BACKWARDS-COMPATIBLE ARRAY BOUNDS CHECKING FOR C WITHVERY LOW OVERHEAD

Dinakar Dhurjati and Vikram Adve, ICSE 2006

Presented by Itay Polack, 02.03.2014

Introduction

We are dealing with the problem of detecting incorrect usage of pointers and arrays.

The purpose is to overcome issues such as buffer overflows, memory corruption and more. The method is to monitor programs, during run-time, and detect memory access errors.

Real-world usage tends to uncover a lot of bugs and edge cases that were not discovered in the testing phase. Therefore the goal is to monitor programs running in production environments. The approach suggested in this paper is pointer analysis, but unlike previous methods, the cost is relatively low (usually) - about 12% - so it can be used in production systems .

Basic problems

1. Existing run-time monitoring frameworks carry a very high performance cost, making it unsuitable for production code.
2. Backward compatibility – we don't want to rewrite applications.
3. External libraries compatibility – we need to work with binary 3rd party libraries; sometimes external libraries modify our memory as well.

The method

We will describe now the original method suggested by Jones and Kelly in a previous paper. Our method will be based on their method.

Identifying Invalid Pointer Operations

Jones and Kelly suggested that the baseline definition for valid pointers should be derived from the ANSI-C standard:

1. Pointers must refer to valid memory (memory allocated on the heap, stack, or global).
2. When performing arithmetic operations on pointers, the result must stay either in the same memory object referred to by the source pointer, or one byte after it.

The basic idea is to keep track of all live object in the memory (earlier approach include only heap object, this paper extends it to all live objects including stack and global variables). Notice that we are not monitoring pointers, we are monitoring live objects.

We will add special run-time checks for each pointer-arithmetic operation – we will locate the object referenced by the source pointer, and make sure that this source pointer is valid and that the arithmetic operation result is still within the allowed bounds of this pointer.

Creating an implementation of this method while the overhead is limited requires careful design choices.

The data structure chosen for storing the live objects is a splay-tree: a variant of binary-tree, which has a "splay" operation that rotates tree nodes, reordering the tree according to our requirements. This data structure gives quick amortized search, insertion and deletion times. It also allows quick searching by range – which we require.

In practice, it works by adding a pre-compilation phase, where a global splay-tree, with all live objects, is updated on each allocation, de-allocation, entering and exiting a function. Each pointer arithmetic operation is inspected to make sure that the source pointer is valid, and that the result of the operation stays within the bound of the object originally referenced by the source pointer .

For example, the following code (assuming the types match):

```
int * p;

p = q + 5;
```

Will include two additional checks:

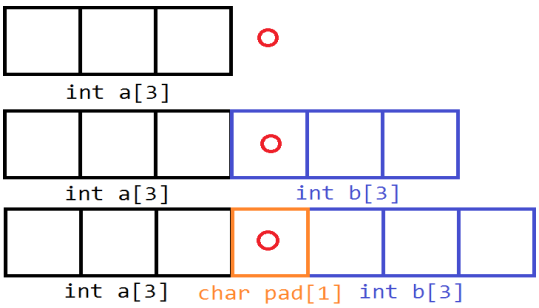
1. The pointer "q" is valid.
2. The memory referenced by q is looked-up. A bound checking makes sure that q + 5 is still within the bounds of this memory object (or one byte after it).

Assuming any of those validation checks failed, the target pointer will be modified to a constant invalid value that will cause immediate program crash when trying to access the memory referenced by this pointer (the authors chose the value -2). Why not crash immediately? Probably because we might assign illegal values to pointers we have no intention to touch anymore or to use as pointers.

The One-Off Problem

One additional complication is neighboring memory objects. It's perfectly legal for a pointer to point to a memory location that is one byte after the memory object it referenced. When we have object which was allocated right near this object, we might not know if pointers that points exactly one byte after the first object is pointing on the end of the first object or the beginning of the second one. We can solve this by changing memory allocation functions to add one byte in the end of each object.

See illustration:



It worth mentioning that modifying allocators in such way introduce a problem when external code is involved, as the external code cannot be aware to the fact that memory was allocated in a different way, and will not use our customized allocators. In addition, changing sizes of objects passed as function parameters can change the memory layout of the parameters. Therefore, this workaround cannot be applied to objects and code that might be involved with external libraries.

Out of Bound Objects

So far, this approach would immediately replace pointers that went out-of-bounds with invalid values, crashing the program when accessing them (and so would be the results of all pointer-arithmetic operations involved with such pointers that were changed to "-2"). In reality, this is causing a huge backward compatibility problem – many programs tend to assign illegal values to pointers while still using them in a perfectly correct manner. Most common example – pointers that hold temporary values mid-calculations might refer to illegal address, but the eventual result of the calculation is legal. With the original approach, this temporary value will be invalidated and so will the eventual result.

In a paper following Jones and Kelly's method, Ruwase and Lam suggested a new method for handling pointers that seemed to go out of bounds: out-of-bound objects. When pointer-arithmetic operation ends in invalid value, we will create an out-of-bound object that will contain the original source pointer address (before the arithmetic operation), and the pointer arithmetic result. The target pointer will be replaced with pointer to the OOB object. Now, when each pointer is accessed, we can see if it's an OOB object; we can check if arithmetic operations resulted in valid values, and then we can replace the OOB object with the original value. Otherwise, we can assign a new OOB object for the pointer arithmetic result.

The OOB objects are stored in a hash-table, with the OOB addresses as keys and the objects as values.

Performance Issues in Current Approaches

The JK methods includes a single splay-tree per application, and it’s getting slow for large apps. The RL method is extending the detection capabilities, but the performance toll is even larger: maintaining OOB objects would now require monitoring of all load/store operations to make sure we are not accessing OOB objects. Also, each time when de-allocating memory, we need to check all OOB objects in memory and see is there were any OOB objects originally referencing this memory. RL suggested an optimization – only monitor

string operation. While this greatly reduces the performance penalty, it is still too high for production, and the detection accuracy is not as good.

Optimizations

The work in this paper is focusing on optimizing and reducing the limitations of the previous works.

Introduction – Automatic Pool Allocation

Automatic Pool Allocation - this is a previous work by Chris Lattner and Vikram Adve (one of the paper writers). The idea - improve locality of objects by assigning separate pool for each data structure. Implementation – create a "points-to" graph: a directional graph which nodes are all memory objects. If one object refers another, there will be an edge from the referring to the referee. Using this graph, we will create a separate pool for each node in this graph, which should practically be any data-structure. For example: each linked list will have its own pool.

Pools are as short-lived as possible, allocated in the beginning of the scope where they are required, de-allocated right afterwards. When an object is transferred to another function, its pool goes with it – the pool is added as another function parameter.

For global objects, we will use global pools that are allocated in the main function.

Leveraging Automatic Pool Allocations

The biggest performance issue in JK's approach was managing the splay-tree that holds all live memory objects. We will now assume that our program was successfully transformed to use automatic pool allocation. We know through compile time which object goes to which pool, so looking up object's pool is highly efficient.

We will change the bound-checking code so that each pool will have its own splay-tree. When pointer-arithmetic operations are performed, we only need to query the splay-tree of this object's pool, which is usually much smaller. The off-by-one problem is mostly solved now, as we can tell which object we "should" refer according to its pool.

Another thing: each pool will have its own OOB hash-table. In this way, going through the OOB table during de-allocation is much shorter – we only go through the pool of a specific object, and it is usually much smaller.

Optimizing Out-of-Bounds Object

OOB object in their original form required checking each load/store operation (because we must know if this pointer is a real pointer or was replaced by OOB object). We can avoid this by creating a new abstraction layer. Pointers that go out-of-bounds will be assigned values from a preserved range (if such exist; for example, user-mode programs can use the range assigned to the kernel. Accessing this range would cause an immediate hardware-trap). We will keep another hash-table to map between values from this range and OOB objects. If no such preserved range exists we will need to fall-back to RL approach.

Improving Effectiveness

For improve the effectiveness, they created an instrumented versions of common library functions that include bounds-checking. This is crucial for getting good results, as misusing library functions is a very common cause of illegal memory access.

And More Optimizations

1. Do not maintain splay-trees for single element objects – we can rely on the pool meta-data in order to monitor bound checking for those objects.
2. Additional cache: avoids looking at the splay-trees, use a 2-elements cache.
3. LICM - loop-invariant code motion: For loop operations, extract operations that are not part of the loop outside the loop, to avoid repeating the same operation (usually done by the compiler).

Implementation

The authors used a compiler framework named LLVM to do all the compile-time transformations. Then, GCC was used for compilation.

Evaluation

Evaluation purpose was to determine the performance of the new approach, both relatively – did we improve the previous works, and objectively – is it good enough for production?

For start, they wanted to know the impact of adding bounds-checking on existing code. They also wanted to know if using per-pool splay tree improves the performance of using one global splay-tree.

They tested a few variances, including one vanilla-compiled code, one with pool allocation enabled (but no bound-checking), with bounds-checking and all optimization, and with bounds-checking that use a single pool.

Their results showed that in many cases, the performance penalty of adding bound-checking with all the optimizations was quite small. However, in few cases, it was very high (39% and 69%). They did not suggest why was this difference, just showed that the average was good.

Regarding effectiveness, this framework (including standard-library bounds checking) gave perfect results with the Zitser's test suite.

Olden benchmark source: http://www.sosy-lab.org/~dbeyer/blast_mc/

Related Works

- Binary instrumentation (valgrind, purify) - binary instrumentation solves the backward compatibility issue, but has huge performance cost.
- “Fat” pointers – add additional information to pointers, describing their bounds: memory cost, huge issue with external libraries - need to explicitly wrap function calls, and in some cases (such as global variables accessible from the external library) it’s not practical.
- Keep pointers meta-data – do not modify pointers, maintain pointers meta data separately: high lookup time cost, restrictive implementation and the compatibility

issue is still not solved (wrappers required for external libraries that modifies the pointer, and if it's global variables that's still a problem).

Discussion

- Not all problems related with manual memory-management is addressed – accessing uninitialized data, double free, memory leaks and more. It looks like the framework suggested here can be used for detecting such problems with relatively low cost.
- Why did we get the great performance difference in the benchmark? Why in some cases the performance was merely affected, while in others the cost was great? Knowing this can lead to both better usage of this framework, and would also open a door for further improving it.