DSI: Automated Detection of Dynamic Data Structures in C Programs and Binary Code*

Thomas Rupprecht, Jan H. Boockmann, David H. White and Gerald Lüttgen

Software Technologies Research Group, University of Bamberg, 96045 Germany

Abstract. Program comprehension is an important task for software engineers who maintain legacy code, as well as for reverse engineers who analyse binary executables such as malware. Detecting dynamic, i.e., pointer-based data structures is a particular challenge due to the complex usage of pointers found in real world software.

This paper presents the key results of the DFG-funded project "Learning Data Structure Behaviour from Executions of Pointer Programs" (DSI), in which dynamic analysis techniques have been developed to identify dynamic data structures in C programs and x86 binary code. DSI's analysis utilizes a novel memory abstraction that allows for a compact description of pointer-based data structures such as linked lists and binary trees, and their interconnections such as parent-child nesting. On top of this abstraction, an evidence-collecting approach calculates a natural language description of the observed data structure with the help of a systematic taxonomy. The inferred data structure information is not only helpful for program comprehension but also for other use cases including software verification and software visualization.

1 Introduction

This paper summarizes the key results of the DFG-funded project DSI (LU 1748/4-1), whose main goal was to develop novel techniques for identifying dynamic data structures in pointer software. Details can be found in [23, 24, 27, 28] and on the DSI webpage located at `http://www.swt-bamberg.de/dsi`.

Motivation. Many software developers are faced with legacy code, or sophisticated program constructs employed in low-level code. In addition, programs often rely on dynamic, i.e., pointer-based data structures involving linked lists and trees. Therefore, developers desire advanced support for program comprehension, which is even more true when only a binary version of a program is available, e.g., when analysing the vastly growing amount of malware [3]. To partially alleviate this obstacle, we propose *Data Structure Investigator* (DSI), a novel dynamic analysis for the automatic identification of dynamic data structures. DSI detects (cyclic) singly and doubly linked lists and binary trees, as

* This research is supported by the German Research Foundation (DFG) under project title "Learning Data Structure Behaviour from Executions of Pointer Programs" (grant no. LU 1748/4-1).

well as other data structures, such as skip lists, that are not handled by related work [8,14,17]. Additionally, DSI allows for arbitrary parent-child nesting combinations of such data structures, which is also out-of-scope of [8,12,14,15,17]. DSI's analysis information can be used in various ways, such as for inferring a natural language description of an observed data structure (e.g., "a doubly linked list with nested singly linked lists"), for generating verification conditions for verification tools such as VeriFast [16], and for enabling advanced visualizations of pointer programs.

State-of-the-art. Prominent examples of dynamic analysis tools for detecting dynamic data structures are ARTISTE [8], DDT [17], and MemPick [14]. These operate on binaries only and instrument the binaries to extract runtime information. DSI also uses instrumentation to record memory events, but operates on both source code and binary code. One of the biggest challenges when identifying data structures comes with data structure operations as they tend to temporarily break a data structure's shape; consider, e.g., the rewriting of pointers during a doubly-linked list insertion operation. We term the true shape of a data structure a *stable shape* and the temporarily broken shape a *degenerate shape*. Both DDT and MemPick avoid degenerate shapes: DDT performs its analysis on operation boundaries and thus requires their accurate detection, while MemPick employs a heuristic to determine quiescent periods during which no changes to the data structure are made. While DSI also analyses degenerate shapes, it uses an evidence-collecting approach so as not to loose overall precision. ARTISTE does not explicitly avoid degenerate shapes, too, but it becomes more conservative with its analysis when it encounters them. Additionally, ARTISTE, DDT, and MemPick have the common assumption that one node of a data structure occupies an allocated chunk of memory as a whole. Instead, DSI has a finer grained heap abstraction, which allows it to seamlessly deal with advanced and low-level constructs such as employed, e.g., by the Linux kernel list [2].

Complementary to dynamic analysis tools are static shape analysis tools such as Predator [12] and Forester [15], which operate on source code. They also infer the shape of a data structure but focus on program verification rather than program comprehension. Other related work are visualization tools that generate a compact view of the heap to ease understanding, e.g., Heapviz [5] and HeapDbg [19]. Both operate on heap snapshots as opposed to DSI's dynamic analysis that observes how data structures evolve over time. Heapviz processes Java-based heap snapshots that provide rich information about objects, their types, and their interconnections. It uses the extracted type information from the heap as is, which can result in less precise data structure naming, as can be seen in an example in [5], where a doubly linked list (DLL) is labeled as `LinkedList`. Instead, DSI infers shapes without relying on preexisting data structure information, and will correctly report a DLL in the mentioned example. HeapDbg is similar to Heapviz but infers shapes by inspecting the interconnections between objects; for example, it classifies objects with a left pointer and a right pointer as a tree. In addition, HeapDbg does not support as many data structures as DSI does, such as skip lists and various parent-child relations.

Contributions. DSI contributes to the state-of-the-art of program comprehension and reverse engineering with a novel approach for identifying dynamic data structures in both source code and binaries. Its dynamic analysis is based on a fine-grained memory abstraction that consists of singly linked lists and their interconnections (such as nesting), where list nodes can cover sub-regions of memory. The detectable data structures are summarized in a taxonomy and reach from (cyclic) singly and doubly linked lists to skip lists to trees. At the heart of DSI is an evidence-collecting algorithm called *DSIcore* (see Sec. 3), which considers the structural complexity of an observed data structure shape as evidence measure, and reinforces evidence by exploiting structural and temporal repetition to discriminate against degenerate shapes.

Two concrete realizations of DSI have been developed as front ends to DSIcore: (i) *DSIsrc* supports the analysis of C source code (see Sec. 4), and (ii) *DSIbin* supports x86 binaries (see Sec. 5). One of the fundamental differences between DSIsrc and DSIbin is the availability of type information in source code but not in (stripped) binaries. This is mitigated in DSIbin by first using the state-of-the-art type excavator Howard [25] to recover (partial) low-level type information, and subsequently refining the types via a new approach termed *DSIref*, which employs DSIcore to assist in the inference of low-level types via high-level data structure information (see Sec. 5).

Extensive benchmarking using example programs from the literature and real-world programs demonstrated that DSI identifies dynamic data structures correctly and robustly. Due to the generality of DSIcore's memory abstraction and in contrast to related work, DSI provides rich descriptions of data structures over a variety of implementation techniques that commonly occur in pointer programs. Three back ends to DSIcore have also been prototyped within the DSI project and testify to DSI's applicability in various domains, namely program verification, data structure visualization, and operation detection (see Sec. 6).

2 DSI Tool Suite

Fig. 1 depicts DSI's architecture: our front-end components DSIsrc and DSIbin, the DSIcore component and various back-end components. Both front ends collect runtime information via a dynamic analysis, which is accomplished by instrumenting C source code via the CIL framework [21] and binaries via Intel's Pin framework [18]. Executing an instrumented program results in an execution trace that records relevant heap and stack events such as memory (de-)allocations and pointer writes. The trace is passed to DSIcore for offline analysis. DSIbin uses Howard [25] for recovering type information from binaries and our additional type refinement component DSIref. DSI's back ends consume the information produced by DSIcore and comprise the *naming* module for giving a natural language description for an identified data structure, the *operation detection* component for localizing data structure operations, the *visualization* component for presenting the data structure consistently over its lifetime, and the *verification* component for interfacing to the program verifier VeriFast [16] (see Sec. 6).

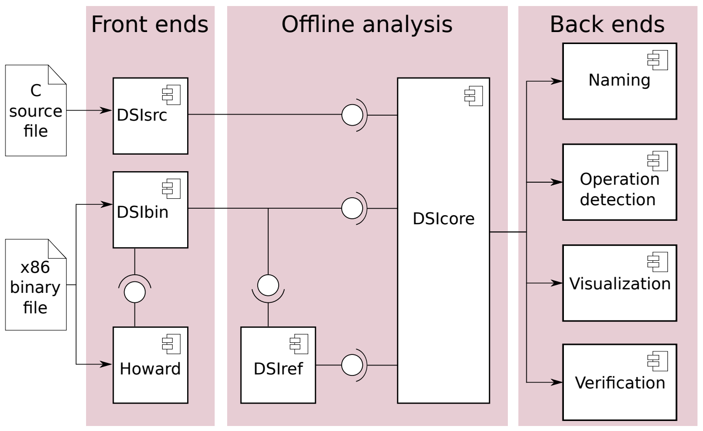

Fig. 1. Overview of the DSI tool chain. (Figure adapted from [23].)

3 DSIcore: DSI's Core Algorithm

This section overviews our general DSI approach implemented by DSIcore, using the illustrative example of Fig. 2. The example shows two time steps t and $t+1$ in the construction of a singly linked list (SLL) of doubly liked lists (DLLs), with entry pointer `ep`. Note that there exists a degenerate DLL child at time step t, and how the pipelined nature of the approach becomes apparent. The pipeline starts by constructing a sequence of *points-to graphs* that model the heap at each time step, where a points-to graph represents allocated memory chunks as vertices and pointers as edges (see Fig. 2(a)).

Memory abstraction. DSI relaxes the common assumption that data structure nodes occupy an allocated memory chunk as a whole [8, 14, 17]. Instead, our approach allows nodes to cover sub-regions of memory, termed *cells*. Sequences of cells that have the same linkage condition, i.e., all pointers originate at the same linkage offset relative to the cell's start address, and that point to the start address of the following cell are called *strands*. Strands are the basic data structure building blocks considered by DSI and highlighted in Fig. 2(a) by the thick coloured arrows that traverse through nodes and that are labeled S_1 to S_6. To infer a data structure shape, the relationships between strands need to be identified, which we term *strand connections* (SCs). A strand connection describes exactly one way in which multiple cells of a strand are related. It aggregates all *cell pairs* of two strands that have the exact same relation; thus, a strand connection consists of a set of cell pairs. Observe that two strands can be related in more than one way, resulting in multiple strand connections between the strands. We construct a *strand graph*, where vertices represent strands and edges represent strand connections (see Fig. 2(b)). The strand connections with the same edge style denote the same relationship type; for example, the DLL strands form a tight strand connection that is bi-directional, e.g., S_3 can be reached from S_2 and vice-versa. Two kinds of loose and uni-directional strand connections are formed between the parent SLL and each child DLL; for example, S_1 cannot be reached from S_2.

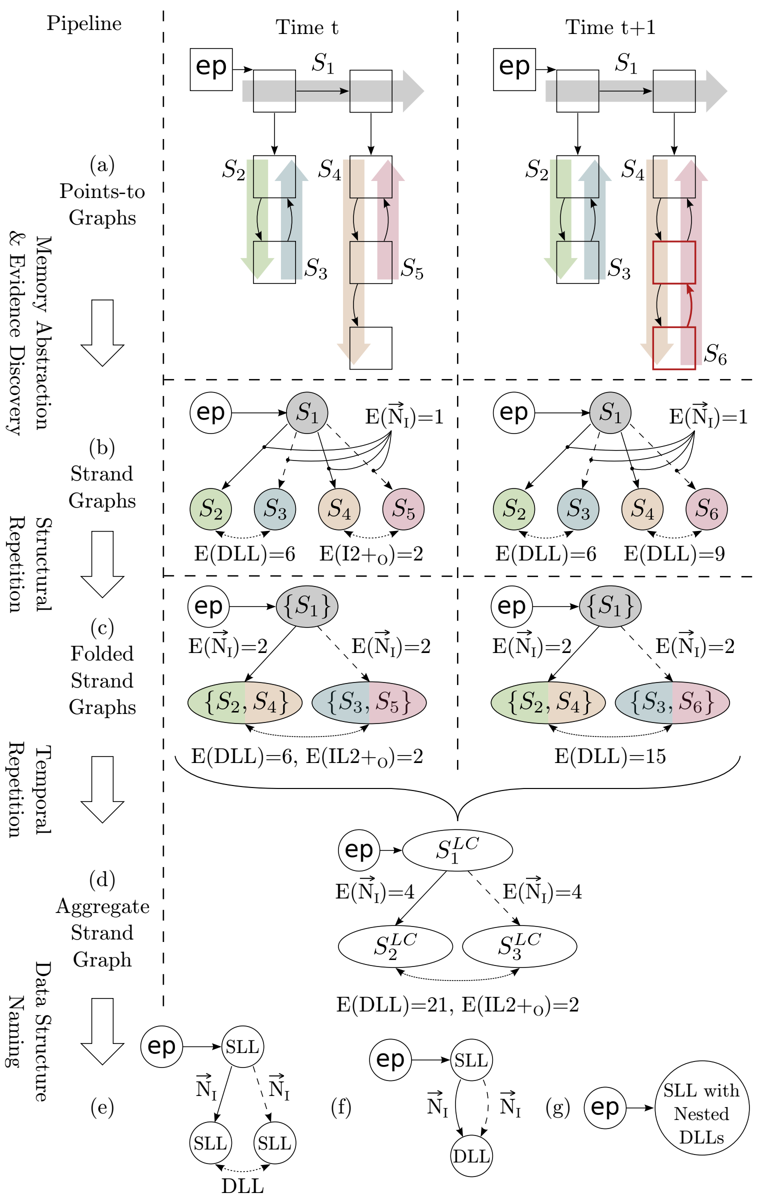

Fig. 2. Left: DSI's pipeline; right: the example of Sec. 3. (Figure adopted from [28].)

In the following, data structures and their interconnections are consolidated under the term *memory structure*. A memory structure observation is made on the strand graph by matching (parts of) the graph against a set of shape predicates. For example, our DLL predicate requires exactly two strands running in opposite directions and connected by a tight strand connection. The predicates available to DSIcore are defined by our data structure taxonomy presented in [28], which also describes the precedence rules on predicates when matching data structures; these are needed to avoid ambiguities and not inadvertently interpret, e.g., a binary tree as parent-child nesting of SLLs.

Evidence collection. Each memory structure observed in a strand graph is associated with a corresponding memory structure label and an evidence count E. The evidence is recorded on the strand connections that are matched by the shape predicate. The evidence count is derived from the structural complexity of the memory structure on the basis of the number of cell pairs comprising the strand connection and the way in which these are accessed by the shape predicate. Evidence weights for our example, are shown in Fig. 2(b). The degenerate DLL in time step t has an evidence count of 2 for "intersecting on two nodes overlay" ($\mathrm{I2{+}_O}$), where the number of connections between the two strands is required to be at least two and the count is simply the number of cell pairs in the connection. The regained stable DLL shape at $t + 1$ is matched by a predicate checking that the forward/reverse property of the DLL holds for each cell pair in the strand connection. This results in a count of 3 for each cell pair: 1 for the existence of the cell pair, plus 2 because both cells in the cell pair must be analysed. As three cell pairs exist, the total evidence count is $3 * 3 = 9$. Nesting ($\mathrm{N_I}$) only requires the existence of one cell pair connecting the parent strand to the child; hence, each occurrence has a count of 1. This evidence is spread over each strand connection that participates in the parent-child nesting and is then accumulated via *structural* and *temporal repetition*:

- Regarding structural repetition, DSI detects and groups those elements of the strand graph that perform the same role within one time step. Consider, e.g., the forward strands $\{S_2, S_4\}$ and backward strands $\{S_3, S_5\}$ of the DLL children at time t in Fig. 2. The grouping is done via a merge algorithm that produces a *folded strand graph* (see Fig. 2(c)). The folding results in vertices that now consist of strand *sets* and merged strand connections, where all evidence counts are summed up. The aggregation of the strand connections reinforces the evidence and is part of our solution to rule out degenerate shapes during data structure operations, as those parts of the data structure that are in stable shape can be aggregated with others that are in degenerate shape. In practice, the degenerate shape is in the minority, such that the majority of stable shapes overrides the minority.
- To track the temporal behavior of a memory structure and enable the identification of temporal repetition, it must be determined which strands represent the same data structure building block over multiple time steps. We tackle this problem by considering the labeling from the point of view of

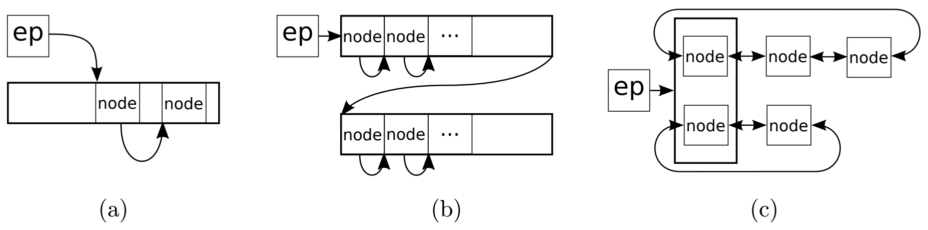

Fig. 3. Complications of C heaps: (a) custom allocator, (b) cache efficient list [7], (c) Linux kernel cyclic DLL [2]; `ep` denotes an entry pointer. (Figure adopted from [28].)

each entry pointer to a data structure separately, because entry pointers are inherently stable over their lifetimes. For each entry pointer and each time step in which an entry pointer exists, temporal repetition is performed by extracting the folded strand graph's subgraph reachable from the entry pointer. All extracted subgraphs are then merged over the lifetime of the entry pointer, resulting in an *aggregate strand graph*, in which the identified evidences are again accumulated. This is the second part of the evidence reinforcement to rule out degenerate shapes. The vertices of the aggregate strand graph are abstract representations of the strands in terms of their *linkage conditions* (S^{LC}).

Our example's aggregate strand graph is shown in Fig. 2(d), where the evidence for DLL is overwhelming. At the end of an entry pointer's lifetime, we interpret DSI's analysis by choosing the label with the highest evidence for each strand connection and label each strand as SLL or cyclic SLL (CSLL) (see Fig. 2(e)).

Naming of data structures. While the aggregate strand graph is useful for program comprehension, a natural language annotation of the program's source code at an entry pointer declaration, which names the data structure to which the entry pointer points, can be preferable. Of course, this assumes that the source code is available. Our according *naming* component iteratively groups the vertices of the aggregate strand graph and assigns a textual label to the resulting group; these grouped elements now form an atomic vertex in subsequent groupings. For example in Fig. 2(f), the result of grouping the two vertices connected by a DLL strand connection is shown. The order in which vertices are grouped must be carefully chosen so that the most suitable naming is generated; see [28] for details. Ultimately, we end up for our example with the graph in Fig. 2(g).

4 DSIsrc: DSI Front End for Analysing C Source Code

DSIsrc instruments the source code using the CIL framework [21], inserting instructions for recording pointer writes and memory (de-)allocations both on the heap and the stack. Subsequently, the instrumented source code is compiled

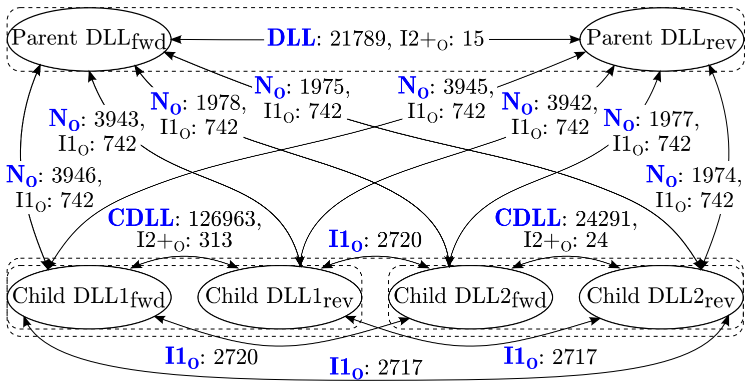

Fig. 4. Aggregate strand graph for `libusb`. (Figure adopted from [28].)

and executed to capture the programs event trace that is then analysed by DSIcore. The following paragraphs are adapted from [28], where a more detailed description of DSIsrc can be found.

The C heap. The reason for choosing C as a concrete realization for DSI is that it is far less restrictive regarding heap manipulations than other programming languages such as C# or Java. This results in programs that utilize the freedom of pointer arithmetic, type casts, macro usage, and (customized) memory allocations, and that are quite often favoring performance over source code readability. Together with C's widespread use in operating systems code and in Unix systems, the amount of difficult-to-comprehend legacy code is overwhelming, making DSI's capabilities a desirable feature for software developers.

A common assumption of related work is that one node of a data structure corresponds to one allocated chunk of memory. This assumption breaks in case multiple nodes of a data structure are placed inside a memory chunk, which is often the case with (i) custom memory allocators (see Fig. 3(a)), (ii) cache-efficient data structures [7] (see Fig. 3(b)), and (iii) head nodes of multiple lists embedded in the same memory chunk (see Fig. 3(c)). The latter is common practice in the cyclic Linux kernel list [2], which embeds the linkage struct `list_head` inside another struct. To model these situations, it is mandatory to allow nodes to only cover some sub-region of memory and permit that the linkage offsets are always from the start of the enclosing sub-region instead of, as is done in [14,17,19], the start of the enclosing memory chunk. DSI explicitly supports such heap states, which are typical for low-level C programs, with its cell-based strand abstraction.

Example. `libusb` [1] is one of the most challenging examples from the DSI benchmark [28]. This benchmark comprises textbook examples, self-written synthetic examples, Forester/Predator examples [4], and the real-world programs `bash` and `libusb`. The latter is a user space usb library, which comprises almost 7k lines of C code. We exercised this code using the included utility `listdevs`, modified to expose several `struct libusb_contexts`. The main data structure of `libusb` forms a parent CDLL, where both child elements for the parent form a CDLL: one for devices and one for associated file descriptors. DSI's output, as depicted in Fig. 4, clearly indicates this structure. Note that the CDLLs are Linux CDLLs that are embedded in structs, thus requiring our cell abstraction to understand the cyclic nature of the lists.

5 DSIbin: DSI Front End for Analysing x86 Binaries

DSIbin opens up our DSI approach to x86 binaries. The challenge here is not the binary format itself but the absence of any type information that is needed by DSIcore for detecting and tracking cells and strands.

Front end. To provide DSIcore with the required information when inspecting binaries, we utilize Intel's Pin framework [18] for capturing, as before, pointer writes and memory (de-)allocations. DSIcore demands low-level type information including types of (nested) structs both on the heap and stack, which is unavailable from stripped binaries. Therefore, DSIbin relies on the type excavator Howard [25] to recover this information. One fundamental problem when inferring types is to identify whether some types are the same. This happens in case one type is allocated at different locations within the binary. Because no explicit information is present in the binary that indicates type equivalence, this needs to be inferred separately. We modified Howard mildly to perform type merging between identified structs by tracking whether instructions and pointers touch binary compatible memory chunks from different allocation sites, which then get merged. However, some situations are not covered; for example, Howard does not merge nested types and misses nested structs when the access pattern is ambiguous, e.g., when a nested struct is placed at the head of the surrounding struct. To overcome these limitations, we devised the type refinement component DSIref.

DSIref: Refinement of type information. DSIref uses Howard's inferred types as starting point. Pointer connections reveal information about the data structure layout; for example, an incoming pointer to the middle of a data structure might indicate a nested struct or a linkage between two objects of different types. Struct types as reported by Howard could be merged when they are binary compatible, i.e., they have the same size and fields of the same primitive data types. Because pointer connections can be ambiguous, we create a set of possible type interpretations and then select the most plausible one. The latter is done by evaluating each interpretation with DSI and choosing the structurally most complex data structure as the most plausible interpretation. The intuition is that correct type merges and nested type detections naturally reveal the most 'complex' data structure. For example, when considering Fig. 6, the cyclicity of the lists can only be detected when the grey nodes are correctly identified.

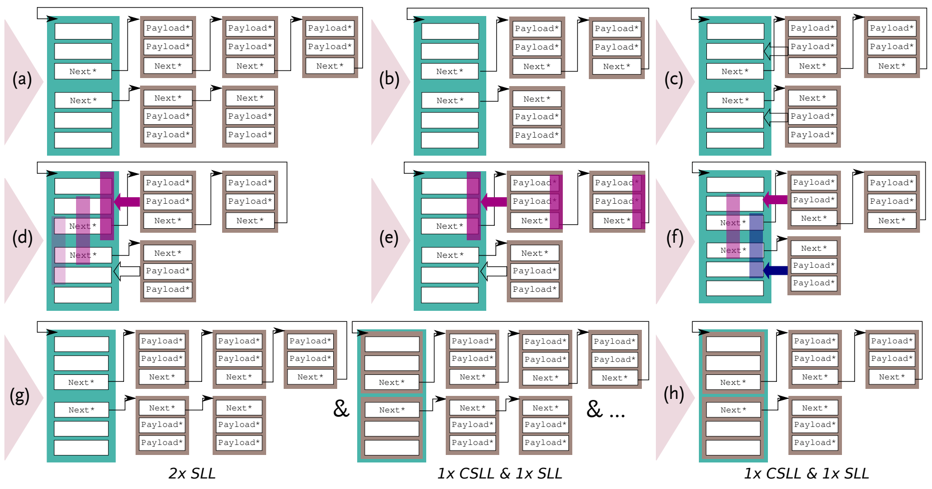

Fig. 5. Overview of the DSIref approach: (a) create sequence of points-to-graphs from program execution (only one shown); (b) construct merged type graph capturing pointer connections between types; (c) exploit pointer connections by mapping type sub-regions (two possibilities shown); (d) observe that multiple interpretations may be possible; (e) propagate each interpretation along pointer connections; (f) rule out inconsistencies; (g) evaluate remaining interpretations via DSI; (h) choose the 'best' interpretation in terms of data structure complexity (indicated by merged type graph with resulting label *1x CSLL & 1x SLL*). (Figure adopted from [23].)

The refinement process involves eight phases (Phases (a)–(h)), see Fig. 5. In Phase (a), DSIref takes all the 'as is' type information from Howard. In Phase (b), DSIref constructs a *merged type graph* similar to [5, 8, 22], where types are vertices and pointers are edges. Each type occurs only once, which reflects the connections between all types that occur in the execution trace under analysis. Notably, heap and stack allocated types are transparent to the merged type graph. This allows us to merge both, something that is not considered in related work [9]. Phases (c)–(f) utilize the merged type graph in order to generate new possible type interpretations. The first of these phases maps binary compatible sub-regions between different types by following pointer connections. Such mappings might be ambiguous as can be seen by the three possible mappings in Phase (d). Consequently, each of the mappings results in one type interpretation. In Phase (e), these types are then propagated along pointer chains as long as binary compatibility allows further propagation. This merges arbitrary combinations of (nested) struct types distributed over the heap and stack. Note that type interpretations can be conflicting as shown in Phase (f), where two interpretations are overlapping. These conflicts are eliminated as nested structs cannot overlap. All remaining interpretations and also Howard's initially inferred interpretation are evaluated with DSI, and the structurally most complex data

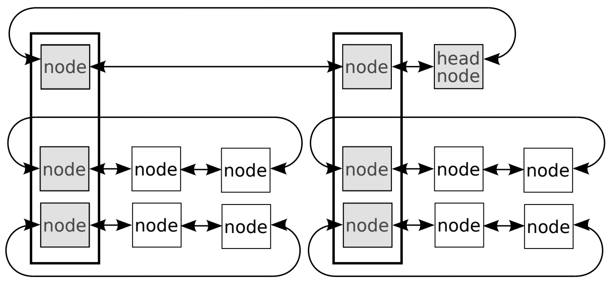

Fig. 6. CDLL parent with two CDLL children per parent node [4].

structure interpretation is selected. This is shown in Phase (h), where the cyclic SLL is chosen over the non-cyclic SLL interpretation.

Example. One challenging example from our benchmark for DSIbin [23] is taken from the literature [4] and happens to have the same dynamic data structure as `libusb` (see Fig. 4). A points-to graph of this example is shown in Fig. 6, where the parent CDLL is located at the top. It consists of two payload nodes, both having two nested CDLL children, and the external head node of the list. The nodes highlighted in grey are required to detect the correct shape of the data structure and are subject to the refinement performed by DSIref. Without DSIref, the parent nested structs and the external head struct are not detected to be of the same type, which results in missing the cyclic property of the DLL, i.e., the external head is cut off. The same is true for the grey head nodes of the child elements, with the addition that the overlay nesting is missed and only a "nesting on indirection" is reported. Therefore, without the refinement, a "DLL with two indirect nested DLL children" is detected. When employing DSIref, the grey nodes are also typed correctly, resulting in the identification of the data structure's true shape, i.e., a "CDLL with two overlay nested CDLL children."

6 DSI's Back Ends

The artefacts generated by DSIcore can be passed to various back-end components (see Fig. 1). Within the DSI project, we have developed early prototypes for operation detection, formal verification, and visualization. The *operation detection* component observes repetitive changes in data structures that occur due to insertion and deletion operations. The repetitive behaviour is identified via a multi-dimensional pattern matching approach driven by a genetic algorithm [11].

Detecting data structure operations is key for an important use case of DSI, namely the automated generation of source code annotations for formal program verification, such as pre-and post-conditions. The intention is to free a verification engineer from the burden of providing the often rather straightforward annotations that relate to (preserving) data structure shape. The utility

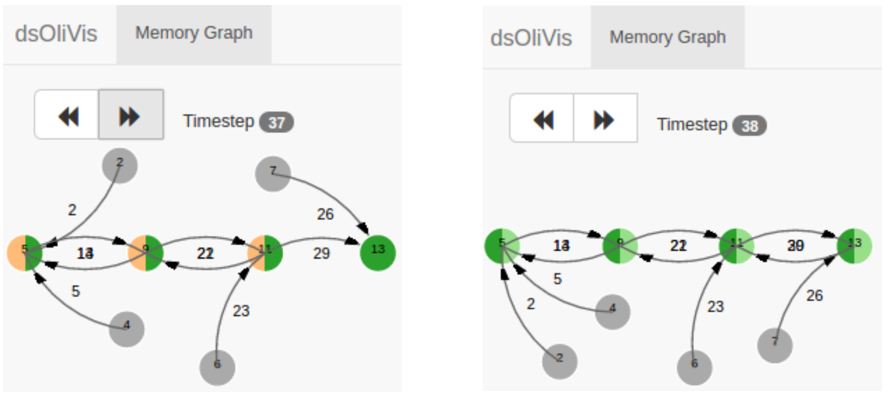

Fig. 7. Visualization of the transition of a DLL from a degenerate shape (left) to a stable shape (right). (Figure adopted from [26].)

of this DSI use case had already been demonstrated by interfacing DSI's predecessor tool dsOli [27] to the verification tool VeriFast [20], and DSI's verification component has been prototyped in [6].

Our third prototypical back end implements an advanced visualization approach [26] that uses the data structure information inferred by DSIcore to consistently display a data structure over its lifetime. This allows for a stepwise inspection even during periods of degenerate shapes, where our layout algorithm still arranges the data structure according to the final data structure interpretation. This is in contrast to, e.g., the *dot* renderer of Graphviz [13] which might rearrange the graph from one time step to the next. An example of adequately visualizing a DLL during an insertion operation is shown in Fig. 7, where grey nodes represent the entry pointers into the DLL. Observe that all DLL nodes, i.e., the nodes 5, 9, 11, and 13, are displayed consistently across both time steps.

7 Conclusions

We provided an overview of the novel DSI approach for reliably identifying dynamic data structures in pointer programs, which we envisage to be a significant help when comprehending C source code or reverse engineering binaries of pointer programs. The data structures in scope for DSI contain list-based structures such as (cyclic) singly and doubly linked lists (with head and/or tail pointers), binary trees, skip lists, and more complex data structures that are build from the mentioned ones via indirect and overlay nesting.

DSI advances over related work in terms of accuracy and data structure variety have been enabled by DSIcore's fine-grained memory abstraction in terms of cells, strands, and strand connections, and by DSI's evidence-collecting approach to data structure identification. In addition, we compensated the loss of type information in binaries via a novel combination of the type excavator Howard with DSIcore, whereby the high-level data structure information obtained by DSIcore is employed to infer low-level type information such as nested struct types. We refer the interested reader to our publications on DSIsrc [28] and DSIbin [23, 24] for more details on the DSI approach and its evaluation. DSI is open source and available for download from `http://www.swt-bamberg.de/dsi`.

The present paper also briefly discussed our proof-of-concept back end implementations, demonstrating how DSI's analysis results can be used for operation detection [11], verification condition generation [6], and data structure visualization [26]. Regarding additional future uses cases for DSI, we envision to employ DSI for generating malware signatures so as to detect polymorphic malware families. This use case is inspired by the virus scanner Laika [10], which detects structure on pointer connections but does not identify the data structures themselves. Finally, we wish to investigate how far DSI's rich analysis can contribute to memory leak detection. DSIcore already performs its own memory leak detection, allowing it to exactly record where and when a reference to memory is lost. By integrating this information into DSI's data structure visualization, a powerful tool could emerge that allows software developers to replay the steps leading to a memory leak and thus to find the leak's cause.

Acknowledgments. We thank the German Research Foundation (DFG) for their support of the DSI project (DFG grant LU 1748/4-1), and also Linus Dietz and Kathrin Welzel who contributed to the project in the context of their bachelor and master theses.

References

1. libusb 1.0.20. `http://www.libusb.info/`. Accessed: 8th May 2017.
2. Linux kernel 4.1 cyclic DLL (`include/linux/list.h`). `http://www.kernel.org/`. Accessed: 31 August 2015.
3. Malware statistics by AV-TEST. `https://www.av-test.org/en/statistics/malware/`. Accessed: 16th June 2017.
4. Predator/Forester GIT repository. `https://github.com/kdudka/predator`. Accessed: 8th May 2017.
5. Aftandilian, E. E., Kelley, S., Gramazio, C., Ricci, N., Su, S. L., and Guyer, S. Z. Heapviz: Interactive heap visualization for program understanding and debugging. In *SOFTVIS '10*, pp. 53–62. ACM, 2010.
6. Boockmann, J. Automatic generation of data structure annotations for pointer program verification. Bachelor thesis, U. Bamberg, Germany, October 2016.
7. Braginsky, A. and Petrank, E. Locality-conscious lock-free linked lists. In *ICDCN '11*, vol. 6522 of *LNCS*, pp. 107–118. Springer, 2011.
8. 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, Spain, 2012.
9. Caballero, J. and Lin, Z. Type inference on executables. *ACM Computing Surveys*, 48(4):1–65, 2016.
10. Cozzie, A., Stratton, F., Xue, H., and King, S. Digging for data structures. In *OSDI '08*, pp. 255–266. USENIX Association, 2008.
11. Dietz, L. Multidimensional repetitive pattern discovery for locating data structure operations. Bachelor thesis, U. Bamberg, Germany, April 2015.

12. Dudka, K., Peringer, P., and Vojnar, T. Byte-precise verification of low-level list manipulation. In *SAS '13*, vol. 7935 of *LNCS*, pp. 215–237. Springer, 2013.
13. Gansner, E. R. and North, S. C. An open graph visualization system and its applications to software engineering. *Software-Practice and Experience*, 30(11):1203–1233, 2000.
14. Haller, I., Slowinska, A., and Bos, H. Scalable data structure detection and classification for C/C++ binaries. *Empirical Software Engineering*, 21(3):778–810, 2016.
15. Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., and Vojnar, T. Fully automated shape analysis based on forest automata. In *CAV '13*, vol. 8044 of *LNCS*, pp. 740–755. Springer, 2013.
16. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., W. Penninckx, W., and Piessens, F. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NFM '11*, vol. 6617 of *LNCS*, pp. 41–55. Springer, 2011.
17. Jung, C. and Clark, N. DDT: Design and evaluation of a dynamic program analysis for optimizing data structure usage. In *MICRO '09*, pp. 56–66. ACM, 2009.
18. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Notices*, 40(6):190–200, 2005.
19. Marron, M., Sanchez, C., Su, Z., and Fähndrich, M. Abstracting runtime heaps for program understanding. *IEEE Transactions on Software Engineering*, 39(6):774–786, 2013.
20. Mühlberg, J. T., White, D. H., Dodds, M., Lüttgen, G., and Piessens, F. Learning assertions to verify linked-list programs. In *SEMF '15*, pp. 37–52. Springer, 2015.
21. 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 '02*, vol. 2304 of *LNCS*, pp. 213–228. Springer, 2002.
22. Raman, E. and August, D. I. Recursive data structure profiling. In *Workshop on Memory System Performance*, pp. 5–14. ACM, 2005.
23. Rupprecht, T., Chen, X., Boockmann, J. H., Lüttgen, G., and Bos, H. DSIbin: Identifying dynamic data structures in C/C++ binaries. In *ASE '17*. IEEE, 2017. Accepted for publication.
24. Rupprecht, T., Chen, X., White, D. H., Mühlberg, J. T., Bos, H., and Lüttgen, G. POSTER: Identifying dynamic data structures in malware. In *CCS '16*, pp. 1772–1774. ACM, 2016.
25. Slowinska, A., Stancescu, T., and Bos, H. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS '11*. The Internet Society, 2011.
26. Welzel, K. Heap visualisation using interactive memory graphs. Bachelor thesis, U. Bamberg, Germany, March 2016.
27. White, D. H., Rupprecht, T., and Lüttgen, G. dsOli2: Discovery and comprehension of interconnected lists in C programs. In 18th Coll. on Programming Languages and Foundations of Programming (Kolloquium Programmiersprachen, KPS '15), 2015. Proceedings available online at `http://www.complang.tuwien.ac.at/kps2015/proceedings`.
28. White, D. H., Rupprecht, T., and Lüttgen, G. DSI: An evidence-based approach to identify dynamic data structures in C programs. In *ISSTA '16*, pp. 259–269. ACM, 2016.