Boundless Memory Allocations for Memory Safety and High Availability

Marc Brünink  Martin Süßkraut  Christof Fetzer

Technische Universität Dresden
Department of Computer Science; 01062 Dresden; Germany
{marc, suesskraut, christof}@se.inf.tu-dresden.de

Abstract—Spatial memory errors (like buffer overflows) are still a major threat for applications written in C. Most recent work focuses on memory safety — when a memory error is detected at runtime, the application is aborted. Our goal is not only to increase the memory safety of applications but also to increase the application’s availability. Therefore, we need to tolerate spatial memory errors at runtime. We have implemented a compiler extension, Boundless, that automatically adds the tolerance feature to C applications at compile time. We show that this can increase the availability of applications. Our measurements also indicate that Boundless has a lower performance overhead than SoftBound, a state-of-the-art approach to detect spatial memory errors. Our performance gains result from a novel way to represent pointers. Nevertheless, Boundless is compatible with existing C code. Additionally, Boundless provides a trade-off to reduce the runtime overhead even further: We introduce vulnerability specific patching for spatial memory errors to tolerate only known vulnerabilities. Vulnerability specific patching has an even lower runtime overhead than full tolerance.

Keywords—Bounds checking; Fault tolerance; Compiler transformation; Availability; Software safety

I. INTRODUCTION

Programming languages like C offer great flexibility on memory management. While this flexibility offers a great opportunity to tweak the performance of programs, it comes at a cost: programs written in C are not safe against temporal and spatial memory errors. A temporal memory error is caused by using a memory object after it was deallocated. Garbage collection tolerates temporal memory errors by making them impossible to happen. In contrast, a spatial memory error violates the isolation between independently allocated memory objects. In 2010, spatial memory errors were ranked as the third most dangerous programming error [1]. Therefore, we focus in this paper on tolerating spatial memory errors, i.e., buffer overflows and buffer underflows.

We propose a new system, Boundless, which not only detects but also tolerates spatial memory errors. In this way, allocations are boundless and spatial memory errors are no longer possible. Most state-of-the-art approaches handle spatial memory errors using a fail-stop approach, e.g., [2, 3]. If a spatial memory error is detected, the application is aborted. But fail-stop behavior decreases the availability of a system. There are approaches to tolerate fail-stop errors, e.g., by restarting the aborted application. In our evaluation, we show that by tolerating the memory error itself Boundless can achieve higher availability than best-practice methods to tolerate fail-stop errors.

Detecting and tolerating spatial memory errors comes at the price of additional runtime overhead. Boundless decreases the performance overhead. First, we merge meta data into pointer values. Even though Boundless not only detects but also tolerates spatial memory errors, in our measurements Boundless achieves a 23.9% better performance than the detection-only, fail-stop approach SoftBound [2]. Second, Boundless can automatically generate patches to tolerate known spatial memory errors. Patching reduces the performance overhead down to 2.6% of the performance overhead of the full tolerance mode.

Boundless has the following features: High Availability By tolerating spatial memory errors we not only increase memory safety but also availability. In our measurements, Boundless increases the availability of the HTTP proxy squid dramatically compared to squid’s built-in fail-stop error handling approach. Thus, we believe tolerance makes Boundless more attractive to practical deployment.

Source Code Compatibility We have implemented Boundless as a compiler plug-in for the LLVM compiler infrastructure [4]. No source code has to be changed to apply Boundless. Boundless adds tolerance against spatial memory errors to C code automatically.

Completeness Boundless tolerates spatial memory errors on the heap, the stack, and on global variables. In contrast to most related work, Boundless handles almost arbitrary integer arithmetic on pointers. For instance, integer arithmetic is often used to align pointers to pages or cache lines.

To decrease the performance overhead of Boundless, we store our meta data in unused bits of pointer values on x86_64. Currently pointers on x86_64 use only 48 bits of the available 64 bits. We have two pointer representations with different trade-offs. Both use the 16 spare bits to store meta data. On architectures that do not offer spare bits or
Tolerance We present a spatial memory error tolerance mechanism (Section II and Section V) and evaluate its performance in Section VII.

Pointer Representations We introduce two novel pointer representations in Section IV. They differ in error tolerance properties and incurred runtime overheads. We can switch between these representations at runtime. Both representations store meta data in the 16 upper spare bits on x86_64.

Integer Arithmetic We support arbitrary casts between pointers and integers including integer arithmetic (Section IV-B). Manipulated integers can be casted back to pointers without losing isolation between memory objects.

Automatic Patching In Section VI, we describe a sound automatic patching mechanism for tolerating known vulnerabilities. Instead of using a probabilistic approach like padding memory objects, we add full tolerance to the vulnerability. Thus, we protect the application not only against a specific instance of an attack, but we fix the whole vulnerability.

Compiler Optimizations Finally, we are the first, to the best of our knowledge, to discuss compile time optimizations which are sound in the context of error tolerance (Section V-D).

II. OVERVIEW

Our goal is to provide memory safety and availability in the presence of spatial memory errors with low performance overhead. For memory safety, we ensure object isolation for every memory object on the heap, on the stack and in the data segment. A memory object is a consecutive buffer allocated on the heap, on the stack, or in the data segment of a running application. Object isolation means that a pointer to memory object A can never be used to access a different memory object B. Every memory object has bounds, i.e., a base address and an end address. A pointer p for a given memory object A points in-bounds, if base address \( \leq p < \) end address. If a pointer is not in-bounds, we call it out-of-bounds. Each pointer is associated with exactly one memory object. This association is established at the time of assignment of a memory object to a pointer. A pointer \( p' \) derived from pointer p is always associated to the same memory object as p.

Most approaches implement memory safety by fail-stop, i.e., when an out-of-bounds memory access is detected, the application is aborted. Fail-stop decreases the application’s availability. We provide availability by tolerating out-of-bounds accesses (Section V).

For spatial memory error tolerance, we need to track the bounds of memory objects and the association between pointers and memory objects. Every memory access needs to be checked at runtime to detect out-of-bounds memory accesses.

A. Boundless

We implemented our approach as a compiler plug-in called Boundless. Out of convenience, we used the LLVM compiler infrastructure [4]. The approach is independent of LLVM and it is possible to port Boundless to other compiler infrastructures.

Boundless adds tolerance to applications by instrumenting intermediate code. Figure 1 gives an overview of Boundless. In order to use Boundless, the application has to be compiled into LLVM’s intermediate representation. Next, Boundless instruments the modules of the application. The developer is free to instrument only a part of the modules, e.g., for performance reasons. Finally, Boundless links all modules (instrumented and uninstrumented) and the Boundless runtime library together to form a hardened executable.

Boundless’ instrumentation reroutes all dynamic allocations (on the heap and on the stack), all memory accesses (load and store), and every pointer arithmetic into the Boundless runtime library. Our custom memory allocator wraps the systems memory allocator. At runtime, our allocator tracks meta data of every dynamically allocated memory object. For global variables, we generate the meta data at compile time. At runtime, Boundless keeps track of the bounds of memory objects and the association between pointers and memory objects. At each memory access, Boundless checks whether the memory access is in-bounds or out-of-bounds. In-bounds accesses are passed through. Out-of-bounds accesses are rerouted to our out-of-bounds store for tolerance (Section V-B).

B. Object Isolation

Object isolation ensures that a pointer to one object cannot be manipulated in such a way that it points to a different object. A pointer should retain its intended referent object [5]. The referent object can only be changed by explicit assignment not by pointer arithmetic.
Figure 2. In addition to detecting memory accesses to unallocated memory, a spatial memory error detector also needs to enforce object isolation.

![Figure 2](image_url)

Figure 3. The spare bits on x86_64 are used to store meta data. The highest bit is used to distinguish between the two runtime pointer types.

![Figure 3](image_url)

Figure 2 depicts a memory image with two objects A and B. First, pointer p is associated with memory object A. Next, p is advanced by d to position p + d which is outside of A. By chance p + d points to a different memory object B. Since the memory at position p + d is allocated to object B, p + d points to valid and accessible memory. Nevertheless, accessing memory object B through pointer p + d violates object isolation. Since the C standard does not guarantee anything about the position of object B in memory [6], it is invalid to intentionally create pointer p + d to access object B. Without object isolation, p + d is not detected as an out-of-bounds pointer of A. Instead, it is classified as an in-bounds pointer to B. To provide object isolation between A and B, we have to treat p + d as an overflow of object A.

We achieve object isolation differently depending on the used runtime pointer type. FastPointers carry the complete meta data as explained in Section IV-A. In contrast, SlowPointers use a separate meta data store and carry only links into this store (Section IV-B). Most of the time Boundless relies on FastPointer. SlowPointers are only used as fall back in case FastPointers cannot be applied.

C. Pointer Representation

Internally, we represent any pointer in one of two pointer representations (Section IV). Both pointer representations share the common structure depicted in Figure 3. Our representation approach makes use of unused bits in pointer values on x86_64. Boundless’ implementation is for x86_64 where every pointer is 64 bits wide. Currently, on x86_64 the upper 16 bits of a 64 bit pointer are unused [7]. We put representation dependent meta data in the upper 16 bits; the remaining 48 bits hold the original pointer value. In case x86_64 will use all 64 bits in the future or if Boundless is used on architectures that do not offer spare bits, we have to restrict the usable address space similar to [3].

The highest bit of the spare bits determines the pointer representation (cf. Figure 3). If it is set to 0 the pointer is a SlowPointer (Section IV-B), otherwise it is a FastPointer (Section IV-A). The semantic of the remaining 15 bits depends on the pointer representation.

The x86_64 standard requires that the unused upper 16 bit of pointer values are sign-extended [7]. Thus, before dereferencing a pointer using one of our pointer representations, we need to mask the upper 16 bits.

D. Comparison with SoftBound

To further define the competitive landscape of Boundless we briefly compare it with SoftBound [2] version 1.1.1, which is freely available¹. In contrast to Boundless, SoftBound only implements detection but no tolerance. Boundless has two further advantages:

- Boundless supports arbitrary integer arithmetic on pointer values. Integer arithmetic is used in practice for example to align pointers to pages or cache lines. In Section IV-B we present an example from one of the benchmarks we used.
- SoftBound needs semantic knowledge about external functions, even if the external functions do not dereference pointers. Listing 1 shows an example that sorts an array of strings using the Standard C library function qsort. In C, strings are pointers to characters. qsort changes the order of these pointers in array.

SoftBound reports a false positive for the code from Listing 1. The reason is that qsort swaps the pointers in array, but SoftBound does not accordingly swap the meta data for these pointers. To solve this, SoftBound would either have to instrument qsort or it has to sort the meta data of the pointers in array according to the content of the strings. The first solution is preferable, but not always practical as the source code of the external library used needs to be available. The latter solution requires semantic knowledge of qsort. In contrast to SoftBound, Boundless merges the meta data into the pointers. Thus, when qsort swaps the pointers it also swaps the meta data.

III. RELATED WORK

Detection Boundless detects spatial memory errors and tolerates them. Many approaches [2, 3, 8, 9, 10, 11] focus on detection only. These approaches implement fail-stop semantics, i.e., if a buffer overflow is detected the application is aborted. CRED [8] uses a splay tree to store buffer bounds and checks every memory access for bound

¹http://www.cis.upenn.edu/acg/softbound/

Listing 1. SoftBound’s implementation reports a false positive for this code.

```c
int cmp(const void *p1, const void *p2) {
    return (**(int**) p1 < **(int**) p2);
}

void example() {
    char array[2] = {"z", "a"};
    qsort(array, 2, sizeof (array[0]), cmp);
    char c = *array[0]; // SoftBound will detect a false positive
}
```
violations. Instead of a single large splay tree as with CRED, it is also possible to partition the heap statically and use several smaller splay trees (one per partition) to hold the buffer bounds [9]. For every memory access only the splay tree of the corresponding partition needs to be searched, and, consequently, the overhead is reduced. This approach is completely orthogonal to ours; we believe it can be merged with Boundless without major problems. DieHard [10] uses a randomized heap to detect bound violations probabilistically. The implementation is based on a randomizing memory allocator. DieHard does not need to instrument memory accesses. It detects spatial memory errors only on the heap. Orchestra detects spatial memory errors only on the stack [11]. We detect violations on the heap, the stack, and the data segment.

Our FastPointer representation is similar to BaggyBounds’ implementation for x86_64 [3]. BaggyBounds uses a heuristic to assign a concrete memory object to an out-of-bounds pointer. If the out-of-bounds pointer is too far away from its associated memory object, then BaggyBounds might falsely associate the wrong memory object with this pointer.

SoftBound avoids meta-data look-ups by adding new local variables and function arguments [2]. Both are used to transport the buffer bounds in parallel with the pointers. For instance, a function foo(char* p), which has a pointer argument p, is changed to foo(char* p, bufferBounds pBounds). In contrast, we use FastPointer to avoid meta-data look-ups without any additional variables or arguments. Instead, we put the buffer bounds in the unused upper 16 bits of the pointer.

Of all these approaches, DieHard is the only one that supports arbitrary integer arithmetic on pointers. However, DieHard detects buffer overflows only on the heap and only probabilistically. The fuller the heap, the more likely it is that DieHard misses a buffer overflow. Through our novel combination of FastPointers and SlowPointers, we support arbitrary integer arithmetic on pointers. It is common to use integer arithmetic on pointer, e.g., to align a pointer. We present a real world example in Section IV-B.

The fail-stop approach decreases the application’s availability. Therefore, we see the fail-stop behavior as one of the reasons why buffer overflow detection is not widely deployed. Tolerating buffer overflows at runtime does not decrease the availability of the application. Thus, we believe tolerance approaches are much more acceptable in practice than detection-only approaches.

**Tolerance** Most tolerance approaches increase the size of an allocation to counteract overflows [12, 13, 14], i.e., if a buffer overflow is detected on a buffer B, the next time B is allocated its size is increased to incorporate the out-of-bounds access. This approach has two problems:

First, it does not tolerate the buffer overflow that just has been discovered. It only tolerates overflows that affect buffers allocated after the first detection. Some approaches [13, 14] use check-pointing and rollback to redo the allocation and pad the memory object accordingly. However, a rollback approach is only possible, if a checkpoint exists from before the allocation. Even if one keeps all checkpoints, rolling back too much into the past of an application is not feasible in practice. Replaying hours or even minutes is just not possible for many applications. Even worse, increasing the allocation size of global variables forces a complete application restart.

Second, increasing the allocation size does not fix the vulnerability. If an attacker is able to overflow a buffer to an arbitrary extend, the attacker can exploited the vulnerability multiple times. Eventually, the overflow offset will be too large to allocate the object at all; the result may be permanent denial-of-service. We believe increasing the allocation size is not the right approach to achieve our goal of high memory safety and availability.

We derived our tolerance strategy from failure oblivious computing [15]. In failure oblivious computing, every out-of-bounds write is ignored. For out-of-bounds reads values are forged based on an heuristic. Instead of ignoring and forging, we use an out-of-bounds store similar to [16]. All out-of-bounds read and write operations are redirected to this store. For uninitialized out-of-bounds reads, we return zero.

**Automatic Patching** In comparison to [16], we not only use our novel overflow detection approach based on the two pointer representations FastPointer and SlowPointer, but we also apply vulnerability specific patching to speedup tolerance even further. Vulnerability specific patching was inspired by vulnerability specific filtering [17]. Vulnerability specific filtering adds code to an application to filter input for a given vulnerability. In contrast, we do not filter input, but tolerate attacks targeting a given vulnerability. Vulnerability specific patching and vulnerability specific filtering share a trade-off between security and performance overhead: On the one hand, the needed instrumentation is minimal; therefore, the performance overhead is also minimal. On the other hand, every patching approach only protects against previously discovered vulnerabilities. Collaborative vulnerability detection and automatic patch deployment systems such as Vigilante [18] should be used to protect against zero-day attacks. Patches created by Boundless can also be used by the application developer to aid debugging and manual patching [19].

**IV. POINTER REPRESENTATION**

Spatial memory errors are detected by comparing the current memory address with the base address and the end address of the underlying memory object. Those addresses are necessary to verify that a pointer points into allocated memory. Together, base and end form a *meta data record*. Each meta data record must be uniquely associated with an allocated memory object, and each pointer must be
associated with its memory object. This meta data record is often stored in a meta data store, for example, a splay tree [5, 8, 9].

We use two different pointer representations. Both representations differ in the way they maintain meta data. The FastPointer encodes the meta data directly into the pointer itself (Section IV-A). The SlowPointer uses a traditional meta data store (Section IV-B). We can convert between both pointer representations at runtime. In this way, we can always choose the most appropriate representation for the current context.

Both pointer representations use the lower 48 bits to represent the pointer value. The highest bit determines the pointer representation. If it is set to 1 the pointer is represented as FastPointer. If it is set to 0 the pointer is a SlowPointer. The usage of the remaining spare bits depends on the pointer representation.

A. FastPointer

In the past, the speed of the CPU grew much faster than the speed of the memory (RAM). Hence, in current architectures it becomes increasingly difficult to keep the processor busy for memory intensive applications. The gap between the performance of the processor and the performance of the memory subsystem widens.

FastPointer is a runtime pointer representation that exploits the memory gap. Our goal is to decrease the pressure on the memory subsystem. In order to achieve this, we trade computational power for memory bandwidth.

FastPointers use the upper spare bits on current x86_64 architectures to store the offset between the current pointer position and the end address of the object into them. Thus, we can extract the offset from the pointer and recover the end address at any point in time. We use this property to detect buffer overflows. In the following we present a small example of how this is accomplished. Subsequently, we will explain how we detect buffer underflows.

Consider memory object A in Figure 4. The object is 5 bytes large; thus, a pointer to the base of this memory object carries 0x0005 in its upper bits. A pointer to the fourth byte of the same memory object can access two bytes without triggering an out-of-bounds error. Therefore, this pointer carries 0x0002 in the upper bits.

In order to bounds-check a pointer in FastPointer representation, we extract the remaining size of the associated memory object from the upper bits. If this size is greater or equal to the size of the requested operation, the operation is in-bounds. If it is smaller, we detect an out-of-bounds error.

To keep the meta data stored in the spare bits correct, we have to update the upper 16 bits of the pointer at any arithmetic affecting the pointer. For example, instead of executing (buffer + i), we have to execute (buffer + i - (i << 48)) to update the meta data properly. These additional operations increase the load on the integer units for all non-constant offsets. For constant offsets the compiler folds all operations automatically into one. Since the meta data is carried on the pointer itself, we do not fetch meta data from memory. Hence, we decrease the load on the memory subsystem.

Until now we can only protect against buffer overflows. FastPointer needs further meta data to protect against underflows: The upper 16 bits only contain the offset between the current position and the end address. However, to detect underflows we need the base address of the associated memory object.

For example, assume the pointer marked 0x0002 in Figure 4 is decremented by 4. It will point to the out-of-bounds position marked 0x0006. Looking only at the meta data stored in the spare bits, we cannot detect the underflow. We need an additional mechanism to detect whether the pointer crossed the lower bound of the memory object. To this end, we pad all allocations and store the base address of the allocation just after the allocated object. Whenever a pointer is decremented, we retrieve the base address of the object stored at the end address (at offset 0x0000 in Figure 4). Next, we check for a buffer underflow using this base address. In the example, the resulting pointer is detected as out-of-bounds.

While the sum of these operations is costly compared to a simple pointer arithmetic, they only have to be performed in case we need to check for buffer underflows. We need to check for buffer underflows if and only if the offset of the pointer arithmetic is negative. In our experience negative pointer arithmetic is rare.

The rareness of negative pointer arithmetic is also the reason why we use the end address as point of reference, i.e., we store the base address at the end of the allocation. Also the offset stored in the spare bits refers to the end address. It would be trivial to use the base address as point of reference instead. However, then we would have to lookup the end address whenever positive pointer arithmetic is used.

Listing 2 shows the pseudo code for handling pointer arithmetic. If the offset of the pointer arithmetic is positive and in-bounds, then the pointer value and the size in the upper 16 bits are updated (line 4). If the offset is positive...
but the resulting pointer is out-of-bounds and does not point to the end address, the pointer is converted into SlowPointer representation on line 5. We will explain the SlowPointer representation in more detail in the next section. If the offset is negative, we strip the meta data of the pointer on line 7. Next, we read the base address from the padding after the memory object. If the resulting pointer stays in-bounds, then the pointer and the size in the upper bits is updated on line 10. Otherwise, the resulting pointer is converted into the SlowPointer representation (line 11).

In sum, we always use SlowPointers in case pointer arithmetic results in an out-of-bounds position and the position is not the end address. While we experimented with different solutions to represent out-of-bounds pointer using the FastPointer representation, in our experience the gains are not worth the effort.

The FastPointer representation can use 15 bits to represent the size. Hence, only memory objects up to a size of $2^{15} - 1 = 32767$ bytes can be represented by FastPointers. Pointers to memory objects with a larger size are always represented as SlowPointer.

### B. SlowPointer

The FastPointer representation has some limitations. First, it cannot be used to access objects larger than $2^{15} - 1$ bytes. Second, it is not worthwhile to represent out-of-bounds pointer using FastPointer. Third, FastPointer are not stable in the presence of integer arithmetic.

The example in Listing 3 is taken from one of the benchmarks we used in our evaluation (Labyrinth from STAMP [20]). Most bounds checkers do not support arbitrary integer arithmetic on pointers like in Listing 3 on line 6. We verified that SoftBound [2] signals a false positive for this example. SoftBound loses the bounds meta data during the integer arithmetic. We have not been able to verify the behavior of BaggyBounds [3], but we expect it to work fine with this specific example. However, this is by accident and not by design. For example, BaggyBounds will fail if integer arithmetic changes the pointer to an out-of-bounds address [3, §7].

To support integer arithmetic both checkers would require additional instrumentation on any integer operations to track and update the meta data of the pointer. These additional instrumentations are difficult to do correctly: Integer operations on non-pointer values must be explicitly excluded from these instrumentations to avoid false positives. In general, it is impossible to decide at compile time whether an integer operation (such as and) will be applied on pointers or integers at runtime. Furthermore, the additional instrumentation would increase the performance overhead.

In contrast, we use a pointer representation that is stable with respect to virtually all integer arithmetic operations. Whenever a pointer is casted to an integer, we switch the pointer representation to SlowPointer. For our SlowPointer representation, we store the meta data in a separate meta data store. The meta data store contains one record per allocated memory object. Figure 5 depicts the meta data store for the SlowPointer representation. The store is organized in $2^{15}$ buckets. Each bucket contains a linked list of meta data records. A pointer’s ID is stored in the unused 15 bits of a SlowPointer.

### Example

```c
// taken from labyrinth/grid.c (STAMP)
const unsigned long CACHE_LINE_SIZE = 32UL;
...
points_unaligned = (long*)malloc((n * sizeof(long) + CACHE_LINE_SIZE);
...
gridPtr->points = (long*)((char*)((unsigned long)points_unaligned & (((unsigned long)CACHE_LINE_SIZE-1)) + CACHE_LINE_SIZE));
```

Listing 3. Pointers are casted to integers to align the memory to the next cache line boundary by using an and operation.

![Figure 5. Meta data store used by the SlowPointer representation. The store consists of 2^{15} buckets. Each bucket is a linked list of meta data records. The bucket’s ID is stored in the unused 15 bits of a SlowPointer.](image)

The system only breaks when the spare bits are deliberately changed using integer arithmetic; Boundless does not protect against a malicious software developer.
the entries in the meta data store. Since we switch pointer representation between SlowPointer and FastPointer at run-time, a meta data record might even exist for FastPointers. For example, a FastPointer might get copied and converted to SlowPointer representation. If the original FastPointer is used to deallocate the memory object, we still need to delete the corresponding meta data record. Therefore, we also have to access the meta data store even if a FastPointer is deallocated. In the common case this access requires only a single load operation and one comparison for FastPointers.

In order to look up a meta data record, we use the ID stored in the SlowPointer representation to get the right bucket. Next, we search the bucket for a meta data record for which the given pointer is in-bounds. If no meta-data record is found, then the pointer is out-of-bounds. For out-of-bounds pointers we access our out-of-bounds store (Section V).

There is a probability that an out-of-bounds pointer is falsely classified in-bounds if a bucket contains more than one meta data record. For example, assume we have two memory objects $A$ and $B$. By chance both meta data records reside in the same bucket. Consider an out-of-bounds pointer $p$ belonging to memory object $A$. There is a likelihood that $p$ points into the address range of memory object $B$. Thus, it is possible that the out-of-bounds pointer $p$ is falsely detected as an in-bounds pointer to memory object $B$. However, Boundless uses SlowPointers only as a fall back. In our evaluation, we found that the SlowPointer representation is rarely needed. Hence, in most experiments each bucket contains at most one meta data record.

The probability of getting a false positive is influenced by the calculation of the ID. We experimented with different approaches, e.g. round robin and hashing. Finally, we chose a very simple one: we use bits 4 to 18 of the end address of the memory object. This approach is fast and sufficient: we never encountered a false positive. Furthermore, in contrast to round robin and hashing, it enforces that two objects sharing an ID have a minimal distance of $2^{18}$ bytes.

V. TOLERANCE OF OUT-OF-BOUNDS ACCESSES

When using a spatial memory error detector, naturally one question arises: What to do if a memory error is detected? Most approaches implement fail-stop behavior. While this can help during development it does not help in many real-world scenarios. Even with the best recovery strategies the application’s availability is decreased. Furthermore, fail-stop enables attackers to perform denial-of-service (DoS) attacks. If an error can be deterministically triggered by an attacker, aborting an application might result in a permanent DoS. Three out of five applications presented in [15] expose permanent DoS behavior in case a fail-stop approach is used. In general, we cannot distinguish between a rightful user doing something untested and a malicious attacker. As a result, fail-stop is an option if and only if denial-of-service is acceptable.

Instead of using a fail-stop approach, we tolerate out-of-bounds accesses. Our tolerance runtime approximates the theoretical concept of an infinite heap [10]. Spatial memory errors are impossible on an infinite heap. We use an out-of-bounds store to store out-of-bounds writes. Out-of-bounds reads retrieve the value from the out-of-bounds store.

A. Infinite Heap

We have adapted the concept of an infinite heap [10] for our tolerance approach. An infinite heap has the following properties:

- The heap area is infinitely large.
- The distance between two allocated memory objects is infinite. We can never reach a different object by overflowing another one. Out-of-bounds errors are contained, because the size of an object is virtually infinite. All objects are isolated from each other.
- The space between memory objects is initialized with zero.

Obviously, it is impossible to implement a infinite heap using a machine with finite memory. However, we approximate the infinite heap by isolating all objects from each other. We achieve object isolation by our pointer representation (see Section IV). Our instrumentation wraps all load and store instructions completely. Any out-of-bounds access is rerouted to our out-of-bounds store.

We apply our protection mechanism to the heap, the stack, and the data segment. In this sense we do not only have an infinite heap, but we also have an infinite stack and an infinite data segment.

B. Out-Of-Bounds Store

All out-of-bounds pointers, except pointers to the end address, are always represented as SlowPointer. However, even in-bounds pointers can access out-of-bounds positions. For example, loading a 64 bit value from a 1 byte large allocation results in an out-of-bounds access. If an out-of-bounds access uses a FastPointer, the pointer is converted to SlowPointer representation first. Next, we get the bucket for this SlowPointer from the meta data store. In the common case there is exactly one meta data record in this bucket. This meta data record is associated with the memory object. If there is more than one meta data record in this bucket, we have to decide heuristically. We pick the meta data record of the memory object closest to the out-of-bounds pointer.

We use a linked list as out-of-bounds store. Each entry in this list corresponds to a currently allocated memory object that experienced out-of-bounds write operations; the list is normally very short. An entry is composed of an object identifier and another linked list. This linked list holds the values stored by out-of-bounds operations. We use the end address of the memory object as identifier to isolate objects.
Listing 4. Wrapping uninstrumented calls to enforce large enough buffers.

```
// If the object is too small return a new
// one. Otherwise return cbuf untouched
cp = enforceBufferSize(cbuf, 100);
/fread(cp, sizeof(char), 100, stdin);
// merge buffers if necessary
if (cp != cbuf) mergeBuffer(cp, cbuf);
```

in the out-of-bounds store. The end address is guaranteed to be unique for all currently allocated objects. All entries are allocated lazily; a load to an uninitialized out-of-bounds position returns zero without any allocation.

When a memory object is freed we also free all out-of-bounds store entries of this memory object. A memory object can be freed explicitly on the heap or implicitly on the stack by destroying the stack frame the memory object belongs to.

C. Protecting Uninstrumented Functions

External libraries are not necessarily instrumented with our compile time transformation, e.g., in case the source code is not available. These uninstrumented functions themselves neither detect out-of-bounds memory access nor do they tolerate them. However, we provide custom wrappers to a set of common external functions. These wrappers tolerate any out-of-bounds access caused by passing too small buffers to these functions. We directly insert the wrapper code into the binary at compile time.

In order to prevent memory errors in these functions we check the size of passed objects before the function is called. Our wrappers encode semantic knowledge about the functions behavior to calculate the necessary object size. For example, for the standard C function `fread` we know the passed buffer needs to be at least as large as the number of requested bytes. Listing 4 shows the added code for a call to `fread`. Function `enforceBufferSize` returns a newly allocated object if the passed object is too small. Otherwise it returns the passed object untouched. The returned object is guaranteed to be large enough for the subsequent call to `fread`. After the call we merge the values of the new object back into the old object if necessary. All values that do not fit into the original buffer are stored in the out-of-bounds store.

D. Compiler Optimizations

Most related work introduces compiler optimizations to reduce the performance overhead. The common goal is to remove runtime checks from the code. We also use static out-of-bounds checks. If Boundless can prove at compile time (using pointer analysis) that a memory access is always in-bounds, the memory access is not checked at runtime. We do not use compiler optimizations like, e.g., pool allocation [9]. However, pool allocation is orthogonal to our approach and can be applied to Boundless, too.

Some common compiler optimizations are targeted at detection only, fail-stop approaches. Redundant check elimination (RCE) and loop hoisting checks [3] cannot be used unmodified for tolerance.

RCE removes redundant checks from the source code. All checks that are dominated by a check of the same address can be removed. For example, if a value is loaded, manipulated, and stored again then the check at the store operation can be omitted in a fail-stop approach. However, for tolerance the second check is still needed: if we tolerate an out-of-bounds pointer on the first access, we need to tolerate again on the second access.

We use dynamic redundant check elimination, i.e., we decide at runtime whether a check can be skipped. For each potentially redundant check, we track whether the dominating check failed. If it failed, the access is redirected into the out-of-bounds store. Otherwise the check is skipped. Instead of performing a full bounds check at the redundant check, we only use a single boolean comparison.

Loop hoisting moves checks out of loops. Again, for tolerance this approach cannot be used unmodified. The argument is the same as for RCE: if we need to tolerate the access to a memory object in one loop iteration, we might need to do it in the next iteration, too. Our approach is similar to how we handle calls to external functions. Before entering the loop we check whether the memory object accessed in the loop is large enough. To check the size of the object, we use the function `enforceBufferSize` that we introduced in Section V-C. The memory object returned by `enforceBufferSize` is guaranteed to be large enough. Within the loop we do not need to check for memory errors. After the loop we merge the buffer with the original memory object using `mergeBuffer` if necessary. In order to apply this approach we need to statically infer symbolic bounds: At compile time we need to be able to find a formula that allows us to calculate the necessary size at runtime. While it is theoretically possible to use over-approximation, we apply loop hoisting only if we can determine exact symbolic bounds.

VI. AUTOMATIC PATCHING

It is a good attitude to fix a bug as soon as it is detected. However, often it takes considerable time until a bug is completely fixed. Automatic patching minimizes the window of vulnerability. It creates patches that are ready for distribution immediately.

Based on Boundless, we developed an automatic patching approach that is safe and non-probabilistic. In combination with automatic patch deployment systems it can be used in scenarios in which full tolerance is too costly. Patching offers a huge potential to decrease performance overhead: usually only a very small subset of the code needs to be instrumented.
Figure 6. The automatic patching process uses error logs of a fully instrumented binary.

Figure 6 illustrates our patching approach. We start with a fully instrumented binary, which tolerates arbitrary spatial memory errors. Whenever an error is tolerated, we add an entry to an error log. The entry contains the instruction which caused the memory error. Additionally, we log the allocation site of the object the instruction operates on. For memory objects on the heap or on the stack the allocation site is the LLVM IR instruction that allocated the object. Note that LLVM allocates every stack variable with an explicit instruction. For objects in the data segment we use the declaration as allocation site.

In order to distinguish between different allocation sites we assign a unique 8 byte identifier to each site at compile time. When an object is allocated at runtime, we pad the allocation with 8 bytes and store the identifier into these bytes. Eight bytes ensures that we can uniquely identify up to $2^{64}$ instructions in an application.

Using the error log we can create a patched executable with minimal instrumentation. The patched executable only contains instrumentation for allocations sites and memory accesses that appear in the error log. However, for some memory access we still need to mask the upper 16 bits of the pointer value; sometimes we do not know at compile time whether the accessed pointer will contain meta data at runtime.

Our approach does not only protect against a specific attack, but against all attacks exploiting a specific vulnerability. We call this approach vulnerability specific patching. To patch multiple vulnerabilities multiple logs can be easily merged by concatenating them.

VII. Evaluation

Our evaluation focuses on the performance overhead of Boundless and on its memory safety and availability. In Section VII-A we compare the performance overhead of Boundless (without automatic patching) with SoftBound. Next, we show that Boundless can tolerate real world memory errors in Section VII-B. In Section VII-C we present performance gains caused by our automatic patching approach. We present our measurement of the availability of the HTTP proxy squid in Section VII-D. Finally, we motivate our use of FastPointer. We show that trading computational power for memory bandwidth can be especially efficient for all applications with high memory load in Section VII-E.

Setup: All measurement are gathered on a 8-core machine with Intel Xeon E5430 processors running at 2.66 GHz. The machine is equipped with 16 GB of main memory. We use Fedora 10 as operating system. We report the trimmed mean (trimmed by 20%) of 10 measurements. We measured the runtime using the time command. All binaries are compiled using LLVM version 2.5.

A. Performance

We measured the performance slowdown caused by Boundless for the Olden [21] and STAMP [20] benchmarks. We used the Olden benchmarks to facilitate easy comparison with related work [2, 3, 9]. We used the STAMP benchmarks because they are CPU and memory intensive. For STAMP benchmarks we used the provided non-simulation workload. For the Olden benchmarks we used workloads at least as large as the largest documented workload. For some Olden benchmarks we increased the workload to get reasonable long execution times. Figure 7 compares the slowdown of Boundless with the slowdown caused by SoftBound. The slowdown is calculated as the runtime of the instrumented benchmark relative to the runtime of the uninstrumented benchmark.

We used version 1.1.1 of SoftBound. Our measurements deviate from those in [2]. Most notably, SoftBound reports non-deterministic out-of-bounds accesses for some benchmarks (intruder, yada, bh, and em3d). SoftBound detects a false positive for bayes, because qsort is not handled correctly (cf. Section II-D). Furthermore, SoftBound reports a false positive for labyrinth because of unsupervised integer arithmetic (cf. Section IV-B). For vacation SoftBound crashes at compile time. Finally, treeadd produces wrong output after being instrumented by SoftBound. Therefore, we exclude the measurements of SoftBound with these benchmarks from Figure 7.

For the STAMP benchmarks Boundless exhibits an average runtime slowdown of 2.55 (arithmetic mean) as shown in Figure 7(a). The smallest slowdown is present at intruder (1.23). The largest is caused by sscat2 (4.2). Boundless has a average slowdown of 1.65 for the Olden benchmarks. The largest is present at bh (2.26). Treeadd has the smallest slowdown (1.35). For all Olden benchmarks except power, Boundless exhibits a smaller overhead than SoftBound. Note that we not only detect memory errors but we also tolerate them. Our mean runtime overhead is 23.9% smaller than SoftBound’s.
B. Memory Error Tolerance

In order to test the effectiveness of Boundless to guard against memory errors, we applied Boundless to 5 different applications containing memory errors (Table I). We took gzip, polymorph, and squid from the BugBench benchmarks suite [22]. Furthermore, we evaluate a known bug in libpng [23]. The error in libpng propagates through libpng and finally manifests in zlib. In order to tolerate this error, we statically link zlib with libpng and instrument both. Finally, ssca2 contains a previously unknown bug which we found using our tool. With carefully chosen arguments ssca2 underestimates the necessary size of a heap object. The allocated object is too small for later accesses. This bug has been confirmed by the maintainers of STAMP.

Boundless detects all errors and redirects all out-of-bounds accesses into the out-of-bounds store. All executions proceed normally. Subsequent runtime checks in gzip, libpng, and polymorph detect an error condition and terminate the application gracefully. Both, squid and ssca2, continue operation, terminate normally, and return the correct result.

C. Automatic Patching

For all 5 applications in Table I we automatically create patches. We measure the runtime slowdown caused by the patch relative to the uninstrumented version. We measure gzip by compressing a 1 MB large text file. For libpng we used pngtest and process a 5 MB large file. We evaluate polymorph by operating on a directory containing 1000 files. All files names are uppercase and are renamed by polymorph. We send 10.000 request sequentially to a patched squid instance via a 100 Mbps network link. Instead of using real HTTP traces, we use an artificial benchmark: We request 13 different binary files in a round robin manner. The files have the following sizes: 100 byte, 1kB, 5kB, 10kB, 15kB, 20kB, 25kB, 50kB, 75kB, 100kB, 150kB, 200kB, 250kB. The round robin approach ensures that all pages are in the cache after the first iteration. As a result we do not add the latency of potential cache misses to our measurement. Since every request, except the first 13, results in a cache hit, the measurement shows the worst case slowdown one should expect. All files are fetched from a locally running Apache server. Finally, we measured ssca2 using the documented, non-simulation workload. We present the results in Figure 8. For comparison we also included the runtime of the full tolerance relative to the uninstrumented version. The runtime overhead of our automatically generated patches ranges from 0.0 (polymorph) to 0.32 (gzip). The average runtime overhead is 14.2% which is 2.6% of the runtime overhead caused by full tolerance.

D. Availability: Fail-stop vs. Tolerance

In Figure 9 we compare the throughput of an uninstrumented version of the squid server with a patched version. We use the same request pattern as in the measurement above. However, we insert an attack every 1000 requests.

During normal operation the patched version of squid has a smaller throughput than the uninstrumented one. This is caused by the runtime overhead of the patch. When
an attack hits the uninstrumented version of squid the processing thread crashes. Squid spawns a new thread to substitute the crashed one. As a result the throughput decreases at each attack.

During an attack the throughput of the patched version decreases only slightly. This is caused by the fact that the attack itself is within the FTP authentication process: squid does not cache those requests. Attacks always result in a cache miss and thus the throughput naturally decrease. After more than 3.75 hours both versions still have a stable throughput. We terminated the measurement at this point in time.

We also measured the behavior with an increased attack frequency. Instead of sending an attack every 1000 requests, we sent an attack every 500 requests. The uninstrumented version of squid cannot handle this high frequency of attacks: The main thread terminates deterministically during the 6th attack. Thus, the whole application terminates and the throughput drops to 0. Boundless is able to cope with this high frequency of attacks and continues to operate until we terminate the measurement again after about 3.75 hours. In sum, Boundless dramatically increases the availability of squid under high attack loads, e.g., denial-of-service attacks.

E. Pointer Representation

We measured the ratio between last level cache misses and unhalted core cycles using perfmon [24]. We used this ratio to sort the STAMP benchmarks. Figure 10 plots the runtime slowdown as a function of this ratio. We plot the slowdown twice: Once for a version without our compiler optimizations and once with all our compile time optimizations enabled. The experienced slowdown declines as a function of the last level cache misses.

An increasing frequency of last level cache misses translates to a higher load on the memory bus. It also means the processor adds more bubbles into the pipeline. Boundless can use these bubbles to perform some of the extra calculations caused by the different runtime pointer representations. Boundless trades computation for memory accesses. It increases the computational load and minimizes the load on the memory subsystem. As a result, Boundless performs best for applications with high utilization of the memory bus.

Since the memory gap is widening, the utilization of the memory bus will increase. We believe Boundless will experience even smaller runtime overheads on future, upcoming processors.

The FastPointer representation is crucial for the performance of Boundless. Figure 11(a) compares the performance overhead of Boundless with a SlowPointer only approach. Except for ssc2 our combination of SlowPointer with FastPointer performs dramatically better that a SlowPointer only approach. Figure 11(b) demonstrates another positive effect of FastPointers. When many pointers are represented as FastPointer the SlowPointer meta data store is much less filled as with a SlowPointer only approach.

Additionally, FastPointer decrease the probability for false negatives with SlowPointer dramatically. With FastPointer most buckets of the SlowPointer meta data store contain at most on record. Hence, false negatives are very unlikely.

VIII. Conclusion

Boundless is a novel approach that increases the memory safety and availability of applications with spatial memory errors. Our measurements show that Boundless can dramatically increase the availability of server applications with spatial memory errors. We exploit spare bits on current x86_64 to track meta data directly in the pointer value. We use two different pointer representations and we switch between them at runtime. Boundless chooses the representation best suited for the current needs dynamically. Boundless is safe, complete, and has a competitive low overhead.

REFERENCES


\[\text{Average search depths smaller than 1 can appear when most accesses to the meta data store are caused by deallocations of FastPointers that have never been converted to SlowPointers (cf. Section IV-B).}\]


