dsOli2: Discovery and Comprehension of Interconnected Lists in C Programs

David H. White, Thomas Rupprecht and Gerald Lüttgen

Software Technologies Research Group, University of Bamberg, 96045 Germany

Abstract. Comprehension of C programs can be a challenging task, especially when they contain pointer-based dynamic data structures. Based on prior experience with our tool dsOli1, we report on work in progress concerning a new dynamic analysis for automated data structure identification that targets C source code. Our technique first applies a novel abstraction on the evolving memory structures observed at runtime for discovering the building blocks of complex data structures. By analyzing the interconnections between the building blocks, we are then able to identify trees, doubly-linked lists, skip lists, as well as relationships between these such as nesting. We give preliminary results from a prototype implementation, which aims to provide a natural language description of the identified data structures. This information will benefit software developers when code must be comprehended or modified.

1 Introduction

C programs are notoriously difficult to comprehend, and this is especially true for legacy or low-level code, e.g., that found in OSs or device drivers. In such situations it is not uncommon to see programmers employ complex usages of pointers, types and memory allocation to achieve the desired behavior or efficiency. These constructs are often used to implement the dynamic data structures of a program, and thus data structures can form a major obstacle in program comprehension, optimization and verification. To partially alleviate this obstacle we propose a dynamic analysis for automatic identification of dynamic data structures in C programs.

The essence of our analysis is to first discover the building blocks of complex data structures, which are essentially singly linked lists (SLLs), and then to analyze any relationships that exist between the lists. Lists may be either *tightly connected*, where they comprise some part of a more complex data structure, e.g., the two lists running in reverse directions through a doubly-linked list (DLL), or *loosely connected*, where they describe relationships between specific data structures, e.g., the parent-child relationship found in nested lists.

The identification of dynamic data structures is made challenging due to manipulation operations that temporally transform a *stable shape* into a *degenerate shape*. For example, consider how the key feature of a DLL is broken during the insertion of a node; if one were to inspect the shape at such an intermediate state, then it may be difficult to give the correct *label*, i.e., name of the data structure. Approaches such as dsOli1 [11] and DDT [7] handle this by trying to find data structure operation boundaries, while MemPick [5] attempts to perform identification only in the quiescent periods of a data structure. In both cases, identification is performed when one can be reasonably sure the data structure has a stable shape.

In our work we include degenerate shapes but override their influence by observing the *context* in which a shape appears. Context arises from two sources: *structural repetition*, which occurs when there exist many structures performing the same role, e.g., the multiple child lists found in parent-child nested lists, and *temporal repetition*, which occurs when the same structures exist over multiple program time steps. By *discovering evidence* for specific occurrences of data structures and then *reinforcing* this evidence through structural and temporal repetition, our approach enables identification even when temporary degenerate shapes are encountered. To illustrate the utility of our approach, we track variables that represent entry points to dynamic data structures and aim to annotate these with natural language descriptions of the reachable data structure, e.g., "Entry point `p` points to a skip list with a parent child nesting to DLLs".

The remainder of this paper is organized as follows. In Sec. 2 we discuss the complexities of data structures in C heaps, which motivates many of the design decisions we have made for dsOli2. Sec. 3 describes our approach from a high level with an illustrative example, and in Sec. 4 we dive into the details. We report preliminary results in Sec. 5 obtained from our prototype implementation, and finally present conclusions and future work in Sec. 6.

Related Work. Our dynamic analysis aims to identify data structures but provides no soundness guarantee. In contrast, modern shape analysis tools, such as Predator [4] and Forester [6], are sound and employ symbolic execution to learn shape predicates that allow memory safety to be checked automatically. In particular, Forester summarizes repetitive graph structures with forest automata to handle skip lists and trees. However, neither approach can handle the recursion commonly found in tree operations, and as their focus is memory safety, it is not clear how naturally the learnt shape predicates fit the goal of program comprehension.

The dynamic analyses HeapDbg [8] and ARTISTE [3] represent multiple concrete data structure nodes with a single abstract node, which is in turn checked for interesting shape properties. Our approach shares much in common with the techniques of HeapDbg: their summarization process employs structural and temporal repetition, but as the process is conservative, temporal joins force the label of an abstract node to be reduced to the most general available. While this works for HeapDbg's tree label, Artiste includes DLLs and, if temporal joins were to be performed, then the precision of the DLL label would be lost.

MemPick [5] functions on object code and excels in distinguishing different types of trees. In contrast, we require source code but handle skip lists and produce a much richer description of the connections between data structures. Finally, dsOli1 [11] and DDT [7] go beyond all these approaches in that they

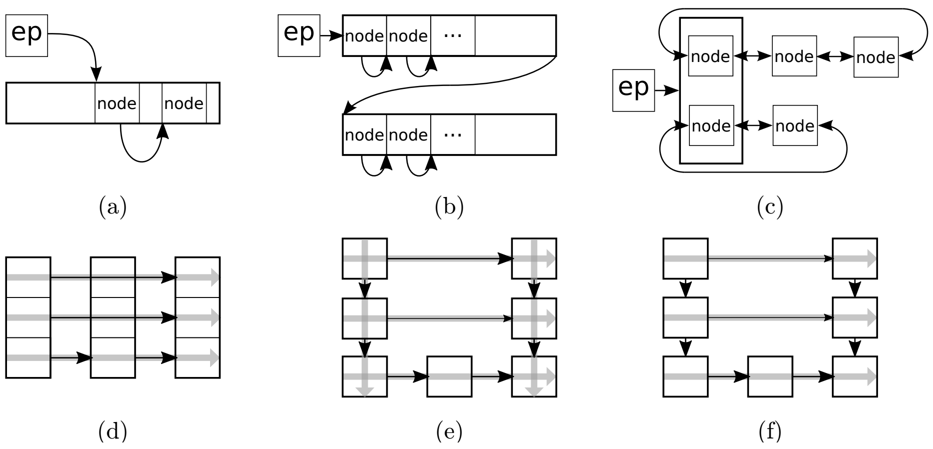

Fig. 1. Complexities of C heaps: (a) custom allocator, (b) cache efficient list [2], (c) Linux kernel DLL [1], (d-f) skip lists where building block lists are indicated by bold gray arrows. Examples of (d) and (e) appear in `tests/skip-list/jonathan-skip-list.c` and `tests/forester-regre/test-f0021.c` of Predator [4], respectively.

also seek to discover the operations that manipulate the data structures. DDT accomplishes this by assuming that data structures are accessed via well-defined interface functions, while dsOli1 employs a machine learning approach to locate repetitive code segments indicative of operations. Compared to dsOli1, we currently don't consider operations but do expand the variety of data structures that are in-scope considerably.

2 Heap Usage in C Programs

The type safety of modern programming languages such as Java and C# constrains the actions that a programmer may take and results in programs having relatively well structured heaps. However, in languages frequently used for OS programming such as C, where pointer arithmetic and type casting may be freely applied and memory management is in the hands of the programmer, the heap can be formed in a more ad-hoc manner. In this section we describe some of the challenging C code we have seen in practice that leads us to this conclusion, and in the next section we outline how our approach copes with this challenge. Firstly, we briefly introduce the notion of a *points-to graph*, which describes a snapshot of program memory by representing *memory chunks*, i.e., stack/global variables and dynamically allocated memory, as vertices and *pointers* as edges.

A typical assumption is that a memory chunk represents a single node of a data structure; however, in practice this is broken in a number of situations. Firstly, if a custom memory allocator is employed, but memory chunks are detected at the level of the system memory allocator, then it may be the case that multiple nodes of potentially multiple data structures appear in the same memory chunk (Fig. 1(a)). Secondly, cache-efficient data structures combine multiple nodes into a single memory chunk to enhance performance (Fig. 1(b)). Thirdly, head nodes of multiple lists may be embedded in the same memory chunk (Fig. 1(c)). This is common practice with the cyclic DLL type `struct list_head` employed by the Linux kernel [1], which is designed to be embedded inside another struct. Given this *cyclic* property, a natural interpretation is to treat the head node uniformly with the remainder of the list. This gives rise to an alternative view, i.e., as a list where the nodes occupy memory chunks of varying sizes. In the above case, a list of length n consists of one node in a memory chunk of type t_1 and $n-1$ nodes each in a memory chunk of type t_2. Macros are provided that allow the outer struct to be reached from a list head struct via pointer arithmetic and casting.

The key insight to model all of the above situations uniformly is to relax the assumption that a list linkage offset should occur at a fixed offset from the memory chunk start address. Thus, it is necessary to track lists in terms of their linkage rather than in terms of memory chunk type. In the next section we show how our approach handles this by determining the minimal subregions of memory chunks needed to establish list linkage.

Now that we have discussed the complexities surrounding list formation, we turn to how lists are connected. A connection may be made either by *overlay*, where at least one node from each list occupies the same memory chunk, or by *indirection*, where there exists a pointer, or a chain of pointers, from the memory chunk holding the node of one list to a memory chunk holding a node of another list. To illustrate this we consider possible skip list constructions. Firstly, if the number of levels are known *a priori*, then it is common to employ a memory chunk with an array of linkages, where the array element at index i represents the linkage to the next node at level i (Fig. 1(d)). Thus, in situations where multiple levels run through the same node, these are connected by overlay. Secondly, all nodes in the skip list may be of the same type; in other words, each memory chunk has a `next` pointer to the next node of the level it represents and a `down` pointer to the level below (Fig. 1(e)). Since all nodes are of the same type, atomic lists are formed both in the horizontal and vertical directions and are again connected by overlays. Lastly, consider a skip list where each level is represented by a node of different type (Fig. 1(f)). Since only the horizontal linkage forms lists, the downward link is an indirect connection between lists.

In the next section we show how our approach uniformly handles the variety of implementation techniques that may be employed by firstly gathering evidence and then employing structural and temporal repetition to consolidate the acquired evidence.

3 Overview of our Approach

In this section we give an overview of our approach and provide motivation with the simple example in Fig. 2, which also shows our approach as a pipeline.

Table 1. Memory structures (with abbreviations) in-scope for our approach, plus the number of strands required for discovery and the priority in the identification phase. Memory structures typically have several implementations, in the case of Head/Tail Pointers this affects the category. Sharing denotes two lists which share a downstream cell sequence, while intersecting lists is a catch-all for any pair of connected strands.

Data Structure	**# Strands Required**	**Priority**	**Connection**	**# Strands Required**	**Priority**
(Cyclic) SLL	1	-	Head/Tail Ptrs. (HT)	1 or 2+	3
(Cyclic) DLL	2	1	Parent Pointers (PP)	2+	4
Tree	2+	2	Intersecting Lists (IL)	2	7
Skip List (SL)	2+	5	Nesting (N)	2	8
Grid	2+	6	Sharing	2	9

The example shows two time steps in the construction of a SLL of DLLs; note that at time step t, there exists a degenerate DLL child. In favor of a succinct explanation, details are delayed until Sec. 4.

We commence from the classic definition of an SLL, which is a sequence of memory chunks all of the same type, where the entirety of each chunk constitutes one node in the list. A subset of pointers between these chunks fulfill a *linkage condition*, which states that all pointers originate at the same *linkage offset* from the start of the chunk and terminate at the start address of the next chunk.

Strands. To handle the scenarios outlined in Sec. 2, we relax the notion that the nodes of the list occupy the whole memory chunk, and instead try to discover what we term *strands*, which will form the basic building blocks of the structures we seek to identify. A strand represents a sequence of *subregions of memory chunks*, each termed a *cell*, such that the same linkage condition can be established between the cells. Thus, the linkage offset is now given relative to the start address of a cell. Strands (S_i) are indicated by bold arrows in Fig. 2(a).

Strand Connections. Our approach is driven by relationships between strands, which we term *strand connections*. Each strand connection describes exactly *one* way in which the cells of two strands are related, hence multiple strand connections between a pair of strands are possible. Merging strand connections that describe the same relationship will be of key importance in the accumulation of evidence, and we define strand connections with offsets relative to the cells in order to handle the scenarios of Sec. 2. We construct a *strand graph* where vertices represent strands and edges represent strand connections, see Fig. 2(b). Since only two time steps of the program are considered in the illustrative example, it is unsurprising that both strand graphs have the same structure. For now note that strand connections with the same edge style denote the same relationship type; for example, the DLL strands form a bi-directional overlay connection, while two kinds of uni-directional indirect connections are formed between the parent SLL and each child DLL.

Memory Structures. We use the term *memory structure* to speak collectively about data structures and connections between data structures, i.e., both the tight and loose connections mentioned in Sec. 1. The list of memory

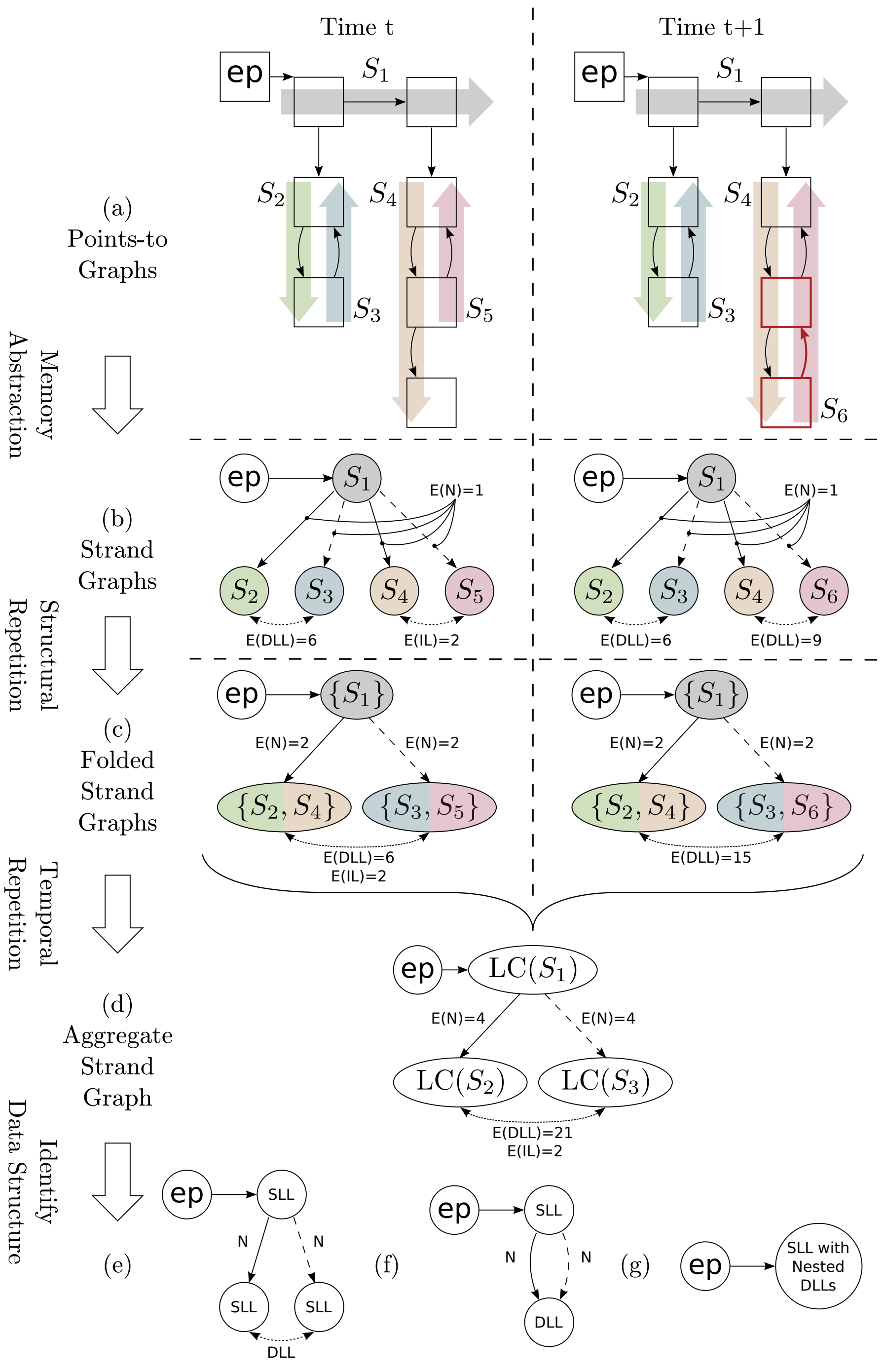

Fig. 2. Left: the pipeline of our approach, right: the illustrative example of Sec. 3.

structures in-scope for our approach are given in Table 1 and are categorized based on the number of strands required for their discovery. For example, a DLL requires exactly two strands to be discovered, while a skip list requires two or more strands. We now refer to memory structures requiring one or two strands as Category 1/2 and those requiring more than two as Category 2+. We will see later that memory structures in the Category 2+ are discovered at a later stage of the approach than those in Category 1/2.

Evidence Gathering. When a strand connection is found we immediately determine the supporting evidence for that strand connection for each Category 1/2 memory structure. The evidence is weighted by the number of cells and connections between cells that must be present for that memory structure to be correctly identified. Essentially, our goal is to count the number of things that have gone "right" for such a memory structure to exist and use this for evidence. For example, the weight of evidence gathered for nested lists on overlays, where the strands must only intersect in one memory chunk, is much weaker than that for DLLs, where the strands must form a very specific connection. Non-zero evidence is shown on the strand connections of Fig. 2(b). The degenerate DLL in the first time step has an evidence count of 2 for Intersecting Lists (IL); in this case, evidence is simply the number of overlay connections between the cells of each strand. When the DLL regains the correct shape at time $t + 1$, it has an evidence count of 9 based on the length of both composite strands $(3 + 3)$ and the number of intersection points (3). Strand connections describing nesting (N) have an evidence count of 1 as the connection is made by a single pointer.

Structural Repetition. The primary use of structural repetition is to group elements of the strand graph that perform the same role within one program time step. This grouping is realized via a merge algorithm that results in a *folded strand graph* and, since this contains merged strand connections, it serves to reinforce the evidence of Category 1/2 memory structures. Observe in Fig. 2(c) that the vertices have now become *sets* of strands. Merging partially addresses the problem of degenerate shapes, i.e., if strands with the correct shape can be grouped with those having degenerate shapes, then the majority can override the minority. The correct shape is generally in the majority since degenerate shapes are produced by manipulations that typically only have a local effect. In Fig. 2(c), this is seen between strands $\{S_2, S_4\}$ and $\{S_3, S_5\}$ at time t.

With the folded strand graph to hand, the identification of Category 2+ memory structures begins. For any suitable subgraph in the folded strand graph, it is checked if that subgraph has the property required of the corresponding Category 2+ memory structure. If found to be true, all strand connections that comprise that memory structure record the associated evidence.

Temporal Repetition. To track the temporal behavior of a memory structure and enable the identification of temporal repetition, we must determine which strands represent the same atomic component of a data structure over multiple time steps. This is a very difficult task to do globally as lists will be split, joined, created and deleted at runtime, and any labeling system will end up with some amount of discontinuity. Instead, we tackle this problem by considering the labeling from the point of view of each entry point separately, since entry points are inherently stable over their lifetimes. For each time step that an entry point exists, we extract the subgraph of the folded strand graph reachable from that entry point. The subgraphs are then merged into an *aggregate strand graph*, and thus temporal repetition is identified whenever multiple graph elements are merged together. Naturally, the evidence embedded in those elements is also merged and, hence, evidence for both Category 1/2 and Category 2+ memory structures is reinforced, further reducing the effect of degenerate shapes. Vertices of this graph become abstract descriptions of the original strands in terms of their *linkage conditions* (LC). The aggregate strand graph is shown in Fig. 2(d); note that the evidence for the DLL shape is overwhelming.

Identifying Memory Structures. The final barrier for memory structure identification arises from the fact that there may be several possible interpretations of the aggregate strand graph. We resolve this as follows: we first set the label of each strand connection in the aggregate strand graph to the one with the most evidence and set the label of all vertices to be `SLL` (Fig. 2(e)). We then group graph elements according to the priorities given in Table 1 and assign a textual label to the group. For example, DLLs have priority 1, so strand connections labeled DLL and their associated strands are grouped first (Fig. 2(f)). These elements then form an atomic vertex in subsequent groupings. Ultimately, we end up with a graph (Fig. 2(g)) of one atomic vertex with a textual label describing the whole data structure reachable from the entry point.

4 Details of our approach

We now formalize the concepts presented in the illustrative example of Sec. 3.

4.1 Memory Abstraction

To identify the data structures employed by the program we reconstruct a sequence of points-to graphs $\langle G_0^{pt}, \ldots, G_n^{pt} \rangle$ from an execution of the program under analysis. This reconstruction is enabled by first instrumenting the program, which results in the runtime capture of *program events* such as pointer writes and dynamic memory (de)allocation. The result of the program event at *time step* t is captured by G_t^{pt}, where $1 \leq t \leq n$ and G_0^{pt} is empty.

Definition 1. *A* **points-to graph** $G^{pt} = (\mathcal{V}, \mathcal{E})$ *is a directed graph comprising a vertex set* $\mathcal{V}$ *representing memory chunks and an edge set* $\mathcal{E} \subseteq \mathcal{V} \times \mathbb{N} \times \mathcal{V} \times \mathbb{N}$ *representing pointers.*

An edge $(v_s, a_s, v_t, a_t) \in \mathcal{E}$ captures the points-to relationship between two memory chunks established by a pointer with source address a_s, encapsulated by vertex v_s, and target address a_t, encapsulated by vertex v_t. A memory chunk is either a *heap chunk* (a memory region returned from dynamic memory allocation, e.g., `malloc`) or a *stack/global chunk*. Our points-to graphs only consist of reachable memory, so if a leak occurs, then all unreachable chunks are removed.

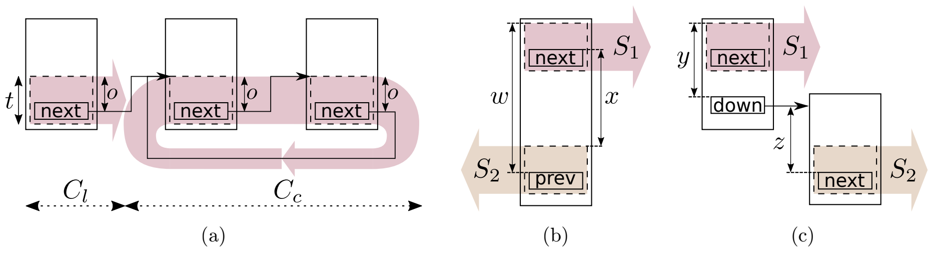

Fig. 3. Details of (a) a strand $S = ((t, o), (C_l, C_c))$, (b) an overlay strand connection $S_1 \xleftrightarrow{wx} S_2$ and (c) an indirect strand connection $S_1 \xrightarrow{yz} S_2$. Memory chunks have black outline, cells are dashed, and strands are indicated with large transparent arrows.

All memory chunks are typed with standard C types, and a heap chunk becomes typed when it is accessed by a non void * pointer. Usages of a memory chunk must be typed consistently, i.e., if a memory address a is accessed via pointer types `t1*` and `t2*`, then `t1` and `t2` must be structurally equivalent. Since structs may be nested, and thus multiple structs may start at an address, the function $\textsc{type}(a)$ returns the set of types starting at address a.

Definition 2. *A stack/global chunk $v \in \mathcal{V}$ is an* **entry point** *if it (a) contains any pointer variable with a target address in the heap or (b) contains a strand cell (e.g., holds the "head" node in a list).*

We begin the formalization of a strand using a pointer that establishes a linkage condition between two cells, see Fig. 3(a) for details in the following. Set operators with a bar, $\bar{\in}$, $\bar{\subseteq}$ and $\bar{\cap}$, function on memory ranges, e.g., $a \mathrel{\bar{\subseteq}} b$ determines if the range of a is included in the range of b.

Definition 3. *A* **cell** *c is a subregion of a memory chunk, i.e., $\exists v \in \mathcal{V} : c \mathrel{\bar{\subseteq}} v$, which begins at address $c.bAddr$ and ends at address $c.eAddr$.*

Definition 4. *A* **linkage condition** *$L = (t, o)$ exists between two cells $c_s \xrightarrow{L} c_t$ with cell type t and linkage offset o if:*

$$\exists(\_, a_s, \_, a_t) \in \mathcal{E} : a_s \mathrel{\bar{\in}} c_s \wedge a_t = c_t.bAddr \wedge o = a_s - c_s.bAddr$$
$$\wedge\, t \in \textsc{type}(c_s.bAddr) \cap \textsc{type}(c_t.bAddr) \wedge c_s \mathrel{\bar{\cap}} c_t = \varnothing.$$

We are interested in the maximal linkage condition, i.e., choose L such that the length of the sequence of cells $c_1 \xrightarrow{L} c_2 \xrightarrow{L} c_3 \ldots$ is maximized. If more than one maximal L exists, then we choose the one with the type t of smallest size.

Definition 5. *A* **strand** *$S = (L, C)$ represents the sequence of cells C captured by a maximal linkage condition L. The cell sequence $C = (C_l, C_c)$ is divided into an optional linear start C_l and an optional cyclic tail C_c, although at least one must be non-empty. When both sequences are non-empty, the following holds:*

$$\forall i \in [1..|C_l|-1] : C_l[i] \xrightarrow{L} C_l[i+1] \;\wedge\; C_l[|C_l|] \xrightarrow{L} C_c[1]$$
$$\wedge\; \forall i \in [1..|C_c|-1] : C_c[i] \xrightarrow{L} C_c[i+1] \;\wedge\; C_c[|C_c|] \xrightarrow{L} C_c[1].$$

Strands are created to capture every unique sequence of cells and are not destroyed unless all their component cells cease to exist.

As the key to our approach is the reinforcement of evidence via grouping elements that perform the same role, we must ensure that identical strand connections may be found and grouped wherever possible. Thus, due to the issues of Sec. 2, all strand connection parameters (w, x, y and z in the following) are given relative to the cells, linkage pointers and target addresses, i.e., quantities that are independent of a cell's position in a memory chunk (see Figs. 3(b) & (c)). It is for this reason that the strand connections of Fig. 2 are drawn with different line styles, those with the same style have identical parameters. Lastly, note that indirect connections can be generalized to sequences of pointers.

Definition 6. *A* **strand connection** $S_1 \overset{\alpha}{\cdots} S_2$ *describes exactly one way in which a subset of the cells of* S_1 *are related to a subset of the cells of* S_2*. A connection is defined by the cells that establish the relationship:* $\textsc{pairs}(S_1 \overset{\alpha}{\cdots} S_2) = \{(c_1, c_2) \in \textsc{cells}(S_1) \times \textsc{cells}(S_2) : c_1 \overset{\alpha}{\cdots} c_2\}$*. The relationship between cell pairs (and by extension between strands) may be (a) overlay* $c_1 \overset{wx}{\longleftrightarrow} c_2$ *if* $\textsc{vertex}(c_1) = \textsc{vertex}(c_2)$ *with parameters* $w = (c_2.bAddr + \textsc{linkageOffset}(S_2)) - c_1.bAddr$ *and* $x = (c_1.bAddr + \textsc{linkageOffset}(S_1)) - c_2.bAddr$*. Alternatively, (b) indirect* $c_1 \xrightarrow{yz} c_2$ *if* $\exists e = (v_s, a_s, v_t, a_t) \in \mathcal{E} : v_s \neq v_t \wedge v_s = \textsc{vertex}(c_1) \wedge v_t = \textsc{vertex}(c_2)$ *and there is no linkage condition on* e*. In this case, the parameters are:* $y = a_s - c_1.bAddr$ *and* $z = (c_2.bAddr + \textsc{linkageOffset}(S_2)) - a_t$*.*

To uniquely track the strands reachable from an entry point over multiple time steps, we introduce *entry point connections* for each type of entry point given in Def. 2. These are essentially specialized strand connections, where the starting offset is given from the memory chunk's start address and is therefore absolute. Thus, when a chain of strand connections are followed by their relative offsets, the chain is still uniquely identifiable due to the absolute offset of the initial entry point connection.

Definition 7. *An* **entry point connection** $v_{ep} \xrightarrow{xy} S$ *from an entry point* $v_{ep} \in \mathcal{V}$ *of type Def. 2(a) to a cell* $c \in \textsc{cells}(S)$ *via a non-linkage condition edge* $e = (v_{ep}, a_s, v_t, a_t) \in \mathcal{E}$ *is defined by two parameters:* $x = a_s - v_{ep}.bAddr$ *and* $y = (c.bAddr + \textsc{linkageOffset}(S)) - a_t$*. An entry point connection* $v_{ep} \xrightarrow{z} S$ *from an entry point* $v_{ep} \in \mathcal{V}$ *of type Def. 2(b) to a cell* $c \in \textsc{cells}(S)$ *such that* $c \mathrel{\bar{\subsetneq}} v_{ep}$ *is defined by one parameter:* $z = (c.bAddr + \textsc{linkageOffset}(S)) - v_{ep}.bAddr$*.*

Definition 8. *A* **strand graph** $G^s = (\mathcal{V}^s, \mathcal{E}^s)$ *is composed of a vertex set* $v \in \mathcal{V}^s$*, where* v *represents either a strand or an entry point, and an edge set* $e \in \mathcal{E}^s$*, where* e *represents either a strand connection or an entry point connection.*

4.2 Evidence Discovery and Reinforcement

With the strand graph for each time step to hand, we proceed to discover and reinforce evidence for the memory structures of Table 1.

Definition 9. *A* **memory structure** $M = (L, P_{shape}, P_{area}, E)$ *has a label* L*, a shape predicate* P_{shape} *to enable discovery of* L*, an area condition* P_{area} *that describes on which graph elements* P_{shape} *is checked, and an evidence count* E*.*

An area condition serves two purposes, firstly, to limit the number of locations that a shape predicate must be checked, and secondly, to expose to the shape predicate only the subset of graph elements necessary for discovery. Such elements include the set $\mathcal{C}'$, containing the cells of all strands mentioned in the area condition, and the set $\mathcal{E}'$, containing all edges that form the linkage of the strands and all pointers included in strand connections mentioned in the area condition. For Category 1/2 memory structures, P_{area} simply limits whether P_{shape} applies to strands connected by an overlay or an indirect strand connection; however, later we will present a shape predicate employing $\mathcal{C}'$ and $\mathcal{E}'$.

During the construction of the strand graph, for each strand connection matching P_{area}, the associated shape predicate P_{shape} is tested. If found to be true, then the pair (L,E) is added to the strand connection identified by P_{area}. We now give concrete examples of these concepts for selected Category 1/2 memory structures:

$$L = \textsc{sharing}, P_{\text{area}} = S_1 \overset{xy}{\longleftrightarrow} S_2 \wedge x = y, E = |\textsc{pairs}(S_1 \overset{xy}{\longleftrightarrow} S_2)|, P_{\text{shape}} = \text{true}$$

$$\begin{aligned} L = \textsc{intersectingLists}, P_{\text{area}} &= S_1 \overset{\alpha}{\cdots\cdots} S_2, E = |\textsc{pairs}(S_1 \overset{\alpha}{\cdots\cdots} S_2)| \\ P_{\text{shape}} &= |\textsc{pairs}(S_1 \overset{\alpha}{\cdots\cdots} S_2)| \geq 1 \wedge \neg\textsc{dll}(S_1 \overset{\alpha}{\cdots\cdots} S_2) \wedge \neg \ldots \end{aligned}$$

$$\begin{aligned} L = \textsc{dll}, P_{\text{area}} &= S_1 \overset{xy}{\longleftrightarrow} S_2, E = |\textsc{pairs}(S_1 \overset{xy}{\longleftrightarrow} S_2)| * 3 \\ P_{\text{shape}} &= |\textsc{cells}(S_1)| = |\textsc{cells}(S_2)| = |\textsc{pairs}(S_1 \overset{xy}{\longleftrightarrow} S_2)| \\ &\quad \wedge \textbf{let } (\_, (C_l^1, C_c^1)) = S_1 \wedge (\_, (C_l^2, C_c^2)) = S_2 \textbf{ in } C_c^1 = C_c^2 = \emptyset \\ &\quad \wedge \forall i \in [0..\textsc{length}(S_1) - 1]\ \exists (c_1, c_2) \in \textsc{pairs}(S_1 \overset{xy}{\longleftrightarrow} S_2) : \\ &\quad\quad C_l^1[i+1] = c_1 \wedge C_l^2[\textsc{length}(S_2) - i] = c_2 \end{aligned}$$

Sharing describes two lists that share a downstream cell sequence, while INTERSECTINGLISTS is a catch-all predicate that matches any pair of connected strands. As such, most memory structures must be explicitly excluded in its P_{shape} to prevent unnecessary evidence being produced. While SHARING and INTERSECTINGLISTS generate evidence in the number of connection points between the two strands, the DLL predicate requires two strands to be connected in a specific way and, thus, uses the length of each strand summed with the number of connection points as evidence. Lastly, note that DLL is easily extended to cyclic DLLs by requiring S_1 and S_2 to have cyclic cell sequences and checking that, under some cyclic permutation of those sequences, the DLL property holds.

With the evidence for Category 1/2 memory structures added to the strand graph, we proceed to identify structural repetition via Alg. 1. This serves two

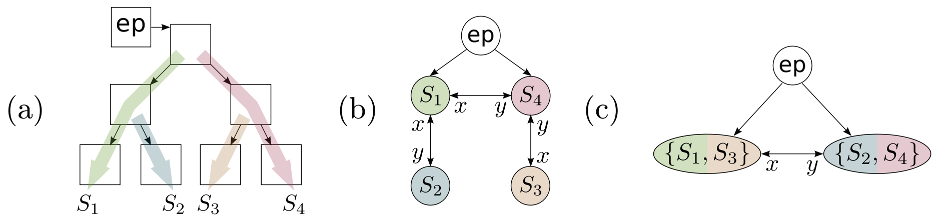

Fig. 4. Tree example: (a) points-to graph, (b) strand graph and (c) folded strand graph.

purposes in our approach. Firstly, it reinforces evidence of Category 1/2 data structures and, thus, alleviates the degenerate shape problem. Secondly, it partially groups the graph elements of Category 2+ data structures, which facilitates their discovery.

Algorithm Sketch 1. Structural repetition *is found by successively locating strands of the strand graph that conceptually perform the same role, and then merging them. Any duplicate strand connections that result from the strand merge are also merged, thus summing any associated evidence. Two strands S_1 and S_2 are merged if they have the same linkage condition L and there exists a merge point S_3 with strand connections $S_3 \overset{\alpha}{\cdots} S_1 \wedge S_3 \overset{\alpha}{\cdots} S_2$. A strand connection between S_1 and S_2 is forbidden unless it describes sharing. Alternatively, two strands S_1 and S_2 are merged if they appear in a disjoint partition of the strand graph, where the only connections between strands of that partition describe sharing. After all merges are performed, the result is a folded strand graph.*

Definition 10. *A* **folded strand graph** G^{fs} *is a summarization of a strand graph. The vertices now represent entry points or* sets *of strands. Edges represent entry point connections or* merged *strand connections.*

Area conditions for the discovery of Category 2+ data structures may describe sets of strands, and for convenience these memory structures are discovered in the folded strand graph. Consider the binary tree shown in Fig. 4(a), which has strands covering the left and right linkages; the associated strand graph is shown in Fig. 4(b). This data structure displays high structural repetition and, in the folded strand graph Fig. 4(c), the strands have been grouped into two classes representing the left and right linkages. As can be seen in the following, P_{area} for a binary tree recognizes the shape of Fig. 4(c) exactly:

$$\begin{aligned}
&L = \textsc{BinaryTree}, P_{\text{area}} = \mathcal{S}_1 \overset{xy}{\longleftrightarrow} \mathcal{S}_2, E = |\textsc{pairs}(\mathcal{S}_1 \overset{xy}{\longleftrightarrow} \mathcal{S}_2)| \\
&\quad P_{\text{shape}} = \textbf{let } \exists \mathcal{E}'_1, \ldots, \mathcal{E}'_n : \mathcal{E}' = \uplus_{i=1}^{n} \mathcal{E}'_i \wedge \exists \mathcal{C}'_1, \ldots, \mathcal{C}'_n : \mathcal{C}' = \uplus_{i=1}^{n} \mathcal{C}'_i \textbf{ in} \\
&\qquad\qquad \forall i \in 1..n \; \exists c_{\text{root}} \in \mathcal{C}'_i : \\
&\qquad\qquad\quad (\nexists (\_, \_, \_, a_t) \in \mathcal{E}'_i : a_t \in c_{\text{root}}) \\
&\qquad\qquad\quad \wedge |\{(\_, a_s, \_, \_) \in \mathcal{E}'_i : a_s \in c_{\text{root}}\}| \in \{0, 1, 2\} \\
&\qquad\qquad\quad \wedge \forall c \in \mathcal{C}'_i - c_{\text{root}} : |\{(\_, \_, \_, a_t) \in \mathcal{E}'_i : a_t \in c\}| = 1 \\
&\qquad\qquad\qquad \wedge |\{(\_, a_s, \_, a_t) \in \mathcal{E}'_i : a_s \in c \wedge a_t \in \mathcal{C}'_i - \{c_{\text{root}}\}| \in \{0, 1, 2\}
\end{aligned}$$

The shape predicate employs the sets $\mathcal{C}'$ and $\mathcal{E}'$ that result from P_{area} to ensure that irrelevant pointers are excluded from the shape test. However, due to the folding of structural repetition, it is possible that P_{area} locates multiple trees. This could occur if, e.g., many trees were nested under an SLL. To handle this, P_{shape} first partitions $\mathcal{C}'$ and $\mathcal{E}'$ into n trees using the disjoint union operator $\uplus$, where $\mathcal{C}'_i$ and $\mathcal{E}'_i$ represent the elements of tree i. Then, for each i, a root c_{root} is found with no incoming pointer in $\mathcal{E}'_i$, and the non-root cells $\mathcal{C}'_i - c_{\text{root}}$ are checked for a suitable number of incoming and outgoing edges in $\mathcal{E}'_i$.

If P_{shape} is found to be true for a Category 2+ data structure, then all strand connections mentioned in P_{area} have (L, E) added. Since the label and evidence may be distributed over multiple elements of the folded strand graph, L is parameterized to ensure the graph elements of that memory structure can be recovered. However, for a binary tree such a parameterization is unnecessary.

To find temporal repetition we must locate strands that perform the same role over multiple time steps. As mentioned previously, we do not attempt a global solution and instead solve the problem from the point of view of each entry point, where that local solution is represented as follows:

Definition 11. *An* **aggregate strand graph** G^{as}_{ep} *is composed of edges describing strand connections and vertices, of which one,* v_{ep}*, will represent the entry point and the remainder will represent linkage conditions.*

Algorithm Sketch 2. Temporal repetition *observed by an entry point* `ep` *is computed as follows. For each time step t in* `ep`*'s lifetime, we extract the subgraph of* G^{fs}_t *reachable from* v_{ep}*, which results in a subgraph set* $\mathcal{G}$*. To abstract over multiple time steps, we relabel all vertices that represent strands in the graphs of* $\mathcal{G}$ *to include only the associated linkage condition, which, unlike strands, is time step independent. The subgraphs in* $\mathcal{G}$ *are merged together in time step sequence, where the result of the last merge* G^{as}_{ep} *is merged with the next subgraph* $G_{next} \in \mathcal{G}$ *and* G^{as}_{ep} *is initially empty.*

To perform the merge, v_{ep} *of each graph is placed in correspondence, and then an inexact graph match is computed. Vertices may be in correspondence if they have identical linkage conditions, while edges may be in correspondence if the strand connections (including parameters) are identical. Graph elements in correspondence imply that temporal repetition has been discovered, and naturally the merge also sums any associated evidence. Elements of* G_{next} *not in correspondence are simply transfered with their associated evidence to* G^{as}_{ep}*.*

The identification algorithm is then applied to each G^{as}_{ep}, resulting in a natural language string. However, due to space limitations we refer the reader to the informal description of this process presented at the end of Sec. 3.

5 Preliminary Results

We have prototyped our approach using a combination of CIL [10] (approx. 1K LOC OCaml & 600 LOC C) to inject instrumentation into C source code, and

Table 2. Preliminary results obtained from our prototype implementation.

Example	Runtime (s)	Memory (GB)	Evidence Count	% Supp. / % Opp.	# Agg. Merges
Binary Tree	10.3	2.70	**Tree**: 102, Nesting: 21	83%/17%	16
Linux DLL [1]	9.7	1.76	**CDLL**: 60, IL: 52, DLL: 6	51%/49%	20
Wolf DLL [12]	71.4	2.86	**DLL**: 1410, IL: 220	87%/13%	123
Skip list with DLL Children	107.2	2.84	**SL**: 24793, N: 487, Tree: 48 **DLL**: 345, IL: 2	98%/2% 99%/1%	101

Scala (approx. 7.5K LOC) to perform the offline analysis. All experiments were run on an Intel i7-4800MQ with 32GB of RAM. We applied the prototype to four examples, the first three of which are self-written: a binary tree, an example exercising the cyclic Linux DLL [1], a skip list with child DLLs and a textbook DLL implementation [12]. We have made the source code of our self-written examples available at `http://www.david-white.net/kps15.zip`.

In Table 2 we report the runtime and memory usage of the offline analysis, although we note that currently no optimization has been performed and we store much redundant data. To simplify presentation of the results, we give details for only the longest running entry point of each example. In the evidence column we list all the non-zero evidence counts for each discovered memory structure, and in the following column we give the ratio of evidence supporting the correct data structure name versus that opposing. In each example, the evidence suggests the correct memory structure, which is shown in bold. For the example with multiple data structures, we separate out the evidence for each sub structure into a separate row. Lastly, in column *# Agg. Merges*, we show the number of merges that were performed by Algorithm 2 to produce the aggregate strand graph, which gives a rough indication of the rate at which evidence was gathered.

All examples are synthetic in the sense that they only manipulate, often just one, data structure. Since real world programs perform other tasks besides data structure manipulation, their data structures typically spend a smaller proportion of the runtime in degenerate states. Therefore, although the examples are quite simple, they effectively represent the worst case for our evidence based analysis. This is especially true for the Linux DLL example, which has approximately three time steps of intersecting lists for every one time step where the full DLL is present, resulting in only 51% evidence supporting a DLL. Behavior more typical of a real-world program can be seen in the skip list with nested DLLs example. This is due to building the skip list first, which is then held in a stable shape while child DLLs are added. This stable portion generates overwhelming evidence for the skip list.

6 Conclusion

We have presented dsOli2, a dynamic analysis that automatically identifies the dynamic data structures appearing in a C program during execution. By decomposing complex structures into strands and then analyzing the resulting strand connections, we are able to identify many data structures typically appearing in C heaps such as (cyclic) singly and doubly linked lists, trees, skip lists and relationships between data structures such as nesting. In contrast to related work that tries to avoid degenerate shapes [11, 7, 5], we permit these in our analysis and employ evidence based on structural complexity that is reinforced by structurally and temporally repetitive heap structures to override degenerate shapes. Preliminary results appear to support this method of dealing with degenerate shapes, and does not require the discovery of data structure operations or quiescent periods.

Ultimately, we aim to use the output of dsOli2 in a number of applications beyond program comprehension, including informing formal verification (which has already been studied in the context of dsOli1 and VeriFast [9]) and reverse engineering, which would be possible after we permit object code as input.

Acknowledgments. This work is supported by DFG grant LU 1748/4-1.

References

1. Linux kernel 4.1 cyclic dll (`include/linux/list.h`). `http://www.kernel.org/`. Accessed: 31/08/2015.
2. Braginsky, A. and Petrank, E. Locality-Conscious Lock-Free Linked Lists. In *ICDCN 2011*, vol. 6522 of *LNCS*, pp. 107–118. Springer, 2011.
3. Caballero, J., Grieco, G., Marron, M., Lin, Z., and Urbina, D. ARTISTE: Automatic Generation of Hybrid Data Structure Signatures from Binary Code Executions. Tech. Report TR-IMDEA-SW-2012-001, IMDEA Software Institute, 2012.
4. Dudka, K., Peringer, P., and Vojnar, T. Byte-Precise Verification of Low-Level List Manipulation. In *SAS 2013*, vol. 7935 of *LNCS*, pp. 215–237. Springer, 2013.
5. Haller, I., Slowinska, A., and Bos, H. MemPick: High-Level Data Structure Detection in C/C++ Binaries. In *WCRE 2013*, pp. 32–41. IEEE, 2013.
6. Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., and Vojnar, T. Fully Automated Shape Analysis Based on Forest Automata. In *CAV 2013*, vol. 8044 of *LNCS*, pp. 740–755. Springer, 2013.
7. Jung, C. and Clark, N. DDT: Design and Evaluation of a Dynamic Program Analysis for Optimizing Data Structure Usage. In *MICRO 2009*, pp. 56–66. IEEE, 2009.
8. Marron, M., Sanchez, C., Su, Z., and Fahndrich, M. Abstracting Runtime Heaps for Program Understanding. *IEEE Trans. Softw. Eng.*, 39(6):774–786, 2013.
9. Mühlberg, J. T., White, D. H., Dodds, M., Lüttgen, G., and Piessens, F. Learning Assertions to Verify Linked-List Programs. *To appear at the 13th International Conference on Software Engineering and Formal Methods*, 2015.
10. Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC 2002*, vol. 2304 of *LNCS*, pp. 213–228. Springer, 2002.
11. White, D. H. and Lüttgen, G. Identifying Dynamic Data Structures by Learning Evolving Patterns in Memory. In *TACAS 2013*, vol. 7795 of *LNCS*, pp. 354–369. Springer, 2013.
12. Wolf, J. *C von A bis Z.* Galileo Computing, 2009.