Seminar Code Generation
Winter Term 2015/2016

Automatic detection of dynamic data structures for C/C++ binaries

Mohammad Zeeshan
Technische Universität München

22.01.2016

Abstract

Reversing data structures from C/C++ binaries depends on low-level programming constructs such as structs or variables. Forensics analysis and reverse engineering is very hard if the detailed information about the pointer structures is not known. In this paper we have proposed a tool called "MemPick" to cater this need. MemPick detects and classifies high-level data structures from binaries. It keeps track of the links between the memory objects that evolve during program execution.Based on the shape of the memory objects and these links, it identifies and classifies many commonly used data structures. These data structures include different types of linked lists, many types of trees and graphs. For evalution of MemPick, four different file system implementations, 10 real world applications and 16 libraries are used. Based on the results obtained, MemPick is categorized as one of the high accuracy data structure identification tools.

Keywords: Data structures, shape analysis, reverse engineering, dynamic binaries analysis

1 Introduction

Reverse engineering is the process of taking binary code of an application and producing the source code from it. One of the key factors in reverse engineering is identification of data structures used in the application as most of the time the program code acts on these data structures to perform specific tasks. Reverse engineering plays a vital role in malware identification and binary optimization. For instance a basic optimizer may keep nodes of a linked list in small and consecutive pages in memory to reduce seek time and latency of the read and writes from the disk into main memory. Furthermore, an optimizer which is aggressive variant of the basic one, may replace the data structure entirely with a more efficient alternate data representation.
This paper summarizes the contents of [3] with its examples and definitions [1] A tool named MemPick created by [3] is discussed. MemPick finds and identifies data structure using C/C++ binaries. Shape of data structure is analysed to identify type of the data structure. For instance, interconnected heap buffers may represent linked list or a binary tree. Currently this tool identifies single and

[1]Results presented in this paper where we test MemPick on different software programs have been taken from runs done by [3].

double linked list, binary and n-ary trees, various balanced trees, sentinel nodes, threaded trees and graphs. Overlapping data structures are also identified.MemPick only focuses on dynamic changing of shape for identifying type of data structures. Functions or characterization of content is not identified in MemPick.
As MemPick is based on dynamic analysis technique hence it only detects data structures based on code coverage of the profiling runs. Other analysis technique is static analysis, in which reasoning is done without execution. This type of analysis is used for components that are difficult to execute. Static analysis is not suitable for C/C++/Assembly because of pointer aliasing and indirect control flow changes. Dynamic analysis reasons about specific execution path resulting in accuracy and precision of results.

2 Outline

Architecture of the application is discussed in detail in section 3. Section 4 explains the low-level manipulation of memory graphs. Section 5 shows the high-level representation of data structures. The intricacies of data structure classification are explained in section 6 which is followed by section 7 elaborating height balanced trees. Section 8 explains the information provided to the user. Extensive evaluation of the tool is performed in section 9. Computational complexity and scalability of the tool is discussed in section 10. Section 11 explains the limitations and future extensions of MemPick. In the last two sections related projects are discussed and conclusion is mentioned.

3 MemPick

Figure 2 is used as running example for understanding approach of MemPick. It also includes overlapping data structures: a child tree, a parent tree and a singly linked list for tree traversal. MemPick works in three stages. First, application binary is executed and sample executions are recorded.

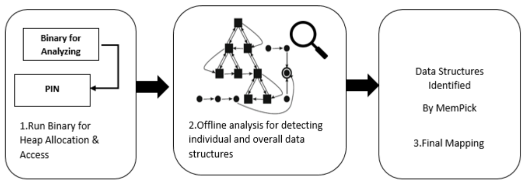

Binary for Analyzing
PIN
1.Run Binary for Heap Allocation & Access
2.Offline analysis for detecting individual and overall data structures
Data Structures Identified
By MemPick
3.Final Mapping

Figure 1: Highlevel Overview of MemPick.

These executions are fed to analysis engine which identifies data structures. Finally results are combined and presented. High-level overview of MemPick is shown in figure 1.
Intel's PIN binary instrumentation framework provides API that is used for tracking and recording the execution of application [10]. MemPick uses PIN for storing address of all buffers allocated on heap and memory allocation functions. In offline analysis phase links between heap buffers are analysed for identifying data structures. This phase consist of following four steps.

1. Heap buffers and associated links are arranged in a memory graph which reflects how connections are evolved.
2. After analysis graph is split into objects according to their types as shown in figure 2a and 2b.
3. MemPick analyses the links in each partition to identify overlapping structures and then categorized each partition
4. In the final step, MemPick tries to identify different types of tree like AVL and red-black tree.

Each step is discussed in detail in respective sections below.

4 Memory Graphs

Memory graph only identifies links that exist between heap objects. For identifying the type of data structure in a given graph, MemPick analyses connection between nodes for identifying different logical types. Two different heap objects share a

logical type if one object can be replaced by other or their low level C/C++ type is same.
Graphs are built in MemPick like RDS [7] by inserting new node in graph when a heap buffer is allocated. Edges are added or deleted between nodes based on instructions that add or delete pointers between objects respectively. MemPick then tags graphs with logical type by first eliminating instructions that are type agnostic. Same logical type objects are identified by checking if both objects can be used as operand in the same instruction.
Type reference algorithm is used for identifying logical type in MemPick. Every instruction is assigned with a pair of unique tags, including tags for both its source and destination operands. A unique tag is assigned to a heap object when it acts as an operand in a given instruction thus ensuring all heap objects, belonging to same instruction share a unique tag. If another tag is already associated with a heap object, then they have same logical type and both tags are merged as one. All heap objects are checked for consistency resulting in quick merging of tags and categorization of logical types used in the program. For avoiding ambiguous typing, MemPick assigns Type Aware for instructions that stores pointer to heap buffer for a memory location at specific constant offset in another heap buffer. Instructions that store non-pointer values or pointers to different offsets are ignored.

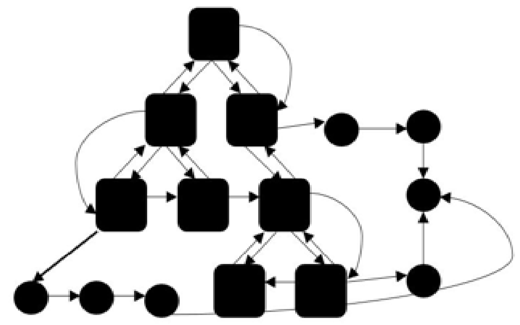
5 Memory Graph for Identification of Individual Data Structures

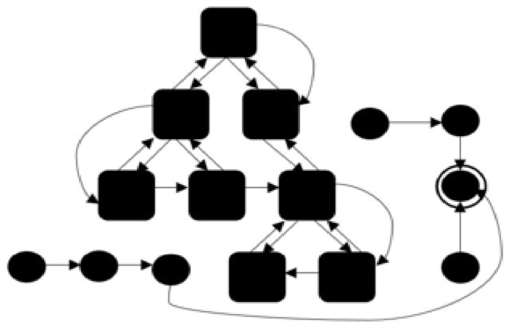
MemPick divides the memory graph into subgraphs, each comprising of individual data structures. Using these data structures we will perform shape analysis. Firstly, MemPick eliminates from the graph those links that connect nodes of different types. In the example from Figure 2, one of the partitions illustrates the growth of the tree.
The specific properties of a data structure do not necessarily remain unaltered all over the execution. For example, if an application uses a red-black tree, the tree is often unstable just before a rotate operation. However when the application does not modify the tree, its shape maintains the expected features. Therefore, MemPick performs its shape analysis only on dormant data structure.
MemPick defines dormant periods as the number of instructions executed. Specifically, we measure the interval of the break between alterations of the data structures in cycles and then decide on the longest n percentage as the inactive phase. To make sure that we never pick active phase, we need to remain adequately choosy. MemPick uses dynamic gap size to familiarize itself to the characteristics of each binary and the data structure.
The method defined in MemPick benefits from two core properties, 1) it assures a lower bound of inactive phase for every data structure, 2) it provides maximum robustness. Since sentinel nodes distort the shape of data structure, we detect and separate them before passing even snapshots of each data structure to the shape analyser.
To locate sentinel nodes in a partition of the memory graph, MemPick searches for outliers in number of incoming edges for each node. While this strategy produces good results for lists, trees, and graphs, it might break some highly customized data structures. Finally, for each partition of the memory graph, we get hold of its snapshots in the inactive interval, and use them in the following stage of MemPicks algorithm discussed in the next section.

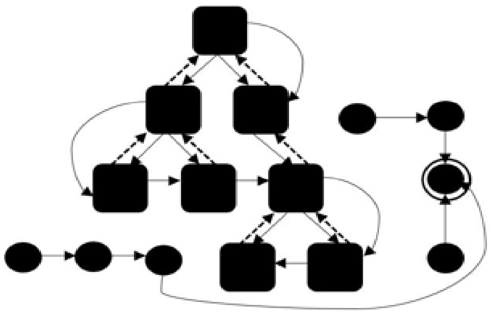
6 Shape Detection

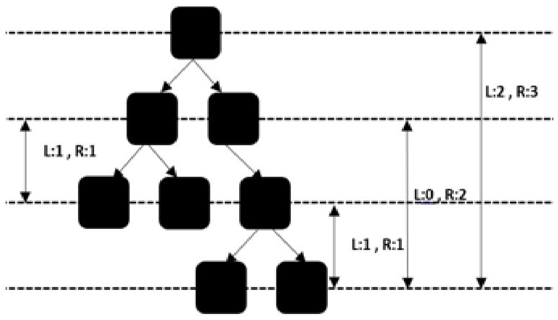
Quiescent period represents stable state of data structures. MemPick focuses on quiescent periods for validating any shape hypothesis. If it represents globally valid property of data structure then shape hypothesis should be true in every snapshot of graph-partition. Overlapping structures are identified first to avoid blurring of actual shape. More advance analysis is also performed for supporting data structures like threaded trees. Each stage is discussed in detail below.

A. Identification of Overlapping Data Structures


(a) Memory Graph after MemPick identified the types of object.

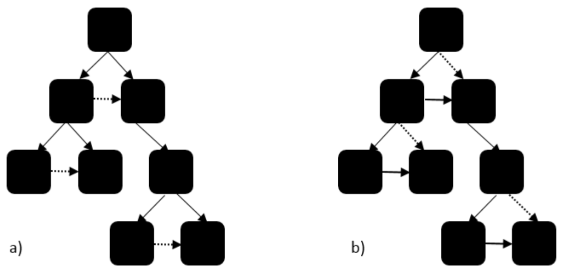

(b) Memory Graph split into two partitions containing objects of the same type. One sentinel node also found.


(c) MemPick detected three overlapping structures: a child tree, a parent tree and a list.



(d) MemPick measures if the tree is balanced. It measures height of subtrees recursively and concludes that it is unbalanced.

Figure 2: A running example illustrating MemPick's detection algorithm.



a)The overlapping structure contains the left and the right children edges.
b)The overlapping structure contains the left child edge and the sibling edge.

Figure 3: An example binary tree with three pointers: left, right and sibling.It shows two overlapping data structures: one depicted on left with solid edges and the other depicted on the right.

Minimal pointer set is searched for finding overlapping data structures. This set comprise of the following properties

- Generating sub-graph by keeping only those edges, corresponding to pointers that are connected
- There should be no subset of minimal pointer set, holding the first property

Overlapping data structures are referred as Overlays in this section. For each partition of the memory in the graph, a set $P = p_1, p_2, p_3, .., p_n$ is maintained. Each p_i corresponds to the offset of a pointer variable representing type of node in class or struct. All maximal subsets of that connect the partition are also listed. There is no redundant element in the maximal subset i.e. removing any pointer from the subset, will result in a subset that does not cover the whole partition. The tree in figure 2 shows the following set of overlays $4, 8, 12, 16$. These structures are disjoint but in many data structures they may have common elements as shown in figure 3.
Overlays also help to identify non-tree shapes like non-circular doubly-linked list. After identification of overlays, rules in Table 1 are applied for classification of each overlay separately. Columns 2 and 3 specify incoming and outgoing edges for ordinary nodes while columns 3 and 4 specify incoming and outgoing edges for special nodes. Last column defines the number of special nodes.

B. Data Structure Classification

Information of all overlays is combined for high-level description of partition being analysed, followed by a decision tree shown in figure 4. For classification of tree in figure 2c, first MemPick checks there are no graph overlays. As there is a binary tree and a parent tree overlay hence MemPick identifies a binary tree with a linear overlay.

C. Refinement Classifier for Special Data Structures

Complex data structures like threaded trees have a specific shape for which the general classification rules are not sufficient. Hence MemPick provides additional classifier to increase the accuracy of its classification. Threaded tree are discussed as example in the following section.
All the child pointers that are null in a binary tree, point to in-order predecessor or in-order successor in a threaded tree. We may assume that the right child of a threaded tree is used to thread to the successor node. A single overlapping structure is formed because of a threaded child pointer as it keeps all nodes of the tree connected. MemPick applies the un-threading algorithm [5] for obtaining a binary child tree, resulting in accurate classification.

7 Classification of Height-Balanced Trees

For more detailed classification of height-balanced tree, properties like size and height of different subtrees within a tree can be taken into account [14]. MemPick identifies height-balanced tree, AVL, B-trees and red-black trees. AVL tree is a type of binary tree where the difference between height of child subtree and the node is at most one. Where as in red-black trees ratio of height of longest branch to the shortest branch is 2:1. All lead nodes are at same height in a B-tree. MemPick measures height

Table 1: MemPicks rules to classify individual overlays

Type	Ordinary Nodes	Ordinary Nodes	Special Nodes	Special Nodes	#
	In	Out	In	Out	
List	1	1	0	1	1
			1	0	1
Circular List	1	1	-	-	-
Binary Child Tree	1	{0,1,2}	0	{1,2}	1
Binary Parent Tree	{0,1,2}	1	{1,2}	0	1
3-ary Child Tree	1	{ 0,...,3}	0	{1,...,3}	1
3-ary Parent Tree	{0,...,3}	1	{1,...,3}	0	1
n-ary Child Tree	1	{0,...,n}	0	{1,...,n}	1
n-ary Parent Tree	{0,...,n}	1	{1,...,n}	0	1
Graph (All remaining cases)					

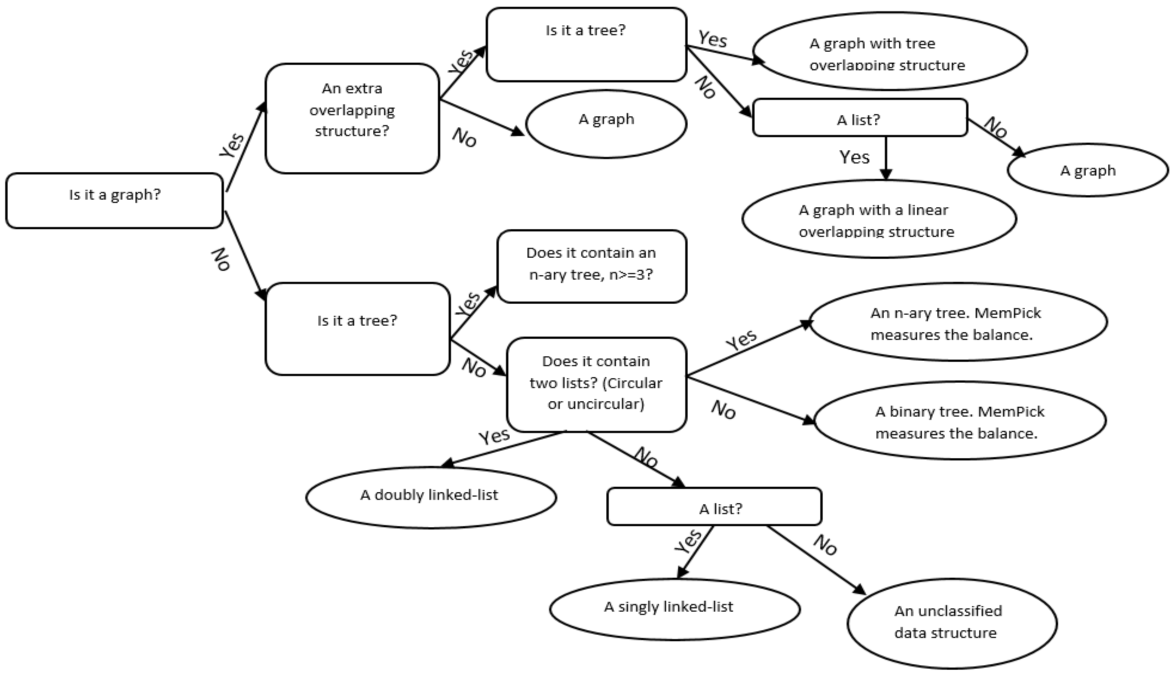

Is it a graph?
Yes
No
An extra overlapping structure?
Yes
No
Is it a tree?
Yes
No
A graph with tree overlapping structure
A graph
A list?
Yes
No
A graph with a linear overlapping structure
A graph
Is it a tree?
Yes
No
Does it contain an n-ary tree, n>=3?
Does it contain two lists? (Circular or uncircular)
Yes
No
An n-ary tree. MemPick measures the balance.
A binary tree. MemPick measures the balance.
A doubly linked-list
A list?
Yes
No
A singly linked-list
An unclassified data structure

Figure 4: MemPick's decision tree used to perform the final classification of a partition of the memory graph.

of a tree starting from the root node and computing left H_L and right h_R sub-trees. Absolute height $|h_L - h_R|$ and relative height imbalance h_L/h_R are also computed. If for all subtrees $|h_L - h_R|1$, tree is classified as AVL tree. B-tree property of $|h_L - h_R| = 0$ is checked for non-binary trees.

8 Final Mapping

In the final stage MemPick summarizes the results of all partitions by each unique partition classification and their occurrence count. Categorization of each data structure into local or global variables can also be detected. During execution of a function, MemPick checks the pointers stored by the function and assigns these pointers to stack frame of the function or global memory location. Information from multiple runs can be merged if the type global between different application runs can be detected. MemPick supports this analyses of individual instructions and the resulting global types can be merged transparently. In future MemPick can be merged with Howard [15] which is used for extraction of low-level data structures from C binaries.

9 Evaluation

For evaluation two sets of applications are used. Fist set comprised of popular libraries for lists and trees and the second set comprised of real-world applications. Test programs from libraries test suit are used for testing relevant functionality. These libraries are also used for evaluation of quiescent period detection to identify gap size requirement. The second set comprised of applications like chromium, lighttpd, wireshark, which showed scalability of approach.

A. Popular Libraries

MemPick is tested on 16 libraries featuring a wide range of data structures. These libraries are well documented and provides mean to evaluate different implementations. Built-in self-test functionality is used for these libraries. Additional tests are also built according to testing requirements.

MemPick is tested on 16 libraries featuring a wide range of data structures. These libraries are well documented and provides mean to evaluate different implementations. Built-in self-test functionality is used for these libraries. Additional tests are also built according to testing requirements

Table 2 shows the summary of test performed on selected libraries. All data structures were successfully categorized by MemPick with only total of two misclassifications. For GDSL, MemPick categorizes perfect balanced as the tree is limited to 3 nodes. In Glib N-ary tree is comprised of parent tree, left child and sibling pointers. A previous pointer in sibling list is also included. MemPick identifies parent pointer, child pointers correctly with an overlay also comprised of left child and previous sibling pointer. This overlay results in classification as graph. As the results of MemPick are also presented to the user, hence reverse engineer can easily interpret results. Hence MemPick deals with a variety of data structures and efficiently classifies them based only on their shape.

B. Applications

MemPick is also evaluated on 10 real world applications comprising of compiler, web browser, web server, multiple networking and graphics applications. MemPick successfully detected large number of data structures defined in third-party libraries. Table 3 summarizes the results of MemPick for real applications. All the results were manually checked for confirmation of classification. Table 3 presents two type of errors. Typing error for misclassification by MemPick and partition errors for data structures that are classified correctly but their partitions contain errors. MemPick only misclassified 3 data structures in 10 applications. Furthermore a wide range of data structures comprising of single linked lists to n-ary trees, were classified accurately. MemPick misclassified three instances. First misclassification is linked list in chromium, reported as parent-pointer tree. This misclassification results because of programming decision. Nodes removed

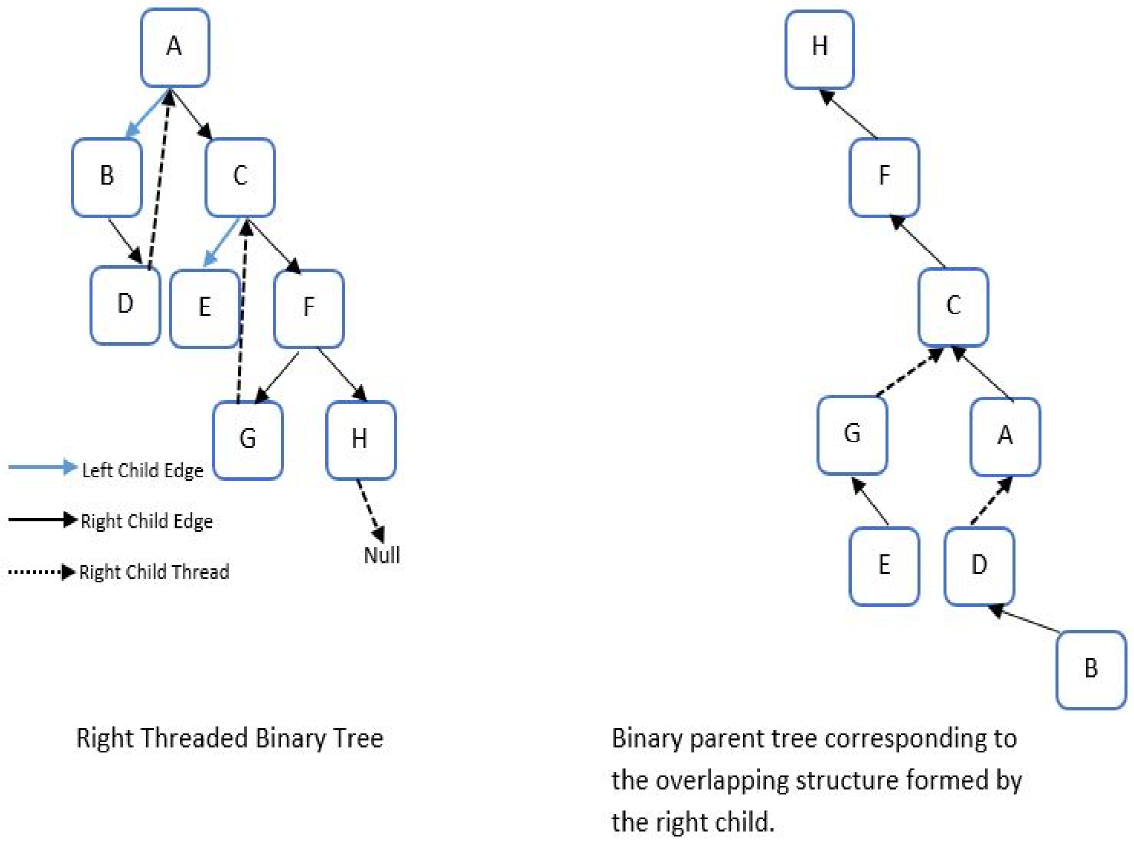

A
B
C
D
E
F
G
H
Null
Left Child Edge
Right Child Edge
Right Child Thread
Right Threaded Binary Tree
H
F
C
G
A
E
D
B
Binary parent tree corresponding to the overlapping structure formed by the right child.

Figure 5: The left-hand side figure presents an example right threaded binary tree and the right-hand side illustrates the corresponding overlapping binary parent tree.

Table 2: MemPick's evaluation across 16 libraries(#Total is the number of implementation variants of the given type available in the library, #TruePos is the number of correctly classified variants, #FalsePos is the number of misclassified variants)

Library	Type	#Total	#TruePos	#FalsePos
boost:container	dlist	1	1	0
	RB tree	1	1	0
clibutils	slist	1	1	0
	dlist	1	1	0
GDSL	dlist	2	2	0
	binary tree	3	2	1
	RB tree	1	1	0
GLib	slist	1	1	0
	dlist	1	1	0
	binary tree	1	1	0
	AVL tree	1	1	0
	n-ary tree	1	0	1
gnulib	dlist	1	1	0
	RB tree	2	2	0
	AVL tree	2	2	0
google-btree	B-tree	1	1	0
libavl	binary tree	4	4	0
	RB tree	4	4	0
	AVL tree	4	4	0
LibDS	dlist	1	1	0
	AVL tree	1	1	0
linux/list.h	slist	1	1	0
	dlist	2	2	0
linux/rbtree.h	RB tree	1	1	0
queue.h	slist	2	2	0
	dlist	2	2	0
SGLIB	slist	1	1	0
	dlist	1	1	0
STDCXX	dlist	1	1	0
	RB tree	1	1	0
STL	dlist	1	1	0
	RB tree	1	1	0
STLport	dlist	1	1	0
	RB tree	1	1	0
UTlist	slist	1	1	0
	dlist	2	2	0

from list, resided in memory because no data was cleared. This resulted in confusion for shape analysis. This can be resolved by use of advanced heap tracking and garbage collection.

MemPick misclassified three instances. Fist misclassification is linked list in chromium, reported as parent-pointer tree. This misclassification results because of programming decision. Nodes removed from list, resided in memory because no data was cleared. This resulted in confusion for shape analysis. This can be resolved by use of advanced heap tracking and garbage collection.

10 Limitations and Future Work

In our current work, we have applied shape analysis to the memory graph to identify and categorize heap based data structures. Heap objects are being handled through the use of memory allocators, so MemPick needs to be familiarized with the custom memory allocators of the application. We can consider the approach by [15] to be implemented for the requirements of MemPick.

For shape analysis of the memory graph in MemPick, simple but inflexible rules are applied to regulate the edge count. This mechanism can discriminate relevant and irrelevant edges in the memory. Our assessment shows the type inference engine designed for MemPick can meet this requirement in practice. To limit false positive we recommend to combine multiple typing information sources, such as Howard [2] or static analyses [8] [9] [13].

In addition, we focus solely on shape based categorization of data structures. MemPick currently recognizes all the instructions relevant to the internal operations of the data structure, but it can be extended for the functional analysis of data structures. We believe that the existing shape analysis can be extended to rapidly recognize code associated with the known semantics of data structures.

11 Related Work

Recovery of data structures is relevant to the fields of shape analysis and reverse engineering. While shape analysis verifies properties of data structures, reverse engineering techniques observe how a binary uses memory. In this section, we summarize the existing approaches and their relation to MemPick.
Low-level data structure identification: The most common method for revealing low-level data structures are based on static analysis techniques like value set analysis [8], aggregate structure identification [9] and combinations thereof [13]. Some recent approache such as Howard [2] has taken up dynamic analysis. As they cannot learn about high-level data structures MemPick can also be used.
High-level data structure identification: The most relevant approaches are the ones that dynamically identify high-level data structures, such as [7], [1], DDT [4] and [6]. The authors suggest a shape graph to explore changes in collection of objects of the same type. MemPicks memory graph enhances the shape graphs for data structure detection.

[1] recovers data structures during execution. After determining the location and size of an object; it converts the objects from raw bytes to sequences of block types. Finally, it categorizes similar objects by clustering objects with similar sequences of block types. In this way, [1] detects abstract data types but the detection is inadequate for reverse engineering.

[6] suggest to replace shape analysis by analyzing the patterns in data structure operations. They label instruction groups in order to merge the label information from all instruction groups to form a final candidate classification. This approach requires manually defined templates for classification and source code access. With MemPick, the result of the shape analysis can be used to limit the search space for analysis of data structure operations.

[12] propose an algorithm to dynamically infer abstract types. It suggests a run-time interaction among values to point out their similar data types. This approach assembles objects that are classified together. MemPicks approach to type identification is specifically customized to our requirements.

Table 3: MemPick's gap size evaluation across 16 libraries(The percentage represents the gap size percentile used for quiescent period selection. The columns represent the number of data structure implementations affected, compared to the basic-line of using 1%)

Library	Type	5%	10%	15%	20%
boost:container	dlist	0	0	0	0
	RB tree	0	0	0	1
clibutils	slist	0	0	0	0
	dlist	0	0	0	0
	RB tree	0	0	1	1
GDSL	dlist	0	0	0	0
	binary tree	1	1	1	1
	RB tree	0	1	1	1
GLib	slist	0	0	0	0
	dlist	1	1	1	1
	binary tree	0	1	1	1
	AVL tree	0	1	1	1
	n-ary tree	0	1	1	1
gnulib	dlist	0	0	1	0
	RB tree	0	1	2	0
	AVL tree	0	1	2	0
google-btree	B-tree	0	1	1	1
libavl	binary tree	2	2	2	2
	RB tree	2	2	2	2
	AVL tree	2	2	2	2
LibDS	dlist	0	0	0	0
	AVL tree	1	1	1	1
linux/list.h	slist	0	0	0	0
	dlist	0	0	0	0
linux/rbtree.h	RB tree	0	0	1	1
queue.h	slist	0	1	2	2
	dlist	2	2	2	2
SGLIB	slist	0	1	1	1
	dlist	0	1	1	1
STDCXX	dlist	0	0	0	0
	RB tree	0	0	0	1
STL	dlist	0	0	0	0
	RB tree	0	0	1	1
STLport	dlist	0	0	0	0
	RB tree	0	0	1	1
UTlist	slist	0	0	1	1
	dlist	0	0	0	0

Currently, the most advanced approach to the data structure detection problem is DDT [4]. DDT uses consistent information obtained using Daikon [11]. The invariant detection reduces the flexibility of the system. The system assumes that links between heap objects are never accessed while the contents of data structures are never altered. Compiler optimization limits application of DDT. In the absence of inlining, DDT works well with popular libraries but its accuracy for custom implementations of data structures remains uncertain. MemPick does not make any assumptions about the structure of the code implementing the operations on data structures. Additionally, DDT ignores the problem of the auxiliary overlays in data structures.

12 Conclusions

In this paper, we presented MemPick, to identify complicated pointer structures in stripped C/C++ binaries. MemPick can only detect data structures that can be distinguished by their shape but it is resistant to compiler optimizations. The results of evaluation of MemPick show that the accuracy of the data structure detection was high. Due to its high accuracy and different approach ,MemPick is one of the best tool tool for reverse engineers.

References

[1] Cozzie A., Stratton F., Xue H., and King ST. Digging for data structures. In *In: Proceedings of USENIX symposium on operating systems design and implementation, OSDI08*, 2008.

[2] Slowinska A., Stancescu T., and Bos H. A dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th annual network and distributed system security symposium*, 2011.

[3] Istvan Haller and Asia Slowinska and Herbert Bos. Scalable data structure detection and classification for C/C++ binaries. , 2015.

[4] Jung C. and Clark N. 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, MICRO-42*, 2009.

[5] Wyk CJV, editor. *Data structures and C programs*, volume Second. Addison-Wesley Longman Publishing Co., Inc., Boston, 1991.

[6] White DH and Luttgen G. Identifying dynamic data structures by learning evolving patterns in memory. In *Proceedings of the 19th international conference on tools and algorithms for the construction and analysis of systems, TACAS13.*, 2013.

[7] Raman E. and August DI. Recursive data structure profiling. In *Proceedings of the 2005 workshop on memory system performance*, 2005.

[8] Balakrishnan G. and Reps T. Analyzing memory accesses in x86 binary executables. In *Proceedings of the conference on compiler construction, CC04*, 2004.

[9] Ramalingam G., Field J., and Tip F. Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on principles of programming languages.*, 1999.

[10] Intel. A dynamic binary instrumentation tool, 2011. http://www.pintool.org/.

[11] Ernst MD, Perkins JH, Guo PJ, McCamant S., Pacheco C., Tschantz MS, and Xiao C. The daikon system for dynamic detection of likely invariants. *Sci Comput Program*, pages 35–45, 2007.

[12] Guo PJ, Perkins JH, McCamant S, and Ernst MD. Dynamic inference of abstract types. In *Proceedings of the 2006 international symposium on software testing and analysis, ISSTA06.*, 2006.

[13] Reps T. and Balakrishnan G. Improved memory-access analysis for x86 executables. In *Proceedings of the joint european conferences on theory and practice of software 17th international conference on compiler construction, CC08/ETAPS08*, 2008.

[14] Cormen TH, Stein C., Rivest RL, and Leiserson CE. Introduction to Algorithms. In , 2001.

[15] Chen X., Slowinska A., and Bos H. Detecting custom memory allocators in C binaries. , 2013.