Memory Safety Without Garbage Collection for Embedded Applications

---

Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner

2018-05-07

Papers

Three papers referenced:

1. “Memory Safety Without Runtime Checks or Garbage Collection” by Dhurjati et al. 2003
2. “Memory Safety Without Garbage Collection for Embedded Applications” by Dhurjati et al. 2005
3. “Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap” by Lattner and Adve 2005

Problem

- Many systems rely on GC and runtime checks before individual memory operations to ensure memory safety

Problem

- Many systems rely on GC and runtime checks before individual memory operations to ensure memory safety
- Embedded systems have energy, memory, and power limitations precluding the use of a high-overhead runtime

Problem

- Many systems rely on GC and runtime checks before individual memory operations to ensure memory safety
- Embedded systems have energy, memory, and power limitations precluding the use of a high-overhead runtime
- SafeC, CCured, and Vault have between 20% to 300% slowdown

Problem

- Many systems rely on GC and runtime checks before individual memory operations to ensure memory safety
- Embedded systems have energy, memory, and power limitations precluding the use of a high-overhead runtime
- SafeC, CCured, and Vault have between 20% to 300% slowdown
- Need (mostly) static techniques to ensure safety of dynamically allocated memory
  - Require no additional programmer annotations
  - Not overly restrict semantics of language (e.g. C)

Solution

- Improve on Control-C and Automatic Pool Allocation for safety

Solution

- Improve on Control-C and Automatic Pool Allocation for safety
- A program is memory safe if
  - It never references a memory location outside address space allocated for or by it
  - It never executes instructions outside code area created by the compiler within that space

Solution

- Improve on Control-C and Automatic Pool Allocation for safety
- A program is memory safe if
  - It never references a memory location outside address space allocated for or by it
  - It never executes instructions outside code area created by the compiler within that space
- Control-C
  - “Strongly-typed” (see next slide)
  - Affine relationships between array’s size and address used to index into it
  - Dynamic memory allocation via single region at a time
  - Goal: 100% static checking using *existing* compiler techniques

Assumptions

- Some runtime errors are safe (e.g. growing stack beyond available space, attempted access to kernel memory)
- Strong typing
- No pointer-to-pointer casts
- Unions must be castable to each other
- Initialization of local pointers before dereference
- Individual data types no larger than size of reserved address space
- Cannot store address of stack location in heap-allocated object, global variable, or function return value

Behaviors

They show that the following behaviors are mitigated:

Behaviors

They show that the following behaviors are mitigated:

Unitialized pointers

- Dataflow analysis to ensure automatic (i.e. stack-allocated) scalar pointers are initialized
- All unitialized global scalar pointers point to base of reserved address space (thus can trigger safe runtime error)

Behaviors

They show that the following behaviors are mitigated:

Unitialized pointers

- Dataflow analysis to ensure automatic (i.e. stack-allocated) scalar pointers are initialized
- All unitialized global scalar pointers point to base of reserved address space (thus can trigger safe runtime error)

Stack safety

- To prevent accessibility of address of local variables after function return
- Traverse Data Structure Graph, computed via Data Structure Analysis
- Check for reachable stack-allocated objects from function pointer arguments, globals, and return values

Data Structure Analysis

- Computes a data structure graph (points-to graph) for memory objects, identifying disjoint instances

Data Structure Analysis

- Computes a data structure graph (points-to graph) for memory objects, identifying disjoint instances
- Separate graph computed for each function

Data Structure Analysis

- Computes a data structure graph (points-to graph) for memory objects, identifying disjoint instances
- Separate graph computed for each function
- All functions within a strongly-connected component share a single graph

Data Structure Analysis

- Computes a data structure graph (points-to graph) for memory objects, identifying disjoint instances
- Separate graph computed for each function
- All functions within a strongly-connected component share a single graph
- Different nodes in the graph represent different objects

Data Structure Analysis

- Computes a data structure graph (points-to graph) for memory objects, identifying disjoint instances
- Separate graph computed for each function
- All functions within a strongly-connected component share a single graph
- Different nodes in the graph represent different objects
- Context sensitive (uses full acyclic call paths)

Data Structure Analysis

- Computes a data structure graph (points-to graph) for memory objects, identifying disjoint instances
- Separate graph computed for each function
- All functions within a strongly-connected component share a single graph
- Different nodes in the graph represent different objects
- Context sensitive (uses full acyclic call paths)
    - allows analysis to distinguish heap objects processed by common functions

Data Structure Analysis

- Computes a data structure graph (points-to graph) for memory objects, identifying disjoint instances
- Separate graph computed for each function
- All functions within a strongly-connected component share a single graph
- Different nodes in the graph represent different objects
- Context sensitive (uses full acyclic call paths)
  - allows analysis to distinguish heap objects processed by common functions
  - enables automatic pool allocation to put distinct instances of same logical data structure into distinct pools

Data Structure Analysis

- Computes a data structure graph (points-to graph) for memory objects, identifying disjoint instances
- Separate graph computed for each function
- All functions within a strongly-connected component share a single graph
- Different nodes in the graph represent different objects
- Context sensitive (uses full acyclic call paths)
  - allows analysis to distinguish heap objects processed by common functions
  - enables automatic pool allocation to put distinct instances of same logical data structure into distinct pools
- Field sensitive (distinguishes pointer fields in structures)

Data Structure Analysis

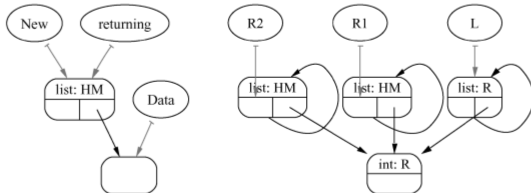
- Computes a data structure graph (points-to graph) for memory objects, identifying disjoint instances
- Separate graph computed for each function
- All functions within a strongly-connected component share a single graph
- Different nodes in the graph represent different objects
- Context sensitive (uses full acyclic call paths)
  - allows analysis to distinguish heap objects processed by common functions
  - enables automatic pool allocation to put distinct instances of same logical data structure into distinct pools
- Field sensitive (distinguishes pointer fields in structures)
- Flow insensitive (order not taken into account)

Data Structure Graphs

```
struct list { list *Next; int *Data; };
list* createnode(int *Data) {
  list *New = malloc(sizeof(list));
  New->Data = Data;
  return New;
}
void splitclone(list *L, list **R1, list **R2) {

  if (L == 0) { *R1 = *R2 = 0; return; }
  if (some_predicate(L->Data)) {
    *R1 = createnode(L->Data);
    splitclone(L->Next, &(*R1)->Next, R2);
  } else {
    *R2 = createnode(L->Data);
    splitclone(L->Next, R1, &(*R2)->Next);
  }}
```

(a) Fragment of C program manipulating linked lists



(b) DS Graphs for `createnode()` (left) and `splitclone()`.

Behaviors (continued)

Heap safety (dangling pointers)

Behaviors (continued)

Heap safety (dangling pointers)

- Problem: pointers to freed memory used to access objects of different types later on

Behaviors (continued)

Heap safety (dangling pointers)

- Problem: pointers to freed memory used to access objects of different types later on
- Solution: type homogeneity principle

Behaviors (continued)

Heap safety (dangling pointers)

- Problem: pointers to freed memory used to access objects of different types later on
- Solution: type homogeneity principle
- “If a freed memory block holding a single object were to be allocated to another object of the same type and alignment, then dereferencing dangling pointers to the previous object cannot cause a type violation”

Behaviors (continued)

Heap safety (dangling pointers)

- Problem: pointers to freed memory used to access objects of different types later on
- Solution: type homogeneity principle
- “If a freed memory block holding a single object were to be allocated to another object of the same type and alignment, then dereferencing dangling pointers to the previous object cannot cause a type violation”
- Don’t prevent dangling pointers, just make safe, via Automatic Pool Allocation

Automatic Pool Allocation Transformation

Overview: create a separate pool for each logical data structure instance on heap (e.g. a particular linked list or graph)

Automatic Pool Allocation Transformation

Overview: create a separate pool for each logical data structure instance on heap (e.g. a particular linked list or graph)

Specifics

1. Identify data structure instances (maximally connected subgraphs containing only heap nodes) in DSG
2. Identify and allocate pools for structures local to procedures
3. Transform function interfaces to include pool pointers as arguments as necessary

Heap Safety from APA

- Pools are type homogeneous

Heap Safety from APA

- Pools are type homogeneous
- Need to ensure memory within some pool P_1 is not used for any other data (i.e. another pool P_2 or trusted library heap allocations) until P_1 is destroyed

Heap Safety from APA

- Pools are type homogeneous
- Need to ensure memory within some pool P_1 is not used for any other data (i.e. another pool P_2 or trusted library heap allocations) until P_1 is destroyed
- Modify run-time so pool memory not released to heap until `pooldestroy`

Heap Safety from APA

- Pools are type homogeneous
- Need to ensure memory within some pool P_1 is not used for any other data (i.e. another pool P_2 or trusted library heap allocations) until P_1 is destroyed
- Modify run-time so pool memory not released to heap until `pooldestroy`
- Dangling pointers will always reference

Heap Safety from APA

- Pools are type homogeneous
- Need to ensure memory within some pool P_1 is not used for any other data (i.e. another pool P_2 or trusted library heap allocations) until P_1 is destroyed
- Modify run-time so pool memory not released to heap until `pooldestroy`
- Dangling pointers will always reference
  - original object

Heap Safety from APA

- Pools are type homogeneous
- Need to ensure memory within some pool P_1 is not used for any other data (i.e. another pool P_2 or trusted library heap allocations) until P_1 is destroyed
- Modify run-time so pool memory not released to heap until `pooldestroy`
- Dangling pointers will always reference
  - original object
  - new object of same type and alignment, in same pool

Pool Example

```
f() {
  ...
  g(p);
  // p->next is dangling
  p->next->val = ... ;
}

g(struct s *p) {
  create_10_Node_List(p);
  initialize(p);
  h(p);
  free_all_but_head(p);
}

h(struct s *p) {
  for (j=0; j < 100000; j++) {
    tmp = (struct s*) malloc(sizeof(struct s));
    insert_tmp_to_list(p,tmp);
    q = remove_least_useful_member(p);
    free(q);
  }
}
```

```
f() {
  Pool *PP;
  poolinit(PP, sizeof(struct s));
  ...
  g(p, PP);
  // p->next is dangling
  p->next->val = ... ;
  pooldestroy(PP);
}

g(struct s *p, Pool *PP) {
  create_10_Node_List(p, PP);
  initialize(p);
  h(p, PP);
  free_all_but_head(p, PP);
}

h(struct s *p, Pool *PP ) {
  for (j=0; j < 100000; j++) {
    tmp = poolalloc(PP);
    insert_tmp_to_list(p, tmp);
    q = remove_least_useful_member(p);
    poolfree(PP, q);
  }
}
```

Memory Consumption

- The change to pool allocation run-time library prevents reuse of memory between two simultaneously live pools
- Better than naïve approach (pools for each static type) because these pools are more short-lived

Memory Consumption

- The change to pool allocation run-time library prevents reuse of memory between two simultaneously live pools
- Better than naïve approach (pools for each static type) because these pools are more short-lived

Three categories of pool use behavior; classification algorithm notifies programmer of case 3 (possible increase in memory usage)

```
p1 = poolinit(s);
t = makeTree(p1);
while(...) {
 processTree(p1,t);
 freeSomeItems(p1,t);


}
freeTree(p1,t);
poolDestroy(p1);
```

(a) No reuse (case 1)

```
p1 = poolinit(s);
t = makeTree(p1);
while(...) {
 processTree(p1,t);
 freeSomeItems(p1,t);
 addItems(p1,t);  // self-reuse


}
freeTree(p1,t);
poolDestroy(p1);
```

(b) Self-reuse (case 2)

```
p1 = poolinit(s);
t = makeTree(p1);
while(...) {
  processTree(p1,t);
  freeSomeItems(p1,t);
  addItems(p1,t);  // self-reuse
  addItems(p2,t);  // cross-reuse
}
freeTree(p1,t);
poolDestroy(p1);
```

(c) Self- and cross-reuse (case 3)

Arrays

Problem: compiler must prove that index expressions in array references lie within array's bounds on *all* execution paths

- Limited by fundamental limits of symbolic integer expressions

Arrays

Problem: compiler must prove that index expressions in array references lie within array's bounds on *all* execution paths

- Limited by fundamental limits of symbolic integer expressions
- Generate Presburger arithmetic constraints $(+, -, *C, \wedge, \vee, \exists, \forall)$

Arrays

Problem: compiler must prove that index expressions in array references lie within array's bounds on *all* execution paths

- Limited by fundamental limits of symbolic integer expressions
- Generate Presburger arithmetic constraints $(+, -, *C, \wedge, \vee, \exists, \forall)$
- Set of language rules for arrays (positive array sizes, affine relationships with index variables, etc.)

Arrays

Problem: compiler must prove that index expressions in array references lie within array's bounds on *all* execution paths

- Limited by fundamental limits of symbolic integer expressions
- Generate Presburger arithmetic constraints $(+, -, *C, \wedge, \vee, \exists, \forall)$
- Set of language rules for arrays (positive array sizes, affine relationships with index variables, etc.)
- Set of trusted library functions with parameter constraints, preconditions that are added to constraint set

Arrays

Problem: compiler must prove that index expressions in array references lie within array's bounds on *all* execution paths

- Limited by fundamental limits of symbolic integer expressions
- Generate Presburger arithmetic constraints $(+, -, *C, \wedge, \vee, \exists, \forall)$
- Set of language rules for arrays (positive array sizes, affine relationships with index variables, etc.)
- Set of trusted library functions with parameter constraints, preconditions that are added to constraint set
- Relevant unconstrained variables cause array access to be marked unsafe

Arrays

Problem: compiler must prove that index expressions in array references lie within array's bounds on *all* execution paths

- Limited by fundamental limits of symbolic integer expressions
- Generate Presburger arithmetic constraints ($+, -, *C, \wedge, \vee, \exists, \forall$)
- Set of language rules for arrays (positive array sizes, affine relationships with index variables, etc.)
- Set of trusted library functions with parameter constraints, preconditions that are added to constraint set
- Relevant unconstrained variables cause array access to be marked unsafe
- Add array bound violations to set of generated constraints, use Omega library to check satisfiability

Arrays

Problem: compiler must prove that index expressions in array references lie within array's bounds on *all* execution paths

- Limited by fundamental limits of symbolic integer expressions
- Generate Presburger arithmetic constraints $(+, -, *C, \wedge, \vee, \exists, \forall)$
- Set of language rules for arrays (positive array sizes, affine relationships with index variables, etc.)
- Set of trusted library functions with parameter constraints, preconditions that are added to constraint set
- Relevant unconstrained variables cause array access to be marked unsafe
- Add array bound violations to set of generated constraints, use Omega library to check satisfiability
- Array access is safe if system is unsatisfiable

Example Array Constraints

```
char A[51];            // last character is set to null
...
k = read(fd, A, 50);   // requires A.size >= 50; implies k <= 50
if (k > 0) {
  len = strlen(A);     // implies len <= A.size
  for (i=0; i < len; i++)
    if (A[i] == '-')
      break;
    ...                // use A and i
}
```

The set of constraints generated are:

(`A.size = 50 && len <= A.size && k <= 50`
`&& i < len && k > 0 && i >= 0`).

Evaluation

- Implemented compiler in LLVM
- Test programs from MiBench (13) and MediaBench (4), 3 others
- Porting effort: few changes to conform with type and array safety (120 out of 42,000 lines)
- Heap and pointer safety effectiveness: all 20 have provably-enforced safety
  - 17 programs have no increase in memory consumption
  - 3 increased because of cross-reuse by other pools (1% - 40% increase)
- Array access checks: difficulties because of non-affine bit operations on index variables, understanding the size of array stored on heap

Evaluation (continued)

Table III. Execution Time and Memory Usage for Heap Safety Approach

Benchmark	Execution Time (s)			Memory Usage (bytes)				
	Orig Time	Heap Safety Time	Exec Ratio	Orig Mem Usage	Pool Alloc Mem. Usage	Mem Ratio 1	Pool Alloc. + Safety Restriction	Mem. Ratio 2
Automotive								
basicmath	1.667	1.672	1.00	16,384	16,384	1	16,384	1
bitcount	0.710	0.727	1.02	16,384	16,384	1	16,384	1
qsort	0.405	0.404	1.00	24,576	24,576	1	24,576	1
susan	0.670	0.675	1.01	253,952	253,952	1	253,952	1
Office								
stringsearch	0.024	0.024	1.00	16,384	16,384	1	16,384	1
Security								
sha	0.145	0.138	0.95	24,576	24,756	1	24,576	1
blowfish	0.713	0.722	1.01	24,576	24,756	1	24,576	1
rijndael	0.340	0.366	1.07	24,576	24,576	1	24,576	1
Network								
dijkstra	0.340	0.349	1.02	32,768	32,768	1	32,768	1
Telecomm								
CRC 32	1.463	1.53	1.04	16,384	16,384	1	16,384	1
adpcm codes	1.255	1.252	1.00	0	0	—	0	—
FFT	0.495	0.478	0.96	540,672	540,672	1	540,672	1
gsm	1.979	1.959	0.98	24,576	24,576	1	24,576	1
Multimedia								
g721	0.354	0.355	1.00	24,576	24,576	1	24,576	1
mpeg(decode)	0.331	0.320	0.97	385,024	401,408	1.04	401,408	1
epic	0.126	0.128	1.01	671,744	681,616	1.01	779,920	1.14
rasta	0.124	0.125	1.01	147,456	212,992	1.44	212,992	1

Exec. ratio is the ratio of execution time after pool allocation to the original time (A ratio of 2 means the program runs twice as long as the original).
Mem. ratio 1 is the ratio of the memory usage of program after pool allocation to that of the original program.
Mem. ratio 2 is the ratio of the memory usage of pool allocated program with our safety restriction to that of just the poolallocated program.