DSI: An Evidence-Based Approach to Identify Dynamic Data Structures in C Programs

David H. White
david.white@swt-bamberg.de

Thomas Rupprecht
thomas.rupprecht@swt-bamberg.de

Gerald Lüttgen
gerald.luettgen@swt-bamberg.de

Software Technologies Research Group
University of Bamberg, Germany

ABSTRACT

Comprehension of C programs containing pointer-based dynamic data structures can be a challenging task. To tackle this challenge we present *Data Structure Investigator* (DSI), 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 to discover data structure building blocks. By analyzing the interconnections between building blocks we are then able to identify, e.g., binary trees, doubly-linked lists, skip lists, and relationships between these such as nesting. Since the true shape of a data structure may be temporarily obscured by manipulation operations, we ensure robustness by first discovering and then reinforcing evidence for data structure observations.

We show the utility of our DSI prototype implementation by applying it to both synthetic and real world examples. DSI outputs summarizations of the identified data structures, which will benefit software developers when maintaining (legacy) code and inform other applications such as memory visualization and program verification.

CCS Concepts

•Software and its engineering → Software maintenance tools; Data types and structures; Software reverse engineering;

Keywords

Data structure identification, program comprehension, dynamic data structures, pointer programs

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 analysis. To partially alleviate this obstacle we propose *Data Structure Investigator* (DSI), a dynamic analysis for the automatic identification of dynamic data structures in C programs.

DSI relies on a front-end module to perform an *online trace recording* of the program under analysis, which we supply for C source code. This is followed by an *offline trace analysis*, which first discovers the building blocks of complex data structures, that are essentially singly-linked lists (SLLs), and then analyzes 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 opposite directions through a doubly-linked list (DLL) or those forming the left and right branches of a binary tree, or *loosely connected*, where they describe relationships between specific data structures, e.g., the parent-child relationship found in nested lists.

Challenges. We address two key challenges in this work. The first is the variety and complexity of implementation techniques used to realize data structures in C programs, which arise due to efficiency concerns or simply the freedom offered by the C language. We tackle this by employing an abstraction of memory based on identifying the building blocks of data structures as *lists*, rather than as *nodes* which is common in other approaches, e.g., ARTISTE [11], DDT [19], HeapDbg [23], and MemPick [16]. This key difference can be seen in a DLL: our approach considers this as two separate artifacts in the abstraction (i.e., lists) that happen to be connected in a specific way, while node-based approaches would consider a DLL as one artifact. Our approach allows for some interesting possibilities, such as identifying a cyclic list with one head node embedded in a node of a different type, recognizing a list with several nodes embedded in one type, and the handling of custom memory allocation out of the box.

The second key challenge arises due to manipulation operations that temporarily 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 it the correct *label*, i.e., name, of the data structure. Approaches such as dsOli [30] and DDT [19] handle this by trying to find data structure operation bound-

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany
ACM. 978-1-4503-4390-9/16/07...$15.00
http://dx.doi.org/10.1145/2931037.2931071

aries, while MemPick [16] attempts to perform identification only in the "quiescent periods" of a data structure. In both cases, identification is performed when it is likely that the data structure has a stable shape.

DSI includes degenerate shapes but overrides 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 observations of data structures and then *reinforcing* this evidence through structural and temporal repetition, our approach enables identification even when transient degenerate shapes are encountered.

Contributions. Our contributions are as follows:

- an abstraction of memory designed to cope with the complexities arising in C programs;
- a taxonomy of data structures and connections between data structures based on this abstraction;
- assignment of evidence to data structure observations, with weight based on the structural complexity of the underlying shape;
- a method to aggregate evidence both within one program time step and over multiple time steps, to mitigate the effect of transient degenerate shapes.

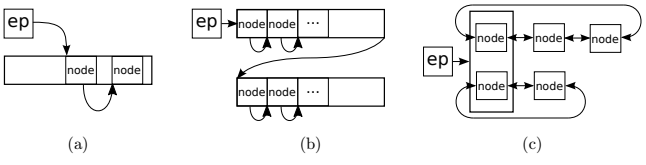
We apply our approach on textbook and synthetic examples, as well as examples taken from the literature [5] and real-world programs such as *libusb* [2]. In all cases, DSI identifies the associated data structure correctly, and the high weight of evidence collected for each correct interpretation confirms the robustness of our approach. Furthermore, due to the generality of our memory abstraction and thus in contrast to related work [16, 19], DSI is able to provide rich descriptions of data structures over a variety of implementation techniques commonly appearing in C programs.

Applications. To illustrate the utility of our approach we track variables that represent entry points to dynamic data structures, and annotate these with summarizations of the reachable data structure, e.g., "Entry point `ep` points to a Skip List with nested DLLs". However, DSI's output has application beyond that of *program comprehension* [29]. For example, our approach can guide *memory graph visualization* [8, 23] to produce visually intuitive layouts. In particular, the transition between DSI's various abstractions provides a natural zoom function to handle large graphs.

The notion of stable/degenerate shape can aid data structure *operation identification* [19, 30]. By monitoring the changes in the data structure, e.g., in terms of how the building blocks change between stable shapes, the operation boundaries can be estimated and the behavior identified, e.g., to determine that an SLL is used as a stack or queue.

Dynamic data structures are one of many contributing factors that make *formal verification* difficult. The approach prototyped in [24] employs information about data structure usage to generate program annotations, such as function contracts, to support the verification process.

Additional applications for DSI include *profiling and optimization* [20, 26], detecting *abnormal data structure behavior* [18], constructing *program signatures* [13] and, with a suitable front-end for object code, *reverse engineering* [16].



(d) SL_{O1}:
```
struct skip {
  struct skip *level1;
  struct skip *level2; }
```

(e) SL_{O2}:
```
struct skip {
  struct skip *next;
  struct skip *down; }
```

(f) SL_{I}:
```
struct level1 {
  struct level1 *next;
  struct level2 *down; }
struct level2 {
  struct level2 *next; }
```

Figure 1: Complexities of C heaps: (a) custom allocator, (b) cache efficient list [9], (c) Linux kernel cyclic DLL [3]; ep denotes an entry point. SL_{O1}, SL_{O2} & SL_{I} are skip list implementations (visualized in Fig. 3). Examples of SL_{O1} and SL_{O2} appear in `tests/forester-regre/test-f0021.c` and `tests/skip-list/jonathan-skip-list.c` of Forester [5], resp.

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 DSI. Sec. 3 describes our approach from a high level with an illustrative example, and in Sec. 4 we dive into the details. We report results obtained from our prototype implementation in Sec. 5, discuss related work in Sec. 6, and finally present our conclusions and suggestions for future work in Sec. 7.

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, and in Sec. 3 we outline how our approach copes with these challenges. Firstly, we briefly introduce 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 corresponds to 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 multiple nodes of potentially multiple data structures may 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 (CDLL) type `struct list_head` employed by the Linux kernel [3], which is designed to be embedded inside another struct. Given the *cyclic* property of the Linux CDLL, 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 one type and $n-1$ nodes each in a memory chunk of another type.

The key insight to model all of these 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, instead of grouping the nodes of a list by the entire memory chunk type, as is done in [16, 19, 23], it is necessary to group the nodes of a list by linkage or, equivalently, allow nodes to be some sub region of an outer 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.

We now consider the complexities regarding connections *between* lists. 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 just some of the possible ways to implement a skip list; C snippets are given in Fig. 1(e-f), and each implementation is visualized in Fig. 3. Firstly, if the number of levels are known *a priori*, then it is common to employ a struct where each member represents one level (SL_{O1}). Thus, when 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 (SL_{O2}); hence, lists are also formed in the vertical direction and overlay connections are present between these and the horizontal lists. Lastly, consider a skip list where each level is represented by a node of different type (SL_{I}). Since only the horizontal linkage forms lists, the downward link is an indirect connection between lists.

3. OVERVIEW OF OUR APPROACH

In this section we give an overview of our approach and provide motivation with the example in Fig. 2, which also depicts our approach as a pipeline. 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 classical implementation 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 chain of pointers between these chunks may 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 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 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 shown by block arrows in Fig. 2(a).

Strand Connections (SCs). Our approach is driven by relationships between strands, which we term *strand connections* (SCs). Each SC describes exactly *one* way in which

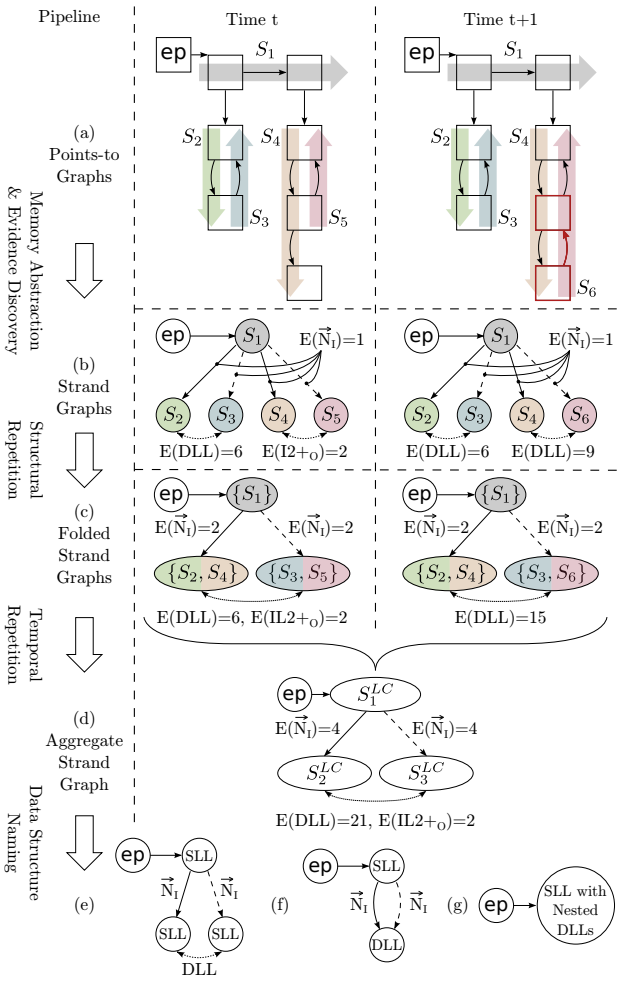


Figure 2: Left: our approach's pipeline; right: the illustrative example of Sec. 3.

the cells of two strands are related and is thus comprised of a set of *cell pairs*. Since cells from two strands may be related in more than one way, there may exist multiple SCs between a pair of strands. We construct a *strand graph*, where vertices represent strands and edges represent SCs (Fig. 2(b)). Since only two adjacent 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 SCs with the same edge style denote the same relationship type; for example, the DLL strands form an overlay SC, which is bi-directional, e.g., S_3 can be reached from S_2 and vice-versa. Two kinds of indirect SCs are formed between the parent SLL and each child DLL, which are unidirectional, e.g., S_1 cannot be reached from S_2.

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 memory structures in-scope for our approach are given in Fig. 3. An illustrative points-to graph accompanies each memory structure, where cells are drawn as circles and strands as block arrows; note that our illustrations do not depict all corner cases. A connection between cells by overlay is represented by a box

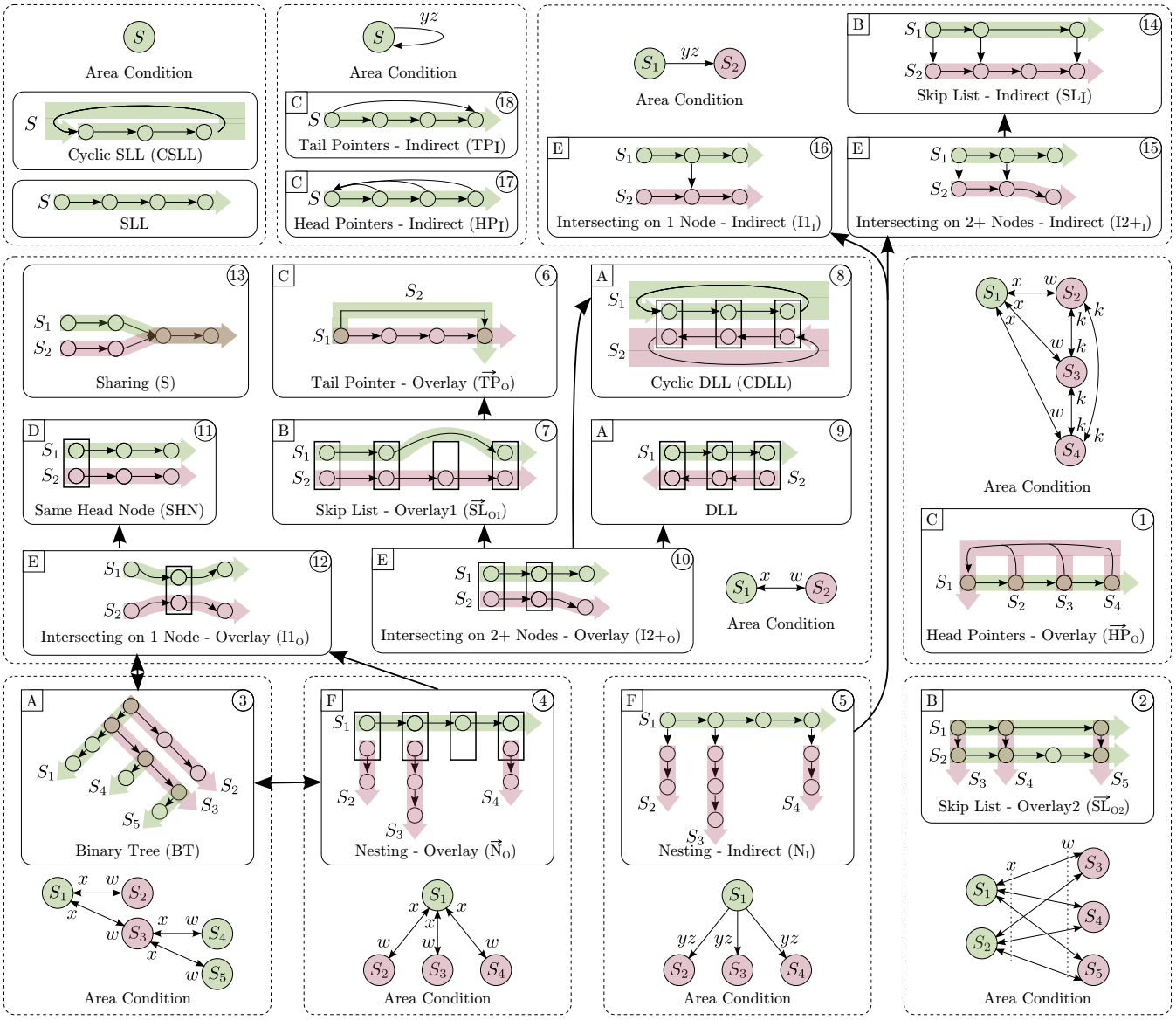


Figure 3: Illustrations of memory structures identified by our approach, grouped by area condition. (Cyclic) SLLs are implicitly identified by our abstraction as strands. I1 and I2+ are catch-all cases for SCs with 1 or 2+ cell pairs connecting the strands, resp., and Sharing (S) represents two strands that share the same tail cell sequence. The O and I subscripts denote whether the SCs are made by overlay or indirect connections, resp. Consult Fig. 1 for details of the three skip lists presented. Large arrows between memory structures represent a generalization → specialization relationship as discussed in Sec. 4.2, and small circled numbers assign priorities necessary to resolve ambiguities due to these relationships (see Alg. 1).

enclosing both cells. Each box should not be understood as a unique memory region; it is quite acceptable for multiple boxes to reside in the same memory region. Connections by indirection are simply represented by pointers.

A memory structure observation is made on the strand graph and requires a particular configuration of strands and SCs to be present. The subgraph of the strand graph corresponding to such a configuration is determined by a memory structure's *area condition*. For example, a DLL's area condition requires exactly two strands connected by an overlay SC, while nesting requires SCs from a parent strand to multiple child strands. The memory structures in Fig. 3 are grouped by area condition (regions with dashed lines), and multiple memory structures may belong to a single region. Thus, satisfying the area condition is necessary but insufficient to confirm the observation of a memory structure, and therefore we additionally check that the subgraph satisfies a memory structure's *shape predicate*, which typically performs a detailed inspection of SCs in terms of cells. Such a two-stage process is often used (c.f. [16,23]) to isolate exactly the elements to be checked by a shape property.

Observe that differences in the implementation of a mem-

ory structure can result in differences when that memory structure is viewed in terms of strands and SCs. This is a consequence of our choice to track linkages rather than nodes to cope with the complexities arising in C heaps, and can clearly be seen when considering the skip lists of Fig. 1.

Evidence Gathering. For each observation of a memory structure on the strand graph we record the associated evidence of that observation. The evidence consists of a label, uniquely identifying the memory structure (i.e., the acronyms introduced in Fig. 3), and a weight. We typically record evidence on all SCs mentioned by the area condition, which provides (a) a convenient mechanism to aggregate evidence via identification of structural and temporal repetition and (b) a method of deriving a suitable weight of an observation from the shape predicate. The weight is guided by the structural complexity of the observed memory structure, which intuitively means we wish to count the number of things that have gone "right" to interpret it as such a memory structure. We represent this count by the number of cell pairs comprising an SC, and the way in which they are accessed by the shape predicate.

For example, the shape predicate $\text{I2}+_{\text{O}}$ (lists intersect on 2+ nodes) must check that the number of connections between the two strands is at least two, thus the weight is simply the number of cell pairs in the connection. On the other hand, the DLL shape predicate must inspect the specific content of each cell pair in the SC to ensure that the forward/reverse property of the DLL holds. This results in a count of 3 for each cell pair: 1 for the existence of the cell pair, and 2 since both cells in the cell pair must be analyzed.

Evidence weights for the example are shown in Fig. 2(b). The degenerate DLL in time step t has an evidence count of 2 for $\text{I2}+_{\text{O}}$; however, when the DLL regains the correct shape at $t+1$, it has a count of 9 based on the three cell pairs that must exist and that their internal structure must be checked, i.e., $3 * 3$. Nesting ($\vec{\text{N}}_{\text{I}}$) only requires the existence of one cell pair connecting the parent strand to the child, hence each occurrence has a count of 1. Further, this evidence is deposited over each SC identified by $\vec{\text{N}}_{\text{I}}$'s area condition.

Some memory structures are specializations of others, and thus, if we are not careful, the more general form can be matched, leading to unnecessarily imprecise results. We give more details of this in Sec. 4, but for now it suffices to say that memory structure recognition is attempted in the order given by the circled numbers on the taxonomy in Fig. 3. On the successful observation of a memory structure, all SCs given by the area condition are removed from further consideration, with the rationale being that we have already found the most interesting interpretation of that structure, and there is no need to bloat the strand graph with the evidence from additional interpretations.

Structural Repetition. Structural repetition is used 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* (Fig. 2(c)), where the vertices have now become *sets* of strands. Since the folding process also merges SCs, it serves to reinforce the evidence assigned during the previous step. Thus, it provides part of our solution to address the problem of manipulation operations obscuring the correct shape, i.e., if SCs representing degenerate shape can be grouped with those representing the correct shape, then the majority can override the minority. In Fig. 2(c), this is seen between strands $\{S_2, S_4\}$ and $\{S_3, S_5\}$ at time t.

The correct shape is generally in the majority, because degenerate shapes are produced by operations that typically only have a local effect. Furthermore, memory structures that match degenerate shapes normally have much lower structural complexity than the memory structure that would match the stable shape, resulting in less contradictory evidence being added.

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 data structure building block 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 to a data structure 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, thus providing the second part of our solution to address degenerate shapes. Vertices of this graph become abstract descriptions of the original strands in terms of their *linkage conditions* (S^{LC}). The aggregate strand graph of our example is shown in Fig. 2(d); note that the evidence for a DLL is overwhelming. Only at the end of the trace do we interpret the evidence by setting the label for each SC to the one with the most evidence, and labeling each strand with SLL/CSLL as appropriate (Fig. 2(e)).

Output & Applications. We gather the artifacts produced by DSI during its offline analysis into an XML file and provide this for use in additional *back-end modules*. Recall from the introduction that we outlined several use cases for DSI's output including memory visualization [8], formal verification [24], optimization [20, 26], signature generation [13] and, with a suitable front-end for recording traces from object code, reverse engineering [16]. In the following we describe our back-end module for program comprehension.

Naming Data Structures. While the aggregate strand graph is highly useful for program comprehension, in some situations it is preferable to have a linear summarization of the identified data structure, e.g., a natural language string may be employed to annotate the source code of an entry point declaration. We propose a simple *naming module* that functions by iteratively grouping the vertices of the aggregate strand graph, and assigning a textual label to the resulting group. These grouped elements now form an atomic vertex in subsequent groupings. For example, in Fig. 2(f) we show the result of grouping the two vertices connected by a DLL SC. The order in which vertices are grouped must be carefully chosen to ensure the most suitable summary is generated. In practice we have found that the following rules work well: (a) group the vertices in the order shown by the letters in square boxes in Fig. 3 (from A to F), and (b) if there should exist multiple conflicting SCs between two vertices, then present all interpretations, with the exception of nesting which overrules all other interpretations. Ultimately, we end up with a graph like in Fig. 2(g).

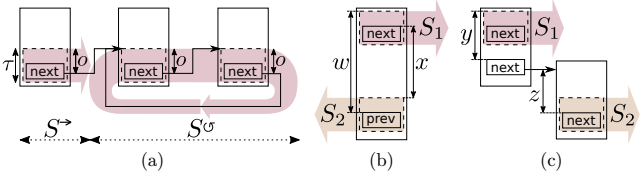


Figure 4: Details of (a) a strand S with linkage condition $S^{LC} = (\tau, o)$ and both linear and cyclic cell sequences, (b) an overlay SC $S_1 \overset{xw}{\longleftrightarrow} S_2$ and (c) an indirect SC $S_1 \overset{yz}{\longrightarrow} S_2$. Memory chunks have black outline, cells are dashed, and strands are indicated with block arrows.

4. DETAILS OF OUR APPROACH

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

4.1 Memory Abstraction

To identify data structures 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 using the front-end module, which results in the capture of *program events* at runtime such as pointer writes and dynamic memory (de)allocation. The result of the program event at *time step* t is represented by G_t^{pt}, where $1 \leq t \leq n$ and G_0^{pt} is empty. In the following we drop time step subscripts (t) until Sec. 4.3, as until then a single time step is sufficient for our presentation.

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, all unreachable chunks are removed.

A heap chunk becomes (partially) typed with a standard C type when it (or a subregion) is accessed by a non-void pointer. Usages of a memory chunk (or subregions) must be typed consistently, i.e., if a memory address a is accessed via pointer types `t1*` and `t2*`, then the overlapping parts of `t1` and `t2` must be structurally equivalent. Since structs may be nested, and thus multiple structs may start at an address, a function $\text{TYPE}(a)$ 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 a 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. 4(a) for details in the following. Set operators with a bar, $\bar{\in}$, $\bar{\subseteq}$ and $\bar{\cap}$, function on memory ranges, e.g., $a \bar{\subseteq} b$ 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.,* $\exists v \in \mathcal{V} : c \bar{\subseteq} v$*, with beginning and end addresses,* $c.bAddr$ *and* $c.eAddr$*, respectively.*

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

$$\exists (\_, a_s, \_, a_t) \bar{\in} \mathcal{E} : a_s \bar{\in} c_s \wedge a_t = c_t.bAddr \wedge o = a_s - c_s.bAddr$$
$$\wedge\, \tau \in \text{TYPE}(c_s.bAddr) \cap \text{TYPE}(c_t.bAddr) \wedge c_s \bar{\cap} c_t = \varnothing.$$

The linkage condition may be extended to n cells ($n \geq 2$) in the expected, straightforward manner. We are interested in the *maximal linkage condition* L that maximizes the length of the cell sequence $c_1 \xrightarrow{L} c_2 \xrightarrow{L} c_3 \ldots$; if L is not unique, then we choose the one with type τ of smallest size.

Definition 5. *A* **strand** S *represents the cell sequence captured by a maximal linkage condition, to which we refer as* S^{LC}*. The cell sequence* $\text{CELLS}(S)$ *comprises a linear start* $S^{\rightarrow}$ *and a cyclic tail* $S^{\circlearrowleft}$*, and at least one must be non-empty. When both sequences are non-empty, we have:*

$$\forall i \in [1, |S^{\rightarrow}| - 1] : S^{\rightarrow}[i] \xrightarrow{L} S^{\rightarrow}[i+1] \wedge S^{\rightarrow}[|S^{\rightarrow}|] \xrightarrow{L} S^{\circlearrowleft}[1]$$
$$\wedge \forall i \in [1, |S^{\circlearrowleft}| - 1] : S^{\circlearrowleft}[i] \xrightarrow{L} S^{\circlearrowleft}[i+1] \wedge S^{\circlearrowleft}[|S^{\circlearrowleft}|] \xrightarrow{L} S^{\circlearrowleft}[1]$$

The set $\mathcal{S}$ of strands captures every unique cell sequence. Strands are removed when all their 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 SCs are grouped wherever possible. Thus, due to the issues discussed in Sec. 2, all SC 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. 4(b) and (c)). It is for this reason that the SCs of Fig. 2 are drawn with different line styles; those with the same style have identical parameters. Lastly, note that indirect SCs can be generalized to sequences of pointers.

The set of SCs present is denoted $\mathcal{C}$ and in the following, $\text{MER}(c)$ finds the *Maximum Enclosing memory sub-Region* of a cell c, i.e., the memory sub-region of the outermost struct that contains c.

Definition 6. *A* **strand connection** *(SC)* $S_1 \overset{\alpha}{\cdots} S_2$ *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*. An SC is defined by the cells that establish the relationship:* $\text{PAIRS}(S_1 \overset{\alpha}{\cdots} S_2) = \{(c_1, c_2) \in \text{CELLS}(S_1) \times \text{CELLS}(S_2) : c_1 \overset{\alpha}{\cdots} c_2\}$*. The relationship between cell pairs (and by extension between strands) may be (a) overlay* $c_1 \overset{xw}{\longleftrightarrow} c_2$ *if* $S_1 \neq S_2 \wedge \text{MER}(c_1) = \text{MER}(c_2)$ *with parameters* $w = (c_2.bAddr + \text{LINKAGEOFFSET}(S_2)) - c_1.bAddr$ *and* $x = (c_1.bAddr + \text{LINKAGEOFFSET}(S_1)) - c_2.bAddr$ *(see Fig. 4). Alternatively, (b) indirect* $c_1 \overset{yz}{\longrightarrow} c_2$ *if* $\exists e = (\_, a_s, \_, a_t) \in \mathcal{E} : a_s \bar{\in} \text{MER}(c_1) \wedge a_t \bar{\in} \text{MER}(c_2)$ *and there is no linkage condition on* e*. In this case, the parameters are:* $y = a_s - c_1.bAddr$ *and* $z = (c_2.bAddr + \text{LINKAGEOFFSET}(S_2)) - a_t$*.*

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 SCs, where the starting offset is given from the memory chunk's start address and is therefore absolute. This is important as it allows us to uniquely identify a data structure building block under an entry point for evidence reinforcement over multiple time steps.

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* $c \in \text{CELLS}(S)$ *via a non-linkage condition edge* $(v_{ep}, a_s, v_t, a_t) \in \mathcal{E}$ *is defined by two parameters:* $x = a_s - v_{ep}.bAddr$ *and* $y = (c.bAddr + \text{LINKAGEOFFSET}(S)) - a_t$*. 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* $c \in \text{CELLS}(S)$ *such that* $c \subseteq v_{ep}$ *is defined by one parameter:* $z = (c.bAddr + \text{LINKAGEOFFSET}(S)) - v_{ep}.bAddr$.

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

4.2 Evidence Discovery and Reinforcement

With the strand graph G^s to hand, we proceed to first discover and then, in Sec. 4.3, reinforce evidence for observations of the memory structures in Fig. 3. As we prefer the discovery of complex memory structures, we check them in the order of area condition size, from large to small. However, recall from Sec. 3 that some memory structures are specializations of others and, thus, must be treated carefully. Generalizations are indicated in the taxonomy of Fig. 3 by bold arrows pointing from the general form to the specialized form. For example, I1_I is a specialization of $\vec{\text{N}}_\text{I}$ corresponding to nesting from a parent to a *single* child. We resolve such ambiguities by imposing a minimum area size on the more general form and always checking for its existence first, i.e., following the priority of circled numbers ⓧ in Fig. 3. In this case, the $\vec{\text{N}}_\text{I}$ memory structure is required to have a minimum of at least two children.

In other situations, the generalization/specialization relationship may be read both ways (double-ended bold arrows in Fig. 3), such as between I1_O and BT. Here, I1_O could be a specialization of BT with only one branch. Alternatively, BT could be a specialization of I1_O, since the position of the cell connection matters in BT (from any cell in the parent strand to the first cell of the child strand) but does not matter in I1_O. Again, such situations are handled by imposing a minimum area size (BT must have at least 3 levels) and an ordering (BT is checked before I1_O). Similar methods are used to resolve the remaining ambiguities; where there is no ambiguity, the checking order has been chosen arbitrarily.

Definition 9. *A* **memory structure** M *comprises a function* AREACONDITION*, which determines the subgraph of* G^s *potentially containing* M*; a* SHAPEPREDICATE*, which confirms an observation of* M *by performing a detailed analysis of the subgraph from* AREACONDITION*; and a function* ASSIGNEVIDENCE*, which imparts suitable evidence for* M *onto the SCs in* G^s*.*

A full formalization of these concepts for all memory structures is available from [6]. In the following, we present selected formalizations to highlight points of interest, starting with the DLL memory structure briefly presented in Sec. 3:

$$\text{DLL.AREACONDITION}(C_\text{init}, \_, \_, \_) =$$
$$\textbf{if } C_\text{init} = S_1 \overset{xw}{\longleftrightarrow} S_2 \textbf{ then ret } \{C_\text{init}\} \textbf{ else ret } \emptyset$$

$$\text{DLL.SHAPEPREDICATE} = S_1^{\circlearrowleft} = \emptyset \wedge S_2^{\circlearrowleft} = \emptyset \wedge$$
$$\forall i \in [0..\text{LENGTH}(S_1) - 1]\ \exists (c_1, c_2) \in \text{PAIRS}(S_1 \overset{xw}{\longleftrightarrow} S_2) :$$
$$\overrightarrow{S_1}[i+1] = c_1 \wedge \overrightarrow{S_2}[\text{LENGTH}(S_2) - i] = c_2$$

$$\text{DLL.ASSIGNEVIDENCE}(G^s) =$$
$$S_1 \overset{xw}{\longleftrightarrow} S_2 \text{ (in } G^s) \leftarrow \text{“DLL”} : |\text{PAIRS}(S_1 \overset{xw}{\longleftrightarrow} S_2)| * 3$$

$\mathcal{C}^\text{rem} = \mathcal{C}$
label LOOP: **while** $\mathcal{C}^\text{rem} \neq \emptyset$ **do**
  $C_\text{init} \leftarrow \text{CHOOSE}(\mathcal{C}^\text{rem})$
  for each M **from** ① **to** ⑱ **do** ▷ ⓧ in Fig. 3
    $\mathcal{C}_\text{area} = M.\text{AREACONDITION}(C_\text{init}, \mathcal{C}^\text{rem}, \mathcal{S}, \mathcal{C})$
    if $\mathcal{C}_\text{area} \neq \emptyset \wedge M.\text{SHAPEPREDICATE}$ **then**
      $M.\text{ASSIGNEVIDENCE}(G^s)$
      $\mathcal{C}^\text{rem} \leftarrow \mathcal{C}^\text{rem} \setminus \mathcal{C}_\text{area}$; **goto** LOOP
  $\mathcal{C}^\text{rem} \leftarrow \mathcal{C}^\text{rem} \setminus \{C_\text{init}\}$

Algorithm 1: Assign Evidence to Strand Graph G^s

AREACONDITION is used by Alg. 1. For now we note that it ensures that C_init, which is the “seed” SC around which an area is found, must be an overlay SC. We often need to refer to graph elements identified by AREACONDITION in SHAPEPREDICATE. To indicate this, any bound variable appearing above in red with a highlight is exported for subsequent usage (usages are shown in blue).

DLL.SHAPEPREDICATE requires that both strands have only linear cell sequences and, for every cell in the forward direction $\overrightarrow{S_1}[i+1]$, there exists a connection to the appropriate cell in the reverse direction $\overrightarrow{S_1}[\text{LENGTH}(S_2) - i]$ (the choice of direction is arbitrary at this point). Note that this can easily be extended to a CDLL by finding a cyclic permutation of the cyclic cell sequences which establishes the DLL property.

Recall that evidence is derived from the SHAPEPREDICATE, i.e., it checks each $pair \in \text{PAIRS}(S_1 \overset{xw}{\longleftrightarrow} eS_2)$ (count of 1 for each pair) and then opens $pair = (c_1, c_2)$ to ensure that the DLL shape property holds by examining c_1 and c_2 (count of 1 for each opened element examined). Thus, DLL.ASSIGNEVIDENCE places “DLL” : $|\text{PAIRS}(S_1 \overset{xw}{\longleftrightarrow} S_2)| * 3$ evidence on $S_1 \overset{xw}{\longleftrightarrow} S_2$ in G^s, in the form Label : Weight.

The process of imparting evidence on G^s is performed by Alg. 1. Recall that the sets $\mathcal{S}$ and $\mathcal{C}$ give the strands and SCs, resp. The set $\mathcal{C}^\text{rem}$ holds the remaining SCs to be checked whether they potentially belong to a whole or part observation of a memory structure. An SC C_init is chosen from $\mathcal{C}^\text{rem}$ and then, for a memory structure M, it is checked whether an area condition can be found for M containing C_init. If so, the shape predicate is checked and, if found to be true, then evidence is imparted on G^s by M.ASSIGNEVIDENCE. Finally, all SCs included in the area condition of a successfully observed memory structure are removed from $\mathcal{C}^\text{rem}$, and the process repeats. The algorithm terminates when $\mathcal{C}^\text{rem}$ is empty.

In some situations we must associate a direction with the evidence label, so that the memory structure can still be understood unambiguously. This is the case with all memory structures in Fig. 3 with an arrow above their acronym. To exemplify this, consider $\vec{\text{N}}_\text{O}$ and observe that, when the children strands are grouped due to the structural repetition step, the overlay SC can be read in either direction; hence the need to add a direction to $\vec{\text{N}}_\text{O}$. We now formalize $\vec{\text{N}}_\text{O}$:

$$\vec{\text{N}}_\text{O}.\text{AREACONDITION}(C_\text{init}, \mathcal{C}^\text{rem}, \mathcal{S}, \mathcal{C}) =$$
$$\textbf{if } \exists S_p \in \mathcal{S}, \mathcal{S}_c \subset \mathcal{S}, \mathcal{C} \subseteq \mathcal{C}^\text{rem}, x, w \in \mathbb{N} : C_\text{init} \in \mathcal{C}$$
$$\wedge \forall S_c \in \mathcal{S} : (S_p \overset{xw}{\longleftrightarrow} S_c) \in \mathcal{C}^\text{rem} \implies \quad \text{(C1)}$$
$$(S_p \overset{xw}{\longleftrightarrow} S_c) \in \mathcal{C} \wedge S_c \in \mathcal{S}_c$$
$$\wedge (\forall S_c \in \mathcal{S}_c : (S_p \overset{xw}{\longleftrightarrow} S_c) \in \mathcal{C}) \wedge |\mathcal{C}| = |\mathcal{S}_c| \quad \text{(C2)}$$
$$\wedge |\mathcal{S}_c| \geq 2$$
$$\textbf{then ret } \mathcal{C} \textbf{ else ret } \emptyset$$

$\vec{\text{N}}_\text{O}$.SHAPEPREDICATE = ALLLINKAGECONDSEQUAL($\mathcal{S}_c$)

$\vec{\text{N}}_\text{O}$.ASSIGNEVIDENCE(G^s) = **for all** $S_c \in \mathcal{S}_c$ **do**
$S_p \overset{xw}{\longleftrightarrow} S_c$ (in G^s) $\leftarrow$ "$\vec{\text{N}}_\text{O}$" : 1

$\vec{\text{N}}_\text{O}$.AREACONDITION showcases a complex area condition that requires the discovery of a parent strand S_p, a set of children strands $\mathcal{S}_c$, a set of SCs $\mathcal{C}$, and a direction on the overlay SCs using x and w. We typically wish to find *maximal area conditions*, and in this case condition (C1) ensures that all possible children are included, and (C2) checks that there are no unwanted strands or SCs.

Finally, note that $\vec{\text{N}}_\text{O}$.SHAPEPREDICATE is this simple, i.e., it only checks that all children have the same linkage condition, because many other possibilities have already been filtered out due to the priority system (Alg. 1, line 4), i.e., trees and skip lists have already failed to match. Thus, nesting becomes the only possible interpretation. Interestingly, this is the only memory structure for which we cannot derive the evidence weight directly from the shape predicate. Instead, by construction, there must be at least one pair in the cell-pairs relationship for each SC, and as we do not inspect this further, the evidence is simply 1 per SC.

4.3 Evidence Reinforcement

With the discovered evidence for all memory structure observations added to the strand graph, we now wish to reinforce the evidence to mitigate the effect of degenerate shapes. The first step in this process is the computation of a *folded strand graph*.

Definition 10. *A* **folded strand graph** G^{fs} *is a summarization of* G^s*. Vertices represent entry points or* sets *of strands; edges represent entry point connections or* merged *SCs.* G^{fs} *is computed by detecting structural repetition.*

Structural repetition *is found by successively locating those strands in* G^s *that conceptually perform the same role, and then merging them. Duplicate SCs resulting from the strand merge are also merged, thus aggregating any associated evidence. Two strands* S_1 *and* S_2 *are merged if* $S_1^{LC} = S_2^{LC}$ *and there exists (a) a merge parent* S_3 *with SCs* $S_3 \overset{\alpha}{\cdots} S_1$ *and* $S_3 \overset{\alpha}{\cdots} S_2$ *or (b) an SC* $S_1 \overset{xw}{\longleftrightarrow} S_2$ *where* $x = w$*, i.e., two lists with a shared tail ("Sharing" in Fig. 3).*

The parent merging process is shown in Fig. 2(b-c), where, e.g., S_1 functions as the parent, and S_2 and S_4 are merged.

The next step in the mitigation of degenerate shapes is the discovery of temporal repetition, i.e., locating strands that perform the same role over multiple time steps and aggregating the evidence of the associated SCs. As mentioned previously, we do not attempt a global solution and instead solve the problem from the point of view of each entry point `ep`, where that local solution is represented as follows:

Definition 11. *An* **aggregate strand graph** G^{as}_{ep} *is composed of a vertex* v_{ep} *and entry point connections originating from* v_{ep}*. The remaining vertices represent linkage conditions, and the remaining edges are SCs.* G^{as}_{ep} *is computed by detecting* **temporal repetition** *using Alg. 2.*

To describe Alg. 2, we first reintroduce the time step subscript t. For each time step $t \in$ [ep.tStart, ep.tEnd] in `ep`'s lifetime, we use EXTRACTREACHABLESG to extract the subgraph G'_t of G^{fs}_t reachable from v_{ep}, and cumulatively merge these subgraphs together resulting in $G^{\text{as}}_{\text{ep}}$. To abstract over multiple time steps, EXTRACTREACHABLESG relabels all vertices that represent a strand S to include only the associated linkage condition S^{LC}, which, unlike strands, is time step independent.

1: $G^{\text{as}}_{\text{ep}} \leftarrow$ EXTRACTREACHABLESG($G^{\text{fs}}_{\text{ep.tStart}}$, ep)
2: **for each** t **from** ep.tStart + 1 **to** ep.tEnd **do**
3: $\quad G'_t \leftarrow$ EXTRACTREACHABLESG(G^{fs}_t, ep)
4: $\quad G_{\text{MCS}} \leftarrow$ MCSINCLUDINGVERTEX($G'_t, G^{\text{as}}_{\text{ep}}, v_{\text{ep}}$)
5: $\quad$ **for each** SC $\in G_{\text{MCS}}$ **do**
6: $\quad\quad$ SC (in $G^{\text{as}}_{\text{ep}}$) $\leftarrow$
7: $\quad\quad\quad$ AGGREGATEEVIDENCE(SC (in $G^{\text{as}}_{\text{ep}}$), SC (in G'_t))
8: $\quad G^{\text{as}}_{\text{ep}} \leftarrow G^{\text{as}}_{\text{ep}} \cup (G'_t \setminus G_{\text{MCS}})$

Algorithm 2: Compute Aggregate Strand Graph $G^{\text{as}}_{\text{ep}}$

To perform the merge, we compute the *maximum common subgraph* (MCS) of G'_t and $G^{\text{as}}_{\text{ep}}$ that contains v_{ep}. Any structure appearing in the MCS means temporal repetition has been discovered and, thus, evidence on an SC of the MCS in G'_t is aggregated with that on the corresponding SC in $G^{\text{as}}_{\text{ep}}$. The remaining elements of G'_t not included in the MCS are unioned with $G^{\text{as}}_{\text{ep}}$. These elements represent the new parts of `ep`'s data structure added in time step t, and will likely be reinforced by appearing in the MCS of subsequent merges. Temporal aggregation in our example is shown in Fig. 2(c-d); the MCS is simply the complete graph since no new structure is created at $t + 1$.

5. EVALUATION

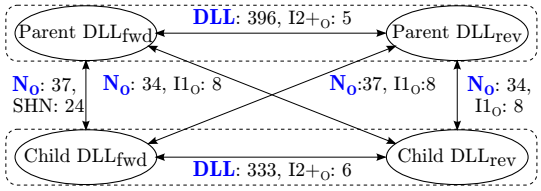
We have prototyped DSI's offline phase in Scala (9K LOC) and also written a front-end module to perform the online trace recording phase for C programs in the style of [30], which employs CIL [25] to inject instrumentation into C source code (1K LOC OCaml & 600 LOC C). All experiments were run on an Intel i7-4800MQ with 32GB of RAM.

Benchmark & Instrumentation. We apply our prototype to examples from textbooks (`tb`), self-written synthetic examples (`syn`), examples taken from Forester / Predator [5] (`lit`), and real-world programs from benchmarks [4, 7], plus `bash` and `libusb`. The latter provides access to usb devices from userspace and is a particularly interesting example as it exercises many of the features offered by DSI. All these 16 examples have one main data structure that may be composed of several memory structures. We have made the source code of our self-written examples available at [6]; the others may be obtained from the corresponding references. We employ the front-end module to perform a full instrumentation of all examples, with the exception of `bash`, for which we only instrument the files `array.c` and `xmalloc.c` to evaluate our approach on a real-world example of a non-Linux CDLL.

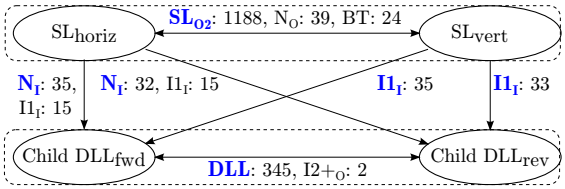
As DSI only observes behavior that a program exhibits at runtime, we provide drivers (available from [6]) for the `syn` and `tb` examples to exercise the data structures. Examples from [4, 5, 7] exercise the data structures out of the box, while we invoke `bash` with a piped command, e.g., `ls | grep pattern`, and exercise `libusb` using the included utility `listdevs`. `libusb` allows multiple simultaneous usages within one executable via several `struct libusb_context`s, which form a parent DLL. This behavior is not visible in `listdevs` by default, so we augment the example with two additional contexts (modified `listdevs` are available at [6]).

Table 1: Results obtained from our prototype implementation

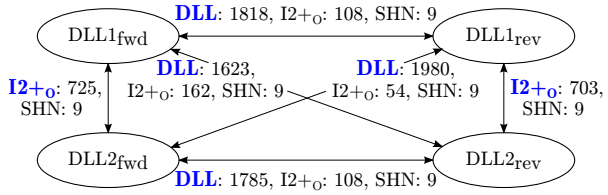
ID	Naming Module Output	Evidence Counts	% Supporting Evidence
`tb1` [28]	SLL	None since no SCs present	
`tb2` [31]	DLL	**DLL**: 1440, I2+$_O$: 220	87%
`syn1`	CDLL	**CDLL**: 15, I2+$_O$: 10, DLL: 6	48%
`syn2`	Binary Tree	**BT**: 248, N$_O$: 6, I1$_O$: 3	97%
`syn3`	SLL + nest. SLL	**N$_I$**: 6, I1$_I$: 2	75%
`syn4`	SLL + nest. SLL	**N$_O$**: 10, SHN: 6	63%
`syn5`	DLL + nest. DLL		Avg. 84%



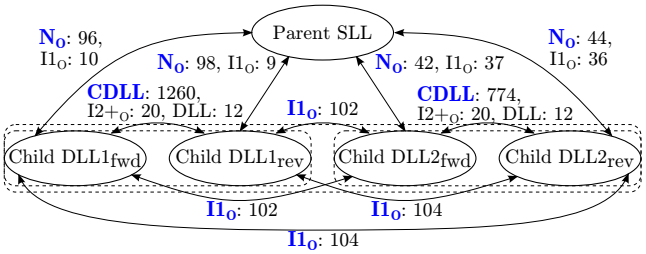
`syn6`	Skip List + nest. DLL		Avg. 89%



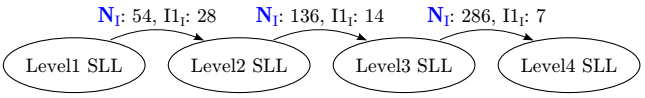
`lit1` [5]	DLL of (DLL, DLL) OR Intersecting(2xDLL)		Avg. 96%



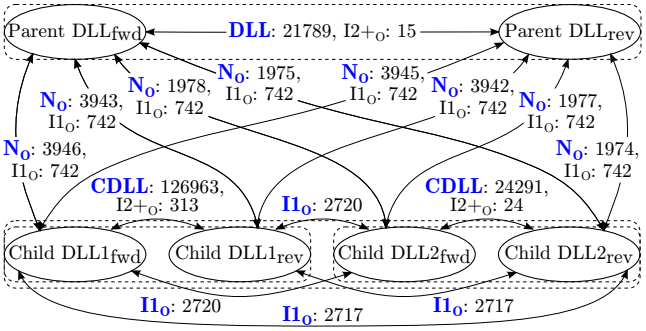
`lit2` [5]	Skip List	**SL$_{O2}$**: 1242, BT: 72, N$_O$: 58, I1$_O$: 8, SHN: 3	90%
`lit3` [5]	SLL + nest. Intersecting(2x CDLL)		Avg. 88%



`lit4` [5]	SLL + nest. SLL + nest. SLL + nest. SLL		Avg. 85%



`bash` [1]	CDLL	**CDLL**: 68529, I2+$_O$: 72, DLL: 6	∼100%
`treeadd` [4]	Binary tree	**BT**: 256	100%
`treebnh` [7]	Binary tree	**BT**: 930	100%
`libusb` [2]	CDLL + nest. Intersecting(2x CDLL)		Avg. 88%



Identification Results & Robustness. In total, the examples exercise 13 out of the 18 memory structures given in Fig. 3 and, as Table 1 shows, evidence for the correct observation is always in the majority (blue text), which naturally leads to the correct naming (green text). The suitability of our abstraction of memory is further substantiated by the variety and quantity of C code in the examples.

We turn to demonstrate that our approach can *robustly* identify data structures even in the presence of degenerate shapes, and to do so we dive into the details of Table 1. To simplify the presentation, the table only contains details for the longest running entry point of each example. In the evidence column we list all evidence counts for each observed memory structure; for examples with more than one SC, we show the aggregate strand graph (minus the entry point) annotated with the aggregated evidence. In the final column we give the percentage of evidence supporting the correct data structure name, and for examples with more than one SC, we give the supporting evidence as an average.

The evidence counts in Table 1 show that evidence for the stable shape always builds much faster than evidence for the competing degenerate interpretations, and often becomes overwhelming. This behavior enables a robust and correct naming of the data structure. Of all memory structures, skip lists build correct evidence at the fastest rate due to their high level of structural complexity. This is seen in `syn6` and `lit2`, where the competing degenerate interpretations N$_O$ and BT are quickly ruled out.

Example `syn1` effectively represents the worst case scenario for our approach, since the trace is short and the CDLL is constantly operated on, meaning it frequently is in a degenerate shape; nevertheless, it is named correctly. However, realistic programs do not spend all their time manipulating a single data structure; thus, while other actions are performed, a memory structure is typically held in a stable shape, which builds evidence. Such realistic scenarios appear in examples `syn5` and `syn6`, where there exist many DLL children and, at each time step, only one is in a degenerate shape, and in examples `bash` and `libusb`, where other actions are performed besides manipulating the CDLL(s).

Utility for Program Comprehension. To show the value of our approach for program comprehension, we consider the aggregate strand graphs and output of the *naming module* (cf. Sec. 3), beginning with the simpler example `syn5`

and then moving on to more complex examples. Note that we manually verify the ground truth we evaluate against.

`syn5` contains a DLL of nested DLLs, where the first node of the child resides in a node of the parent. The naming algorithm first groups the strands of the DLLs together (indicated by dashed boxes) and then combines them with nesting, which results in the name: "DLL + nest. DLL".

`lit1` comprises two DLLs running in parallel through a sequence of nodes, and is the only example with an ambiguous reading of the aggregate strand graph, i.e., the forward strand of one DLL can be matched with either DLL's reverse strand. Interestingly, the naming algorithm's output "DLL of (DLL, DLL) OR Intersecting (2x DLL)" highlights a different ambiguity regarding the combination of the two DLLs. Thus, the example serves to highlight the richness of our approach in describing connections between lists, including those that are non-obvious via code inspection.

`lit3` and `libusb` represent the most structurally challenging examples. To illustrate this we concentrate on `libusb`, which employs a parent CDLL of `libusb_context`. Child elements record both a CDLL of devices and a CDLL of associated file descriptors. The naming algorithm's output "CDLL + nest. Intersecting(2x CDLL)" clearly indicates this structure, and we believe this is very helpful in understanding the high-level data structure of the almost 7k LOC of libusb. Note that all CDLLs in `lit3` and `libusb` are Linux CDLLs that are embedded in structs, and thus our abstraction based on cells is mandatory to understand the cyclic nature of the lists (cf. Sec. 2).

6. RELATED WORK

Dynamic analyses such as TIE [21], Howard [27] and Rewards [22] perform a low-level identification of data structures, e.g., by discovering the internal layout of a struct, but do not give details on the high-level usages of the discovered structs, e.g., that a struct type forms a list or tree. We believe MemPick [16] and DDT [19] represent the state-of-the-art in heuristic guided identification of data structures. Both function on object code; in contrast, DSI's current front-end requires source code, which facilitates the discovery of data structures running through nodes of different types and those allocated via custom memory allocation. Regarding data structure variety, MemPick and DDT perform a detailed classification of trees (which could be incorporated into DSI), while DSI handles skip lists and produces a richer description of the connections between data structures. Lastly, as discussed earlier, both MemPick and DDT rely on degenerate shapes not appearing in their analysis, in contrast to DSI's evidence-based approach.

The dynamic analysis HeapDbg [23] employs *abstract interpretation* [12] to provide safe reasoning about program heaps. Information joins are similar to our usage of structural and temporal repetition but, as an over-approximation is required, information is inevitably lost. While this works for HeapDbg's tree label, because trees do not typically form degenerate shapes during manipulations, a related approach ARTISTE [11] includes DLLs and, if temporal joins were to be performed on degenerate shapes, then the precision of the DLL label would be lost. In contrast, our evidence-based approach essentially performs the join at the end of the analysis and is thus robust against such situations.

Modern *shape analysis* tools, such as Predator [15] and Forester [17], 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 deal with the recursion commonly found in tree operations, and their focus is on memory safety, not program comprehension. Nevertheless, it would be interesting to compare such synthesized predicates to DSI's strand-based abstraction.

Finally, dsOli [30] and DDT [19] seek to additionally discover the operations that manipulate the data structures. DDT accomplishes this by assuming that data structures are accessed via well-defined interfaces, while dsOli employs a machine learning approach to locate repetitive code segments indicative of operations. Recently, an approach [14] based on Predator has been used to transform low-level pointer assignments into equivalent high-level operations.

7. CONCLUSIONS & FUTURE WORK

We presented DSI, 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. To handle the complexities arising in C heaps, we proposed an abstraction of memory that allows memory chunks and structs to contain multiple nodes of data structures. Furthermore, in contrast to related work that tries to avoid degenerate shapes [16, 19, 30], we permit degenerate shapes in our analysis and employ evidence based on structural complexity – which is reinforced by structurally and temporally repetitive heap structures – to override degenerate shapes.

Experimental evaluation with a prototype DSI implementation showed that a significant variety and quantity of complex C code can be handled and that, in each case, our evidence-based approach leads to the correct data structure identification. With the exception of a detailed analysis of trees, we believe that DSI today provides the most robust, rich description of data structures for program comprehension when C source code is available.

Regarding future work, we plan to move DSI away from a research prototype by addressing some performance considerations. DSI's execution times range from a few seconds for simple examples up to tens-of-minutes, and memory consumption is in the range of 500MB to 4GB. The trace of our largest example comprises 6.2MB and 32861 time steps. Note that the majority of time (> 95 %) is spent discovering evidence, which is an embarrassingly parallel problem but for which our prototype uses a sequential solution.

Furthermore, we plan a number of improvements to DSI's functionality, including a richer classification of nesting and trees, e.g., unique/shared child strands and balance properties [16, 19], respectively. Lastly, we wish to further investigate applications of DSI's output including combining operation detection with DSI in a style similar to that of [30], and employing techniques from machine learning to inform formal verification as is done in [10, 24].

8. ACKNOWLEDGMENTS

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

9. REFERENCES

[1] GNU `bash` v4.3.30. https://www.gnu.org/software/bash/. Accessed: 17th May 2016.

[2] libusb 1.0.20. http://www.libusb.info/. Accessed: 17th May 2016.

[3] Linux Kernel 4.1 Cyclic DLL(`include/linux/list.h`). http://www.kernel.org/. Accessed: 17th May 2016.

[4] Olden Benchmark v1.01. http://www.martincarlisle.com/olden.html. Accessed: 17th May 2016.

[5] Predator/Forester GIT Repository. In particular, examples:
`lit1`: `tests/forester/dll-duplicate-sels.c`,
`lit2`: `tests/skip-list/jonathan-skip-list.c`,
`lit3`: `tests/forester/cav13_tests/sll-listoftwoclists-linux.c`,
`lit4`: `tests/predator-regre/test-0234.c`.
https://github.com/kdudka/predator. Accessed: 17th May 2016.

[6] Supplemental Material for DSI. http://www.swt-bamberg.de/research/data-structures.html. Accessed: 17th May 2016.

[7] The Computer Language Benchmarks Game: Binary Tree. https://benchmarksgame.alioth.debian.org/u64q/program.php?test=binarytrees&lang=gcc&id=1, Contributed by Kevin Carson. Accessed: 17th May 2016.

[8] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In *SOFTVIS 2010*, pages 53–62. ACM, 2010.

[9] A. Braginsky and E. Petrank. Locality-Conscious Lock-Free Linked Lists. In *ICDCN 2011*, volume 6522 of *LNCS*, pages 107–118. Springer, 2011.

[10] M. Brockschmidt, Y. Chen, B. Cook, P. Kohli, S. Krishna, D. Tarlow, and H. Zhu. Learning to Verify the Heap. Technical Report MSR-TR-2016-17, 2016.

[11] J. Caballero, G. Grieco, M. Marron, Z. Lin, and D. Urbina. ARTISTE: Automatic Generation of Hybrid Data Structure Signatures from Binary Code Executions. Technical Report TR-IMDEA-SW-2012-001, IMDEA Software Institute, Spain, 2012.

[12] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *POPL 1979*, pages 269–282. ACM, 1979.

[13] A. Cozzie, F. Stratton, H. Xue, and S. King. Digging for Data Structures. In *OSDI 2008*, pages 255–266. USENIX, 2008.

[14] K. Dudka, L. Holík, P. Peringer, M. Trtík, and T. Vojnar. From Low-Level Pointers to High-Level Containers. Technical Report FIT-TR-2015-03, Faculty of Information Technology, Brno University of Technology, October 2015.

[15] K. Dudka, P. Peringer, and T. Vojnar. Byte-Precise Verification of Low-Level List Manipulation. In *SAS 2013*, volume 7935 of *LNCS*, pages 215–237. Springer, 2013.

[16] I. Haller, A. Slowinska, and H. Bos. Scalable Data Structure Detection and Classification for C/C++ Binaries. *Empirical Software Engineering*, pages 1–33, 2015.

[17] L. Holík, O. Lengál, A. Rogalewicz, J. Šimáček, and T. Vojnar. Fully Automated Shape Analysis Based on Forest Automata. In *CAV 2013*, volume 8044 of *LNCS*, pages 740–755. Springer, 2013.

[18] M. Jump and K. S. McKinley. Dynamic Shape Analysis via Degree Metrics. In *ISMM 2009*, pages 119–128. ACM, 2009.

[19] C. Jung and N. Clark. DDT: Design and Evaluation of a Dynamic Program Analysis for Optimizing Data Structure Usage. In *MICRO 2009*, pages 56–66. IEEE, 2009.

[20] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective Selection of Data Structures. *SIGPLAN Not.*, 46(6):86–97, June 2011.

[21] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *NDSS 2011*. The Internet Society, 2011.

[22] Z. Lin, X. Zhang, and D. Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *NDSS 2010*. The Internet Society, 2010.

[23] M. Marron, C. Sanchez, Z. Su, and M. Fähndrich. Abstracting Runtime Heaps for Program Understanding. *IEEE Trans. Softw. Eng.*, 39(6):774–786, 2013.

[24] J. T. Mühlberg, D. H. White, M. Dodds, G. Lüttgen, and F. Piessens. Learning Assertions to Verify Linked-List Programs. In *SEFM 2015*, volume 9276 of *LNCS*, pages 37–52. Springer, 2015.

[25] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC 2002*, volume 2304 of *LNCS*, pages 213–228. Springer, 2002.

[26] E. Raman and D. August. Recursive Data Structure Profiling. In *MSP 2005*, pages 5–14. ACM, 2005.

[27] A. Slowinska, T. Stancescu, and H. Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *NDSS 2011*. The Internet Society, 2011.

[28] M. Weiss. *Data structures and algorithm analysis in C*. Cummings, 1993.

[29] D. H. White. dsOli: Data Structure Operation Location and Identification. In *ICPC 2014*, pages 48–52. ACM, 2014.

[30] D. H. White and G. Lüttgen. Identifying Dynamic Data Structures by Learning Evolving Patterns in Memory. In *TACAS 2013*, volume 7795 of *LNCS*, pages 354–369. Springer, 2013.

[31] J. Wolf. *C von A bis Z*. Galileo Computing, 2009.