Empir Software Eng (2016) 21:778–810
DOI 10.1007/s10664-015-9363-y

Scalable data structure detection and classification for C/C++ binaries

Istvan Haller · Asia Slowinska · Herbert Bos

Published online: 7 March 2015

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, 4 file system implementations and 16 popular libraries. The results show that MemPick can identify the data structures with high accuracy.

Keywords Data structures · Dynamic binary analysis

1 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

Communicated by: Romain Robbes, Massimiliano Di Penta and Rocco Oliveto

I. Haller (✉) · A. Slowinska · H. Bos
Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
e-mail: i.haller@student.vu.nl

A. Slowinska
e-mail: asia@few.vu.nl

H. Bos
e-mail: herbertb@few.vu.nl

propose that aggressive optimizers automatically replace the data structures themselves by more efficient variants (e.g., an unbalanced search tree by an AVL tree) (Jung and Clark 2009; Jung et al. 2011).

Accurate data structure detection is also useful for other analysis techniques. For instance, dynamic invariant detection (Ernst et al. 2007) 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 (Guo et al. 2006). Likewise, principal components analysis (PCA) (Hotelling 1933) 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 (Wang and Marron 2007; Aydin et al. 2009).

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) (Guo et al. 2006; Lin et al. 2010; Slowinska et al. 2011; Lee et al. 2011). 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 (Cozzie et al. 2008). 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 (2009). 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.

1.1 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 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 (Chipounov et al. 2011; Godefroid et al. 2008), 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.

1.2 Outline

We start by giving more context about binary analysis in Section 2 and provide two example applications in Section 3. In Section 4 we describe the overall architecture of the system. Section 5 presents details about the low-level manipulation of the memory graph, that is the basis of our high level data structure representation from Section 6. Section 7 deals with the intricacies of data structure classification, that are extended with additional details about height balanced trees in Section 8. In Section 9 we give an overview of information offered to the user, followed by an extensive evaluation in Section 10. We also discuss the computational complexity in Section 11 to argue about the scalability of the proposed approach. In Section 12 we look at the observed limitations and possible extensions to MemPick. Finally we discuss related projects on data structure reverse engineering in Section 13 and conclude the paper in Section 14.

This paper is an extended version of our WCRE 2013 publication (Haller et al. 2013).

2 Static Versus Dynamic Analysis

There are two main approaches to reverse engineering, static and dynamic analysis. In this section we discuss the pros and cons of each for data structure discovery and explain why we opted for dynamic analysis.

Static analysis reasons about the application without executing it. This enables the analysis of components that are difficult to execute normally. While static analysis is the most suitable to reason about the application as a whole, it is fundamentally imprecise for weakly typed languages such as C/C++/Assembly due to pointer aliasing and indirect control flow changes. This typically manifests itself in either false positives (Johnson et al. 2013) or false negatives (Engler and Musuvathi 2004) depending on the analysis model. In the area of binary analysis even the most powerful static technique has problems handling even the most basic aggregate arrays (Balakrishnan et al. 2005). On the other hand dynamic analysis

is inherently context sensitive since it reasons about a concrete execution path. While this approach can only cover what is executed, the added run-time information improves the precision and the scalability of MemPick. In the following we discuss the validity of detecting pointer structures using only dynamic analysis.

Complex data structures represent the core of most algorithms, and are thus used pervasively throughout the application. Data structures implementations are also typically designed as reusable software components, and employed in multiple contexts within the same application. In consequence, complete code coverage is not an appropriate measure for the effectiveness of data structure discovery. Functional coverage, which measures coverage in program features is more relevant for this type of analysis. We believe that in practice a reasonable sized set of unit tests can provide enough functional coverage to extract the core data structures using MemPick. Our evaluation confirms this assumption, since we discovered more than 100 different data structures in 10 applications using basic inputs.

3 Example Applications

To demonstrate the usefulness of MemPick, we describe two possible applications: malware analysis and retrofitting security to legacy binaries not designed with security in mind. We target MemPick at binary analysis where no source code or high level information is available directly from the application. This scenario is typical of malware analysis, which is critical in taking down large scale malicious infrastructures such as botnets (Rossow et al. 2013). MemPick is also applicable to benign applications, not for reverse engineering their contents, but to discover and secure existing structures (Slowinska et al. 2012).

3.1 Case Study 1: Malware Analysis

With the transition towards online services, there is an ever greater incentive to infect user machines with different varieties of malware. Botnets go beyond the traditional single instance attacks, connecting the infected machines in a custom network, used to control and monetize the system. Security experts and law enforcement on the other hand aim to infiltrate and disrupt this network in a botnet takedown. This process is exceedingly difficult due to the encryption and custom protocols employed in botnets.

Rossow et. al. (2013) provide an insightful analysis on the resilience of the state-of-the art Peer-To-Peer botnet families and potential avenues of attacks. This work relies heavily on the reverse engineering of the communication protocols and the underlying encryption algorithms. This process is currently mostly manual effort relying on the experience of the reverse engineer to discover high level code structures. With MemPick, we aim to provide additional information about pointer structures in order, to complement the low-level data structure information from Howard (Slowinska et al. 2011). Pointer structures are highly relevant, since malware designers can easily integrate them to distribute information within different memory objects. Further extensions of MemPick could also detect and semantically annotate the code manipulating the data structure, allowing the reverse engineer to focus his attention on application logic instead.

3.2 Case Study 2: Security Hardening for Third Party Applications

System administrators frequently have to deal with the integration of closed-source components into their systems, such as third party libraries or device drivers. These components

may contain vulnerabilities, that impact the security of the whole system. For example researchers at Microsoft confirm that many vulnerabilities in device drivers stem from the misuse of pointer structures (Yang et al. 2008). Recent research in system security started to shift focus from compiler based security enhancements towards binary hardening to overcome this challenge (Slowinska et al. 2012; Zhang and Sekar 2013).

MemPick detects the pointer data structures within the potentially vulnerable code as a starting point for the hardening process. The vulnerabilities include race conditions on the data structure, as well as malicious out-of-band changes by an attacker. Slowinska et al. (2012) developed a solution that leverages low-level data structure detection to harden these constructs against potential attacks. In the future we hope to infer semantic information about the underlying interface functions as-well. This will enable fully automatic monitoring of data structure validity to uncover attacks or bugs at run-time.

4 MemPick

We now discuss our approach in detail. Throughout the paper, we will use the data structure in Fig. 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 (Intel 2011). 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. (2013) to detect custom memory allocators and instrument those instead.

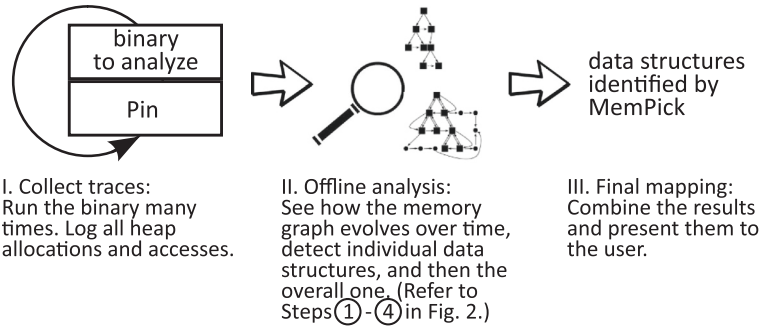


Fig. 1 MemPick: high-level overview

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 Fig. 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, Fig. 2a illustrates a fragment of the memory graph at a point when the tree contains 8 nodes, and Fig. 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 example, in Fig. 2c, all squared nodes would end up in one partition. Since it is common for data structure implementations to use auxiliary pointers, e.g., to form a parent tree, or a list facilitating traversal, the overall shape can become convoluted, such as it is the case in Fig. 2c. By looking at the overlapping structures, MemPick first learns that the collection of squared nodes contains a child tree, a parent tree, and a list, while each circled partition is a list. It then classifies the first 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 Fig. 2d.

We discuss each step in detail in Sections 5–8. Once all the execution traces are analyzed, we combine the results, and present them to the user (Section 9).

5 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 Fig. 2a, we do not want to report a graph comprising all the nodes, but rather classify the tree and the lists separately. For this purpose, MemPick strips connections between nodes featuring different logical types. We introduce the term logical type, to be able to reason about a weak form of type equality, necessary when dealing with potentially polymorphic types. Two objects share a logical type if either their low-level C/C++ type is identical or one object can be used in place of another, i.e. `sub-classes` or equivalent behavior in C.

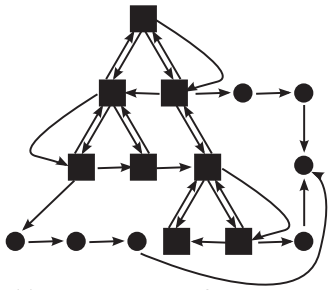
Building the graph Like RDS (Raman and August 2005) and DDT (Jung and Clark 2009), MemPick inserts new nodes in the graph whenever a heap allocation occurs, and deletes existing ones upon deallocation. Edges represent connections between the allocated buffers. MemPick adds or removes them whenever the 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 logical type if they could both be used in the same operand position of a given instruction. This follows the intuition that an instruction carries implicit typing of its

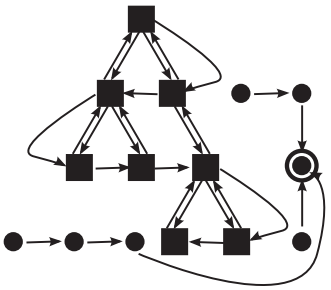
```
typedef struct list_node {
  data_t data;
  lnode_t *next;
} lnode_t;

typedef struct tree_node {
  data_t data;
  tnode_t *left, *right;
  tnode_t *parent, *next;
  lnode_t *list_elem;
} tnode_t;
```

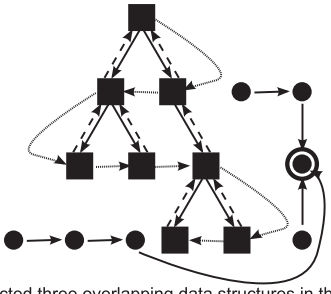
Data structures used in the memory graph. As MemBrush operates at the binary level, it does not have access to this information.



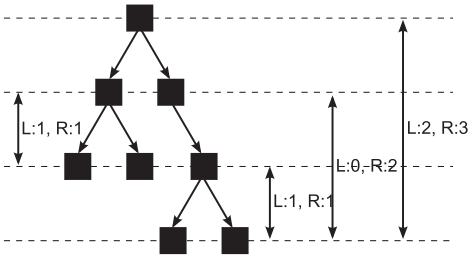
1 (a) The memory graph after MemPick identified the types of the objects.



2 (b) The memory graph split into two partitions containing objects of the same type. MemPick also found one sentinel node (denoted by the double circle).



3 (c) MemPick detected three overlapping data structures in the collection of the squared nodes: a child tree, a parent tree, and a list. Each of them is denoted by a different line type. Next, the collection of the squared nodes is classified as a binary tree with a linked list, and the collection of the circled nodes as three sinlgly-linked lists.



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

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

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.

Next we describe the type inference algorithm in more detail. The algorithm initially associates each instruction with a pair of unique tags, one for its source and one for its destination operand. Whenever a heap object is used as operand for a given instruction, it inherits the corresponding instruction tag. This ensures that different heap objects used within the same instruction operand all share a unique tag. If the heap object is already associated with another tag, than both tags correspond to the same logical type, and have to be merged into one. The tag change is then propagated across all heap objects to ensure consistency. While this algorithm is very simple, it successfully captures our assumptions about logical types. Even though the system starts out with zero assumptions and a highly fragmented tag space, the merging operation quickly converges and identifies a small set of logical types within the program.

During this algorithm it is critical to avoid any instructions with ambiguous typing. 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 (Jung and Clark 2009). 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 1, 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 (Slowinska et al. 2011) or static analyses (Balakrishnan and Reps 2004; Ramalingam et al. 1999; Reps and Balakrishnan 2008). 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.

6 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 7). As illustrated in Fig. 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 Fig. 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 Fig. 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.

7 Shape Detection

We identify the shape of the graph-partitions based on observations during the quiescent periods. As we described above, MemPick focuses on quiescent periods since they represent the stable state of the data-structure. 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 Fig. 2c, it is not simple to tell that the component composed of squares 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.

7.1 Overlapping Data Structure Identification

To find overlapping data structures, we search for sets of pointer variables that keep all nodes of the data structure connected. To exclude unnecessary pointers from such sets, we define the term `minimal pointer set`. Each of these sets features the following properties: *the subgraph generating by maintaining only the edges corresponding to the pointers remains connected; also no subset of a minimal pointer set holds the first property*. Intuitively eliminating any entry from such a set leads to a disconnected data structure. The problem of finding the minimal pointer set is analogous to the maximal set and set cover problems in complexity theory. 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 Fig. 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 Fig. 2, we identify the following set of overlays: {{4, 8}, {12}, {16}}. The first overlay uses the `left` and `right` pointers to connect the tree from a top-down perspective. The second overlay uses the `parent` pointer for the potential of a bottom-up traversal. The last overlay is an alternate linked-list style overlay using `next` pointer to access all nodes without having to traverse the tree. While the overlays in this example involve a disjoint set of pointers, in practice pointers may also be shared, as it is the case in the example from Fig. 3. In this example neither pointer can form an overlay by itself, but any combination of two results in a connected data-structure and should be reported for furter analysis.

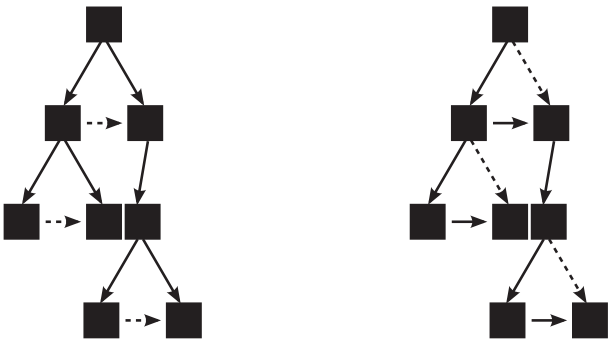


Fig. 3 An example binary tree with three pointer variables: `left`, `right`, and `sibling`. It has three overlapping data structures: { `left`, `right`} depicted on the left (denoted by the solid edges), { `left`, `sibling`} depicted on the right and { `right`, `sibling`}. Observe that the overlapping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

Finally, overlays are also interesting for non-tree shapes, as the can help to disambiguate non-standard list variants. For example, the non-circular doubly-linked list within the `utlist` library uses a peculiar implementation. While the `next` pointer is implemented as a traditional non-circular linked list, the `prev` pointer is made to be circular. Such hybrid implementations can difficult to identify, due to the lack of symmetry, without separating the individual overlays.

Once the overlays are identified, we apply the rules in Table 1 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.

7.2 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 Fig. 4. To classify the tree in Fig. 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 8.

7.3 Refinement Classifiers for Special Data Structures

Some popular data structures have very specific shapes for which the general classification rules of Section 7.2 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

Table 1 MemPick's rules to classify individual overlays

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				

They specify the number of incoming and outgoing edges for ordinary and special nodes

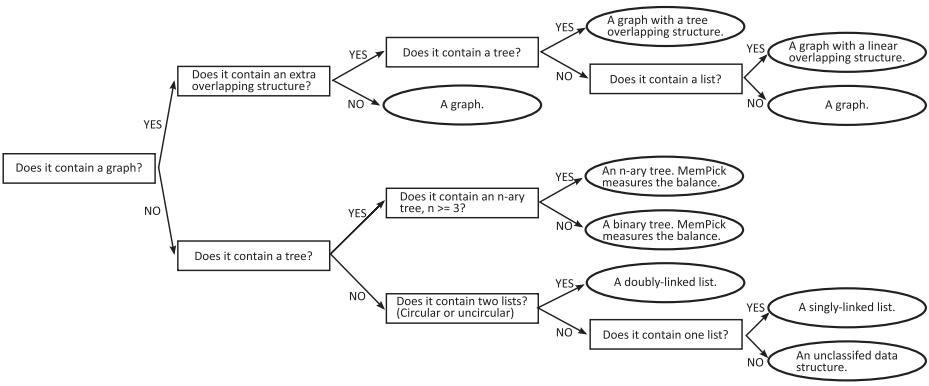


Fig. 4 MemPick's decision tree used to perform the final classification of a partition of the memory graph

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 10).

Threaded trees represent a variation on the binary tree representation that is used in practice. 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, without relying on parent pointers or recursion. The additional links are known as *threads*, and can use either one or both child pointers. Refer to Fig. 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,

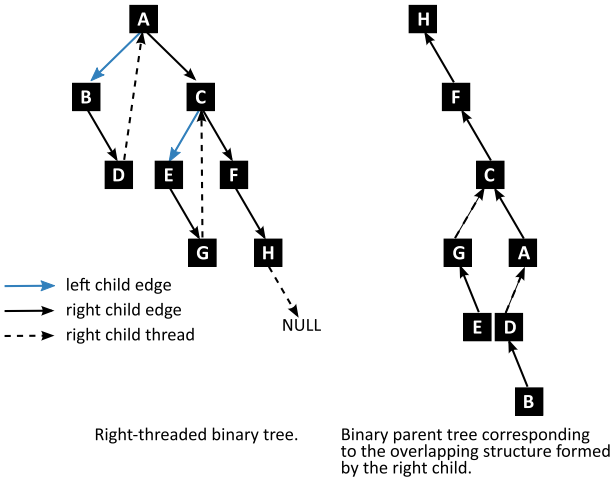


Fig. 5 The left-hand side figure presents an example right-threaded binary tree, and the right-hand side one illustrates the corresponding overlapping binary parent tree

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 (Wyk 1991). After this step, it obtains a binary child tree overlapping structure, and the final classification is straightforward.

8 Classification of Height-Balanced Trees

Once the shape detection step in Section 7.2 classifies a data structure as a tree, MemPick also attempts to reason about the properties of the tree to allow for a more precise classification. We have identified that the size and height properties of the different sub-trees within a tree offer enough information for more detailed classification of height-balanced trees (Cormen et al. 2001). These properties also map well to the shape-only classification algorithm presented so far. MemPick recognizes height-balanced trees (Cormen et al. 2001), 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.

9 Final Mapping

In the final step, MemPick combines the results of the partitions from the different execution traces, and presents them to the user. The summaries generated by MemPick include each unique partition classification and their occurrence count. Thus no outliers are excluded, while the user is not overwhelmed with 100s of copies of the same classification.

MemPick can also project the detected data structures to local or global variables of the application. 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 MemPick already operates with the notion of multiple independent partitions for a given data structure, it is straightforward to merge information from multiple runs. The only requirement for this process is the ability to detect types globally, between different application runs. The type inference engine in MemPick supports this requirement, by operating at the level individual instructions, which do not change between runs. The resulting global types allow MemPick to merge the results from multiple runs transparently to the user.

In the future, we plan to merge MemPick with Howard (Slowinska et al. 2011), 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 more detailed information about the data structures used by the application. For instance, in Fig. 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.

10 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. We also evaluated the quiescent period detection mechanism on these libraries to identify the requirements for gap size selection. Next, we evaluated MemPick on a set of real-world applications, like `chromium`, `lighttpd`, `wireshark`, and the `clang compiler`. We also evaluate the analysis time for these applications to show the scalability of the proposed approach. Finally we also looked into low-level system code with two major file system implementations, `ZFS` and `ntfs`.

10.1 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 2 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.

Table 2 MemPick's evaluation across 16 libraries

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

#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

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 results is still valuable, since MemPick reports all other classification details accurately, this error is also unlikely to occur in applications that deal with real data. 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. One observation is that both errors were found in libraries supporting a large variety of data structure implementations. This is not surprising, since the chance of non-standard data structures is increased with the size of the library. Still our results show that the overlay based classifier is resilient to unexpected data structure shapes, by correctly classifying all basic overlays contained within. Even if the overall classification fails, the partial results are still beneficial as an anchor for the reverse engineering process.

When testing the `utlist` library, we ran across an interesting classification report. MemPick reported a cyclic and a non-cyclic list overlay for the non-cyclic doubly linked list in the library. This behavior was confirmed to be a design decision, when correlating the results with the source code. This example shows the importance of the overlay based classification employed in MemPick. Without this approach the observed shape and behavior would not match any assumption about linked lists as the overall structure is neither properly cyclic nor non-cyclic.

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.

Next we aimed to identify the impact of the gap size percentile used with quiescent periods. In Section 6 we suggested the use of 1 % longest gaps as signal for quiescent periods. In this part of the evaluation we vary this number all the way to 20 % and observe its impact on classification accuracy. We perform this part of the evaluation on libraries, instead of applications, since they offer more precise ground truth information. Table 3 presents the results, counting for all the data-structure implementations where the classification was degraded in comparison to the original proposal. In some instances this degradation was in the form of missing overlays, while in other instances MemPick was unable to offer any valid classification. In general, one can observe that search-trees are more sensitive to the gap size, especially the implementations within the libavl library. This library offers a cloning interface for trees, which in some implementation variants does not respect the validity of the tree throughout the operation. If a quiescent period intervenes during the clone operation, the system will observe an invalid tree. One must also take into account, that we tested the libraries using the built-in unit tests whenever they were available. In this scenario data-structure operations are typically executed in quick sequence, without any intervening application code.

Table 3 MemPick's gap size evaluation across 16 libraries

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

The percentages represent the gap size percentile used for quiescent period selection. The columns represent the number of data-structure implementations affected, compared to the base-line of using 1 % gaps in Table 2

This explains the gap size sensitivity for some of the list implementations. Overall, these results suggest that the quiescent periods should be considered conservatively, especially

in the presence of heavily used, complex data-structures. Our suggestion when performing manually assisted reverse engineering is to start with a highly conservative gap size, which can progressively be increased to detect detect data-structures potentially missed by the initial setting.

10.2 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. Table 4 presents the number of code lines for each of these applications, giving an idea of their size.

As we discussed in the Section 4, 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. (2013).

Table 5 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 5. 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.

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

Table 4 Number of C/C++ lines of code for the 10 real-life applications, excluding potential third party libraries

Application	Version	Lines of code
Chromium	29.0.1548	4190k
Clang	3.2	1045k
inkscape	0.48.4	396k
Lighttpd	1.4.32	40k
Pachi	10.0	13k
povray	3.7.0.RC7	106k
quagga	0.99.22	194k
tor	0.2.4.12-alpha	119k
wget	1.14	68k
Wireshark	1.10.0	1727k

Table 5 MemPick's evaluation across 10 real-life applications

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

Table 5 (continued)

Application	Type	#T	#MT	#P	#MP
	slist + graph	1	0	12	0
	graph	1	0	1	0

#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

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 implementations 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 5, 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 8 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.

As part of evaluating, we also look at the analysis times required when processing these applications. We broke down the analysis times to different stages to identify potential problem areas within the analysis pipeline. We exclude the tracing component from this evaluation, since none of the proposed contributions relate to application tracing. The explicit tracing overhead can also be mitigated when combined with multi-path analysis. The KLEE family of multi-path analysis tools (Cadar et al. 2008, Marinescu and Cadar 2012, 2013 is a prime example within the software engineering research community. Tools within the KLEE family emulate memory operations, by first looking up detailed information about the allocation site at the target address. The tracing within MemPick performs a similar look-up to identify the target heap object, while also performing a look-up on the value as-well. Thus, the desired tracing functionality could also be integrated within tools from the KLEE family with an additional 2X overhead in the worst case.

Table 6 presents the running time of the different analysis stages. The TypeGen stage includes type inference and the detection of the quiescent periods. The GraphGen stage represents graph generation, while the OverlayGen stage identifies al potential overlays. Finally the Classificaton stage is the time it takes to perform the final classification. Applications with limited heap usage finish within a matter of seconds as expected. Once the heap usage increases, so does the analysis time, especially for the TypeGen stage. This stage operates on raw traces and its execution time is unaffected by the semantics of the heap objects. This is highlighted within Lighttpd and Pachi, which make good use of heap memory, but few heap objects are members of high-level data-structures. For these applications the bulk of the analysis is performed within the the first stage, after which all non-desirable heap objects are purged from further analysis. Another particular application is Clang, where the OverlayGen stage is significantly more costly than the rest. This behavior is due to some heap objects featuring a large set of the pointer elements. Overlay identification requires testing an exponential number of pointer combinations, but for most data-structures (except B-trees) the number of pointers is limited. Since we don't expect B-trees to come up often

Table 6 MemPick's analysis time evaluation across 10 real-life applications

Application	TypeGen	GraphGen	OverlayGen	Classification
Chromium	2s	<1s	<1s	2s
Clang	4s	<1s	178s	4s
inkscape	8s	4s	5s	9s
Lighttpd	23s	<1s	<1s	<1s
Pachi	21s	<1s	<1s	<1s
povray	38s	1s	1s	4s
quagga	<1s	<1s	<1s	<1s
tor	848s	542s	42s	361s
wget	8s	4s	1s	3s
Wireshark	160s	1030s	247s	453s

Averaged across 3 runs. Measured in seconds. TypeGen includes the type inference and quiescent period detection. GraphGen involves graph generation, while OverlayGen includes the overlay identification. Classification includes the remaining classification steps to get the final results

during analysis, this behavior can be considered an outlier and not the general case. Finally, for applications with heavy data-structure usage, such as Tor and Wireshark, the execution time can increase to the range of minutes, but the total analysis time is still only around 30 minutes. These execution times suggest that the proposed methodology is well suited for the offline analysis of complex applications. Further optimizations can also be applied to reduce the analysis time within a production setting. For more detailed discussions about scalability, we refer the reader to Section 11.

10.3 System Code

One of the proposed use cases for MemPick was to analyze low-level system code for potentially vulnerable data structures. In this section we analyze the effectiveness of MemPick when dealing with this application class. MemPick relies on the PIN (Intel 2011) framework for dynamic instrumentation, thus currently cannot analyze kernel-space code. However this does not mean that the mechanics behind MemPick cannot be applicable to system code. To overcome this technical limitation we leverage the FUSE project (Szeredi http://fuse.sourceforge.net), which allows file system implementations to reside in user-space. Two major file-system implementations `NTFS-3g` and `ZFS-FUSE` are built on top of this framework on Linux. For this evaluation we choose two additional projects,`s3fs`, which allows mounting buckets from the S3 online storage service of Amazon and `sshfs`, which allows mounting remote folders via ssh.

Table 7 presents the overview of the results from MemPick when analyzing these four systems. The format is the same as the one used for the evaluation of real-world applications. For these four systems no typing errors were observed, only partition errors where small red-black trees were mistakenly classified as being AVL trees. This type of error does not affect the ability of the reverse engineer to identify the underlying data structure since the results offer a comprehensive summary of all partitions, including the right classification. One peculiar detail is the lack of complex data structures for the two systems dealing with real file systems, `NTFS-3g` and `ZFS-FUSE`. No tree-like data structures related to inodes

Table 7 MemPick's evaluation for the 4 FUSE-based file systems

Application	Type	#T	#MT	#P	#MP
NTFS-3g	slist	1	0	1	0
	dlist	2	0	2	0
	slist + graph	1	0	24	0
ZFS-FUSE	slist	2	0	2	0
	dlist	1	0	1199	0
s3fs	slist	3	0	245	0
	dlist	1	0	1	0
	RB tree	1	0	62	37
	n-ary tree	1	0	19	0
	slist + graph	1	0	13	0
sshfs	dlist	2	0	16	0

#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

were discovered in the case of these two systems. By examining the intermediate results, we discover that MemPick correctly identifies the inode objects, but detects no direct pointer links between them. This discovery was confirmed by examining the underlying source code, which uses additional levels of indirection between inode objects. This programming pattern does not match our initial definition of homogeneous data structures. Future work may look into the discovery of heterogeneous data structures consisting of different object types. We conclude that MemPick was successful in analyzing these four systems and shows great promise in handling system code.

11 Complexity Analysis

In this section we analyze the computational complexity of the algorithms within MemPick, with the goal of proving that MemPick does not incur any hidden overhead that would impact its scalability. Our main argument in favor of dynamic analysis for the purpose of data structure detection is the inherent accuracy of run-time information that enables accuracy and scalability to coexist. To ensure that we meet our proposed performance goals, we analyze all components of our proposed approach, both from a theoretical and practical perspective. The research question we issue in this section is the following: "Is it possible to perform shape analysis using the same asymptotic complexity it takes to run basic test-suites for the application?". This research question stems from our assumption: the reverse engineer is capable of exercising the application with different inputs (as described in Section 2) and that he has access to powerful compute resources like clouds. The hope is that once the reverse engineer can execute the application itself, the analysis only involves a constant overhead that is independent of the application size and complexity. This constant overhead can be tackled by additional hardware resources if necessary. In the following, we analyze the complexity for five aspects of MemPick: executing the application, trace generation, type inference, graph generation and shape analysis. During the analysis we will use the following notations:

- N denotes the number of instructions executed and
- M denotes the number of heap objects generated by the application

11.1 Executing the Application

We perform the dynamic analysis in MemPick by instrumenting the application under test and monitoring its behavior. This setup incurs two inherent sources of complexity that we consider in this section: the overhead of the instrumentation framework itself and the number of execution paths necessary to provide appropriate coverage. We show that neither of these two components hinders the goal proposed in our research question.

MemPick uses PIN as its instrumentation framework (Intel 2011), which incurs a 4x overhead on the SPECint 2000 benchmark. In conclusion, the framework does not affect our complexity analysis, since our research question considers the native execution time as the baseline.

The second source of complexity we have to consider is the potential path explosion required for application coverage. In theory, an exponential number of possible execution paths exist for any application. We observed in Section 2) that full code coverage is not necessary for the purposes of data structure discovery. The core data structures of any application are used throughout the code, without concern for concrete execution

details (considering realistic inputs). This is especially true for pointer-based data structures where life-times surpass function boundaries. Functional coverage is the primary metric for the purpose of MemPick and is typically provided by existing unit tests. While the size of unit tests can range up to several hundred inputs, the number is always bound to ensure the ability to finish testing in a reasonable time. Bounding the number of execution paths ensures that the asymptotic complexity metric is not affected, by our coverage requirements.

11.2 Trace Generation

In this section we evaluate different techniques for monitoring and logging execution, to find an optimal solution that maintains the required linear complexity. MemPick generates traces as persistent and abstract representations of execution scenarios. The traces allow repeated analysis without having to deal with non-determinism during execution. They contain all events necessary for our analysis: heap object allocation and deallocation and writes into heap objects. The traces also abstract away low level details such as memory addresses into object identifiers to simplify analysis.

From an algorithmic perspective, tracing requires the system to instrument the individual instructions and generate short summaries whenever necessary. Whenever instructions deal with heap addresses, we abstract those into object identifiers and the corresponding offsets. Since an application may access various fields inside a memory object, each having a different offset, we cannot use a simple hash table to map addresses to object identifiers. The intuitive solution is to use an interval-tree like structure, where every allocation represents an interval. A look-up operation using any given memory address automatically results in both the object identifier and offset. While this is an elegant solution to our problem, it does imply that for each memory write we incur an $O(logM)$ time look-up, where M is the number of heap objects. Thus the overall complexity of tracing is increased to $O(N * logM)$. An alternative solution is to use pointer tracking similar to Howard (Slowinska et al. 2011), which allows constant speed look-up for the root pointer of each memory operand. While pointer tracking can introduce a significant execution overhead, it does not change the asymptotic complexity itself. The overall tracing complexity thus remains $O(N)$.

11.3 Type Inference

In this section we look into the complexity of the type inference algorithm presented in Section 5. In MemPick, type inference is the process of grouping objects into equivalence classes, called types. These equivalence classes do not necessarily correspond to the static types from source code. Each object is associated with a single type, while each type features a set of objects corresponding to it. MemPick leverages the definition of functional equivalence (as defined in Section 5) for this association. Whenever two objects are observed to be functionally equivalent, their types are merged. From an algorithmic perspective, this requires the two object sets to be merged as well as individual object types to be updated, resulting in a complexity linear with the smaller set size. Thus the worst case complexity occurs when equal sized sets are merged in a hierarchical fashion as presented in Fig. 6. Considering the total number of objects to be M, this merging process operates with a worst case complexity of $O(M * logM)$. This is beyond the level of complexity we desire for our algorithm. In practice we found the algorithm to be scalable, thus we perform a more accurate complexity analysis to check for a potential over-approximation in the previous results.

For a more pragmatic approach on complexity analysis, we will follow the progression starting from a newly created object. When this object is first used as an operand for an instruction involved in data structure manipulation, its type is immediately merged with all other previous operands of the same instruction. We denote this new type as a first level type, as presented in Fig. 6. Note, that every other object using the same instruction for the first time, will also be added to the same first level type. Thus, the potential number of first level types is bound by the different instructions that manipulate the data structure itself. These instructions are encapsulated within interface functions, and their numbers are directly proportional. In practice we never observed more than 10 interface functions for any data structure implementation. Since the type merging hierarchy from Fig. 6 is independent for each individual data structure type, its height will be bound by the constant number of interface functions for that given type. This observation reduces the complexity of type merging to $O(M)$.

The overall complexity of type inference also needs to consider the time it takes to inspect all instructions in the trace file. However this is independent of the merging operations themselves from a complexity standpoint. The number of instructions in the trace also dominates the number of objects created, thus the total complexity is maintained at $O(N)$.

11.4 Graph Generation

Now, we consider the complexity of maintaining a graph representation of the heap in memory and generating periodic snapshots. Graph representation and storage is a well studied research field, that offers a wide variety of alternatives, depending on the problem requirements. In the case of MemPick, the graph contains a large number of nodes, potentially up to M and is highly dynamic, with potentially $O(N)$ operations performed. MemPick employs adjacency lists for its internal representation, since matrix-based solutions are prohibitive due to the potential node count. While the theoretical complexity of node and edge removal in adjacency lists is $O(E)$ (where E is the number of edges), we can bound the practical one by $O(D)$ where D is the maximum in/out-degree of each graph node.

This complexity is still beyond our requirements, thus we attempt to bound the value of D in practice, using the constraints of our problem. Our graphs do not represent generic shapes, but well formed data structures from within applications. Each data structure node

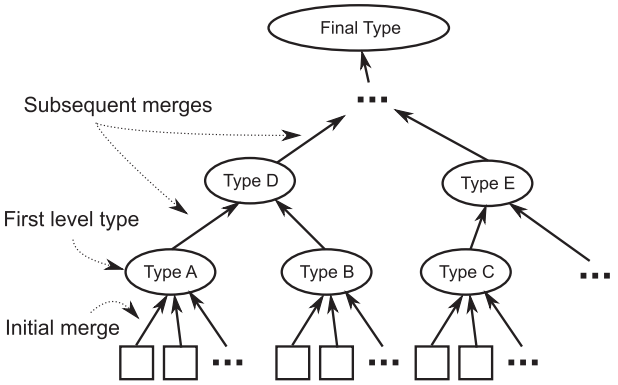


Fig. 6 Visual representation of type merging process for objects of the same type. Objects are initialized with first level types, based on the first instruction manipulating them. Subsequent instructions trigger additional merge operations, generating new types. Finally all objects are merged together into one final type

is a structure, which contains a fixed number of pointers, that represent outgoing edges. In practice all the data structures we observed, except B-trees, featured up to 12 incoming and outgoing edges.The B-tree family of data structures can possibly contain up to thousands of pointers in each node. Even in this corner-case, D can be considered a constant value, albeit a large one. In scenarios where large B-trees may significantly impact performance, adjacency lists can be replaced by hash tables. The latter trades space overhead for additional performance guarantees. Thus we observe that choosing the appropriate graph representation allows MemPick to maintain the graph in memory without incurring additional asymptotic complexity.

Besides the in-memory representation, MemPick is also required to provide periodic graph snapshots at quiescent periods. While generating an individual snapshot depends on the graph size ($O(M + E)$), the total number of quiescent periods is also trace dependent ($O(N)$). Thus we observe that it is not scalable to generate all possible graph snapshots. To mitigate this issue, the system selects a random sample containing K snapshots that are forwarded to graph analysis. The reverse engineer can control this number, based on the level of confidence desired. While in theory the accuracy of MemPick may be affected, in practice we did not observe such issues. The evaluation in Section 10.2 was performed using up to 100 snapshots for each data structure instance and none of the classification errors resulted from this limitations.

Combining all components of graph generation results in a complexity of $O(N + K * (M + E))$. The final formula is controlled by the value of K chosen by the reverse engineer. Constant values ensure that the complexity is still within the desired $O(N)$ class. Our evaluation shows that detection accuracy is not affected by this decision, but for the purpose of precise representation we will keep K as a variable for the rest of the analysis.

11.5 Shape Analysis

Finally, we analyze the complexity of the classification algorithm which operates on the generated snapshots to produce the final results. This algorithm consists of two components, namely overlay identification and rule-based classification of the overlays.

In Section 7.1, we described the notion of minimal pointer sets which define the possible overlays in a given data structure. Since this problem is analogous to the maximal set and set cover problems, the potential number of sets can be considered exponential in the potential pointer count. MemPick validates each candidate pointer set, by analyzing the connectivity in the K snapshots generated in the previous step. This process is performed in $O(K * (M + E))$ for each pointer set highlighting the necessity to deal with the exponential set count. Previously we observed that the number of potential pointers, D, is limited to a constant in practice, thus ensuring a constant number of potential sets.

The final shape analysis requires MemPick to analyze the in/out degrees for the discovered overlays. Classification itself is performed in a constant time, once the statistics are gathered for each snapshot. The resulting time complexity is $O(K * (M + E))$, identical to the result for overlay detection.

11.6 Summary of Complexity Analysis

In summary, by adding together all steps of MemPick, the total time complexity of data structure detection is $O(N + K * (M + E))$. Our evaluation has shown that

choosing a reasonable constant number for K does not impact the accuracy of MemPick regardless of the application under analysis. As such the time complexity formula is simplified to $O(N)$, since the trace size dominates both the potential number of memory objects and pointers. Going back to our research question, we have shown that it is possible to perform data structure detection using the same time complexity required for regular application execution. This ensures the scalability of our solution even for complex applications. While the constant overhead incurred is not negligible, hardware and parallel execution may be used in the future to mitigate it. Even now, the current implementation of MemPick finished the analysis of each application from our evaluation (Section 10.2) within one hour on a 2.6 GHz dual-core machine. This result and our complexity analysis proves that MemPick is applicable for the overnight analysis of these application classes.

12 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. (2013) 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 (Slowinska et al. 2011) or static analyses (Balakrishnan and Reps 2004; Ramalingam et al. 1999; Reps and Balakrishnan 2008) 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.

13 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 (Ghiya and Hendren 1996; Sagiv et al. 1999; Kuncak et al. 2006; Bogudlov et al. 2007; Zee et al. 2008) 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 (Balakrishnan and Reps 2004), aggregate structure identification (Ramalingam et al. 1999) and combinations thereof (Reps and Balakrishnan 2008). Some recent approaches such as Rewards (Lin et al. 2010), Howard (Slowinska et al. 2011), and TIE (Lee et al. 2011), 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. (Raman and August 2005), Laika (Cozzie et al. 2008), DDT (Jung and Clark 2009), and White et al. (2013).

Raman et al. (2005) 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 Raman and August (2005).

Laika (Cozzie et al. 2008) 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. (2007), SigGraph (Lin et al. 2011), and MAS (Cui et al. 2012), are all concerned with identifying data structures in memory dumps. However, they all rely on the type related information or debug symbol tables.

White and Lüttgen (2013) 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. (2006) 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 5) 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 (Jung and Clark 2009). DDT uses invariant information extracted using Daikon (Ernst et al. 2007). This allows DDT to go beyond shape information and to refine its classification based on content information. Unfortunately the same invariant detection also imposes additional assumptions on the system, reducing its flexibility. For one, DDT relies on well-structured detect interface functions which 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". However, this assumption does not always hold in practice.

14 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

Aydin B, Pataki G, Wang H, Bullit E, Marron J (2009) A principal component analysis for trees. Ann Stat 3(4):1597–1615

Balakrishnan G, Gruian R, Reps T, Teitelbaum T (2005) Codesurfer/x86—a platform for analyzing x86 executuables. In: Lecture notes in computer science, pp 250–254. Springer

Balakrishnan G, Reps T (2004) Analyzing memory accesses in x86 binary executables. In: Proceedings of the conference on compiler construction, CC'04

Bogudlov I, Lev-Ami T, Reps T, Sagiv M (2007) Revamping TVLA: making parametric shape analysis competitive. In: Proceedings of the 19th international conference on computer aided verification

Cadar C, Dunbar D, Engler D (2008) KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX symposium on operating systems design and implementation, OSDI'08

Chen X, Slowinska A, Bos H (2013) Detecting custom memory allocators in C binaries. Tech. rep., Vrije Universiteit Amstetrdam

Chipounov V, Kuznetsov V, Candea G (2011) S2E: A platform for in vivo multi-path analysis of software systems. In: Proceedings of the 16th international conference on architectural support for programming languages and operating systems, ASPLOS'11

Cormen TH, Stein C, Rivest RL, Leiserson CE (2001) Introduction to Algorithms

Cozzie A, Stratton F, Xue H, King ST (2008) Digging for data structures. In: Proceedings of USENIX symposium on operating systems design and implementation, OSDI'08

Cui W, Peinado M, Xu Z, Chan E (2012) Tracking rootkit footprints with a practical memory analysis system. In: Proceedings of the 21st USENIX conference on security symposium, SSYM'12

Engler D, Musuvathi M (2004) Static analysis versus software model checking for bug finding. In: Steffen B, Levi G (eds) Verification, model checking, and abstract interpretation. Lecture notes in computer science, vol 2937. Springer, Berlin, pp 191–210. doi:10.1007/978-3-540-24622-0_17

Ernst MD, Perkins JH, Guo PJ, McCamant S, Pacheco C, Tschantz MS, Xiao C (2007) The daikon system for dynamic detection of likely invariants. Sci Comput Program 69(1–3):35–45. doi:10.1016/j.scico.2007.01.015

Ghiya R, Hendren LJ (1996) 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, POPL'96. doi:10.1145/237721.237724

Godefroid P, Levin MY, Molnar DA (2008) Automated whitebox fuzz testing. In: Proceedings of the 15th annual network and distributed system security symposium, NDSS'08

Guo PJ, Perkins JH, McCamant S, Ernst MD (2006) Dynamic inference of abstract types. In: Proceedings of the 2006 international symposium on software testing and analysis, ISSTA'06. doi:10.1145/1146238.1146268

Haller I, Slowinska A, Bos H (2013) Mempick: High-level data structure detection in c/c++ binaries. In: Proceedings of the 20th working conference on reverse engineering (WCRE). Koblenz, Germany

Hotelling H (1933) Analysis of a complex of statistical variables into principal components. J Educ Psych 24

Intel (2011) Pin - A dynamic binary instrumentation tool. http://www.pintool.org/

Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don't software developers use static analysis tools to find bugs? In: Proceedings of the 2013 international conference on software engineering, ICSE '13, pp 672–681. IEEE Press, Piscataway. http://dl.acm.org/citation.cfm?id=2486788.2486877

Jung C, Clark N (2009) 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

Jung C, Rus S, Railing BP, Clark N, Pande S (2011) Brainy: Effective selection of data structures. In: Proceedings of the 32Nd ACM SIGPLAN conference on programming language design and implementation, PLDI '11, pp 86–97. ACM, New York. doi:10.1145/1993498.1993509

Kuncak V, Lam P, Zee K, Rinard M (2006) Modular pluggable analyses for data structure consistency. IEEE Trans Softw Eng 32(12). doi:10.1109/TSE.2006.125

Lee J, Avgerinos T, Brumley D (2011) TIE: Principled reverse engineering of types in binary programs. In: Proceedings of the 18th annual network & distributed system security symposium, NDSS'11

Lin Z, Rhee J, Zhang X, Xu D, Jiang X (2011) SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures. In: Proceedings of 18th annual network & distributed system security symposium, NDSS'11

Lin Z, Zhang X, Xu D (2010) Automatic reverse engineering of data structures from binary execution. In: Proceedings of the 17th annual network and distributed system security symposium, NDSS'10

Marinescu PD, Cadar C (2012) Make test-zesti: A symbolic execution solution for improving regression testing. In: Proceedings of the 34th international conference on software engineering, ICSE '12. IEEE Press, Piscataway, pp 716–726. http://dl.acm.org/citation.cfm?id=2337223.2337308

Marinescu PD, Cadar C (2013) Katch: High-coverage testing of software patches. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering, ESEC/FSE 2013, pp. 235–245. ACM, New York. doi:10.1145/2491411.2491438

Polishchuk M, Liblit B, Schulze CW (2007) Dynamic heap type inference for program understanding and debugging. In: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL'07. doi:10.1145/1190215.1190225

Ramalingam G, Field J, Tip F (1999) Aggregate structure identification and its application to program analysis. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on principles of programming languages. doi:10.1145/292540.292553

Raman E, August DI (2005) Recursive data structure profiling. In: Proceedings of the 2005 workshop on memory system performance, MSP'05. doi:10.1145/1111583.1111585

Reps T, Balakrishnan G (2008) 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, CC'08/ETAPS'08

Rossow C, Andriesse D, Werner T, Stone-Gross B, Plohmann D, Dietrich CJ, Bos H (2013) P2PWNED: Modeling and evaluating the resilience of peer-to-peer botnets. In: Proceedings of the 34th IEEE symposium on security and privacy (S&P). San Francisco, CA

Sagiv M, Reps T, Wilhelm R (1999) Parametric shape analysis via 3-valued logic. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL'99. doi:10.1145/292540.292552

Slowinska A, Stancescu T, Bos H (2011) Howard: a dynamic excavator for reverse engineering data structures. In: Proceedings of the 18th annual network & distributed system security symposium, NDSS'11

Slowinska A, Stancescu T, Bos H (2012) Body armor for binaries: Preventing buffer overflows without recompilation. In: Proceedings of USENIX annual Technical conference, USENIX ATC'12

Szeredi M File system in user space. http://fuse.sourceforge.net

Wang H, Marron JS (2007) Object oriented data analysis: Sets of trees. Ann Stat 35(5):1849–1873

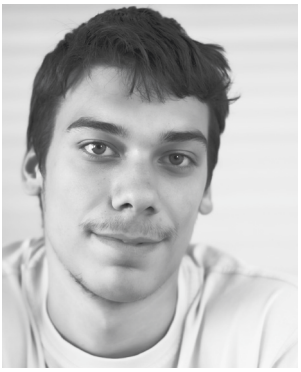
White DH, Lüttgen G (2013) Identifying dynamic data structures by learning evolving patterns in memory. In: Proceedings of the 19th international conference on tools and algorithms for the construction and analysis of systems, TACAS'13. doi:10.1007/978-3-642-36742-7

Wyk CJV (1991) Data structures and C programs, 2nd Ed. (Addison-Wesley series in computer science), 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston

Yang H, Lee O, Berdine J, Calcagno C, Cook B, Distefano D, O'Hearn PW (2008) Scalable shape analysis for systems code. In: Gupta A, Malik S (eds) CAV, Lecture notes in computer science, vol 5123, pp 385–398. Springer. http://dblp.uni-trier.de/db/conf/cav/cav2008.html#YangLBCCDO08

Zee K, Kuncak V, Rinard M (2008) Full functional verification of linked data structures. In: Proceedings of the 2008 ACM SIGPLAN conference on programming language design and implementation, PLDI'08. doi:10.1145/1379022.1375624

Zhang M, Sekar R (2013) Control flow integrity for cots binaries. In: Proceedings of the 22nd USENIX Conference on Security, SEC'13, pp 337–352. USENIX Association, Berkeley. http://dl.acm.org/citation.cfm?id=2534766.2534796



Istvan Haller is a PhD student in the Systems and Network Security group at the Vrije Universiteit Amsterdam. His current research focuses on automatic analysis of software systems and its application to enhance system security.

Asia Slowinska is an assistant professor in the Systems and Network Security group at the VU Amsterdam. Her work focuses on developing techniques to automatically analyze and reverse engineer complex software that is available only in binary form. Further, she looks into mechanisms that proactively protect software from malicious activities.

Herbert Bos is a full professor in Systems and Network Security at Vrije Universiteit Amsterdam. He obtained his Ph.D. from Cambridge University Computer Laboratory (UK). He is very proud of all his (former) students, three of whom have won the Roger Needham Ph.D. Award for best Ph.D. thesis in systems in Europe. In 2010, Herbert was awarded an ERC Starting Grant for a project on reverse engineering that is currently keeping him busy.