MG++: Memory Graphs for Analyzing Dynamic Data Structures

Vineet Singh
University of California, Riverside
Riverside, CA, USA
vsing004@cs.ucr.edu

Rajiv Gupta
University of California, Riverside
Riverside, CA, USA
gupta@cs.ucr.edu

Iulian Neamtiu
University of California, Riverside
Riverside, CA, USA
neamtiu@cs.ucr.edu

***Abstract*—Memory graphs are very useful in understanding the behavior of programs that use dynamically allocated data structures. We present a new memory graph representation, MG++, and a memory graph construction algorithm, that greatly enhance the utility of memory graphs. First, in addition to capturing the shapes of dynamically-constructed data structures, MG++ also captures how they evolve as the program executes and records the source code statements that play a role in their evolution to assist in debugging. Second, MG++ captures the history of actions performed by the memory allocator. This is useful in debugging programs that internally manage storage or in cases where understanding program behavior requires examining memory allocator actions. Our binary instrumentation-based algorithm for MG++ construction does not rely on the knowledge of memory allocator functions or on symbol table information. Our algorithm works for custom memory allocators as well as for in-program memory management. Experiments studying the time and space efficiency for real-world programs show that MG++ representation is space-efficient and the time overhead for MG++ construction algorithm is practical. We show that MG++ is effective for fault location and for analyzing binaries to detect heap buffer overflow attacks.**

Keywords*—*memory graph, evolution history, memory allocator history, fault location, buffer overflow attacks

I. Introduction

A memory graph, where nodes represent allocated memory chunks and edges represent links between them created by memory stores, is effective in visualizing the shapes of heap-allocated data structures constructed at runtime. Memory graphs are useful in program understanding [1], or identifying data structures used by a program to replace them with more efficient ones [2]. In programs with bugs, execution of faulty code often results in anomalies that can be observed in the memory graph. Thus memory graphs are useful for helping locate memory bugs (e.g., memory leaks and illegal memory access patterns [3]–[5]) as well as in general-purpose debugging [6]. However, prior representations [1], [2], [7] fail to capture important information and their construction algorithms make assumptions that limit their utility:

- *Lack evolution history.* Existing representations [1], [2], [7] are a snapshot of the heap at a program point but do not capture the runtime evolution of the memory graph. This deprives the user of critical information useful in verifying data structure properties and understanding how anomalies were introduced in the memory graph [8].
- *Lack mapping to source code.* Memory graphs used in prior work do not capture the program statements whose execution constructs and modifies the memory graph. This makes it hard for the user to relate memory graph anomalies to faulty source code statements.
- *Lack memory allocator history.* Since existing memory graphs do not capture the behavior of memory allocators, they are not effective when understanding program's (faulty) behavior requires examining the internal actions of the memory allocators (e.g., updates to the internally-maintained free list). This limitation is particularly problematic when programs use custom memory allocators.
- *Allocator information requirement.* Existing methods for constructing memory graphs [1], [2], [7] must know what functions allocate/free memory—information from these functions (e.g., starting address and size of allocated memory chunk) is required during graph construction. The allocator-based approaches can only be applied when allocator information is available.

Keeping our focus on dynamic data structures, we overcome all of the above shortcomings by developing MG++, a new representation of heap memory graphs, and a novel approach to construct them. In addition to information traditionally captured by memory graphs, MG++ also captures the runtime evolution history of data structures and its mapping to the source code (Section II-A). Intuitively, MG++ compactly represents the memory graph at the end of the execution, as well as the evolution history; from this history, the memory graph at any earlier program execution point can be extracted. MG++ also captures the internal actions of the memory allocator (Section II-B). This is useful in debugging programs that internally manage the storage or where understanding program behavior requires examining the interaction between program actions and memory allocator actions. We provide examples of real bugs where this information is critical for understanding faulty behavior. We also found the additional information available in MG++ representation useful for manually analyzing program data structures when coupled with Graphviz [9] to visualize the memory graph.

Our novel technique for MG++ construction is based on binary instrumentation and captures memory allocator behavior without requiring knowledge of the allocator function. The technique is based on the key observation that each field within an allocated chunk of memory is accessed via an address computed as an offset from the starting address of the allocated chunk. This enables us to construct the memory graph without assuming that the allocator functions will supply us with

978-1-4799-8469-5/15 © 2015 IEEE

SANER 2015, Montréal, Canada

the starting address and size information for each newly-allocated chunk. Rather, we are able to construct the memory graph by simply monitoring heap references and operations involving them. Runtime information is analyzed to construct the graph by grouping heap references together to form nodes and using stores in memory to create edges between graphs nodes (Section III-B).

We have implemented our memory graph construction technique using the PIN dynamic binary instrumentation framework [10] for Linux executables running on the IA-32 architecture. We have evaluated the efficiency and effectiveness of our techniques on various real-world programs; we now highlight the results. The space required for storing the complete memory graph evolution history of a large real-world program (the CPython interpreter) using MG++ representation is less than 150 MB; using prior memory graph representations would require about 100 GB (capturing snapshots after each memory graph change). For the benchmarks evaluated, our MG++ construction approach manages to keep execution time of instrumented code to an average of 1.7x in comparison to an allocator-based approach while the worst case slowdown is less than 5x. This shows that our approach provides a practical method for constructing memory graphs in scenarios where allocator information is not available. We illustrate the benefits of our representation in locating faults in GNOME and Mozilla and detecting heap buffer overflows using the RIPE test suite [11].

The key contributions of this paper are:

1) The MG++ memory graph representation that captures the runtime evolution of the memory graph and maintains a mapping to the program code responsible for the graph's evolution.
2) Additional MG++ features that handle cases where memory management actions must be included in the analysis.
3) A method for constructing memory graphs that is independent of allocator or symbol table information and can handle custom memory allocators as well as in program memory management.
4) Evaluation of MG++ illustrating its usefulness for (1) fault location in real-world programs, and (2) detection of heap buffer overflows.

II. MG++ Representation

We first present the MG++ representation that captures the evolution of heap data structures as well as the mapping to relevant source code. Next we present the additions to MG++ which capture the behavior of the memory allocator functions as well as splitting and merging of allocated memory chunks. Finally, we show how the memory graph at any program execution point can be extracted from MG++.

A. MG++ for Heap Data Structures

A straightforward approach for tracking the evolution of heap data structures is to capture the traditional memory graph at each program execution point where it is modified. For example, Figure 2 shows the execution of a sequence of statements from a C program that creates a singly-linked list by creating two nodes (statements 11 and 13) and linking them to form the list (statement 15). The last statement (19) is faulty, mistakenly breaking the linked list via the NULL assignment. The programmer can examine the corresponding series of traditional memory graphs [7] and understand how the link list grows and is finally broken by the execution of the faulty statement (19). While examining the sequence of memory graphs allows the programmer to observe the evolution of the link list, including its corruption, this approach is impractical due to its memory cost.

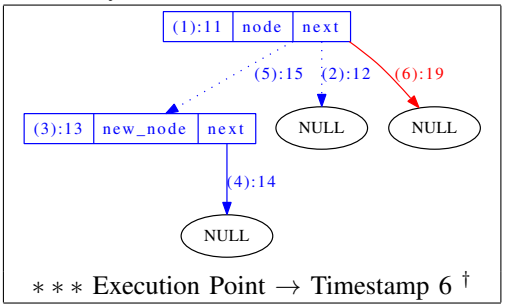


Figure 1. The compact MG++ representation.

To efficiently capture the memory graph's evolution we introduce a compact representation, MG++, from which the memory graph at any execution point can be extracted. As we can see in Figure 1, MG++ is compact because, by construction, MG++ eliminates redundancy across the series of memory graphs corresponding to the six execution points uniquely identified by timestamps 1 through 6. The additional annotations in MG++ represent *timestamps* for capturing the order in which nodes/edges are created/deleted and identities of *source code statements* responsible for changes to the memory graph. In particular, in Figure 1:

- The two non-NULL nodes are labeled with (1):11 and (3):13 indicating their creation at timestamps (1) and (3) by execution of statements 11 and 13, respectively;
- The outgoing edge from new_node−>next to NULL is labeled (4):14 as it was created at timestamp (4) by execution of statement 14; and
- Since node−>next is assigned at timestamps (2), (5), and (6) by statements 12, 15, and 19, it has three outgoing edges labeled (2):12, (5):15, and (6):19. The lifetimes of these edges can be inferred from the timestamps – the (solid) edge labeled with timestamp (6) is the most recent edge; the earlier (dashed) edges exist from the time of their creation to when the next edge is created.

We observe that the use of timestamps prevents *redundancy* across multiple memory graphs and thus makes the MG++ compact. In particular, if a node or an edge is created at timestamp t, and it remains unchanged until the end of execution, represented by timestamp T, then the MG++ will have a single copy of the node or edge labeled with t implying that it has remained unchanged until T.

Given a MG++, the memory graphs corresponding to series of execution points can be extracted and shown to the user. The user can then observe the evolution of the dynamic data structure, identifying steps in execution at which the data structure appears to get corrupted, and then, using the statement numbers contained in MG++, identify the faulty code. We now provide a formal definition of MG++.

†Throughout the paper, *Execution Point* → *Timestamp* t stands for "at execution point corresponding to timestamp t".

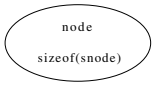
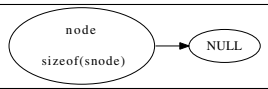
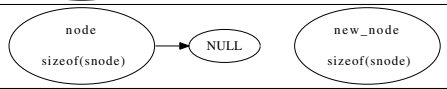
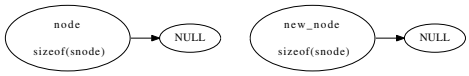
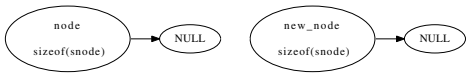
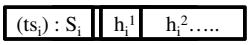
Execution trace	Traditional Memory Graph
11.node = malloc(**sizeof**(snode)); $***$ Execution Point $\rightarrow$ Timestamp 1	
12.node$->$next = NULL; $***$ Execution Point $\rightarrow$ Timestamp 2	
13.new_node=malloc(**sizeof**(snode)); $***$ Execution Point $\rightarrow$ Timestamp 3	
14.new_node$->$next = NULL; $***$ Execution Point $\rightarrow$ Timestamp 4	
15.node$->$next = new_node; $***$ Execution Point $\rightarrow$ Timestamp 5	
19.node$->$next = NULL; $***$ Execution Point $\rightarrow$ Timestamp 6	

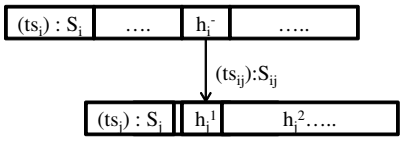
Figure 2. Executed statements and corresponding traditional Memory Graphs.

The MG++ is defined as a tuple (V, E) such that:

- V is a set of nodes such that each node v_i consists of $\langle (ts_i) : S_i; H_i \rangle$, where H_i is a set of heap addresses $\{h_i^1, h_i^2, \ldots\}$ that the node represents, ts_i is the timestamp at which the node was created, and S_i is the source code statement which led to creation of the node v_i. The statement is identified by its location in the source code, i.e., $\langle$file name:line number$\rangle$.



- E is a set of directed edges $H_i.h_i^k \rightarrow H_j.h_j^1$ ($H_i.h_i^k$ represents the heap address that contains a pointer to the heap address $H_j.h_j^1$ where $H_j.h_j^1$ is the first heap address of node v_j); each edge has a label $\langle (ts_{ij}) : S_{ij} \rangle$, where ts_{ij} is the timestamp at which the edge was created and S_{ij} is the source code statement that created the edge. There can be multiple edges corresponding to the same heap address. The edge with the highest timestamp is marked as the current edge.



B. Modeling the Memory Allocator

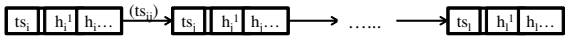
The MG++ representation presented so far does not capture the behavior of the memory allocator itself. Therefore it may be ineffective in cases where understanding program behavior requires allocator information, or when the program has a custom memory allocator for dynamic data structures. In such cases, a memory graph node can no longer be simply defined as an allocated chunk of memory, since the allocator's actions may split a big memory chunk into smaller chunks (during allocation) or join two smaller chunks into a bigger one (following a free).

To capture the history of splitting and merging of memory chunks, we introduce two new kinds of nodes and edges, called *cluster nodes* and *merge edges*, in the MG++ representation. A *cluster node* marks a big consolidated memory chunk formed by joining multiple smaller memory chunks. Representing the node as smaller nodes joined by merge edges enables us to track the history of memory allocation and deallocation operations. This action is captured in the memory graph by joining the two nodes using a *merge edge*. For the purpose of interaction with other nodes, a cluster node is a single node although it internally stores multiple nodes corresponding to earlier smaller chunks.

Figure 3 shows a sample execution trace of a C program that uses an allocator based on Lea's *dlmalloc* allocator [12] along with corresponding timestamps. In addition, we also show the MG++ immediately before the execution and at the end of the execution. *Dlmalloc* maintains the free memory chunks in a doubly-linked list. The oval *head* and *tail* nodes have been shown in the figure for clarity. The MG++ at timestamp 0 shows such a free list with a big memory chunk having starting address tmp1. *Dlmalloc* serves different memory requests by splitting this big chunk into smaller chunks, and stores back the freed memory chunks in the same doubly-linked list. When two contiguous memory chunks are freed, we consolidate them to form a bigger memory chunk. Such a chunk is formed in this example when adjacent memory chunks corresponding to tmp4 and tmp5 are freed (lines 12 and 15) and are consolidated via internal malloc actions. MG++ stores this information using a cluster node – the diamond-shaped node shown in Figure 3. The cluster node has timestamp 26 and points to the two smaller chunks joined by a *merge edge* (edge corresponding to timestamp 26). A cluster node enables the MG++ to retrieve the earlier heap snapshot using the timestamp information.

The formal definitions of the set of *cluster nodes* and *merge edges* follow:

- V' is a set of *cluster nodes* such that each cluster node v'_i is defined as $< (ts_i) : starting\_address; N_i >$ where N_i is an ordered list of nodes $v_i \in V$ joined together by *merge edges*.



- A *merge edge* m connects two nodes $v_i, v_j \in V$ inside a cluster node and has a label $< (ts_{ij}) >$ such that the timestamp marks the merging of the node v_j in the cluster node.

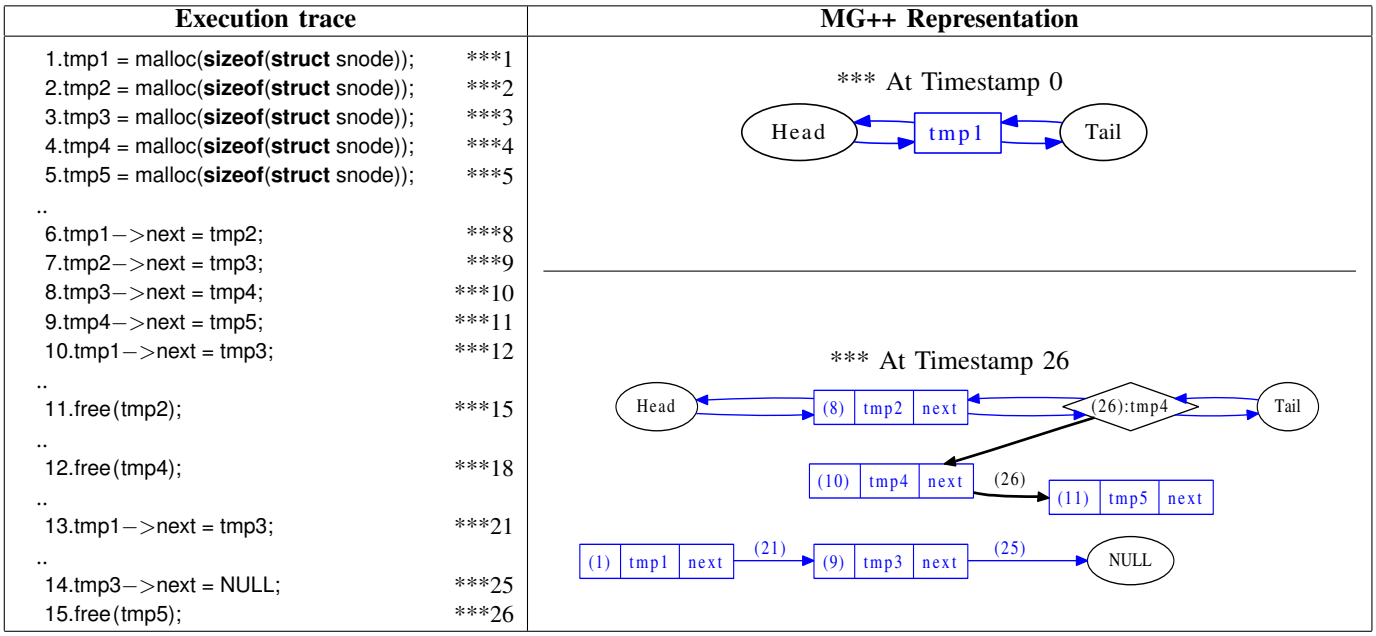


Figure 3. MG++ capturing the actions of the memory allocator.

Algorithm 1 Memory Graph retrieval algorithm

1: /* n_i: node in MG++; n_j: node in memory graph; MG++$_{target}$: MG++ at target timestamp; MG$_{target}$: Memory Graph at target time stamp */
2: **INPUT:** MG++$_{final}$ - the MG++ at final timestamp ts_{final}; target timestamp ts_{target} where $ts_{target} \leq ts_{final}$
3: **function** GRAPH_RETRIEVE()
4: **Step 1:** /* retrieve MG++$_{target}$ */
5: Remove all the nodes created after ts_{target}
6: Remove all the edged created after ts_{target}
7: Join all the nodes split after ts_{target}
8: Separate all the nodes merged after ts_{target}
9: **for** all Heap addresses h_i in MG++$_{target}$ **do**
10: Set the outgoing edge with highest timestamp as the current Edge
11: **end for**
12: **Step 2:** /* retrieve MG$_{target}$ */
13: **for** all nodes n_i in MG++$_{target}$ **do**
14: starting_address(n_j) $\leftarrow$ Head(n_i)
15: Size(n_j) $\leftarrow$ Size(addrList(n_i))
16: MG$_{target}$ $\leftarrow$ $MG_{target} + n_j$
17: **end for**
18: **for** All edges e_i in MG++$_{target}$ **do**
19: add corresponding edges in MG$_{target}$
20: **end for**
21: **return** MG$_{target}$
22: **end function**

- Each node $v_i \in V$ carries a $sourceID$ which marks the parent nodeID corresponding to the node out of which the node v_i is formed after a split.

The definitions of the set of nodes V and the set of edges E are similar to those in Section II.

C. MG++ Rollback and Retrieval

Given the MG++ at timestamp ts_{final}, the memory graph MG for any time stamp $t \leq ts_{final}$ can be efficiently reconstructed by selecting appropriate subsets of nodes and edges. We can reconstruct the step-by-step evolution snapshots of the memory graph enabling us to navigate back and forth over the changes in memory graph during the execution.

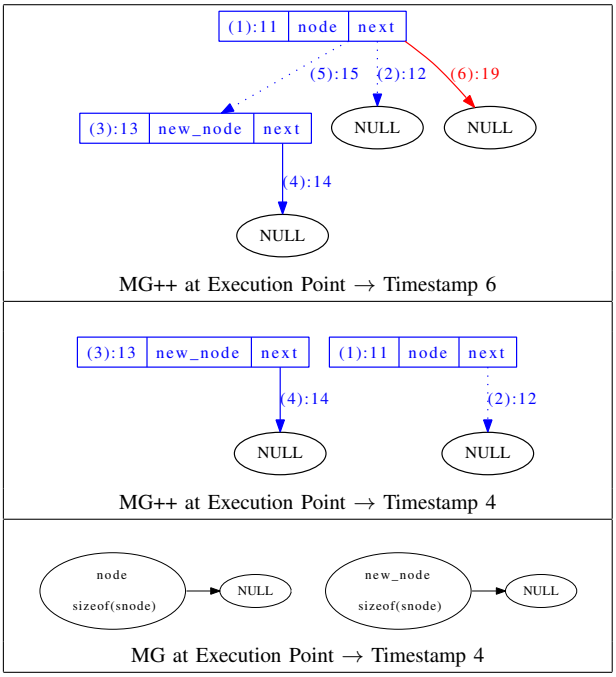


Figure 4. Memory Graph rollback and retrieval.

Algorithm 1 shows how we retrieve MG$_{target}$, the memory graph at target timestamp ts_{target} from MG++$_{final}$, the MG++ corresponding to final timestamp ts_{final}, such that $ts_{target} \leq ts_{final}$. The retrieval takes place in two steps. In the first step, we retrieve MG++$_{target}$, the MG++ at the target timestamp ts_{target}. For this, all the nodes, edges, and merge edges having timestamp greater than ts_{target} are removed from the graph. Also, the addresses of any nodes that were split after the target timestamp are joined together. Removal of nodes may result in isolated data nodes, which are removed. Edges which were overwritten by a store executed after the target timestamp are restored as follows. For each of the heap addresses, the edge with the highest timestamp is set as current edge. Similarly, for each node, merge edge with the highest timestamp is set as

current merge edge. In the second step, MG_{target}, the memory graph at the target timestamp is constructed from $MG{+}{+}_{target}$. This is done by creating nodes and edges in the memory graph corresponding to the nodes and edges in MG++. The starting address of a node is the same as the head of the address list in the corresponding MG++ node. The size of a memory graph node is calculated by joining the sizes of addresses in the address list of the corresponding MG++ node.

Figure 4 illustrates retrieval of the memory graph at time stamp 4 from a MG++ at timestamp 6. In the first step, the node timestamps are examined. Since both nodes have timestamps less then 4, they are retained. The edges with timestamps ≥ 4, i.e., edges with timestamps 5 and 6, are deleted. This leads to an isolated data node which is removed, yielding the MG++ at program point corresponding to timestamp 4. The starting addresses of the two nodes are node and new_node, respectively. The sizes of these nodes are equal to the size of snode, i.e., size of head address + size of next.

III. Portable Memory Graph Construction

We have developed a novel MG++ construction algorithm based on binary instrumentation. By not relying on allocator function information, we develop a portable algorithm that can build the MG++ for programs using different memory allocators, custom allocators, and in-program memory managers. The implementation does not rely on source code or symbol table information. To our knowledge, this is the first attempt to capture the memory usage of a program without relying on source code or symbol table information.

A. Key Observations

The MG++ construction is based on two observations:
(1) *Accesses to fields within a memory node.* Each address inside an allocation site (i.e., node in the memory graph) is always derived from the starting address. The fields can be accessed as an offset from the starting address or the field address is explicitly calculated by adding the offset to the starting address. For example, in Figure 5 the starting address of the memory allocated is stored in register *eax*, the node field *val* in instruction number 6 is accessed as $eax+0$, and in instruction number 9 the node field *next* is accessed as $eax+4$. We observed this behavior in a variety of compilers: GCC, LLVM, Microsoft VC++, and Intel's C compiler. Even when using registers to pass pointers to fields, or to pass the contents of a **char** array from within a structure as a string, an address inside the allocated memory chunk is accessed transitively from the starting address (address C is accessed as an offset from address B, while B is accessed as an offset from starting address A). This observation can be understood in terms of allocator behavior: the allocator returns the starting address of an allocated memory chunk to the program and the program can access the internal addresses of an allocated memory chunk only through the starting address of the memory chunk. *The above observation lets us join all the addresses being derived from the same starting address into a memory graph node, i.e., nodes can be identified without knowledge of memory allocator functions used.*

(2) *Pointers point to the head address of a memory graph node.* When the internal actions of a memory allocator are also being considered, we cannot rely only on the first observation for constructing a memory graph node. The allocator gets the starting address of the memory space from the system (using the brk() or mmap() system calls) and derives all the internal addresses from this starting address. Using only the first observation we will end up with a single node (multiple nodes in case of mmap()) for the whole program.

```
struct item {  int val;                          | 1. main:18 mov dword ptr [esp], 0x8
               item * next; };                   | 2. main:18 call 0x8048350
Main()                                           | 3. main:18 mov dword ptr [esp+0x1c] , eax
{                                                | 4. main:19 mov eax, dword ptr [esp+0x1c]
.....                                            | 5. main:19 mov edx, dword ptr [esp+0x14]
18 curr = (item *)malloc(sizeof(item));          | 6. main:19 mov dword ptr [eax], edx
19 curr->val = i;                                | 7. main:20 mov eax, dword ptr [esp+0x1c]
20 curr->next  = head;                           | 8. main:20 mov edx, dword ptr [esp+0x18]
21 head = curr;                                  | 9. main:20 mov dword ptr [eax+0x4] , edx
.......                                          | 10. main:21 mov eax, dword ptr [esp+0x1c]
}                                                | 11. main:21 mov dword ptr [esp+0x18], eax
```

Figure 5. Memory access example.

Therefore we form memory graph nodes using the observation that all pointers point to the head address of a memory graph node. Any address being pointed to becomes the starting address of a new memory graph node.

In prior works memory graph nodes correspond to allocated memory chunks and the construction techniques rely on the knowledge of calls to the allocator function. Moreover, if the program uses a custom memory allocator or it manages memory internally, then traditional memory graph representations fail to provide any useful information because they will simply show a single memory graph node.

B. Construction Algorithm

Given a program execution trace which captures the operations on heap references during the program execution, our algorithm builds the MG++ by grouping together heap references to form a memory graph node. The technique for identifying heap references is described in the implementation description (Section IV). Whenever a heap address-handling instruction is encountered, it is analyzed for its effects on the graph. The *timestamp* attached to each node and edge marks the order of their creation during the execution of the program. For example, a node will always have a lower timestamp than an edge pointing to it because the node was formed earlier in program execution and the edge was added later. The timestamp is initialized at the start of memory graph construction and is incremented with each change to the graph. Note that we do not capture information about node deallocation, as the MG++ maintains information about nodes even after their deallocation (complete evolution history) so handling of deallocation does not require any special treatment. Algorithm 2 summarizes memory graph construction, changing the memory graph according to the instructions being executed. For a heap address $haddr$, in Algorithm 2, $node(haddr)$ denotes the node corresponding to $haddr$. Given an instruction i, the following four cases arise:

Case 1) If the current instruction i operates only on *one heap address* which has never been encountered before the current execution point, then a new node is created (call to Create_Node, line 8). In Create_Node, the address is added to the address list of the new node. The node is also given a new timestamp which marks its creation.

Case 2) If instruction i operates on *heap address + offset*, the algorithm proceeds as follows. If both the base heap

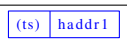
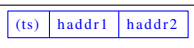
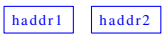
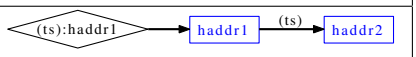
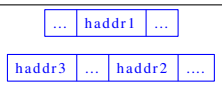
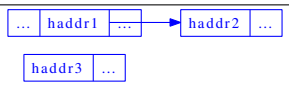
Operation	Instruction	MG++ Before	MG++ After
Join_Addresses($haddr_1, haddr_2$)	$haddr1$ + offset (=$haddr2$)		
Merge_Nodes($haddr_1$, $haddr_2$)	$haddr1$ + offset (=$haddr2$)		
Split_Node($haddr_2$)	$haddr_1 \leftarrow haddr_2$		

Figure 6. MG++ construction operations.

Algorithm 2 Memory Graph construction

```
1: /* haddr: heap address; node(haddr): node corresponding to
   haddr; data: non heap address value */
2: INPUT: Execution Trace
3: OUTPUT: Memory Graph (MG)
4: function GRAPH_CONSTRUCTION()
5:     switch instruction i :
6:         case i has haddr_1
7:             if haddr_1 ∉ MG then
8:                 Create_Node(haddr_1)
9:             end if
10:        case i has haddr_1 + offset(= haddr_2)
11:            if haddr_2 ∉ MG then
12:                Join_Addresses(haddr_1, haddr_2)
13:            else if node(haddr_2) ≠ node(haddr_1) then
14:                Merge_Nodes(haddr_1, haddr_2)
15:            end if
16:        case i is haddr_1 ← data
17:            Create_Edge(haddr_1, data)
18:        case i is haddr_1 ← haddr_2
19:            if haddr_2 ∉ MG then
20:                Create_Node(haddr_2)
21:            else if haddr_2 is not head of node then
22:                Split_Node(haddr_2)
23:            end if
24:            Create_Edge(haddr_1, haddr_2)
25: end function
```

address and offset heap address have not been encountered earlier, then a new node is created and both addresses are added to the address list of the node (omitted from the algorithm for simplicity).

– If the base heap address has been encountered earlier (i.e., it corresponds to a node) and the offset heap address has not been encountered until that execution point, then the offset heap address is added to the address list of the node corresponding to the base address (Figure 6, row 1).

– If both the base heap address and the offset address have already been encountered before, and correspond to different nodes, then the nodes are merged together using the merge edge (call to Merge_Nodes, line 14). This leads to the creation of a cluster node (Figure 6, row 2). An example of this situation is the memory allocator consolidating two adjacent free memory chunks into one bigger chunk.

Case 3) If the instruction i is a *memory write*, A←B where A is a *heap address* and B is a *data value* (e.g., tmp3−>next = NULL in Figure 3) then a data edge from the address is created (call to Create_Edge, line 17). The edge is assigned a new timestamp that marks its creation.

Case 4) If instruction i is a *memory write*, A←B where both A and B are *heap addresses* (e.g., tmp1−>next = tmp2 in Figure 3), an edge is created, from the address written to, to the node corresponding to the address value written (call to Create_Edge, line 24). If the address value written is not the starting address of a memory graph node then the node is split such that the address value written forms the head of the newly created node (Figure 6, row 3). A common example of such a situation is when memory allocator allocates a smaller chunk out of a bigger free memory chunk. If the address value written is a new address, a new node is created as in step 1 (call to Create_Node, line 20).

Node merging and splitting operations come into play only when the internal actions of the memory allocator are included in the analysis. If such internal actions are not considered in the analysis then accessing an address from a node as an offset from the address of a different node (line 13: $haddr_1 + offset(= haddr_2)$ and node($haddr_2$) ≠ node($haddr_1$)) is considered to be suspicious program behavior, e.g., a potential heap buffer overflow.

An Example. Figure 7 illustrates our approach when run on the example given in Figure 3. The first column shows the sample C source code statements executed at each step during construction of a linked list. The second column shows the memory graph formed once the source code in the same row is executed. When the execution starts on line 1, a new heap address tmp1 is encountered because of the call to malloc. A new node is created with initial time stamp of 1 (call to Create_Node, line 8 in Algorithm 2). Executing statements up to line 5, heap addresses tmp2, tmp3, tmp4 and tmp5 are encountered inside malloc. All these memory addresses are derived as an offset of the initial address inside malloc so they are added inside the same MG++ node (call to Join_Addresses, line 12 in Algorithm 2). When the next field is written with value tmp2, the original node is *split* into two nodes (Split_Node, line 22 in Algorithm 2) to form tmp2 as the starting address of the new node. Similarly, other nodes are created due to memory writes in the lines 7, 8 and 9. The free operation on tmp2 leads to the node's addition to the doubly-linked list of free chunks internally maintained by malloc. We have represented the head and tail of the doubly linked list of free chunks with oval nodes (not present in the C code) for clarity. Similarly, free(tmp4) adds the node to the free list. When tmp5 which is adjacent to tmp4 is freed, malloc consolidates the two chunks. A *merge edge* is created to join the nodes corresponding to tmp4 and tmp5 and a *cluster node* is added to the MG++ (Merge_Nodes, line 14 in Algorithm 2). Note that the algorithm performs all of the above actions without knowledge of the memory allocator.

IV. Implementation and Evaluation

Implementation overview: Our system takes a program binary as input. The binary is instrumented to capture the heap references and operations on them. The instrumented program is executed to generate a trace, containing the information about the heap references, operations on heap references and timing information. If debugging information is present in the binary then information about mapping to source code is

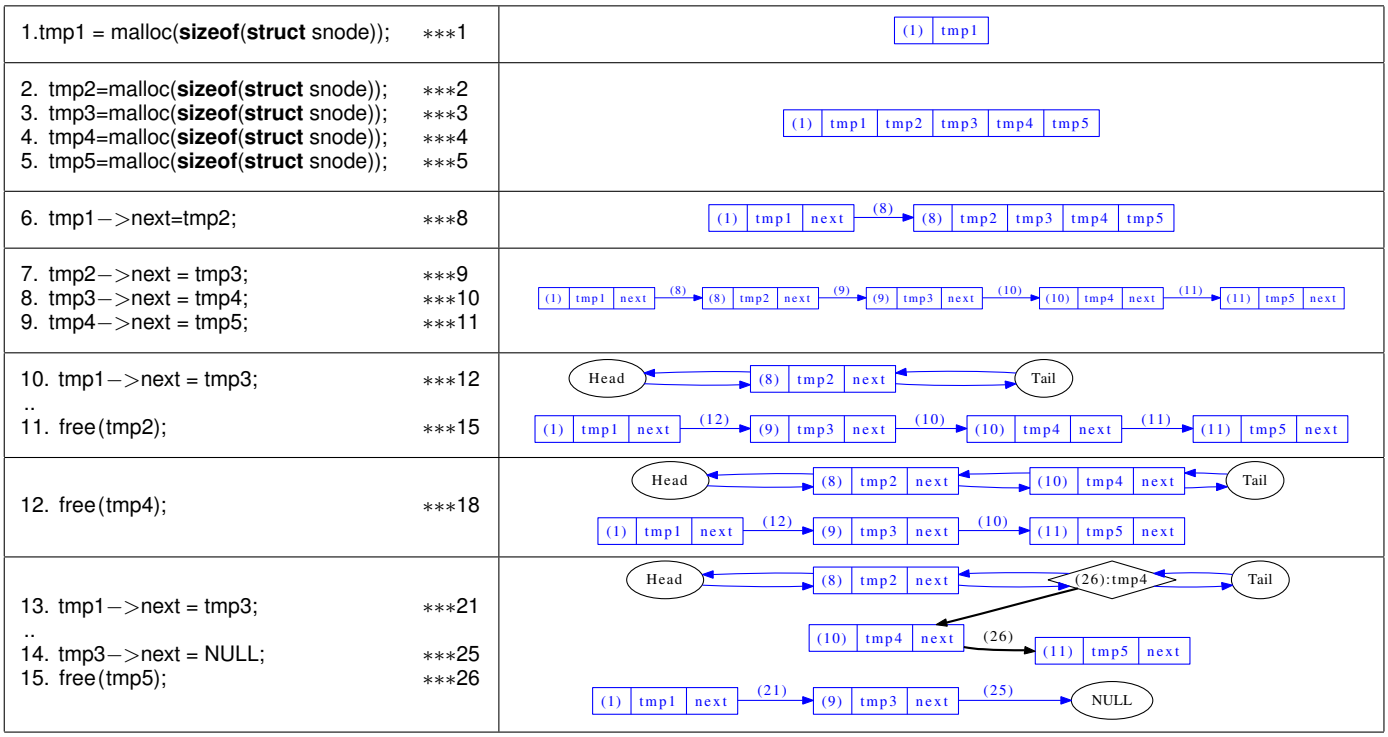


Figure 7. An illustration of MG++ construction.

also generated. The trace is analyzed off-line to generate the memory graph.

Our binary instrumentation is based on PIN-2.6 [10]. We only instrument loads, stores and a set of heap reference-handling instructions (ADD, LEA, SUB, INC, XCHG). Other instructions may have to be instrumented depending on the compiler. The instrumented binary is then executed, generating the execution trace. At runtime, PIN determines the heap address range for the program by monitoring the system calls (brk() and mmap()) as well as the address space of the program, and outputs the valid heap address range and those instructions which handle heap references. These optimizations help reduce the size of our execution trace. The execution trace contains the timestamp information and the statement identifier for each of the executed instructions along with the address references involved in the instruction. An automated analysis of the execution trace constructs the memory graph. The valid heap address range is used to identify heap references in the trace.

Experimental setup: All measurements were performed on an Intel Core 2 6700 @ 2.66GHz with 4 GB RAM, running Linux kernel version 2.6.32. The experiments indicate that our memory graph representation and construction technique are efficient and effective in real-world scenarios.

Benchmarks: We measured the cost of using MG++ in terms of space and execution time to show that our approach is practical. A summary of benchmarks used in our evaluation is shown in Table I. Our benchmarks are divided into three categories. First, widely-used programs: the **ls** GNU core utility, **Tidy** HTML checking&cleanup, and the GNU **bison** parser generator. Second, programs that perform internal memory management: the commonly-used **CPython** and **Perl** interpreters. In the third category, we have graph applications (graph coloring, independent set, and shortest path) with small real-world graphs as input, which we believe reflect common debugging scenarios. The benchmarks in the last category are read-intensive, "build-then-traverse data structure" style programs.

A. *Cost of Constructing MG++*

Space Required: Table II's columns 2 and 3 show the space required for MG++, without and with allocator analysis, respectively. In order to show the space-efficient nature of MG++, we compared the space requirement of MG++ with that of a snapshot approach for capturing the data structure evolution history. The snapshot approach stores snapshots of memory graphs after each change using representations similar to the ones mentioned in [1], [2], [7]. The memory required to capture the sequences snapshots of memory graph is approximated by assuming a snapshot of the average number of nodes stored upon each change to the memory graph (Table II, column 4). The size of each node is assumed to be 8 bytes (a single field). The space required by MG++, excluding (including) the allocator ranges from 3 MB to 138 MB (7 MB to 141 MB, respectively). This is two orders of magnitude less than that of capturing sequences of memory graphs using prior representations. In fact, this data shows that capturing sequences of memory graphs is impractical, while MG++ is highly space-efficient. The memory required for MG++ depends on the number of objects allocated (MG++ nodes) and manipulated (leading to creation of MG++ edges).

Program Execution Time: Table II's columns 8 and 9 show the execution time for running the instrumented code for our technique, without and with allocator analysis, respectively. Table II's column 5 (Original) shows the cost of running the original program; column 6 (Null PIN) shows the cost of running the program under PIN without any instrumentation; column 7 shows the cost of collecting trace for memory graph using allocator information [1], [2], [7], i.e., only calls to allocator functions and memory writes are instrumented. Although program execution time increases significantly due

Table I. OVERVIEW OF BENCHMARKS.

Program Description	Data Structure	Input	Details
ls (GNU core utilities [13])	Array & Linked list	coreutils-8.0 source dir	ls -R coreutils-8.0/
Tidy (HTML check & clean [14])	Binary tree variant	Tidy Project Page.html	Version 1.46
Bison (GNU parser generator [15])	Array & Linked list	ANSI C grammar	Version 2.4.1
CPython [16]	Linked lists & Trees	Page rank program	Version 2.7.8
Perl [17]	Arrays & Linked lists	File search program	Version 5.20.0
Graph Coloring	Linked list of arrays	NetScience [18], [19]	1,589 nodes; 2,742 edges
Independent Set	Linked list	YeastL (protein interaction network) [18], [20]	2,361 nodes; 7,182 edges
Shortest Path	Shard	Wiki-vote (Wikipedia network) [21]	7,115 nodes; 103,689 edges

Table II. TIME AND MEMORY COSTS OF CAPTURING MEMORY GRAPHS.

1	2	3	4	5	6	7	8	9	10
Program Description	**MG++ Representation Space Required (MB)**			**MG++ Construction Method**					
				Program Execution Time (seconds)					**Slowdown** (col 8/ col 7)
	MG++		**Snapshot Approach**	**Original**	**Null PIN**	**Instrumented**			
	w/o allocator	with allocator				Allocator approach	MG++ w/o allocator	MG++ with allocator	
ls	3.8	6.9	≈ 1,895	0.02	1.25	1.97	2.14	5.41	1.08
Tidy	22.4	32.1	≈ 2,484	0.01	1.46	6.46	8.90	9.63	1.37
Bison	2.5	32.3	≈ 283	0.06	2.04	2.23	2.68	9.82	1.20
CPython	137.8	140.5	≈110,242	0.09	7.84	24.60	30.74	32.85	1.25
Perl	48.1	50.2	≈ 65,652	1.11	11.73	26.09	26.41	32.52	1.01
Graph Coloring	82.3	83.5	≈ 311	0.05	0.68	8.69	37.33	38.08	4.29
Independent Set	6.3	10.2	≈ 451	0.05	0.53	13.44	47.27	50.79	3.51
Shortest Path	47.9	84.7	≈ 50,827	0.06	0.57	8.25	13.68	45.84	1.68

to instrumentation, it is a necessary cost of dynamic analysis. Column 10 shows the relative time overhead for our approach in comparison to the allocator-based approach. The average slowdown for the read-intensive benchmarks (graph-coloring, independent-set, shortest-path) is 3.16x while for the other benchmarks the average slowdown is 1.18x. The slowdown is higher in case of read-intensive programs because our MG++ construction algorithm has to monitor all the heap reference-handling instructions, including reads. The total average slowdown is 1.7x. These results show that the MG++ construction algorithm gives a feasible method of memory graph construction for cases where allocator information is not available. Instrumenting the memory allocator increases the execution times by 2x to 4x in comparison to when only application code is instrumented.

B. Fault Location using MG++

Fault location for data structure errors is a novel application of MG++. Data structures have structural properties that must hold during execution. However, bugs can cause these properties to be violated and the violations can be detected by analyzing the memory graphs. Prior approaches have used this idea for detecting corrupted data structures [22] and data structure repair [23]–[25]. The evolution history of data structures and the source code mapping in MG++ make it possible to detect violations and locate the faulty code responsible for causing the violations.

We show the usefulness of MG++ in fault location using *Mozilla BugID 588187*. Due to this bug, Mozilla crashes while traversing a linked list. The linked list in the program has a simple constraint: each entry→next field must point to another linked list node or NULL. Figure 8 shows an excerpt of the traversal code. If entry→next contains a value which is neither NULL nor a heap address then the execution will enter the loop (line 1) and crash at line 2. We detect this bug by matching the given constraint over the MG++ of the corrupted linked list. Upon detecting corruption, the MG++ is rolled back to the point where the MG++ is consistent with respect to the constraint. Using the source code information embedded in MG++, the statement which introduces the corruption in the linked list is identified as the faulty statement.

```
1   while (entry) {
2     if (entry→Key == key) { <−−− crash here
3       return entry→Data;
4     }
5     entry = entry→Next;
6   }
```

Figure 8. Linked list traversal.

The above bug was handled without making the memory allocator as part of the analysis. There are cases where internal actions of the memory allocator are required to be a part of analysis and therefore MG++ can enable fault location. The Glibc memory allocator keeps track of free chunks via a doubly-linked list. Numerous real-world bugs *(GNOME BugIDs 697397, 449433, 318401; KDE BugIDs 119108, 281770, 224877, 237913)* are associated with the corruption of the free chunks list. Figure 9 shows a simple C program which causes Glibc corruption (**** glibc detected *** corrupted double-linked list*) later in the program. The programmer accidentally passes the address instead of the value of tmp→target in line 4. This overwrites the free chunks metadata, in turn corrupting the doubly-linked list. The program will crash later in the execution when malloc tries to access the elements of the doubly-linked list. We detect the violation of the doubly-linked list invariant by matching the invariant *if element **e** points to element **e'** then **e'** should point back to **e*** over MG++. Upon detecting corruption, the MG++ is rolled

back and the statement responsible for corruption is identified.

We have developed a fault location framework based on the MG++ representation. The framework consists of a language to write data structure constraints, a tool to automatically generate a constraint matcher and a fault locator. Invariants are matched based on the type of the MG++ node (this requires debugging information). Detailed discussion about the fault location framework is out of scope of this paper.

```
1 struct node * tmp;
2 tmp = (struct node *)malloc(sizeof(struct node));
3 tmp→target = malloc(value_size);
4 memcpy(&tmp→target, value, value_size);
```

Figure 9. Sample code that corrupts Glibc's free chunks list.

C. Detecting Buffer Overflow Attacks using MG++

In this experiment we tested the use of MG++ to detect heap buffer overflows on the basis of observation 1 in Section III-A. Our assumption was that if an address of a memory node X is being accessed as an offset from an address inside memory graph node Y then it is a heap buffer overflow. We tested our technique against 12 attack benchmarks from the RIPE buffer overflow testbed [11]. For this experiment we did not include allocator internals in the analysis. These benchmarks exercise various types of heap buffer overflows including return address, function pointers, and vulnerable structs, and have been commonly used in prior work to test the effectiveness of attack detection systems. As shown in Table III, our technique was able to detect heap buffer overflow for all vulnerabilities, except for vulnerable structs. In the case of vulnerable structs, an attack-prone buffer resides along with a target function pointer inside the same memory graph node. There is no buffer overflow across memory graph nodes in this case and thus our technique fails to capture it. Valgrind's memcheck [26] can also detect other attacks except for vulnerable structs; it uses a valid bit for each byte of allocated memory and reports an error in case unallocated memory is written. Memcheck cannot be applied in the absence of allocator information while our technique will be able to detect buffer overflow without allocator information.

Table III. HEAP BUFFER OVERFLOW DETECTION RESULTS.

Vulnerabilities	# of benchmarks	Exception raised
Return Address	1	Yes
Old base Pointer	2	Yes
Function Pointers	7	Yes
Vulnerable Structs	2	No

V. LIMITATIONS

- Our algorithm identifies a heap address by checking if the value falls in the valid heap address range. If a non-heap address operand with value in the valid heap address range is encountered, our approach creates a one-field node in the memory graph for it. Such a node is unnecessary and can be removed using a post-construction analysis.
- The size of a memory node constructed by our algorithm is different from the allocated size in cases where trailing fields of an allocated node are never read or written.

VI. RELATED WORK

A. Memory Graphs

Prior heap analyses fall into two main categories: *static* and *dynamic*. Static techniques include shape analysis [27], [28] and other techniques that are aimed at deriving a compile-time approximation of the heap structure [29]. However, static analysis does not give a clear picture of the runtime heap activity in a particular execution. Dynamic analysis techniques like ours collect data from program runs and analyze it either online or offline. Several prior dynamic techniques are aimed at visualizing and navigating a single snapshot or a series of snapshots of the heap [5], [30]–[32]. These works are orthogonal to our work and can make use of MG++ internally for memory efficiency. DDD [33] and Zimmermann & Zeller [30] use symbol table information for constructing memory graph while Raman & August [7] use allocator function information for memory graph construction. Work on visualizing memory management behavior [34], [35] has focused on analyzing memory allocator's behavior and performance. Unlike our work, these approaches are not aimed at capturing the memory graph for program understanding or debugging applications. The technique presented in [36] is specifically aimed at detecting recursive data structures and dynamic degree invariants. Our approach captures all pointer mutations of the heap and the resulting MG++ can be used for detecting data structures and their links. Finally, MG++ representation is similar to that of persistent data structures [37].

B. Applications of Memory Graphs

Memory graphs are at the heart of a number of different applications aimed at program understanding and debugging.

Memory debugging and general-purpose debugging. A number of approaches have aimed at displaying memory uses of the program using memory graphs thereby enabling programmers to detect memory leaks or identify memory corruption patterns [3]–[5]. In Zeller's work [6], program state is captured as a memory graph and state differences between passing and failing runs are used to isolate cause-effect chains for a program failure.

Program Understanding. Myers and Duke [1] extract design abstractions from memory graphs to provide users an intuitive representation. Malik [38] analyzes spectra of heap graphs to extract dynamic invariants.

Data Structure Extraction. Jung and Clark [2] apply invariant detection on a memory graph to identify the data structure used by the program. Lin et al. [39] extract type information from program binary and provide an abstract representation of the variables used in the program. They rely on standard library function calls for the typing information.

All of the above techniques rely on memory graphs and therefore can benefit from our improved representation as it captures program behavior with greater precision, including the behavior of the memory allocator. Moreover, the compact nature of our representation will allow the above techniques to scale to longer program runs.

VII. CONCLUSION

We have presented MG++, a new memory graph representation that captures the evolution history of heap data structures and carries a direct mapping from runtime data structures to the relevant source code. MG++ enables analyses that must consider internal actions of the allocator. We have also presented a MG++ construction technique which does not rely on source code or symbol table information. This construction

technique is useful in scenarios where allocator information is unavailable and none of the allocator-based approaches can be applied. The experiments indicate that the representation and construction technique are practical, efficient, and effective in real-world applications.

ACKNOWLEDGMENTS:

This work was supported in part by NSF grants CCF-0963996 and CCF-1149632.

REFERENCES

[1] C. Myers and D. Duke, "A map of the heap: revealing design abstractions in runtime structures," in *Proceedings of the 5th international symposium on Software visualization*, ser. SOFTVIS '10, 2010, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/1879211.1879223

[2] C. Jung and N. Clark, "Ddt: design and evaluation of a dynamic program analysis for optimizing data structure usage," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, pp. 56–66. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669122

[3] S. P. Reiss, "Visualizing the java heap to detect memory problems," in *Proceedings of the IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2009, pp. 73–80.

[4] W. D. Pauw and G. Sevitsky, "Visualizing reference patterns for solving memory leaks in java," in *Proceedings of the 13th European Conference on Object-Oriented Programming*. Springer-Verlag, 1999, pp. 116–134. [Online]. Available: http://dl.acm.org/citation.cfm?id=646156.679713

[5] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer, "Heapviz: interactive heap visualization for program understanding and debugging," in *SoftVis'10*, 2010, pp. 53–62. [Online]. Available: http://doi.acm.org/10.1145/1879211.1879222

[6] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2002, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/587051.587053

[7] E. Raman and D. I. August, "Recursive data structure profiling," in *Proceedings of the 2005 workshop on Memory system performance*, ser. MSP '05, 2005, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/1111583.1111585

[8] X. Xiao, J. Zhou, and C. Zhang, "Tracking data structures for postmortem analysis (nier track)," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 896–899. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985938

[9] J. Ellson, E. Gansner, L. Koutsofios, S. North, G. Woodhull, S. Description, and L. Technologies, "Graphviz open source graph drawing tools," in *Lecture Notes in Computer Science*. Springer-Verlag, 2001, pp. 483–484.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[11] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime intrusion prevention evaluator," in *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*, 2011.

[12] D. Lea. (1987) A memory allocator. http://g.oswego.edu/dl/html/malloc.html.

[13] Gnu core utilities. http://www.gnu.org/software/coreutils/.

[14] D. Raggett. (2009) Html tidy program. http://tidy.sourceforge.net/.

[15] Bison-gnu parser generator. http://www.gnu.org/software/bison/.

[16] Python interpreter. https://www.python.org/.

[17] Perl programming language. http://www.perl.org/.

[18] V. Batagelj and A. Mrvar. (2006) Pajek datasets. http://vlado.fmf.uni-lj.si/pub/networks/data/.

[19] M. E. J. Newman, "Finding community structure in networks using the eigenvectors of matrices," *pre*, vol. 74, no. 3, p. 036104, Sep. 2006.

[20] S. Sun, L. Ling, N. Zhang, G. Li, and R. Chen, "Topological structure analysis of the protein-protein interaction network in budding yeast," *Nucleic Acids Research*, vol. 31, no. 9, 2003.

[21] Stanford large network dataset collection. http://snap.stanford.edu/data/index.html.

[22] B. Demsky, C. Cadar, D. Roy, and M. Rinard, "Efficient specification-assisted error localization," in *WODA'04*.

[23] B. Demsky and M. Rinard, "Data structure repair using goal-directed reasoning," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, May 2005, pp. 176–185.

[24] ——, "Automatic detection and repair of errors in data structures," in *OOPSLA'03*, pp. 78–95. [Online]. Available: http://doi.acm.org/10.1145/949305.949314

[25] M. Malik, J. Siddiqi, and S. Khurshid, "Constraint-based program debugging using data structure repair," ser. ICST, 2011, pp. 190–199.

[26] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07, 2007, pp. 89–100. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250746

[27] R. Ghiya and L. J. Hendren, "Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '96, 1996, pp. 1–15. [Online]. Available: http://doi.acm.org/10.1145/237721.237724

[28] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '99, 1999, pp. 105–118. [Online]. Available: http://doi.acm.org/10.1145/292540.292552

[29] M. Marron, D. Kapur, and M. Hermenegildo, "Identification of logically related heap regions," in *Proceedings of the International Symposium on Memory Management*, ser. ISMM '09, 2009, pp. 89–98.

[30] T. Zimmermann and A. Zeller, "Visualizing memory graphs," in *Revised Lectures on Software Visualization, International Seminar*. London, UK, UK: Springer-Verlag, 2002, pp. 191–204. [Online]. Available: http://dl.acm.org/citation.cfm?id=647382.724787

[31] J. Sundararaman and G. Back, "Hdpv: Interactive, faithful, in-vivo runtime state visualization for c/c++ and java," in *Proceedings of the 4th ACM Symposium on Software Visualization*, 2008, pp. 47–56. [Online]. Available: http://doi.acm.org/10.1145/1409720.1409729

[32] S. Pheng and C. Verbrugge, "Dynamic data structure analysis for java programs," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, 2006, pp. 191–201.

[33] A. Zeller and D. Lütkehaus, "Ddd—a free graphical front-end for unix debuggers," *SIGPLAN Not.*, vol. 31, no. 1, pp. 22–27, Jan. 1996. [Online]. Available: http://doi.acm.org/10.1145/249094.249108

[34] A. M. Cheadle, A. J. Field, J. W. Ayres, N. Dunn, R. A. Hayden, and J. Nystrom-Persson, "Visualising dynamic memory allocators," in *ISMM '06*, pp. 115–125. [Online]. Available: http://doi.acm.org/10.1145/1133956.1133972

[35] T. Printezis and R. Jones, "Gcspy: An adaptable heap visualisation framework," in *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '02, 2002, pp. 343–358. [Online]. Available: http://doi.acm.org/10.1145/582419.582451

[36] M. Jump and K. S. McKinley, "Dynamic shape analysis via degree metrics," in *Proceedings of the International Symposium on Memory Management*, ser. ISMM '09, 2009, pp. 119–128. [Online]. Available: http://doi.acm.org/10.1145/1542431.1542449

[37] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *J. Comput. Syst. Sci.*, vol. 38, no. 1, pp. 86–124, Feb. 1989. [Online]. Available: http://dx.doi.org/10.1016/0022-0000(89)90034-2

[38] M. Z. Malik, "Dynamic shape analysis of program heap using graph spectra (nier track)," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 952–955. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985956

[39] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *NDSS*, 2010.