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

Dinakar Dhurjati and Vikram Adve
ICSE 2006

Itay Polack
02.03.2014

Introduction

- Unsafe programming language gives unlimited freedom to programmers
  - Direct access to memory
  - Manual resource handling
- This has many benefits:
  - Performance
  - Flexibility
  - Simpler compilers

Resulting Problems

- The programmer is responsible for maintaining correct code
  - In particular, only access valid memory
    - Stack, global or heap
  - But things can easily go wrong
    - Accessing memory that was not allocated or already released
    - Pointer arithmetic that goes out of bounds

Resulting Problems - Cont.

- Consequences are severe and unpredictable
  - Program crashes
    - Unexpected
    - Hard to debug
  - Unexpected behavior
  - Security vulnerabilities

Goal

- Detect out-of-bounds bugs
- Input-sensitve bugs
  - For example: string manipulation
  - Not always caught on development or testing systems
  - Detect on production systems
  - And then what?

Proposed Solution: Runtime Monitoring

- Monitor programs during runtime
- Detect out-of-bounds errors during runtime
  - Illegal pointer access
  - Out-of-bounds arrays access
- Crash and burn!

Runtime Monitoring

- Keep track of all pointers during runtime
- Detect illegal access and react immediately
- This is not easy to achieve
  - Performance cost
  - Memory cost
  - Compile time cost
  - Compatibility
    - External libraries
    - Legacy code

How Can We Achieve That?

- Runtime bounds checking
  - Add checks during compile time
  - Keep and validate pointers state during runtime
- Standard library function wrappers
- Optimizations!

Pointer Tracking

- Based on the ANSI-C standard
  - Pointers must point at valid memory
  - Pointer arithmetic result must stay within the same object or one byte after it
- We will keep track of all **objects**
  - For a given pointer value, search the object it's pointing on
  - Make sure that arithemtic operations are not getting out of the same object bounds(+1)
- Assign illegal pointer values for illegal operations
  - Immediate crash when the program tries to access it

Pointer Tracking - Example

```
int *p = (int*)malloc(sizeof(int)*4);
p[0] = 1;
if (p[4] != 5) {
  ...
}
```

Pointer Tracking - Example

```
int *p = (int*)my_alloc(sizeof(int)*4);
int *tmp_p = bounds_check(p + 0);
*tmp_p = 1;
int *tmp_p2 = bounds_check(p + 4);
if (*tmp_p2 != 5) {
  ...
}
```

Pointer Tracking - Example

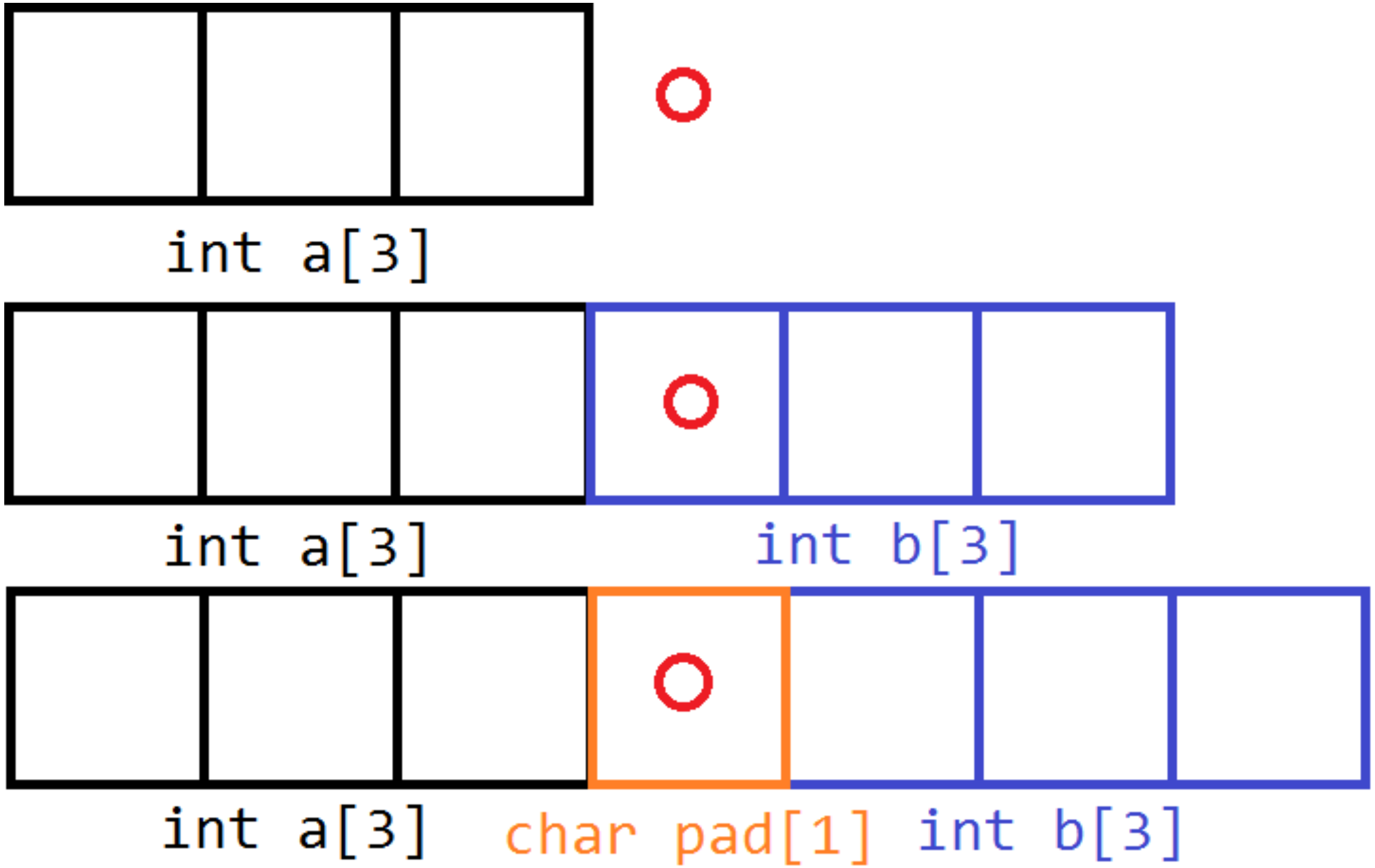
```
my_alloc(void *ptr, size_t size) {
  ptr = malloc(size);
  add_object(ptr, size);
}

bounds_check(void * ptr) {
 if (ptr == -2 || !find_object(ptr)) return -2;
 return ptr;
}
```

Pointer Tracking - One Off

- We are allowed to point one byte past an object
  - How can we distinguish pointing one byte past an object, and pointing on another one?
  - Solution: padding
  - ...but what about backward compatibility?

Pointer Tracking - One Off



Pointer Tracking - Data Structures

- What data structure should we use to store the memory objects mapping?
- Splay-tree: quick insertion and lookup, range searching, good locality
  - Provides O(log(N)) for basic operations
  - Use a global splay-tree for the whole application. What happens when N is large

Splay Tree - Reminder

- Binary tree
- Self-adjusting (splay)
- O(log(N)) amortized time for basic operations
- Splay operation
  – Re-arrange the tree - bring elements to the top
    - Perform tree rotations
  – Faster access
  – Can perform on local variables

Monitoring Out-Of-Bounds Pointers: Improvement

- Real-world: many programs (~60%) do not follow the rules
  - Illegal values are fine if we do not access them
  - We need to keep track of pointers even if they are out-of-bounds
- Introduce out-of-bound objects
  - When pointer arithmetic operation results in illegal values, replace in a special out-of-bounds (OOB) object
  - Keep track of the pointer using this OOB object
    - Holds the original pointer value, and the pre-OOB operation value
    - Use a hash-table to map address-to-OOB
  - Restore the pointer when its value got back to safety

Maintaining OOB Objects Is Pricey

- Allocate a new object if arithmetic operation led to illegal value
- Search the OOB for any pointer arithmetic operation resulting in unknown memory
- All load/store operations must be checked for OOB
- De-allocation of any object requires extensive search
  - Any OOB might originally pointed on this object
  - Must search the whole OOB table

Are We Done?

- We have good detection of illegal pointer access, illegal arithmetic operations, and maybe more
- Backward compatibility is still an issue
  - Padding requirement
- Performance cost is very high, it is not really suitable for production
  - We can limit the checks to string operations only
  - Still not good enough

Introduction: Automatic Pool Allocation

- Original purpose: memory access optimization and easier analysis
- Allocate whole data-structures in a designated pool
  - All data-structure nodes are in the same memory pool
  - Locality – better cache and prefetching performance
  - Easier to analyze

Automatic Pool Allocation - Implementation

- Pointer analysis
- Build a data-structure graph
  - Each node represent memory object
  - Edge between memory objects that might point to each other
  - **Merge nodes that point on the same data-structure**
- Order all nodes of a data-structure subgraph in their own pool
- Pools are short-lived, and follow the call-graph

Automatic Pool Allocation

```
struct List { Patient *data; List *next }

void addList(List *list,
             Patient *data){
  List *b = NULL, *nlist;

  while (list ≠ NULL) {
    b = list;
    list = list→next;
  }

  nlist = malloc(List);
  nlist→data = data;
  nlist→next = NULL;
  b→next = nlist;
}
```

Automatic Pool Allocation

```
void addList(List *list,
             Patient *data);
void ProcessLists(int N) {
 List *L1 = calloc(List);
 List *L2 = calloc(List);

 /* populate lists */
 for (int i=0; i≠N; ++i) {
  tmp1 = malloc(Patient);
  addList(L1, tmp1);

  tmp2 = malloc(Patient);
  addList(L2, tmp2);
 }
}
```

Automatic Pool Allocation

```
void ProcessLists(unsigned N) {
 PoolDescriptor_t L1Pool, PPool;

 poolinit(&L1Pool, sizeof(List));
 poolinit(&PPool, sizeof(Patient));

 List = poolalloc(&L1Pool);

 for (unsigned i=0;i≠N;++i)
  tmp = poolalloc(&PPool);

  pa_addList(L1, tmp, &L1Pool)

 }
 pooldestroy(&PPool);
 pooldestroy(&L1Pool);
}
```

Leveraging Automatic Pool Allocation

- Reminder: previous works used a single data-structure to maintain all memory objects
  - Huge splay-tree - high performance cost
- Apply automatic pool allocation
  - Each pool will have its own splay-tree
  - Computed in compile time

Leveraging Automatic Pool Allocation

- Validating pointer arithmetic operations is much faster now
  - Result of pointer arithmetic must stay in the same pool
  - We only need to search one, smaller, splay tree
  - The pool ID is already known - low overhead

Leveraging Automatic Pool Allocation

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

Leveraging Automatic Pool Allocation

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

Leveraging Automatic Pool Allocation - Challenges

- Do we always have the pool descriptor?
  - Casting
  - External code
  - Just ignore
- What about non-heap objects?
  - Global variables
  - Stack allocated objects
  - Create dummy pool descriptors

Handling Out-Of-Bound Objects

- Reminder: previous works used a special “out-of-bounds” objects
  - Keep track of pointer arithmetic operations that went out-of-bounds
  - Very high cost

Handling Out-Of-Bound Objects

- Assign special memory values to out-of-bounds pointers
  - Use a reserved range
    - For example: kernel-reserved memory range
  - Unique address for each OOB pointer
  - Maintain an additional table for mapping those addresses to OOBs
  - Hash-table per pool
- Immediate crash on load/store – no need to monitor
- Very little to search on free

Compatibility With External Code

- The modifications we introduced cannot always work with external libraries
- Memory allocation and deallocation is changed
  - External libraries are not aware of it
  - Sometimes they modify variables
- Functions interfaces change
  - Functions passed as callbacks cannot change their interface

Compatibility - Solutions

- Do not change calls to external code
- Suspect pointers that were passed to external code
  - Check if they still reside in the same pool
- Callback functions
  - Maintain "checked" and "unchecked" versions of the function
  - Not always possible - exclude functions from bound checking

Library Functions

- Incorrect usage of library functions is extremely common
- Considered as an external code
  - But too important to skip
- Create instrumented standard library wrappers
  - Bounds checking based on parameters and pointers status
  - Optional

Library Functions - Example

```
memcpy(void *p1, void *p2, size_t
n) {
  // Is n > 0?
  // Are p1 and p2 valid?
  // Is (p1 + n) valid?
  // Is (p2 + n) valid?
}
```

Library Functions - Challenges

- Wrapper functions need to be hand-crafted
- We don't always have all the information
  - For example: strlen()
  - Wrapper might not be always enough

More Optimizations

- Single-object elements objects are common
  - Scalar values
  - Single-element arrays
  - We still need to check for out-of-bounds errors
- Avoid entering such objects to splay-trees
  - Detection: pool size equals the object size
  - If it has no splay tree but belongs to the pool – it's a single-object element

And Even More Optimizations

- Caching
  - Very small cache, before even checking the splay-tree
- LICM
  - Do we really need to check the same object each loop iteration?

Implementation

- LLVM
  - Compiler infrastructure
  - Supports automatic pool allocation
- Apply optimizations and then use GCC for generating the binaries

Evaluation

- Performance
  - How are we doing compared to previous works?
  - How is the overall performance?
- Effectiveness
  - Did we spot all the bugs?

Evaluation - Benchmarks

- Use the Olden benchmark and Linux daemons for comparing performance
  - Common benchmark used in many relevant works
- Use Zitser's suite for testing the detection ratio

Evaluation - Baselines

- Baselines: standard compilation with no instrumentation
- We want to evaluate each of the steps
  - Are they really effective?
  - Pool allocation, with no bounds checking (PA)
  - Pool allocation, with bound checking (BoundsCheck)
  - Pool allocation with one pool
  - Pool allocation with one pool and bound checking

Evaluation – Performance Results

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**

Evaluation - Discussion

- Automatic pool allocation by itself usually improves performances
- Average slowdown ratio is 12%
  - In some cases, it's much worse
  - In some cases, it's *better*
  - Why?

Evaluation – Efficiency Results

- All known bugs were found in the test-suite
- Checking standard-library functions was mandatory

Evaluation – Conclusions

- Low overhead in many scenarios
  - Could be useful for non-critical production systems
  - Is it possible to evaluate the possible overhead?
- Good bug-detection ratio
  - Still limited
  - Do we really cover anything?

Related Works– Augmented Pointers

- Pointers hold additional meta-data
  - Pointer base address and size
  - Efficient lookup
- Compatibility with external code is problematic
  - Need to strip pointers before calling external functions
  - Need manually written wrappers
  - What if external library modifies a global variable?

Related Works– Augmented Pointers

- Suggested improvement: decouple meta-data
  - Keep the pointers meta-data in a separate table
  - High performance cost
  - Global variables issue is not resolved

Related Works – Binary Instrumentation

- Tools such as Valgrind and Purify
- Binary instrumentation
  - No backward compatibility problem

Performance cost is too high for production

Questions?

Discussion

- Is it ready for production?
- What about other problems?
  - Double-free?
  - Accessing initialized data?
  - Memory leaks?
- Could we use a better data-structure?
  - Hash-map with partial keys?
- Other suggestions?