MemPick: High-Level Data Structure Detection in C/C++ Binaries

Istvan Haller Asia Slowinska Herbert Bos
Vrije Universiteit Amsterdam, The Netherlands
{i.haller,asia,herbertb}@few.vu.nl

***Abstract*—Many existing techniques for reversing data structures in C/C++ binaries are limited to low-level programming constructs, such as individual variables or `structs`. Unfortunately, without detailed information about a program's pointer structures, forensics and reverse engineering are exceedingly hard. To fill this gap, we propose MemPick, a tool that detects and classifies high-level data structures used in stripped binaries. By analyzing how links between memory objects evolve throughout the program execution, it distinguishes between many commonly used data structures, such as singly- or doubly-linked lists, many types of trees (e.g., AVL, red-black trees, B-trees), and graphs. We evaluate the technique on 10 real world applications and 16 popular libraries. The results show that MemPick can identify the data structures with high accuracy.**

I. Introduction

Modern software typically revolves around its data structures. Knowing the data structures significantly eases the reverse engineering efforts. Conversely, not knowing the data structures makes the already difficult task of understanding the program's code and data even harder. In addition, a deep knowledge of the program's data structures enables new kinds of binary optimization. For instance, an optimizer may keep the nodes of a tree on a small number of pages (to reduce page faults and TLB flushes). On a wilder note, some researchers propose that aggressive optimizers automatically replace the data structures themselves by more efficient variants (e.g., an unbalanced search tree by an AVL tree) [1].

Accurate data structure detection is also useful for other analysis techniques. For instance, dynamic invariant detection [2] infers relationships between the values of variables. Knowing the types of pointer and value fields, for instance in a red-black tree, helps to select the relevant values to compare and to avoid unnecessary computation [3]. Likewise, principal components analysis (PCA) [4] is a technique to reduce a high-dimensional set of correlated data to a lower-dimensional set with less correlation that captures the essence of the full dataset but is much easier to process. PCA is used in a wide range of fields for many decades. In recent years, it has become particularly popular as a tool to summarize tree data structures [5], [6].

Unfortunately, most reversing techniques for data structures in C/C++ binaries focus on "simple" data types: primitive types (like int, float, and char) and their single block extensions (like arrays, strings, and structs) [3], [7]–[9]. They do not cater to trees, linked lists, and other pointer structures.

Existing work on the extraction of pointer structures is limited. For instance, the work by Cozzie et al. is unabashedly imprecise [10]. Specifically, they do not (and need not) care for precise type identification as they only use the data structures to test whether different malware samples are similar in their data structures. Of course, this also means that the approach is not suited for reverse engineering or precise analysis.

A very elegant system for pointer structure extraction, and perhaps the most powerful to date, is DDT by Jung and Clark [1]. It is accurate in detecting both the data structures and the functions that manipulate them. However, the approach is limited by its assumptions. Specifically, it assumes that the programs access the data structures exclusively through explicit calls to a set of access functions. This is a strict requirement and a strong limitation, as *inline* manipulation of data structures—without separate function calls—is common. Even if the programmer defined explicit access functions, most optimizing compilers inline short access functions in the more aggressive optimization levels. Also, in their paper, Jung and Clark do not address the problem of overlapping data structures—like a linked list that connects the nodes of a tree. Overlapping data structures, sometimes referred to as *overlays*, are very common also.

A. Contributions

In this paper, we describe MemPick: a set of techniques to detect and classify heap data structures used by a C/C++ binary. MemPick requires neither source code, nor debug symbols, and detects data structures reliably even if the program accesses them inline. Detection is based on the observation that the shape of a data structure reveals information about its type. For instance, if an interconnected collection of heap buffers *looks* like a balanced binary tree throughout the program's execution, it probably is one. Thus, instead of analyzing the instructions that modify a datastructure, we observe dynamically how its shape evolves. As a result, MemPick does not make any assumptions about the structure of the binary it analyzes and handles binaries compiled with many different optimization levels, containing inline assembly, or using various function calling conventions.

Since our detection mechanism is based solely on the shape of data structures, we do not identify any features that characterize their contents. For instance, we cannot tell whether a binary tree is a binary search tree or not. Nor do

978-1-4799-2931-3/13/$31.00 © 2013 IEEE

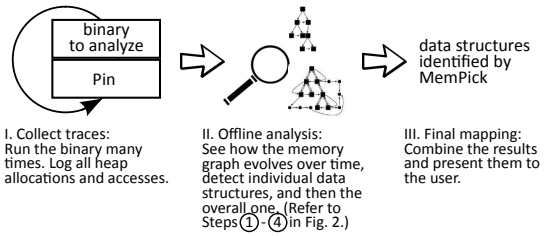

Fig. 1. MemPick: high-level overview.

we pinpoint the functions that perform the operations on data structures.

On the other hand, MemPick is suitable for all data structures that are distinguishable by their shape. The current implementation handles singly- and doubly-linked lists (cyclic or not), binary and n-ary trees, various types of balanced trees (e.g., red-black treed, AVL trees, and B-trees), and graphs. Additionally, we implemented measures to recognize sentinel nodes and threaded trees.

One of the qualitatively distinct features of MemPick is its generic method for dealing with overlays (overlapping data structures). Overlays complicate the classification, as the additional pointers blur the shape of the data structures. However, even if all nodes in a tree are also connected via a linked list, say, MemPick will notice the overall data structures and present them as a "tree with linked list" to the user.

Since MemPick relies on dynamic analysis, it has the same limitation as all other such techniques—we can only detect what we execute. In other words, we are limited by the code coverage of the profiling runs. While code coverage for binaries is an active field of research [11], [12], it is not the topic of this paper. In principle, MemPick works with any binary code coverage tool available, but for simplicity, we limited ourselves to existing test suites in our evaluation.

B. Outline

In Section II we describe the overall architecture of the system. Section III presents details about the low-level manipulation of the memory graph, that is the basis of our high level data structure representation from Section IV. Section V deals with the intricacies of data structure classification, that are extended with additional details about height balanced trees in Section VI. In Section VII we give an overview of information offered to the user, followed by an extensive evaluation in Section VIII and a discussion about the observed limitations and possible extensions in Section IX. Finally we discuss related projects on data structure reverse engineering in Section X and conclude the paper in Section XI.

II. MemPick

We now discuss our approach in detail. Throughout the paper, we will use the data structure in Figure 2 as our running example. The example is a snapshot of a binary tree with three overlapping data structures: a child tree, a parent tree and a list facilitating tree traversal. Each node of the tree has a pointer to a singly-linked list of some unrelated objects that ends with a sentinel node. The example is sufficiently complex to highlight some of the difficulties MemPick must overcome and sufficiently simple to track manually.

Figure 1 illustrates a high-level overview of MemPick. The detection procedure consists of three major stages. First, we record sample executions of an application binary. Next, we feed each of them to an offline analysis engine that identifies and classifies the data structures. Finally we combine the results and present them to the user.

The first stage requires tracking and recording the execution of the application, and for this we use Intel's PIN binary instrumentation framework [13]. PIN provides a rich API that allows monitoring context information, e.g., register or memory contents, for select program instructions, function- and system calls. We instrumented Pin to record memory allocation functions, along with instructions storing the addresses of all buffers allocated on the heap. In the remainder of this paper, we assume that applications use general-purpose memory allocators like `malloc()` and `free()`, or `mmap()`. It is straightforward to use the approach by Chen et al. [14] to detect custom memory allocators and instrument those instead.

For the offline analysis stage, MemPick analyzes the shape of links between heap buffers to identify the data structures. It consists of four further steps. In this section, we describe them briefly and defer the details to later sections (see also the circled numbers in Figure 2).

① MemPick first organizes all heap buffers allocated by the application, along with all links between them, into a *memory graph*. It reflects how the connections between the buffers evolve during the execution.

② Next, MemPick performs type analysis to split the graph into collections of objects of the same type. For instance, Figure (2a) illustrates a fragment of the memory graph at a point when the tree contains 8 nodes, and Figure 2b partitions the data structure into objects of the same type.

③ Given the partitions, MemPick analyzes the shape of the data structures by considering the links in each partition, searching for overlapping structures, and finally identifying types. For instance, in Figure (2c), MemPick first learns that the collection of squared nodes contains a child tree, a parent tree, and a list. It then classifies the data structure as a binary tree by means of a decision tree.

④ Finally, MemPick measures how each tree used by the application is balanced in order to distinguish between various types of height-balanced trees, e.g., red-black and AVL trees. This is illustrated in Figure (2d).

We discuss each step in detail in Sections III-VI. Once all the execution traces are analyzed, we combine the results, and present them to the user (Section VII).

III. Memory Graphs: Interconnected Heap Objects

A memory graph illustrates how links between heap objects evolve during the execution of an application. By itself, this is not enough to extract the links that are relevant to identify a data structure. For instance, in Figure 2a, we do not want

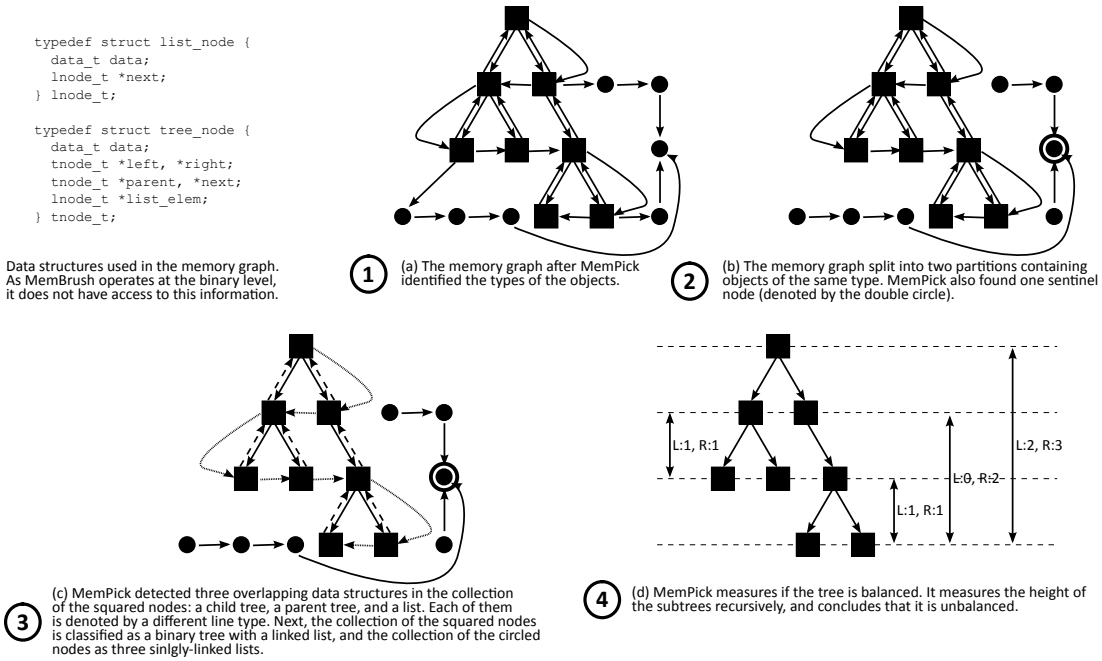

Fig. 2. A running example illustrating MemPick's detection algorithm.

to report a graph comprising all the nodes, but rather classify the tree and the lists separately. For this purpose, MemPick separates memory nodes that belong to unrelated low-level C/C++ types, i.e., different `structs` or `classes`.

Building the graph Like RDS [15] and DDT [1], MemPick inserts new nodes in the graph whenever a heap allocation occurs, and deletes existing ones upon deallocation. Edges represent connections between the allocated buffers. MemPick adds or removes them whenever the application modifies a link between two heap objects. This happens either on instructions that store a pointer to one object in another one, on instructions that clear previous pointers, or on calls to the memory deallocation functions.

Tagging the graph with type information Conceptually MemPick assigns two objects the same type if they are both either source or destination of the same instruction. Intuitively, an instruction carries implicit typing of its operands. However, to avoid false-positives (classifying two different objects together), MemPick first excludes instructions that might be *type agnostic* and handle objects of various types, such as instructions in `memcpy`-like functions.

Specifically, we aim for instructions that modify links in heap objects of a single type. To do so, MemPick classifies an instruction as *type aware* if it consistently stores a pointer to a heap buffer (or `NULL`) to a memory location at a specific, constant offset in another heap buffer. In other words, we do not consider instructions that store non-pointer values to the heap objects, or that store the pointers at different offsets at different times, etc. However, as it is common for applications to keep sentinel nodes in static or stack memory, we need to relax these filtering condition a little to allow for pointers to sentinel nodes.

The way MemPick extends the memory graph with type information is different from the approach used by DDT [1]. In particular, DDT applies typing based on allocation site, which poses problems when an application allocates an object of the same type in multiple places in the code—which is quite common in real software. For example, linked lists allocate memory nodes when inserting elements both in the STL (`push_front` and `push_back`) and the GNOME GLib (`g_list_append` and `g_list_prepend`) libraries. To handle these cases, DDT further examines if objects from different allocation sites are modified by the same interface functions in another portion of the application. As discussed in Section I, relying on the interface is a strong assumption that fails in the case of code that uses macros, say, rather than function calls to access the data structures, or in the face of aggressive optimization where the access code is inlined.

If necessary, we can further refine our analysis with existing approaches to data structure detection, like Howard [8] or static analyses [16]–[18]. While none of these techniques can combine objects from different allocation sites, they would help reduce possible false positives. For all experiments in this paper, however, we use MemPick's mechanism exclusively.

IV. From Memory Graph to Individual Data Structures

Given the memory graph, MemPick divides it into subgraphs, each containing individual data structures. These data structures will form the basis for our shape analysis (see Section V). As illustrated in Figure (2b), the output partitions are connected subgraphs whose nodes all have the same type.

MemPick starts by removing from the graph all links that connect nodes of different types. Doing so splits the graph into components that each reflect transformations of individual

structures during execution. In the example from Figure 2, one of the partitions would illustrate the growth of the tree.

Observe that not every snapshot of the memory graph is suitable for shape analysis. The problem is that properties characteristic to a data structure are not necessarily maintained at each and every point of the execution. For example, if an application uses a red-black tree, the tree is often *not* balanced just before a rotate operation. However, in all the *quiescent* time periods when the application does not modify the tree, its shape retains the expected features, e.g., it is balanced, every node has at most one parent, there are no cycles, and so on. Therefore, MemPick performs its shape analysis only when a data structure is quiescent.

MemPick defines quiescent periods in the number of instructions executed. Specifically, we measure the duration (in cycles) of the gaps between modifications of the data structures and then pick the longest n% as the quiescent periods. As long as we are sufficiently selective, we will never pick non quiescent periods. For instance, in our experiments, we picked only the longest 1% gaps as quiescent periods. The dynamic gap size allows MemPick to adapt to the characteristics of each binary and data structure. Compiler options and data usage patterns all contribute to the observed gap sizes. The method defined in MemPick benefits from two core properties, 1) it guarantees a lower bound of quiescent periods for every data structure, 2) it provides maximum robustness, by selecting the largest possible gap size that still satisfies the quiescent period frequency desired by the reverse engineer.

Before we pass the stable snapshots of each data structure to the shape analyzer, we detect and disconnect sentinel nodes. The problem of sentinels is that they blur the shape of data structures. For example, if we did not disconnect the sentinel node in Figure (2b), it would be difficult to see that the partition of circled nodes is in fact a collection of three lists.

To pinpoint sentinel nodes in a partition of the memory graph, MemPick counts the number of incoming edges for each node, and searches for outliers. While this strategy works well for lists, trees, and graphs, it might break some highly customized data structures. For example, in the case of a star-shaped data structure, it disconnects the central node, and MemPick reports a collection of lists.

Finally, for each partition of the memory graph, we acquire its snapshots in the quiescent periods, and use them in the following stage of MemPick's algorithm, discussed in the next section.

V. Shape Detection

We identify the shape of the graph-partitions based on observations during the quiescent periods. Any given shape hypothesis needs to hold for every "snapshot" of the graph-partition, since it represents a globally valid property of the data structure. Outliers are not allowed, as they would reduce the certainty of the final hypothesis. Since data structures often overlap and each of the overlapping substructures blur the actual shape, we identify them first. For instance, in Figure (2c), it is not simple to tell that the component composed of squares

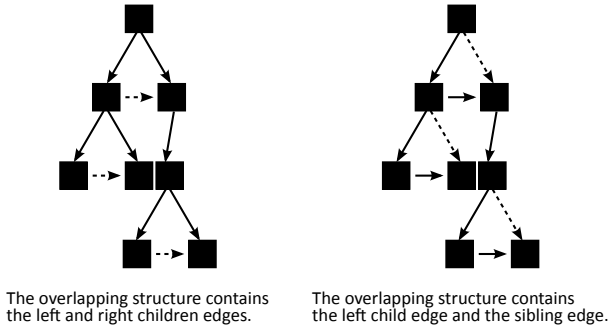

Fig. 3. An example binary tree with three pointer variables: `left`, `right`, and `sibling`. It has two overlapping data structures: {`left`, `right`} depicted on the left (denoted by the solid edges), and {`left`, `sibling`} depicted on the right. Observe that the overlapping data structures are not disjoint — both contain the left child edge.

actually represents a tree. Only after we distinguish between the child tree, the parent tree, and so on, does the identification become straightforward. Given the overlapping structures, we employ a hand-crafted decision tree that finally classifies the data structure. As a final step, we offer support for refined classifiers cases that discover data structures requiring more advanced analysis, e.g., threaded trees. We now discuss the stages in turn.

A. Overlapping Data Structure Identification

To find overlapping data structures, we search for maximal sets of pointer variables that keep all nodes of the data structure connected. In the remainder of this section, we refer to the constituent overlapping data structures as *overlays*, following a similar notion in network graphs.

In particular, for each partition of the memory graph, we consider a set $\mathcal{P} = \{p_1, \ldots, p_n\}$, where each p_i is the offset of a pointer variable in the `struct` or `class` representing the node type. For example, in Figure 2, we have $\mathcal{P} = \{4, 8, 12, 16\}$, which maps to the set of pointers in the tree: {`left`, `right`, `parent`, `next`}. Next, we list all maximal subsets of $\mathcal{P}$ that keep the partition connected. The subsets are maximal in the sense that they do not contain any redundant elements, i.e., if we remove an element from a subset, the remaining pointers do not cover the whole partition. In the tree in Figure 2, we identify the following set of overlays: $\{\{4, 8\}, \{12\}, \{16\}\}$. Even though these structures are disjoint, in many other data structures, they may have common elements. Refer to Figure 3 for an example.

We can now apply the rules in Table I to classify each overlay individually. Columns 2–5 specify the number of incoming and outgoing edges for ordinary and special nodes, while the last one defines how many special nodes there are . For instance, each "ordinary" (internal) node of a list has one incoming and one outgoing edge; additionally each list has one node with just one outgoing edge (the head), and one node with just one incoming edge (the tail). Currently, we do not distinguish between different classes of graph, e.g., cyclic and acyclic graphs, but extending the list of rules is straightforward.

TABLE I
MEMPICK'S RULES TO CLASSIFY INDIVIDUAL OVERLAYS. THEY SPECIFY THE NUMBER OF INCOMING AND OUTGOING EDGES FOR ORDINARY AND SPECIAL NODES.

Type	Ordinary 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 the remaining cases				

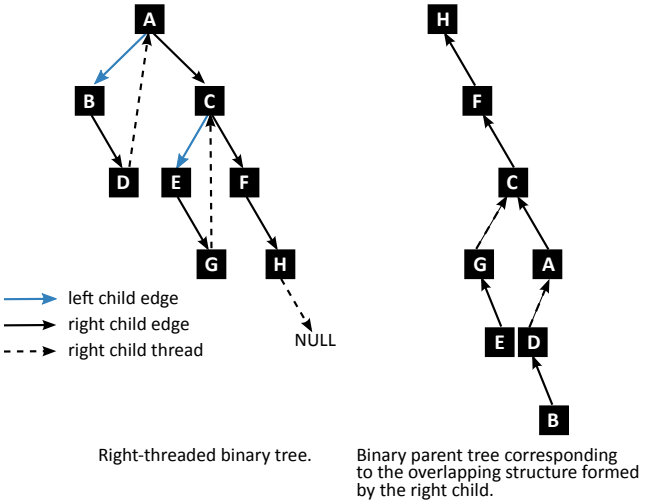

Fig. 5. The left-hand side figure presents an example right-threaded binary tree, and the right-hand side one illustrates the corresponding overlapping binary parent tree.

B. Data Structure Classification

Finally, MemPick combines the information about all overlays, and reports a high-level description of the partition being analyzed. This step follows a decision tree presented in Figure 4. To classify the tree in Figure (2c), MemPick first checks that the data structure has no graph overlays. Since it contains a binary child and a parent tree overlays, MemPick reports a binary tree (with an additional linear overlay). To refine the results, MemPick additionally measures the balance of the tree in Section VI.

C. Refinement Classifiers for Special Data Structures

Some popular data structures have very specific shapes for which the general classification rules of Section V-B are not sufficient. Threaded trees are one such example, currently support by MemPick. In order to increase the accuracy of it classification, MemPick allows for the addition of refinement classifiers that are tailored for specific data structures. We will discuss the threaded tree as an example, as it is the only common data structure with an "exceptional" shape we encountered throughout our extensive evaluation (Section VIII).

Threaded trees are a variant of the well-known binary trees. In a threaded tree, all child pointers that would be null in a binary tree, now point to the in-order predecessor or the in-order successor of the node. For instance, the left child could point to the predecessor and the right child to the successor. Alternatively, the data structure may use only one of the children for threading. Without loss of generality, assume the threaded tree uses the right child node to thread to the successor node. Threading facilitates tree traversal. The additional links are known as *threads*. Refer to Figure 5 for an example threaded tree. In our experiments, threaded trees appear in three libraries, including the GNOME GLib library.

Since a threaded child pointer keeps all nodes of the tree connected, it forms a single element overlapping structure. Observe that it has the shape of a binary parent tree. Thus, a child pointer is either threaded, and it forms a binary parent tree overlapping structure itself, or it is not threaded, and never included in an overlapping structure. Since it is an extraordinary situation, MemPick performs further analysis to test if this partition of the memory graph is a threaded tree. It searches for a root candidate to which it can successfully apply the *un-threading* algorithm [19]. After this step, it obtains a binary child tree overlapping structure, and the final classification is straightforward.

VI. CLASSIFICATION OF HEIGHT-BALANCED TREES

When the shape detection step in Section V-B classifies a data structure as a tree, MemPick additional measures measures its internal balance. It recognizes height-balanced trees [20], and identifies AVL, red-black and B-trees. AVL trees are binary trees where the heights of the child subtrees of any node differ by at most one. Red-black trees on the other hand allow the height of the longest branch to be at most two times that of the shortest branch. B-trees combine high child count with perfect balance, all leaf nodes are located at the same height. Other balanced tree variants, like binomial or 2-3-4 trees can also be described based on their worst case imbalance and maximum child count.

MemPick measures the height of a tree recursively, starting at the root (i.e., the node with no incoming edges). While computing the height of the left and right subtrees, h_L and h_R, respectively, MemPick keeps track of both the absolute and relative height imbalance, i.e., $|h_L - h_R|$ and h_L/h_R. It classifies the tree as an AVL tree, if for all its subtrees $|h_L - h_R| \leq 1$, and as a red-black tree — under the condition that $\frac{1}{2} \leq h_L/h_R \leq 2$. For non-binary trees it checks the B-tree property of $|h_L - h_R| = 0$. Since MemPick focuses its analysis on the shape of a tree, it might misclassify a *too* balanced red-black tree as an AVL tree. However, if a red-black tree is always perfectly balanced, this behavior is very useful for an analyst to know about.

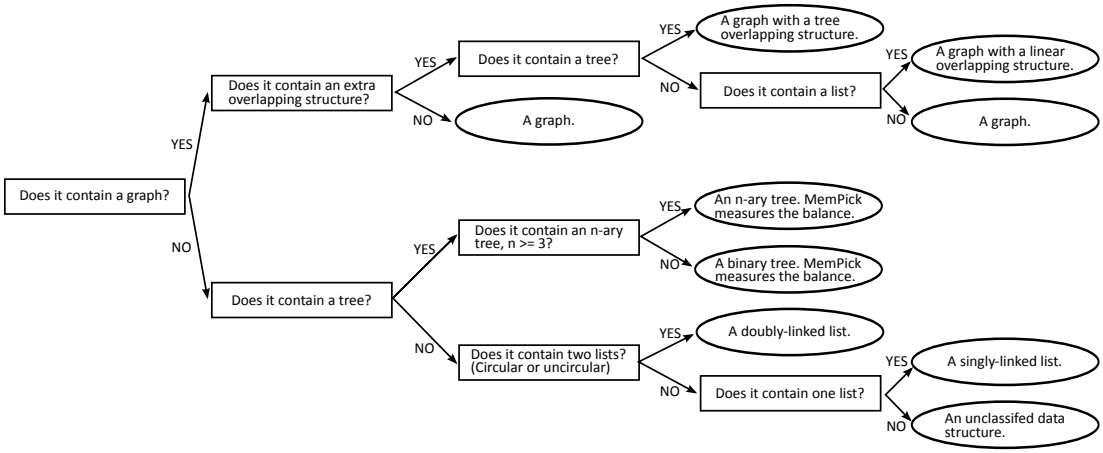

Fig. 4. MemPick's decision tree used to perform the final classification of a partition of the memory graph.

VII. Final Mapping

In the final step, MemPick combines the results of the partitions from the different execution traces, and presents them to the user.

Currently, the output of MemPick is a list of data structures that the application assigns to its local or global variables. Whenever the application stores a pointer to an identified heap buffer in either stack or static memory, MemPick maps the destination to the stack frame of the currently executing function or a global memory location, respectively.

Since the output relates to static locations from the binary, it is straight-forward to merge together data structure information from different execution traces. Using the mapping between data structures and variables, MemPick can detect partitions of the same data structure from different execution runs and generate global sets of data structure partitions. MemPick already operates with the notion of multiple independent partitions for a given data structure, making the merger transparent to the end user.

In the future, we plan to merge MemPick with Howard [8], a solution to extract low-level data structures from C binaries. As Howard automatically generates new debug symbol tables for stripped binaries, MemPick's results will fit in perfectly, providing the user with much more useful details. For instance, in Figure 2, instead of information that the application has a pointer to a `struct` consisting of one integer and five other pointers, the user will learn that the application has a pointer to an unbalanced binary tree with three overlays, and that each node of the tree has a pointer to a singly-linked list of some other data structures. However, this extension is beyond the scope of this paper.

VIII. Evaluation

We evaluated MemPick on two sets of applications. For the first set, we gathered as many popular libraries for lists and trees as we could find. We then exercised the most interesting/relevant functions using test programs from the libraries' test suites. These synthetic tests allow us to control exactly the features we want to test. Next, we evaluated MemPick on a set of real-world applications, like chromium, lighttpd, wireshark, and the clang compiler.

A. Popular Libraries

We tested MemPick on 16 popular libraries that feature a diverse set of implementations for a wide range of data structures. Including libraries in the evaluation has multiple benefits. Firstly, they provide strong ground-truth guarantees since the data structures and their features are well documented. Secondly they provide a means to evaluate a wider range of implementation variants, since most applications typically rely on a few standard implementations like STL and GLib in practice. For all the libraries we tried to use the built-in self-test functionality. Only if such a test was not available, we built simple test harnesses to probe all functionalities.

In the following we present a short summary of the reasoning behind some of the library choices. The evaluation set contains 4 major STL variants and GLib, the libraries typically used by major Linux applications. In addition, libavl brings a large variety of both balanced and unbalanced binary trees, with different overlay configurations, like the presence of parent pointers or threadedness. Several libraries (like UTlist, BSD queues, and Linux lists) implement inline data structures with no explicit interface in the final binary—typically by offering access macros instead of functions. We also include the Google implementation of in-memory B-Trees to validate the ability of MemPick to detect balanced non-binary trees. Typically B-Trees are implemented in database applications, which operate on persistent storage, leading to a lack of pointers in the data structure nodes.

Table II presents a summary of our results gathered from the libraries. We do not present results for individual data structure partitions as that number is dependent on the specific test applications. In all scenarios MemPick classified all partitions for any given data structure the same way.

For all tests executed, we encountered a total of two misclassifications, while all other data structures were successfully identified by MemPick (no false negatives). In the case of GDSL, the shape of the misclassified binary tree is detected

appropriately, however MemPick reports perfect balancedness since the tree is limited to 3 nodes. The misclassification in GLib is more subtle. The implementation of the N-ary tree uses parent, left child and sibling pointers. For optimization purposes the authors also include a *previous* pointer in the sibling list. MemPick correctly identifies the presence of an N-ary parent pointer and binary child pointer (left child + next sibling) trees, but it also detects an overlay using the left child and previous sibling pointers. This overlay does not match any basic shape and is reported as a graph, bringing the overall classification to a graph. Since MemPick also reports the overlay classification to the user, a human reverse engineer can accurately interpret the results. Alternatively, the user can add a (trivial) refinement classifier for this scenario, since the presence of two overlays does imply more structure than a generic graph, but we wanted to keep the number of refinement classifiers to a minimum.

Summarizing these results, we see that MemPick successfully deals with a large variety of data structure implementations. It is capable of correctly identifying the underlying type, independent of the presence of interface functions and independent of overlay variations. The results also show the efficiency of classifying balanced binary trees based only on shape information, provided the tree is sufficiently large.

B. Applications

MemPick is designed as a powerful reverse-engineering tool for binary applications, so it is natural to evaluate its capabilities on a number of frequently used real applications. For this purpose we have selected 10 applications from a wide range of classes, including a compiler (Clang), a web browser (Chromium), a webserver (Lighttpd), multiple networking and graphics applications. In all of these instances MemPick finishes its analysis within one hour, confirming its applicability to real-world application.

As we discussed in the section II, MemPick operates under the assumption that it can track all memory allocations. Two of the selected applications, namely Clang and Chromium, use custom memory allocators to manage the heap. In the case of Clang we also instrumented the custom memory allocators to gain insight to the internal data structures. For Chromium we were currently unable to perform such instrumentation. MemPick was still able to detect a large number of data structures that are defined in third-party libraries which still employ the system allocation routines. In principle, it would be straightforward to detect custom memory allocators automatically using techniques developed by Chen et al. [14].

Table III presents an overview of the results from all applications. It is important to note that for applications there exists no ground-truth information that we can compare against. For every application reported by MemPick we manually checked the corresponding source code to confirm the classification. We report two types of errors in table III. One is typing errors, when a given data structure is misclassified by MemPick. The other is partition errors. They refer to data structures that were classified accurately overall, but for which a number of their partitions contained errors.

TABLE II
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
	RB tree	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

The accuracy of MemPick is demonstrated by the fact that only 3 type misclassifications were detected in all tests on all 10 applications. MemPick was successful in identifying a wide-range of data structures, from custom designed singly-linked lists to large n-ary trees used for ray-tracing. MemPick also highlights different developer trends in the use of data structures. Some application developers prefer static storage such as arrays over complex heap structures. Examples for this pattern include wget and lighttpd. To ensure that this observation is not the result of false negatives, we manually inspected these two applications for undetected data structure implementations. As far as our evaluation goes, no data structures were missed by MemPick in these two applications.

Now let us focus our attention on the analysis of the erroneous classification reported by MemPick. The first example is a type misclassification in one of the linked list imple-

mentations in chromium. In this scenario MemPick reported a parent-pointer tree between the memory nodes. Browsing the source reveals the root of the error to be a programming decision. Nodes removed from the list never have their internal data cleared, nor are they freed until the end of the application. These unused memory links will stay resident in memory and confuse our shape analysis. A potential solution for this problem is a more advanced heap tracking mechanism with garbage collection. The latter would identify dead objects in memory and ensure that they are removed from the analysis. However we feel that this is not in the scope of the current paper.

The other two type misclassifications both stem from composite data structures. Templated libraries such as STL make it possible for the programmer to build composite data structures like list-of-trees or list-of-lists. MemPick correctly identifies the data structure boundaries in situations where node types are mixed, but is unable to do so if both components have the same type, like dealing with list-of-lists. Without such boundaries, MemPick will evaluate the shape of the data structure as a whole. Intuitively, the resulting data structure still has a consistent shape, but features increased complexity. A combination of singly-linked lists turns into a child-pointer tree, while binary trees turn into ternary trees with the addition of the ”root of sub-tree” pointer. This is also exactly what MemPick reports in these two scenarios. Pure shape analysis is not sufficiently expressive to distinguish between this pattern and regular child-pointer or ternary-trees, respectively. A reverse-engineer using MemPick can still identify this pattern with good confidence, by observing that the other partitions of the same type are classified as lists or trees.

Looking at the partition errors in table III, the reader can notice that the vast majority belong to binary trees. We focus our attention on this class of errors first. For all misclassifications of this category, MemPick erroneously detects AVL balancedness instead of the weaker red-black or unbalanced properties. As presented previously in section VI measuring the balancedness of a tree does carry uncertainty if the tree is too small. We confirmed that for each of the erroneous partitions, the tree contained no more than 7 nodes, a number too small to identify the difference between the two tree types. For all trees larger than this size our algorithm has an error rate of 0%.

Outside of the 3 main groups of errors, MemPick reports a few more misclassified partitions. Considering the total number of partitions reported across the 10 applications, these errors represent less than 1% and do not impact the overall analysis.

TABLE III
MEMPICK'S EVALUATION ACROSS 10 REAL-LIFE APPLICATIONS. #T IS THE NUMBER OF UNIQUE DATA STRUCTURES BELONGING TO THE GIVEN TYPE, #MT IS THE NUMBER OF TYPE MISCLASSIFICATIONS, #P IS THE NUMBER OF PARTITIONS BELONGING TO THE GIVEN TYPE, #MP IS THE NUMBER OF PARTITION MISCLASSIFICATION

Application	Type	#T	#MT	#P	#MP
chromium	slist	16	0	303	0
	dlist	5	0	24	0
	list of lists	1	1	8	8
	n-ary tree	1	0	16	0
	n-ary tree	1	0	2	0
	slist + graph	1	0	169	2
	graph	2	0	10	1
clang	slist	3	1	5	1
	dlist	5	0	8	0
	RB tree	1	0	6	2
	graph	4	0	13	0
inkscape	slist	9	0	186	0
	dlist	5	0	14	0
	RB tree	1	0	7	4
	tree of trees	1	0	5	0
	n-ary tree	1	0	28	0
	slist + graph	1	0	13	0
	graph	1	0	1	0
lighttpd	slist	2	0	2	0
	dlist	1	0	1	0
	binary tree	1	0	1	0
pachi	n-ary tree	1	0	1	0
povray	slist	9	0	36	0
	dlist	3	0	66	2
	RB tree	1	0	1	0
	n-ary tree	1	0	17	0
	n-ary tree	1	0	16	1
	slist + graph	1	0	12	0
quagga	slist	2	0	7	0
	dlist	5	0	8	0
	binary tree	1	0	4	2
tor	slist	12	0	413	4
	graph	1	0	1	0
wget	slist	3	0	8	0
	dlist	1	0	6	0
	slist + graph	1	0	13	0
wireshark	slist	3	0	99	0
	dlist	1	0	1071	0
	binary tree	1	0	1	0
	n-ary tree	1	0	3	0
	RB tree	1	0	95	47
	AVL tree	1	0	2	0
	slist + graph	1	0	12	0
	graph	1	0	1	0

IX. LIMITATIONS AND FUTURE WORK

In this work we aim to detect and classify heap based data structures using shape analysis applied to the memory graph. Applications use memory allocators to manage heap objects, a facility instrumented in MemPick to maintain an accurate representation of the memory graph. While most applications employ system allocation routines like `malloc()` or `free()`, some applications implement custom memory allocators for performance benefits. In the latter scenario MemPick needs to be made aware of the custom memory allocators in use by the application. While this information is not readily available in stripped binaries, the approach by Chen et al. [14] is straightforward to adapt for the requirements of MemPick.

The shape analysis of the memory graph in MemPick is based on a set of simple, but stringent rules geared towards edge counts. This classification mechanism assumes the ability to discerning relevant and irrelevant edges in the memory

graph via some typing information. Our evaluation shows that the type inference engine designed for MemPick can meet this requirement in practice, but some theoretical corner cases still exist. Typeless pointers, unions or inner structs could confuse our current solution in theory. For the the future we propose the fusion of multiple typing information sources, such as Howard [8] or static analyses [16]–[18] to limit potential false positives.

In addition, we focus on data structures that can be classified based solely on their shape, and not the contents or algorithms used to handle them. For example, we cannot distinguish binary search trees from the generic binary trees.

A natural extension of MemPick is the functional analysis of data structures. MemPick currently identifies all the instructions involved in the internal operations of the data structure, but is unable to reason about them. The reverse engineering value would be expanded by labeling the instructions with their functional purpose(*insertion, deletion*). We believe that the existing shape analysis results significantly reduce the space of possible operations, enabling a robust and intuitive functional classification. This extension will allow reverse engineers to quickly identify code related to the known semantics of data structures and focus their attention on application logic instead.

X. Related Work

Recovery of data structures is relevant to the fields of shape analysis and reverse engineering. While shape analysis aims to prove properties of data structures (e.g., that a graph is acyclic), reverse engineering techniques observe how a binary uses memory, and based on that identify properties of the underlying data structures. In this section, we summarize the existing approaches and their relation to MemPick.

Shape analysis. Shape analysis [21]–[25] is a static analysis technique that discovers and verifies properties of linked, dynamically allocated data structures. It is typically used at compile time to find software bugs or to verify high-level correctness properties of programs. Although the method is powerful, it is also provably undecidable, and so conservative. It has not been widely adopted.

Low-level data structure identification. The most common approaches to low-level data structure detection, i.e., primitive types, `structs` or arrays, are based on static analysis techniques like value set analysis [16], aggregate structure identification [17] and combinations thereof [18]. Some recent approaches such as Rewards [7], Howard [8], and TIE [9], have resorted to dynamic analysis to overcome the limitations of static analysis. Even though they achieve high accuracy, they cannot provide any information about high-level data structures, such as lists or trees. MemPick is thus complementary to them.

High-level data structure identification. The most relevant to our work are approaches that dynamically detect high-level data structures, such as Raman et al. [15], Laika [10], DDT [1], and White et al. [26].

Raman et al. [15] focus on profiling recursive data structures. The authors introduce the notion of a shape graph, that tracks how a collection of objects of the same type evolves throughout the execution. MemPick's memory graph extends the shape graphs to facilitate data structure detection, which is beyond the scope of the profiler [15].

Laika [10] recovers data structures during execution. First, it identifies potential pointers in the memory dump—based on whether the contents of 4 byte words look like a valid pointer—and then uses them to estimate object positions and sizes. Initially, it assumes an object to start at the address pointed to and to end at the next object in memory. It then converts the objects from raw bytes to sequences of block types (e.g., a value that points into the heap is probably a pointer, a null terminated sequence of ASCII characters is probably a string, and so on). Finally, it detects similar objects by clustering objects with similar sequences of block types. In this way, Laika detects lists and other abstract data types. However, the detection is imprecise, and insufficient for debugging or reverse engineering. The authors are aware of this and use Laika instead to estimate the similarity of malware. Similarly to Laika, Polishchuk et al. [27], SigGraph [28], and MAS [29], are all concerned with identifying data structures in memory dumps. However, they all rely on the type related information or debug symbol tables.

White et al. [26] propose an alternative to shape analysis, by focusing the analysis on the patterns in data structure operations. They label instruction groups based on the local changes observed in the pointer graph. Finally they merge the label information from all instruction groups to form a final candidate classification. The main issue with this approach lies in the complexity of the underlying model, which requires a repository of manually defined templates to perform classification. The authors also require source code access to extract typing information for the pointer graph. Finally, their evaluation is limited to very simple applications which use a single data structure internally. With MemPick we have shown that shape analysis can provide the necessary accuracy, while benefiting from simple and intuitive models. While MemPick does not yet support the analysis of data structure operations, we strongly believe, that the result of the shape analysis is highly valuable to limit the search space of such analysis.

Guo et al. [3] propose an algorithm to dynamically infer abstract types. The basic idea is that a run-time interaction among (primitive) values indicate that they have the same type, so their abstract types are unified. This approach groups together objects that are classified together, e.g., array indices, counts or memory addresses. MemPick's approach to type identification (Section III) is less generic, but also simpler and specifically tailored to our needs.

Currently, the most advanced approach to the data structure detection problem is DDT [1]. DDT relies on well-structured interface functions that encapsulate *all* operations performed on data structures. The distinction is very strict: the system assumes that an application never accesses any links between heap objects, while the interface functions never modify the contents they store in the data structures. Thus, the applicability of DDT is limited when due to compiler optimizations,

the interface functions are inlined, their calling conventions do not follow the standard ones, or when a program simply uses data structures defined with macros or some less strict interfaces (e.g., `queue.h`). In the absence of inlining, DDT works well with popular and mature libraries, such as the C++ Standard Template Library (STL) or the GNOME C-based GLib, but it is unclear what accuracy it would achieve for custom implementations of data structures (let alone malware). MemPick does not make any assumptions about the structure of the code implementing the operations on data structures, so it has no problems analyzing applications that use `queue.h`, say. Additionally, DDT does not address the problem of the auxiliary overlays in data structures. For each data structure type, it relies on a *graph invariant* that summarizes its basic shape. For example, one of the invariants specifies that "each node in a binary tree will contain edges to at most two other nodes". I practice this assumptions does not hold.

XI. Conclusion

In this paper, we presented MemPick, a set of techniques to detect complicated pointer structures in stripped C/C++ binaries. MemPick works solely on the basis of shape analysis. The drawback of such an approach is that it will only detect data structures that can be distinguished by their shape. On the other hand, we showed that MemPick is impervious to compiler optimizations such as inlining and accurately detects the overall data structure even if it is composed of multiple overlapping substructures. We evaluated MemPick first on a set of 16 common libraries and then on a diverse set of ten real-world applications. In both cases, the accuracy of the data structure detection was high, and the number of false positives quite low. In conclusion, we believe that MemPick will be powerful tool in the hands of reverse engineers.

Acknowledgment

This work is supported by the European Research Council through project ERC-2010-StG 259108-ROSETTA, the EU FP7 SysSec Network of Excellence and by the Microsoft Research PhD Scholarship Programme through the project MRL 2011-049.

References

[1] C. Jung and N. Clark, "DDT: design and evaluation of a dynamic program analysis for optimizing data structure usage," in *Proc. of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-42, 2009.

[2] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 35–45, Dec. 2007.

[3] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst, "Dynamic inference of abstract types," in *Proc. of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA'06, 2006.

[4] H. Hotelling, "Analysis of a complex of statistical variables into principal components," *J. Educ. Psych.*, vol. 24, 1933.

[5] W. H. and J. Marron, "Object oriented data analysis: Sets of trees," *Annals of Statistics*, vol. 35, no. 5, pp. 1849–1873, 2007.

[6] B. Aydin, G. Pataki, H. Wang, E. Bullit, and J. Marron, "a principal component analysis for trees," *Annals of Statistics*, vol. 3, no. 4, pp. 1597–1615, 2009.

[7] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proc. of the 17th Annual Network and Distributed System Security Symposium*, ser. NDSS'10, 2010.

[8] A. Slowinska, T. Stancescu, and H. Bos, "Howard: a dynamic excavator for reverse engineering data structures," in *Proc. of the 18th Annual Network & Distributed System Security Symposium*, ser. NDSS'11, 2011.

[9] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," in *Proc. of the 18th Annual Network & Distributed System Security Symposium*, ser. NDSS'11, 2011.

[10] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures," in *Proc. of USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI'08, 2008.

[11] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in vivo multi-path analysis of software systems," in *Proc. of the 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS'11, 2011.

[12] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated Whitebox Fuzz Testing," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, ser. NDSS'08, 2008.

[13] Intel, "Pin - A Dynamic Binary Instrumentation Tool," http://www.pintool.org/, 2011.

[14] X. Chen, A. Slowinska, and H. Bos, "Detecting custom memory allocators in C binaries," Vrije Universiteit Amstetrdam, Tech. Rep., 2013.

[15] E. Raman and D. I. August, "Recursive data structure profiling," in *Proc. of the 2005 workshop on Memory system performance*, ser. MSP'05, 2005.

[16] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 binary executables," in *Proc. Conf. on Compiler Construction*, ser. CC'04, 2004.

[17] G. Ramalingam, J. Field, and F. Tip, "Aggregate structure identification and its application to program analysis," in *Proc. of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 1999.

[18] T. Reps and G. Balakrishnan, "Improved memory-access analysis for x86 executables," in *CC'08/ETAPS'08: Proc. of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, 2008.

[19] C. J. V. Wyk, *Data Structures and C Programs, 2nd Ed. (Addison-Wesley Series in Computer Science)*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1991.

[20] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2001.

[21] 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 *Proc. of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL'96, 1996.

[22] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," in *Proc. of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL'99, 1999.

[23] V. Kuncak, P. Lam, K. Zee, and M. Rinard, "Modular Pluggable Analyses for Data Structure Consistency," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, 2006.

[24] I. Bogudlov, T. Lev-Ami, T. Reps, and M. Sagiv, "Revamping TVLA: making parametric shape analysis competitive," in *Proc. of the 19th international conference on Computer aided verification*, 2007.

[25] K. Zee, V. Kuncak, and M. Rinard, "Full functional verification of linked data structures," in *Proc. of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI'08, 2008.

[26] D. H. White and G. Lüttgen, "Identifying dynamic data structures by learning evolving patterns in memory," in *Proc. of the 19th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'13, 2013.

[27] M. Polishchuk, B. Liblit, and C. W. Schulze, "Dynamic heap type inference for program understanding and debugging," in *Proc. of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL'07, 2007.

[28] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures," in *Proc. of 18th Annual Network & Distributed System Security Symposium*, ser. NDSS'11, 2011.

[29] W. Cui, M. Peinado, Z. Xu, and E. Chan, "Tracking rootkit footprints with a practical memory analysis system," in *Proc. of the 21st USENIX conference on Security symposium*, ser. SSYM'12, 2012.