*, pages 29–40, 2011.