Backwards-Compatible Array Bounds Checking for C with Very Low Overhead[*]

Dinakar Dhurjati and Vikram Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave., Urbana, IL 61801
{dhurjati,vadve}@cs.uiuc.edu

ABSTRACT

The problem of enforcing correct usage of array and pointer references in C and C++ programs remains unsolved. The approach proposed by Jones and Kelly (extended by Ruwase and Lam) is the only one we know of that does not require significant manual changes to programs, but it has extremely high overheads of 5x-6x and 11x–12x in the two versions. In this paper, we describe a collection of techniques that dramatically reduce the overhead of this approach, by exploiting a fine-grain partitioning of memory called Automatic Pool Allocation. Together, these techniques bring the average overhead checks down to only 12% for a set of benchmarks (but 69% for one case). We show that the memory partitioning is key to bringing down this overhead. We also show that our technique successfully detects all buffer overrun violations in a test suite modeling reported violations in some important real-world programs.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.2.5 [**Software**]: Software Engineering—*Testing and Debugging*

General Terms

Reliability, Security, Languages

Keywords

compilers, array bounds checking, programming languages, region management, automatic pool allocation.

1. INTRODUCTION

This paper addresses the problem of enforcing correct usage of array and pointer references in C and C++ programs. This remains an unsolved problem despite a long history of work on detecting array bounds violations or buffer overruns, because the best existing solutions to date are either far too expensive for use in deployed production code *or* raise serious practical difficulties for use in real-world development situations.

The fundamental difficulty of bounds checking in C and C++ is the need to track, at run-time, the intended target object of each pointer value (called the *intended referent* by Jones and Kelly [10]). Unlike safe languages like Java, pointer arithmetic in C and C++ allows a pointer to be computed into the middle of an array or string object and used later to further index into the object. Because such intermediate pointers can be saved into arbitrary data structures in memory and passed via function calls, checking the later indexing operations requires tracking the intended referent of the pointer through in-memory data structures and function calls. The compiler must transform the program to perform this tracking, and this has proved a very difficult problem.

More specifically, there are three broad classes of solutions:

- *Use an expanded pointer representation ("fat pointers") to record information about the intended referent with each pointer*: This approach allows efficient look-up of the pointer but the non-standard pointer representation is incompatible with external, unchecked code, e.g. precompiled libraries. The difficulties of solving this problem in existing legacy code makes this approach largely impractical by itself. The challenges involved are described in more detail in Section 6.

- *Store the metadata separately from the pointer but use a map (e.g., a hash table) from pointers to metadata*: This reduces but does not eliminate the compatibility problems of fat pointers, because checked pointers possibly modified by an external library must have their metadata updated at a library call. Furthermore, this adds a potentially high cost for searching the maps for the referent on loads and stores through pointers.

- *Store only the address ranges of live objects and ensure that intermediate pointer arithmetic never crosses out of the original object into another valid object* [10]: This approach, attributed to Jones and Kelly, stores the address ranges in a global table (organized as a splay tree) and looks up the table (or the splay tree) for the intended referent before every pointer arithmetic operation. This eliminates the incompatibilities

[*]This work is supported in part by the NSF Embedded Systems program (award CCR-02-09202), the NSF Next Generation Software Program (award CNS 04-06351), and an NSF CAREER award (EIA-0093426).

caused by associating metadata with pointers themselves, but current solutions based on this approach have even higher overhead than the previous two approaches. Jones and Kelly [10] report overheads of 5x-6x for most programs. Ruwase and Lam [17] extend the Jones and Kelly approach to support a larger class of C programs, but report slowdowns of a factor of 11x–12x if enforcing bounds for all objects, and of 1.6x–2x for several significant programs even if only enforcing bounds for strings. These overheads are far too high for use in "production code" (i.e., finished code deployed to end-users), which is important if bounds checks are to be used as a security mechanism (not just for debugging). For brevity, we refer to these two approaches as JK and JKRL in this paper.

Note that compile-time checking of array bounds violations via static analysis is not sufficient by itself because it is usually only successful at proving correctness of a fraction (usually small) of array and pointer references [2, 6, 7, 8, 19]. Therefore, such static checking techniques are primarily useful to reduce the number of run-time checks.

An acceptable solution for production code would be one that has no compatibility problems (like the Jones-Kelly approach and its extension), but has overhead low enough for production use. A state-of-the-art static checking algorithm can and should be used to reduce the overhead but we view that as reducing overhead by some constant fraction, for any of the run-time techniques. *The discussion above shows that none of the three current run-time checking approaches come close to providing such an acceptable solution*, with or without static checking.

In this paper, we describe a method that dramatically reduces the run-time overhead of Jones and Kelly's "referent table" method with the Ruwase-Lam extension, to the point that we believe it can be used in production code (and static checking and other static optimizations could reduce the overhead even further). We propose two key improvements to the approach:

1. We exploit a compile-time transformation called Automatic Pool Allocation to greatly reduce the cost of the referent lookups by partitioning the global splay tree into many small trees, while ensuring that the tree to search is known at compile-time. The transformation also safely eliminates many scalar objects from the splay trees, making the trees even smaller.

2. We exploit a common feature of modern operating systems to eliminate explicit run-time checks on loads and stores (which are a major source of additional overhead in the Ruwase-Lam extension). This technique also eliminates a practical complication of Jones and Kelly, namely, the need for one byte of padding on objects and on function parameters, which compromises compatibility with external libraries.

We also describe a few compile-time optimizations (some novel and some obvious) that reduce the sizes of the splay trees, sometimes greatly, or reduce the number of referent lookups. As discussed in Section 3.4, our approach preserves compatibility with external libraries (the main benefits of the JK and JKRL methods) and detects all errors detected by those methods except for references that use pointers cast from integers.

Automatic Pool Allocation uses a pointer analysis to create fine-grain, often short-lived, logical partitions ("pools") of memory objects [13]. By maintaining a separate splay tree for each pool, we greatly reduce the typical size of the trees at each query, and hence the expected cost of the tree lookup. Furthermore, unlike some arbitrary partitioning of memory objects, the properties of pool allocation provide three additional benefits. First, the *target pool* for each pointer variable or pointer field is unique and known at compile-time, and therefore does not have to be found (tracked or searched for) at run-time. Second, because pool allocation often creates type-homogeneous pools, it is possible at run-time to check whether a particular allocation is a single element and avoid entering those objects in the search trees. Finally, we believe that segregating objects by data structure has a tendency to separate frequently searched data from other data, making search trees more efficient (we have not evaluated this hypothesis but it would be interesting to do so).

We evaluate the net overhead of our approach for a collection of benchmarks and three operating system daemons. Our technique works "out-of-the-box" for all these programs, with no manual changes. We find that the average overhead is only about 12% across the benchmarks (and negligible for the daemons), although it is 69% in one case. We also used the Zitser's [23] suite of programs modeling buffer overrun violations reported in several widely used programs — 4 in `sendmail`, 3 in `wu-ftpd`, and 4 in `bind` — and found that our technique successfully detects all these violations. Overall, we believe we have achieved the twin goals that are needed for practical use of array bounds checking in production runs, even for legacy applications: overhead typically low enough for production use, and a fully automatic technique requiring no manual changes.

The next section provides a brief summary of Automatic Pool Allocation and the pointer analysis on which it is based. Section 3 briefly describes the Jones-Kelly algorithm with the Ruwase-Lam extension, and then describes how we maintain and query the referent object maps on a per-pool basis. It also describes three optimizations to reduce the number or cost of referent object queries. Section 5 describes our experimental evaluation and results. Section 6 compares our work with previous work on array bounds checking, and Section 7 concludes with a summary and a brief discussion of possible future work.

2. BACKGROUND: AUTOMATIC POOL ALLOCATION

Automatic Pool Allocation [13] is a fully automatic compile-time transformation that partitions memory into pools corresponding to a compile-time partitioning of objects computed by a pointer analysis. It tries to create pools that are as fine-grained and short-lived as possible. It merges all the target objects of a pointer into a single pool, thus ensuring that there is a unique pool corresponding to each pointer.

We assume that the results of the pointer analysis are represented as a points-to graph. Each node in this graph represents a set of memory objects created at run-time, and two distinct nodes represent disjoint sets of objects. We associate additional attributes with each node; the ones relevant to this work are a type, τ, a flag A indicating whether any of the objects at the node are ever indexed as an array, and an array of fields, F, one for each possible field of the type τ. τ is either a (program-defined) scalar, array, record or function type, or $\perp$ representing an unknown type. $\perp$ is used when the objects represented by a node are of multiple incompatible types, which most often happens because a pointer value is actually *used* as two different types (cast operations are ignored), but can also happen due to imprecision in pointer analysis. Scalar types and $\perp$ have a single field, record types have a field for each element of the record, array types are treated as their element type (i.e. array indexing is ignored), and functions do not have fields.

We also assume that the compiler has computed a call graph for the program. In our work, we use the call graph implicitly provided by the pointer analysis, via the targets of each function pointer variable.

Given a program containing explicit `malloc` and `free` operations and a points-to graph for the program, Automatic Pool Allocation transforms the program to segregate heap objects into distinct pools. Pools are represented in the code by pool descriptor variables. Calls to `malloc` and `free` are rewritten to call new functions `poolalloc` and `poolfree`, passing in the appropriate pool descriptor. By default, pool allocation creates a distinct pool for each points-to graph node representing heap objects in the program; this choice is necessary for the current work as explained later. For a points-to graph node with $\tau \neq \perp$, the pool created will only hold objects of type τ (or arrays thereof), i.e., the pools will be *type homogeneous* with a known type.

In order to minimize the lifetime of pool instances at runtime, pool allocation examines each function and identifies points-to graph nodes whose lifetime is contained within the function, i.e., the objects are not reachable via pointers after the function returns. This is a simple escape analysis on the points-to graph. The pool descriptor for such a node is created on function entry and destroyed on function exit so that a new pool instance is created every time the function is called. For other nodes, the pool descriptor must outlive the current function so pool allocation adds new arguments to the function to pass in the pool descriptor from the caller. Finally, pool allocation rewrites each function call to pass any pool descriptors needed by any of the potential callees. Ensuring backwards-compatibility of the pool allocation transformation in the presence of external libraries is discussed later in Section 3.4.

We have shown previously that Automatic Pool Allocation can significantly improve memory hierarchy performance for a wide range of programs and does not noticeably hurt performance in other cases [13]. It's compilation times are quite low (less than 3 seconds for programs up to 200K lines of code), and are a small fraction of the time taken by GCC to compile the same programs.

3. RUNTIME CHECKING WITH EFFICIENT REFERENT LOOKUP

3.1 The Jones-Kelly Algorithm and Ruwase-Lam Extension

Jones and Kelly rely on, and strictly enforce, three properties of ANSI C in their approach: (1) Every pointer value at run-time is derived from the address of a unique object, which may be a declared variable or memory returned by a single heap allocation, and must only be used to access that object. Jones and Kelly refer to this as the *intended referent* of a pointer. (2) Any arithmetic on a pointer value must ensure that the source and result pointers point into the same object, or at most one byte past the end of the object (the latter value may be used for comparisions, e.g., in loop termination, but not for loads and stores). (3) Because of the potential for type-converting pointer casts, it is not feasible in general to distinguish distinct arrays within a single allocated object defined above, e.g., two array fields in a `struct` type, and the Jones-Kelly technique does not attempt to do so.

Jones and Kelly maintain a table describing all allocated objects in the program and update this table on `malloc/free` operations and on function entry/exit. To avoid recording the intended referent for each pointer (this is the key to backwards compatibility), they check property (2) strictly on every pointer arithmetic operation, which ensures that a computed pointer value always points within the range of its intended referent. Therefore, the intended referent can be found by searching the table of allocated objects.

More specifically, they insert the following checks (ignoring any later optimization) on each arithmetic operation involving a pointer value:

JK1. check the source pointer is not the invalid value (-2);

JK2. find the referent object for the source pointer value using the table;

JK3. check that the result pointer value is within the bounds of this referent object plus the extra byte. If the result pointer exceeds the bounds, the result -2 is returned to mark the pointer value as invalid.

JK4. Finally, on any load or store, perform checks [JK1-JK3] but JK3 checks the source pointer itself.

Assuming any dereference of the invalid value (-2) is disallowed by the operating system, the last run-time check (JK4) before loads and stores is strictly not necessary for bounds checking. It is, however, a useful check to detect some (but not all) dereferences of pointers to freed memory and pointer cast errors. The most expensive part of these checks is step (JK2), finding the referent object by searching the table. They use a data structure called a splay tree to record the valid object ranges (which must be disjoint). Given a pointer value, they search this tree to find an object whose range contains that value.

If no valid range is found for a given pointer value, the pointer must have been derived from an object allocated by some uninstrumented part of the program, e.g., an external library, or by pointer arithmetic in such a part of the program (since no legal pointer can ever be used to compute an illegal one). Such pointers values cannot be checked and therefore step (JK3) is skipped, i.e., any array bound violations may not be detected.

One complication in their work is that, because a computed pointer may point to the byte after the end of its referent object, the compiler must insert padding of one-byte (or more) between any two objects to distinguish a pointer to the "extra" byte of the first object from a pointer to the second object. They modify the compiler and the `malloc` library to add this extra byte on all allocated objects. Objects can also be passed as function parameters, however, and inserting padding between two adjacent parameters could cause the memory layout of parameters to differ in checked and unchecked code. To avoid this potential incompatibility, they do not pad parameters to any function call if the call may invoke an unchecked function and do not pad formal parameters in any function that may be called from unchecked code. In the presence of indirect calls via function pointers, the compiler must be conservative about identifying such functions.

A more serious difficulty observed by Ruwase and Lam is that rule (2) above is violated by many C programs (60% of the programs in their experiments), and hence is too strict for practical use. The key problem is that some programs may compute illegal intermediate values via pointer arithmetic but never use them. For example, in the sequence `{q = p+12; r = q-8; N = *r;}`, the value q may be out-of-bounds while r is within bounds for the same object as $*p$. Jones and Kelly would reject such a program at `q = p+12` because the correct referent cannot be identified later (q may point into an arbitrary neighboring object).

Ruwase and Lam extend the JK algorithm essentially by tracking the intended referent of pointers explicitly but only in the case where a pointer moves out of bounds of its intended referent. For every such out-of-bounds pointer, they allocate an object called the OOB (Out-Of-Bounds) object to hold some metadata for the pointer. The pointer itself is modified to point to the OOB object, and the addresses of live OOB objects are also entered into a hash table. This hash table is checked only before accessing the OOB object to ensure it is a valid OOB object address. The OOB object includes the actual pointer value itself plus the address of the intended referent (saved when the pointer first goes out of bounds). All further arithmetic on the pointer is performed on the value in the OOB object. If the pointer value comes back within bounds, the original pointer is restored to its current value and the OOB object is deallocated.

The extra operations required in the Ruwase-Lam extension are: (1) to allocate and initialize an OOB object when a pointer first goes out-of-bounds; (2) on any pointer arithmetic operation, if the pointer value does not have a valid referent *and* cannot be identified as an unchecked object, search the OOB hash table to see if it points to an OOB object, and if so, perform the operation on the value in the OOB object; (3) When an object is deallocated (implicitly at the end of a program scope or explicitly via `free` operation), scan the OOB object hash table to deallocate any OOB objects corresponding to the referent object that is being deallocated.

The first two operations add extra overhead only for out-of-bounds pointers (which would have caused the program to halt with a run-time error in the JK scheme). The third operation is required even in the case of strictly correct program behavior allowed by J-K. Perhaps more importantly, step JK4 of Jones-Kelley, is now necessary for bounds checking since dereferencing OOB objects is disallowed. In particular, if we wish to combine this approach with other techniques for detecting *all* dereferences to freed memory ([20, 4]) or all pointer cast errors ([15, 5]), we would still need to perform JK4 (or a variant which checks that OOB objects are never dereferenced).

3.2 Our Approach

Our approach is based on the Jones-Kelley algorithm with the RL extension, but with two key improvements that greatly reduce the run-time overhead in practice and makes the approach useful in production level systems. In fact, the improvements are dramatic enough that we are even able to use our system for checking all array operations (not just strings), and still achieve much lower overheads than the JK or RL approaches (even compared with the RL approach applied only to strings). The two improvements are: (1) Exploiting Automatic Pool Allocation [13] for much faster searches for referent objects; and (2) An extra level of indirection in the RL approach for OOB pointers that eliminates the need for run-time checks on most loads and stores.

The Jones-Kelley approach, and in turn Ruwase-Lam extension, rely on one splay data structure for the entire heap. Every memory object (except for a few stack objects whose address is not taken) is entered in this big data structure. This data structure is looked up for almost every access to memory or pointer arithmetic operation. For a program with large number of memory objects, the size of the data structure could be very large, making the lookups quite expensive.

The main idea behind our first improvement is to exploit the partitioning of memory created by Automatic Pool Allocation to reduce the size of the splay tree data structures used for each search operation. Instead of using one large splay tree for the entire program, we maintain one splay tree per pool. The size of each individual splay tree is likely to be much smaller than the combined one. Since the complexity of searching the splay tree for uniform accesses is amortized O(log_2n) (and better for non-uniform accesses), the lookup for each pointer access is likely to be much faster than in the JK or RL approaches.

A key property that makes this approach feasible is that the pool descriptor for each pointer is known at compile-time. Without this, we would have to maintain a run-time mapping from pointers to pools, which would introduce a significant extra cost as well as the same compatibility problems as previous techniques that maintain metadata on pointers.

3.2.1 Algorithm

The steps taken by the compiler in our approach are as follows:

1. First, pool-allocate the program. Let `Pools` be the map computed by the transformation giving the pool descriptor for each pointer variable.

2. For every pointer arithmetic operation in the original program, `p = q + c`, insert a run-time check to test that `p` and `q` have the same referent. We use the function `getreferent(PoolDescriptor *PD, void *p)` to look up the intended referent of a pointer, p. The pool descriptor, `PD`, identifies which splay tree to lookup. For the instruction `p = q + c`, we compute p, then invoke getreferent(Pools[q], q), and finally check that p has the same referent as q using the function call `boundscheck(Referrent *r, void *p)`.

3. The correct pool descriptor for a pointer q may not be known either if the value `q` is obtained from an integer-to-pointer cast or from unchecked code (e.g, as a result of a call to an external function). The latter case is discussed in Section 3.4, below. The two

```
f() {
  A = malloc(...)
  ...
  while(..) {
    ...
    A[i] = ...
  }
}
```

```
f() {
  PoolDescriptor PD
  A = poolalloc(&PD,...)
  ...
  while(..) {
    ...
   Atmp =  getreferent(&PD, A);
   boundscheck(Atmp, A+i);
  }
}
```

Figure 1: Sample code before and after bounds checking instrumentation

cases can be distinguished via the flags on the target points-to graph node: the former case results in a **U** (Unknown) flag while the latter results in a missing **C** (complete) flag, i.e., the node is marked incomplete. In the former case, the pointer may actually point to an object allocated in the main program, i.e., which has a valid entry in the splay tree of some pool, but we do not know which pool at compile-time. We do not check pointer arithmetic on such pointers. This is weaker than Jones-Kelly as it might allow bound violations on such pointers to go undetected.

3.2.2 *Handling Non-Heap Data*

The original pool allocation transformation only created pools to hold heap-allocated data. We would like to create partitions of globals and stack objects as well, to avoid using large, combined splay trees for those objects. The pointer analysis underlying pool allocation includes points-to graph nodes for all memory objects, including global and stack objects. In our previous work on memory safety, we have extended pool allocation so that it assigns pool descriptors to all global and stack objects as well, *without changing how the objects are allocated*. Pool allocation already created pool descriptors for points-to graph nodes that include heap objects as well as global or stack objects. We only had to modify it to also create "dummy" pool descriptors for nodes that included only global or stack objects. The transformation automatically ensures that the objects are created in the appropriate function (e.g., in `main` if the node includes any globals). We call these "dummy" pool descriptors because no heap allocation actually occurs using them: they simply provide a logical handle to a compiler-chosen subset of memory objects.

For the current work, we have to record each object in the splay tree for the corresponding pool. We do this in `main` for global objects and in the appropriate function for stack-allocated variables (many local variables are promoted to registers and do not need to be stack-allocated or recorded). The bounds checks for operations on pointers to such pools are unchanged.

3.3 Handling Out-Of-Bounds Pointers

The Ruwase-Lam extension to handle OOB pointers requires expensive checks on all loads/stores in the program (before any elimination of redundant checks). In this work, we propose a novel approach to handle out of bounds values (in user-level programs) without requiring checks on any individual loads or stores.

Whenever any pointer arithmetic computes an address outside of the intended referent, we create a new OOB object and enter it into a hash-table recording the OOB object address (just like Ruwase-Lam). We use a separate OOB hash-table per pool, for reasons described below. The key difference is that, instead of returning the address of the newly created OOB object and recording that in the out-of-bounds pointer variable, we return an address from a part of the address space of the program reserved for the kernel (e.g., addresses greater than 0xbfffffff in standard Linux implemenations on 32-bit machines). Any access to this address by a user level programs will cause a hardware trap[1]. Within each pool, we maintain a second hash table, mapping the returned value and the OOB object. Note that we can reuse the high address space for different pools and so we have a gigabyte of address space (on 32 bit linux systems) for each pool for mapping the OOB objects.

A load/store using out of bounds values will immediately result in a hardware trap and we can safely abort the program. However all pointer arithmetic on such values needs to be done on the actual out of bounds value. So on every pointer arithmetic, we first check if the source pointer lies in the high gigabyte. If it is, we lookup the OOB hash map of the pool to get the corresponding OOB object. This OOB object contains the actual out of bounds value. We perform the pointer arithmetic on the actual out of bounds value. If the result after arithmetic goes back in to the bounds of the referent then we return that result. If the result after arithmetic is still out of bounds, we create a new OOB object and store the result in the new OOB. We then map this new OOB to an unused value in the high gigabyte, store the value along with the OOB object in the OOB hash map for the pool and return the value. Note that just like Ruwase-Lam, we need to change all pointer comparisons to take in to account the new out of bound values.

Step 2 in our approach is now modified as follows:
For every pointer arithmetic operation in the original program, `p = q + c`, we first check if q is a value in the high gigabyte. This is an inexpensive check and involves one comparison. There are two possibilities.

- Case 1: q is not in the high giga byte.
  Here we do the bounds check as before but with one key differnce. If the result p is out of bounds of the referent of q, then instead of flagging it as an error, we create a new OOB object to store the out of bounds value just like Ruwase-Lam extension. Now we map this OOB object to a value in the high address space and assign this high address space value to p.

[1]If no such reserved range is available, e.g. we are doing bounds-checking for kernel modules, then we will need to insert checks on individual loads and stores just like the Ruwase-Lam extension.

- Case 2: q is a value in the high address space.
  We do the following new check (from the discussion above): We first get the corresponding OOB object for that address using the hash map in the pool. We then retrieve the actual out of bounds value from the OOB object and do the arithmetic. If the result is within the bounds of the referent then we assign the result to p and proceed. If the result is still outside the bounds of the referent, then we create a new OOB object just like in Case 1.

3.4 Compatibility and Error Detection with External Libraries

Although Automatic Pool Allocation modifies function interfaces and function calls to add pool descriptors, both that transformation and our bounds checking algorithm can be implemented to work correctly and fully automatically with uninstrumented external code (e.g., external libraries), although some out-of-bound accesses may not be detected. First, to preserve compatibility, calls to external functions are left unmodified. Second, in any points-to graph node reachable from an external function (such nodes are marked as "incomplete" by omitting a **C** (Complete) flag), the `poolfree` for the corresponding pool must determine if it is passed a pointer not within its memory blocks (this is a fast search we call it `poolcheck` [5]), and simply pass the pointer through to `free`. Third, if an internal function may be called from external code, we must ensure that the external code calls the original function, not the pool-allocated version. This ensures backwards-compatibility but makes it possible to miss bounds errors in the corresponding function. In most cases, we can directly transform the program to pass in the original function and not the pool-allocated version (this change can be made at compile-time if it passes the function name but may have to be done at run-time if it passes the function pointer in a scalar variable). In the general case (which we have not encountered so far), the function pointer may be embedded inside another data structure. Even for most such functions, the compiler can automatically generate a "`varargs`" wrapper designed to distinguish transformed internal calls from external calls. When this is not possible, we must leave the callback function (and all internal calls to it), completely unmodified.

Except in call-back functions, bounds checks can still be performed within the available program for *all* heap-allocated objects (internal or external). Like JK, we intercept all direct calls to malloc and record the objects in a separate global splay tree. For pointer arithmetic on a pointer to an incomplete node, we check both the splay tree of the recorded pool for that node and the global splay tree. All heap objects must be in one of those trees, allowing us to detect bounds violations on all such objects.

Internal global and stack objects will be recorded in the splay tree for the pool and hence arithmetic on pointers to them can be checked. We cannot check any static or stack objects allocated in external code since we do not know the size of the objects. The JK and JKRL techniques have the same limitation.

3.5 Errors in Calling Standard Library Functions and System Calls

More powerful error checking is possible for uses of recognized standard library functions and system calls. Many bugs triggered inside such functions are due to incorrect usage of library interfaces and not bugs within the library itself. We can guard against these interface bugs by generating wrappers for each potentially unsafe library routine; the wrappers first check the necessary preconditions on buffers passed to the library call and then invoke the actual library call. For example, for a library call like `memcpy(void *s1, const void *s2, size_t n)`, we can generate a wrapper that checks (1) `n` > 0, (2) the object pointed to by `s2` has atleast `n` more bytes starting from `s2` and (2) the object pointed to by `s3` has atleast `n` more bytes starting from `s3`. These checks can be done using the same `getreferent` and `boundscheck` functions as before.

Note that the wrappers referred to here are not for compatibility between checked code and library code, and are only needed if extra bug detection is desired. We have written the wrappers for many of the standard C library functions because our compiler does not yet generate them automatically.

3.6 Optimizations

There are a number of ways to reduce the overheads of our run-time checks further. We briefly describe three optimizations that we have implemented. The first optimization below is specific to our approach because it requires a key property of pool allocation. The other two are orthogonal to the approach for finding referents and can also be used with the Jones-Kelly or Ruwase-Lam approaches.

First, we observe that a very large number of single-element objects (which may be scalars or single-element arrays) are entered into the splay trees in all three approaches. Since a pointer to any such object can be cast and then indexed as a pointer to an array (e.g., an array of bytes), references to all such objects (even scalars) must be checked for bounds violations. While many local scalars of integer or floating point type are promoted to registers, many other local and all global scalars may still stay memory-resident. Entering all such scalars into the search trees is extremely wasteful since few programs ever index into such scalars, legally or illegally. We propose a technique to avoid entering single-element objects into search trees while still detecting bounds violations for such objects.

To achieve this goal, two challenges must be solved: (1) to identify single-element object allocations, and (2) to detect bounds violations even if such objects are not in the splay trees. For the former, we observe that most pools even in C and C++ programs are *type-homogeneous* [13], i.e., all objects in the pool are of a single type or are arrays of that type. For non-type-homogeneous pools, the pool element type is simply a byte. Furthermore, all objects in such a pool are aligned on a boundary that is an exact multiple of the element size. The size of the element type is already recorded in each pool at pool creation time. This means that the run-time can detect allocations of scalars or single-element arrays: these are objects whose size is exactly the size of the pool element type. We simply do not enter such objects into the splay tree in the pool.

For the latter problem, the specific issue is that a referent look-up using a valid pool descriptor will not find the referent object in the splay tree. This can only happen for two reasons: (i) the object was a one-element object, or (ii) the object was an unchecked object or a non-existent object but the pointer being dereferenced was assigned the same pool during pool allocation. The latter can happen, for example, with code of the form:

```
  T* p = some_cond? malloc(..) : external_func(..);
```

Here, the pointer `p` is assigned a valid pool because of the possible malloc, but if it points to an object returned by the external function `external_func`, the referent lookup will not find a valid referent. The same situation arises if the pointer `p` were assigned an illegal value, e.g., from an uninitialized pointer or by casting an integer. To distinguish the first case from the second, we simply use the pool metadata to check if the object is part of the pool. This check, which we call a `poolcheck`, is a key runtime operation in our previous work on memory safety [5], and the pool run-time has been optimized to make it very efficient. Combining these techniques, we can successfully identify and omit single element arrays from the splay trees, and yet detect when they are indexed illegally.

The next two optimizations are far simpler and not specific to our approach. They both exploit the fact that it is very common for a loop nest or recursion to access very few arrays (often one or two) repeatedly. Since all accesses to the same array have the same referrent, we can exploit this locality by using a small lookup cache before each splay tree. We use a two-element cache to record the last two distinct referents accessed in each pool. When an access finds the referent in the cache, it reduces overhead because it avoids the cost of searching the splay tree to find the referrent (we found this to be more expensive even if the search succeeeds at the root), and also of rotating the root node when successive references to the same pool access distinct arrays. It increases the overhead on a cache miss, however, because all cache elements must be compared before searching the splay tree. We experimented with the cache size and found that a two-element cache provided a good balance between these tradeoffs, and improved performance very significantly over no cache or a one-element cache.

The third optimization attempts to achieve the same effect via a compile-time optimization, viz., loop-invariant code motion (LICM) of the referent lookup. (We find that the two-element cache is important even with this optimization because LICM sometimes fails, e.g., with recursion, or if the loop nest is spread across multiple functions, or the referent lookup does not dominate all loop exits. Implementing this optimization is easy because the referent lookup is a pure function: the same pointer argument always returns the same referent object (or none). Therefore, the lookup is loop-invariant if and only if the pointer is loop-invariant.

4. COMPILER IMPLEMENTATION

We have implemented our approach using the LLVM compiler infrastructure [12]. LLVM already includes the implementation of Automatic Pool Allocation, using a context-sensitive pointer analysis called Data Structure Analysis (DSA). We implemented the compiler instrumentation as an additional pass after pool allocation. We also run a standard set of scalar optimizations needed to clean up the output of pool allocation [13]. Because DSA and pool allocation are interprocedural passes, this entire sequence of passes is run at link-time so that they can be applied to as complete a program as possible, excluding libraries available only in binary form. Doing cross-module transformations at link-time is standard in commercial compilers today because it preserves the benefits of separate compilation.

Our implementation includes three optimizations described earlier: leaving out single-element objects from the splay tree in each pool, the two-element cache to reduce searches of the splay tree, and moving loop-invariant referent lookups out of loops. In previous work, we have also implemented an aggressive interprocedural static array bounds checking algorithm, which can optionally be used to eliminate a subset of run-time checks [6].

We compile each application source file to the LLVM compiler IR with standard intra-module optimizations, link the LLVM IR files into a single LLVM module, perform our analyses and insert run-time checks, then translate LLVM back to ANSI C and compile the resulting code using GCC 3.4.4 at -O3 level of optimization. The final code is linked with any external (pre-compiled) libraries.

In terms of compilation time, DSA and Automatic Pool Allocation are both very fast, requiring less than 3 seconds combined for programs up to 130K lines of code that we have tested. This time is in fact a small fraction of the time taken by gcc or g++ at -O3 for the same programs) [13]. The additional compiler techniques for bounds checking described and implemented in this work add negligible additional compile time.

5. EXPERIMENTS

We present an experimental evaluation of our bounds checking technique, with the following goals:

- To measure the net overhead incurred by our approach.
- To isolate the effect of using multiple distinct splay trees and the associated optimizations, which is our key technical improvement over the Ruwase-Lam (and so Jones-Kelley) approaches.
- To evaluate the effectiveness of our approach in detecting known vulnerabilities. For this purpose, we use Zitser's suite of programs modeling vulnerabilities found in real-world software [23].

It is also interesting to confirm the backwards-compatibility of our approach. In our experience so far, we have required *no* changes to any of the programs we have evaluated, i.e., our compiler works on these programs "out-of-the-box." This is similar to Jones-Kelly and Ruwase-Lam but significantly better than other previous techniques that use metadata on pointers, applied to the same programs, discussed in Section 5.3 below.

5.1 Overheads

We have evaluated the run-time overheads of our approach using the Olden [3] suite of benchmarks, and the unix daemons, ghttpd, bsd-fingerd, and wu-ftpd-2.6.2. We use the Olden benchmarks because they are pointer-intensive programs that have been used in a few previous studies of memory error detection tools [20, 15, 21]. We compare our overheads with these and other reported overheads in Section 5.3. The benchmarks and their characteristics are listed in Table 2. The programs are compiled via LLVM and GCC, as described in the previous section. For the benchmarks we used a large input size to obtain reliable measurements. For the daemon programs we ran the server and the client on the same machine to avoid network overhead and measured the response times for client requests.

The "LLVM (base)" column in the table represents execution time when the program is compiled to the LLVM IR with all standard LLVM optimizations (including the standard optimizations used to clean up after pool allocation, but not pool allocation itself), translated back to C code, and the resultant code is compiled directly with GCC -03. The "PA" column shows the time when we run the above passes as well as the pool allocator but do not insert any run-time checks. Notice that in a few cases, pool allocation speeds up the program slightly but doesn't significantly degrade the performance in any of these cases. We use the LLVM(base) column as the baseline for our experiments in calculating the net overhead of our bounds checking approach because we believe that gives the most meaningful comparisons to previous techniques. Since Automatic Pool Allocation can be used as a separate optimization, the PA column could be used as a baseline instead of LLVM(base), but the two are close enough for the benchmarks in the table that we do not expect this choice to affect our conclusions.

The "BoundsCheck" column shows the execution times with bounds checking. Here, we have turned on the three optimizations that we have discussed in Section 3.6: caching on top of the the splay tree, loop invariant code motion, and not storing single-element objects in the splay tree. The "Slowdown" ratio shows the net overhead of our approach relative to the base LLVM. In almost half of the benchmarks, we found that overheads are within 3%. Only two programs (em3d, health) have overheads greater than 25%.

In order to isolate the benefits of smaller splay data structures, we conducted another experiment. The pool allocator pass provides an option to force it to merge all the pools in the program in to one single global pool. This pool uses the same memory allocation algorithm as before but puts all tracked objects into a single splay tree. This allowed us to isolate the effect of using multiple splay trees instead of the single splay tree used by **JK** and **JKRL**. Note that we cannot use optimization 1 (leaving singleton objects out of the splay tree) because after merging pools, type information for the pool is lost and we cannot identify singleton object allocations. The other two optimizations – caching splay tree results and LICM for referent lookups – are used, which is again appropriate because they can also be used with the previous approaches. Columns "PA with one pool" and "PA with one pool + bounds checking" show the execution times of this single-global-pool program without and with our run-time checks, and the last column shows the ratio of these. The benchmark `health` used up all system memory and started thrashing. The main reason is because we could not eliminate singleton objects from the splay tree, making the single global splay tree much larger than the combined splay trees in the original code. Comparing the last column with the column labelled "Our Slowdown Ratio" shows that in atleast three cases (health, mst, perimeter) the overheads when using multiple search data structures is dramatically better (more than 100%) than using a single datastructure for the entire heap. The benefits are also significant in tsp and bisort. The remaining programs show little difference in overheads for the two cases.

5.2 Effectiveness in detecting known attacks

We used Zitser's suite of programs modeling real-world vulnerabilities [23] to evaluate the effectiveness of our approach in detecting buffer overrun violations in real software. The suite consists of 14 model programs, each program containing a real world vulnerability reported in bugtraq. 7 of these vulnerabilties were in `sendmail`, 3 were in `wu-ftpd`, and 4 were in `bind`. This suite has been used previously to compare dynamic buffer overflow detection approaches [22].

The results of our experiments are reported in Figure 5.2. We are able to detect all the vulnerabilities in all three programs out of the box. In each case, the illegal memory reference was detected and the program was halted with a run-time error. The four bugs in bind are not triggered in the main program but in a library routine (e.g. due to passing a negative argument to memcpy). These bugs are automatically detected by our approach using the wrappers described earlier because they are due to incorrect usage of the library functions (and not bugs within the library).

5.3 Performance comparison with previous approaches

Finally, we briefly compare the overheads observed in our work with those reported by other work, to the extent possible. We can make direct comparisons in cases where there are published results for Olden suite of benchmarks. When such numbers are not available, only a rough comparison is possible, and then only in cases where the differences are obviously large. Note also that some previous techniques including [20, 16] detect a wider range of bugs than we do in the current work. Where possible, we try to compare the overheads they incur due to bounds checking alone.

The two previous approaches with no compatibility problems, JK and JKRL, have both reported far higher overheads than ours, as noted in the Introduction. Jones and Kelly say that in practice, most programs showed overheads of 5x-6x. Ruwase and Lam report slowdowns up to a factor of 11x–12x if enforcing bounds for all objects, and up to a factor of 1.6x–2x for several significant programs even if only enforcing bounds for strings. Their overheads are even higher than those of Jones and Kelly because of the additional cost of checking all loads and stores and also of checking for OOB objects that may have to be deallocated as they go out of bounds. While two of our optimizations (the two-element cache and LICM for loop-invariant referent lookups) might reduce these reported overheads, it seems unlikely that they would come close to our reported overheads. Our overheads are dramatically lower than these previous techniques because of a combination of using multiple splay trees (whose benefit was shown earlier), not requiring checks on loads and stores, and the additional optimizations.

Xu. et al [20] have proposed to use metadata for pointer variables that is held in a separate data structure that mirrors the program data in terms of connectivity. They use the metadata to identify both spatial errors (array bounds, uninitialized pointers) and temporal errors (dangling pointer errors). Their average overheads for Olden benchmarks for just the spatial errors are 1.63 while ours are far less at 1.12. Moreover, their approach incurs some difficulties with backwards compatibility, as described in Section 6.

CCured [15] divides the pointers of the program into safe, seq pointers (for arrays) and wild (potentially unsafe) pointers at compile-time. At run-time CCured checks that seq pointers never go out of bounds and wild pointers do not clobber the memory of other objects. While CCured checking for WILD pointers is more extensive than ours, in the case of Olden benchmarks, they did not encounter any wild

Benchmark	LOC	Base LLVM	PA	BoundsCheck	**Our slowdown ratio**	PA with one pool	PA with one pool + boundschecks	**One-pool ratio**
bh	2053	9.146	9.156	9.138	**1.00**	9.175	10.062	**1.10**
bisort	707	12.982	12.454	12.443	**0.96**	12.425	14.172	**1.14**
em3d	557	6.753	6.785	11.388	**1.69**	6.803	11.419	**1.68**
health	725	14.305	13.822	19.902	**1.39**	13.618	-	**-**
mst	617	12.952	12.017	15.137	**1.17**	12.203	28.925	**2.37**
perimeter	395	2.963	2.601	2.587	**0.87**	2.547	6.306	**2.48**
power	763	2.943	2.920	2.928	**0.99**	2.925	2.931	**1.00**
treeadd	385	17.704	17.729	17.310	**0.98**	17.706	21.063	**1.19**
tsp	561	7.086	6.989	7.219	**1.02**	6.978	8.897	**1.27**
AVG					**1.12**			
Applications								
fingerd	336	2.379	2.384	2.475	**1.04**	2.510	2.607	**1.04**
ghttpd	837	11.405	9.423	9.466	**0.83**	11.737	12.182	**1.03**
ftpd	23033	1.551	1.539	1.542	**0.99**	1.551	1.546	**1.00**

Figure 2: Benchmarks and Run-time Overheads. The One-Pool Ratio compared with Our Slowdown Ratio isolates the benefit of partitioning the splay-tree.

Program	No. of vulnerabilities	No. of vulnerabilties detected	No. of vulnerabilties detected with std. lib. check
sendmail	7	7	7
bind	4	0	4
wu-ftpd	3	3	3

Figure 3: Effectiveness of our approach in detecting known attacks/vulnerabilities

pointers [15]. It is important to note, however, that CCured uses garbage collection for dynamic memory management and the overhead due to garbage collection is unknown. The reported average overheads for Olden are 1.28, which is only slightly higher than our observed overheads. However, they needed to change 1287 lines of code in total to achieve these results while our technique works out of the box.

Yong et al [21] describe a technique to identify many illegal write references and free operations via pointers, by identifying a set of pointers that might be unsafe using a pointer-analysis and tagging the memory corresponding to the objects those pointers may point to. They use a shadow memory with 1 tag bit per byte of memory, setting this tag bit on allocations and clearing them on deallocations. They check these tag bits on every write or free of a potentially unsafe pointer, allowing them to detect a number of potential security attacks and some errors such as accesses to a freed memory that has not been reallocated. They report an average overhead of 1.37x for the Olden benchmarks (the fraction of overhead due to array references is unknown). Unlike our work and the previous papers described above, they do not perform any checks on read operations and read operations are far more frequent than writes.

6. RELATED WORK

We focus our comparisons on techniques for run-time bounds checking, and any optimizations directly related to those techniques. We do not discuss existing compile-time techniques for bounds checking here (including our own), because these techniques are complementary and can be used to eliminate some run-time checks in any of the approaches discussed here.

There are a number of debugging tools like `purify` and `valgrind` that use binary instrumentation to detect a wide range of memory referencing errors. Using binary instrumentation allows these tools to add arbitrary metadata to pointers without the compatibility problems of other approaches. These tools, however, incur very high run-time overheads, e.g., often greater than a factor of 10x for `purify` and `valgrind`. Also, in case of purify it does not catch some pointer arithmetic violations if the arithmetic arithmetic yields a pointer to a valid region [10].

A number of other approaches target debugging but work at the source level. These include Loginov's work on runtime type checking [14], Kendall's bcc [11], Steffens' rtcc [18]. All of these approaches focus on debugging and usually performance is not a serious consideration. For instance, the reported overheads for Loginov's work are up to 900%.

Some tools including SafeC [1] and Cyclone [9] use an augmented pointer representation that includes the object base and size of the legal target object for every pointer value. Such "fat pointers" require significant changes to programs to allow the use of external libraries, typically introducing wrappers around library calls to convert pointer representations. Furthermore, writing such wrappers may be impractical for indirect function calls, and for functions that access global variables or other pointers in memory. Unlike the remaining techniques, below, however, fat pointers have the major advantage that there is no cost to find the metadata for each pointer value.

To reduce the compatibility problems caused by fat pointers, several recent systems store pointer metadata separately from the pointer variables themselves, at the cost of significantly greater overhead for finding the metadata associated with each pointer. This approach was used by Patil and Fisher [16], CCured [15], and Xu et al. [20]. Separating the metadata eliminates the potential for program failures mentioned above, and reduces the need for wrappers on library calls. This technique does not require wrappers for pointers passed *to* library functions or pointer values explicitly returned by such functions. Wrappers are still needed for checked pointers that may be modified indirectly as a side-effect of a library call, because the metadata before the call would be invalid if the call overwrites the pointer. Such wrappers are likely to be needed less often but, if needed, may be impractical to write for the same reasons as with fat pointers, described above. The work of Xu et al. is also more restrictive than ours because they restrict pointer casts between structures of incompatible types. Finally, and most important from a practical viewpoint, all these techniques have significantly higher overhead than ours, as discussed in more detail in Section 5.3.

As noted in the Introduction, the compatibility problems of both fat pointers and pointers with separately stored metadata occur because the metadata is associated with the pointer itself, and not the object that is the target of a pointer. The work of Jones and Kelly [10] and Ruwase and Lam [17] associate metadata with objects instead of pointers, which greatly reduces the compatibility problem. However, the overheads of these two approaches are quite high. As the comparison in Section 5.3 shows, our approach is able to reduce these overheads greatly, sufficient (we believe) for the technique to be used in production code.

7. SUMMARY AND FUTURE WORK

We have described a collection of techniques that dramatically reduce the overhead of an attractive, fully automatic approach for run-time bounds checking of arrays and strings in C and C++ programs. Our techniques are essentially based on a fine-grain partitioning of memory. They bring the average overhead of run-time checks down to only 12% for a set of benchmarks we have evaluated. Thus, we believe we have achieved the twin goals that have not been simultaneously achieved so far: overhead low enough for production use, and fully automatic checking, i.e., not requiring manual effort to circumvent compatibility problems or to assist the compiler's checking techniques.

We have two goals for the future. First, we aim to evaluate our overheads for a wider range of real-world application programs in the future. Second, we aim to integrate our array bounds checks into the SAFECode system [5, 4], which detects pointer cast errors and dangling pointer errors but not all array bounds errors.

8. REFERENCES

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 1994.

[2] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2000.

[3] M. C. Carlisle. *Olden: parallelizing programs with dynamic data structures on distributed-memory machines*. PhD thesis, 1996.

[4] D. Dhurjati and V. Adve. Detecting all dangling pointer uses in production servers. In *International Conference on Dependable Systems and Networks (DSN)*, June 2006.

[5] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2006.

[6] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without garbage collection for embedded applications. *ACM Transactions on Embedded Computing Systems*, Feb. 2005.

[7] N. Dor, M. Rodeh, and M. Sagiv. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. In *SIGPLAN Conference on Programming Language Design and Implementation*, Sandiego, June 2003.

[8] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM conference on Computer and communications security*, New York, NY, USA, 2003.

[9] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in cyclone. In *Proc. of the 4th international symposium on Memory management (ISMM)*, 2004.

[10] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[11] S. C. Kendall. Bcc: Runtime checking for c programs. In *In Proceedings of the USENIX*, 1983.

[12] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*, San Jose, Mar 2004.

[13] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun 2005.

[14] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. *Lecture Notes in Computer Science*, 2001.

[15] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[16] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Software–Practice and Experience*, 27(1):87–110, 1997.

[17] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 159–169, Feb. 2004.

[18] J. L. Steffen. Adding run-time checking to the portable c compiler. In *Software: Practice and Experience*, 1992.

[19] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Softw. Eng. Notes*, 28(5):327–336, 2003.

[20] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *Proc. 12th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 117–126, 2004.

[21] S. H. Yong and S. Horwitz. Protecting c programs from attacks via invalid pointer dereferences. In *Foundations of Software Engineering*, 2003.

[22] M. Zhivich, T. Leek, and R. Lippmann. Dynamic buffer overflow detection. In *BUGS : Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[23] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT symposium on Foundations of software engineering*, 2004.