DSIbin: Identifying dynamic data structures in C/C++ binaries

Rupprecht, Thomas; Chen, Xi; White, David H.; Boockmann, Jan H.; Luttgen, Gerald; Bos, Herbert

published in
2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)
2017

DOI (link to publisher)
10.1109/ASE.2017.8115646

document version
Publisher's PDF, also known as Version of record

DSIbin: Identifying Dynamic Data Structures in C/C++ Binaries

Thomas Rupprecht
University of Bamberg,
Germany
trupprecht@swt-bamberg.de

Xi Chen
VU University Amsterdam,
The Netherlands
x.chen@vu.nl

David H. White
University of Bamberg,
Germany
dwhite@swt-bamberg.de

Jan H. Boockmann
University of Bamberg,
Germany
jboockmann@swt-bamberg.de

Gerald Lüttgen
University of Bamberg,
Germany
gluettgen@swt-bamberg.de

Herbert Bos
VU University Amsterdam,
The Netherlands
herbertb@cs.vu.nl

***Abstract*—Reverse engineering binary code is notoriously difficult and, especially, understanding a binary's dynamic data structures. Existing data structure analyzers are limited wrt. program comprehension: they do not detect complex structures such as skip lists, or lists running through nodes of different types such as in the Linux kernel's cyclic doubly-linked list. They also do not reveal complex parent-child relationships between structures. The tool DSI remedies these shortcomings but requires source code, where type information on heap nodes is available.**

We present DSIbin, a combination of DSI and the type excavator Howard for the inspection of C/C++ binaries. While a naive combination already improves upon related work, its precision is limited because Howard's inferred types are often too coarse. To address this we auto-generate candidates of refined types based on speculative nested-struct detection and type merging; the plausibility of these hypotheses is then validated by DSI. We demonstrate via benchmarking, that DSIbin detects data structures with high precision.

***Index Terms*—Data structure identification, reverse engineering, dynamic data structures, pointer programs**

I. Introduction

Understanding the internals of software binaries is an important challenge, especially for tackling challenges in comprehending legacy code [18] and security threats posed by malware [14]. A particular challenge in reverse engineering is the identification of *dynamic data structures* (DS) in pointer programs and, in this context, also of nested structs that occur frequently in C/C++ code, e.g., the Linux kernel's cyclic doubly-linked list (CDLL). The demand for automated analysis of pointer programs can, e.g., be seen by the recent acquisition of Infer by Facebook [1]. While there is a wealth of related work on *type recovery* [12], [20], [24], [26], [31], [34], [35], with Divine [12] and Howard [34] being examples of such static and respectively dynamic analysis tools, *DS identification* tools are scarce [16], among which MemPick [21], DDT [22] and ARTISTE [15] are examples, of state-of-the-art tools for dynamic DS identification. They are based on a dynamic analysis but have limitations (Sec. II). Firstly, some make strong assumptions on the binary under analysis; for example, DDT requires that interface functions can be revealed easily, which is, e.g., the case when the C++ Standard Template Library (STL) has been used. However this assumption is not necessarily true for low-level, inlined or optimized code. Secondly, tools such as ARTISTE are not robust against DS operations that temporarily break the structure's shape invariant. Thirdly, the mentioned tools cannot identify lists if these run through differently typed nodes, for which the Linux CDLL is an example. They also do not recognize complex relationships between structures such as arbitrary parent-child nesting.

The program comprehension tool DSI [38] implements a dynamic analysis that overcomes these limitations but works on C sources only. Its heap abstraction breaks with the common assumption that each node of a dynamic structure resides in a memory chunk of its own; instead, it employs a notion of *cell* that can, e.g., be either a struct or a nested struct. Moreover, DSI does not analyze at the node level, but uses *strands* – which can be thought of singly-linked lists – as the building blocks of a DS. Strands can be interconnected loosely, i.e., via pointer-based nesting, or tightly, i.e., via overlay. For example, DSI represents a doubly-linked list (DLL) as a strand graph that consists of two strands that run in opposite directions and are connected by 'nodewise' overlay. Its subgraphs are then annotated by DSI with quantitative evidence for a structure having a certain interpretation. This evidence is accumulated by structural and temporal repetition not unlike as in ARTISTE (Sec. II), which typically leads to overwhelming evidence for a DS's true shape.

This paper develops the novel tool chain DSIbin that enables DSI's core algorithm to work on binaries. Our first 'naive' approach simply replaces the source code instrumentation framework CIL [29] that is employed by DSI with Intel's Pin binary instrumentation framework [27], and utilizes Howard [34] for extracting type information that is then used by DSI (Sec. III). We evaluate this naive tool chain on an benchmark, comprising of real-world examples [7], [4], [10], standard textbook examples [39], [36], examples taken from the shape analysis literature [8], and handwritten examples

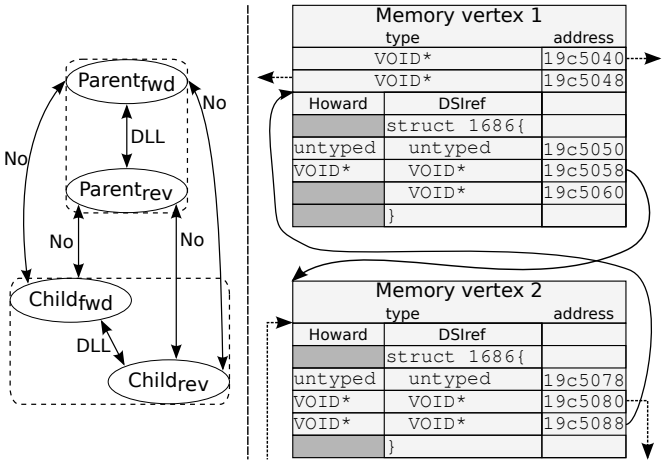

Fig. 1. DSI's aggregate strand graph (left) and a snippet of the points-to graph (right) for example `syn-08`: "Parent DLL with nested child DLLs".

that challenge our tool chain in various aspects. Our results demonstrate that the naive tool combination leads to the correct identification of 10 of the benchmark's 30 examples, which already improves upon the capabilities of related tools, especially regarding the recognition of skip lists and various forms of nesting. A detailed analysis of the 20 negative examples reveals that recognition fails due to insufficient and imprecise type information excavated by Howard, particularly regarding nested structs and type mergings.

Therefore, we propose a new approach for refining the low-level type information excavated by Howard which utilizes DSI's core algorithm (Sec. IV-B). We infer some of the missing type information by propagating existing information within a type graph [11], [15], [31]. This leads to speculative type mergings and struct nestings, which we term *type hypotheses*. Their plausibility is then quantitatively evaluated via DSI's core algorithm, which identifies the most probable correct typing on the basis of the complexity of the DS that is implied by a typing hypothesis. We complete DSIbin by adding this refinement component DSIref to our naive tool chain, and can now correctly identify the DSs in 26 of the 30 benchmark examples.

We briefly illustrate our improvement in the state-of-the-art of dynamic DS identification and related type recovery mechanisms. Consider example `syn-08` of our benchmark, which contains a "Parent DLL with nested DLL children" (Fig. 1) and is not recognized by other tools. DSIbin reports the DS in an aggregated form of a strand graph (figure's left-hand side). It reveals the strand aggregates `Parent`$_{\texttt{fwd}}$, `Parent`$_{\texttt{rev}}$, `Child`$_{\texttt{fwd}}$, `Child`$_{\texttt{rev}}$, the first and second pair of which each forms a DLL (`DLL` label between the `fwd` and `rev` strands). It also highlights the nesting-on-overlay (`No`) between the parent and child DLLs, where the head node of the child DLL is embedded inside the parent node. This correct interpretation is only possible due to DSIref's type refinement. To see this, consider the snippet of a points-to graph that DSIbin constructs when analyzing `syn-08` (figure's right-hand side), where a parent vertex is displayed at the top and a child vertex at the bottom. We have annotated the type information extracted by Howard and DSIref, resp., which shows how much more information DSIref reveals. Firstly, it reveals a nested struct encompassing addresses 19c5050 to 19c5060 (the nested head of the child DLL). Secondly, this enables DSIref to identify the equality between the type of vertex 2 and the type of the nested struct of vertex 1 and, thus, to detect the child DLL.

II. Background & Related Work

Several dynamic analyses of binary programs have been introduced for identifying contained dynamic DSs, in particular MemPick [21], DDT [22] and ARTISTE [15]. We first discuss their underlying analysis approaches and point out their limitations, then introduce the DSI approach [38] that overcomes these limitations but requires source code and some type information included therein. This leads us to surveying tools such as Howard [34] that excavate type information from binaries and that will ultimately enable our desired application of DSI to binaries.

MemPick, ARTISTE & DDT. MemPick aims at determining the periods in the program trace under consideration when pointer operations are absent. For these *quiescent* periods, it considers the heap's structure as points-to graphs, tries to type the graphs' nodes on the basis of information revealed by CPU instructions, and clusters the nodes according to types. Each cluster is then processed via a set of rules to identify the corresponding DS. For example, if an analyzed cluster contains only nodes with two outgoing arcs and one incoming arc, plus one node with two outgoing arcs but no incoming arc, then MemPick reports the cluster as a binary tree. However, MemPick does not analyze the relationships between DSs, such as parent-child nesting, and thus cannot identify more complex DSs such as the Linux kernel CDLL that involves nodes of different types.

ARTISTE samples every n^{th} step of the executed trace, which does away with quiescence analysis but may result in considering execution points at which the dynamic DSs are in some degenerate shape, i.e., a shape that arises in the middle of a DS manipulation and temporally breaks the DS's invariant. The points-to graphs at the sampled execution points are checked for structural repetition within a graph and temporal repetitions across graphs, where subgraphs of the same structure are folded. The naming of the overall DSs is then left to a rule-based selection algorithm. While this approach can discover even complex structures involving some nesting, ARTISTE looses precision if sample points containing degenerate shapes are picked.

DDT aims at finding interface functions to DSs, i.e., functions that, e.g., insert and remove nodes from a dynamic data structure. As a prerequisite, the types of the nodes in the points-to graphs of a trace are inferred by tracking allocation sites during program execution, and merging types from different allocation sites, e.g., if nodes are accessed through a common interface function. Thereby, DDT can build

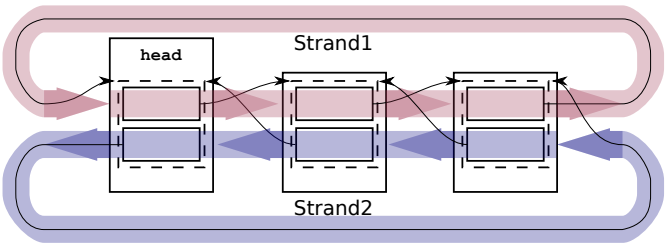

Fig. 2. DSI's view of the Linux kernel CDLL.

up signatures of interface functions, which are then matched against a library of known DSs for identification. This works well if interface functions are easily revealed, e.g., if programs use well-structured libraries such as the C++ STL. However the approach is hampered in the presence of obfuscation, inlining, or optimization, which are often employed in legacy and kernel-space software.

DSI: Data Structure Investigator. DSI aims at overcoming the abovementioned limitations of DS identification, but operates on C code rather than binary code. The workings of DSI are not unsimilar to ARTISTE, but the detection of structural and temporal repetition is not directly conducted on points-to graphs but instead on *strands*. Strands are an abstraction from the core ingredient of dynamic DSs: singly-linked lists. During program execution, DSI tracks strands and their interconnections, such as overlay or nesting, in a strand graph. It is then the evolving sequence of strand graphs on which repetition detection is performed and on which folding takes place.

On the one hand, the difference of DSI to competing tools lies in the finer, more expressive concept of strands. A strand links *cells* rather than nodes, which can be thought of as memory subregions: while a node is normally a struct, a cell could be, e.g., the whole struct or just a nested struct within the whole struct. Thus, strands can traverse through nodes of different types, while the traversed cells themselves are obviously required to be of the same type. Fig. 2 shows DSI's view of the recognized Linux kernel CDLL, where strands are drawn using wide, colored arrows and involve linkages between nodes of different types (head node vs. tail nodes).

On the other hand, DSI does not rely on quiescent periods but instead employs an evidence-based approach that considers all points of an execution trace. At each point, a subgraph of the strand graph is given a count on the basis of its strands' sizes and their kinds of interconnections, which reflects the likelihood of the subgraph representing a certain DS; for example, the count for the subgraph capturing a DLL might be higher than the count for two singly-linked lists (SLLs) intersecting. These counts, or evidences, are accumulated when folding strand graphs structurally within a strand graph and temporally across all strand graphs of the considered execution trace. This typically leads to overwhelming evidence for the true DS shape, and makes DSI's approach robust against degenerate DS shapes. These degenerate shapes are just injecting interpretation noise that is becoming irrelevant when accumulating evidence.

In summary, DSI overcomes the limitations of related work in that it (i) does not make strong assumptions on the program under analysis, (ii) is not mislead by temporarily degenerate shapes, and (iii) is more general as it supports the identification of lists through different node types and arbitrarily nested combinations of lists. Because of this, DSI can reliably handle custom implementations including those involving (cyclic) singly-/doubly-linked lists, various skip lists and binary trees, and interconnections between those, including indirect and overlay parent-child nesting. Consequently, DSI can deal with real-world software, e.g., the Linux kernel CDLL [6], the region clipping library of VNC (`hvnc2/libs/libvncsrv/rfbregion.c` found in Carberp [10]) and libusb [5].

DSI would be a useful tool for understanding complex code such as malware [33], but currently – and in contrast to the related work discussed above – it cannot handle binary code. While its instrumentation that captures pointer-based events such as memory (de)allocations and pointer writes can be switched from the C Intermediate Language (CIL) [29] to, e.g., Intel's Pin framework [27], the core DSI algorithm requires type information on cells, which is accessible in source code but not in (stripped) binaries. Thus, the essential step for our desired opening of DSI to binaries is the type recovery for structs and nested structs.

Type recovery tools. A multitude of tools exist for recovering type information from binaries [12], [20], [24], [26], [31], [34], [35]. They differ in (i) whether a static, a dynamic, or a combined analysis is performed, (ii) whether they operate on a binary file or a memory snapshot, and (iii) the spectrum of discovered type information, ranging from simply identifying pointers and sizing memory chunks to detecting structs and their primitive types. A closer investigation reveals that especially the tools Divine [12] and Howard [34] can deal with nested structs.

Divine conducts a static analysis of Windows binaries and discovers types of memory regions, including regions on the heap, which may be primitive data types or complex types, e.g., consisting of arbitrarily nested structs. Its machinery combines a classic value-set analysis with an algorithm for aggregate structure identification in an iterative manner. Memory access patterns in a binary are exploited for making educated guesses as to how data is laid out in memory. For example, if a code instruction accesses a sequence of eight bytes at a particular offset this corresponds to a variable or field of this size at that location within the region. However, memory access patterns can be blurred by `memcpy`-like functions, which restricts Divine's utility [34].

In contrast, Howard conducts a dynamic analysis but handles `memcpy` and similar functions in C/C++ binaries, in accordance with its practical aims of supporting forensics, reverse engineering and protecting existing binaries against memory corruption attacks. Howard also tracks memory access patterns and, additionally, monitors mallocs for heap

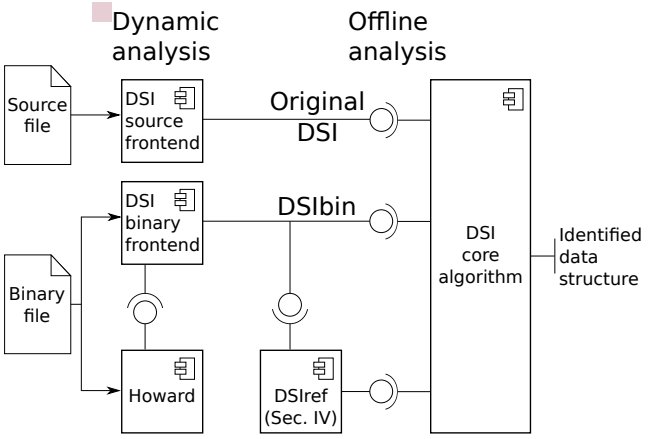

Fig. 3. Overview of our tool chain.

memory, stack frames for local variables, and pointers to reveal primitive types and nested structs. For typing heap memory, Howard identifies allocation sites by their call stack and stack frames by their associated function address. Howard reports the typed memory for each allocation site and stack frame.

Howard is arguably the best candidate for extracting type information for use by DSI: it (i) supports nested structs, (ii) deals with C and C++ binaries that include `memcpy`-like functions, and (iii) operates under Linux as does DSI. Below we see that Howard still has some limitations and, show that DSI's core algorithm can be employed to improve the type information reported by Howard.

III. Naive Approach: Howard+DSIbin

This section describes a naive approach for combining DSI and Howard, evaluates it on an benchmark, and discusses the results. Our observations pave the way for a much improved approach (Secs. IV & V).

A. Howard+DSIbin Tool Chain

The naive tool chain is depicted in Fig. 3 when ignoring component DSIref, and consists of three components: the core DSI algorithm that was already used in the source code DSI version, a newly created DSI binary frontend, and Howard. We refer to the core DSI algorithm simply as DSI and to the binary frontend, as well as the overall tool chain, as DSIbin. Recall that DSI performs an offline-analysis for detecting dynamic data structures. For this purpose, the original source-code version of DSI is provided with an execution trace captured by executing instrumented C source code, where the instrumentation of memory (de)allocation and memory-write events is done via the CIL framework [29]. We treat DSI as a black box and replace the instrumentation part with DSIbin, which utilizes Intel's Pin framework [27]. Because DSI relies on primitive data type information including structs and nested structs on the heap and stack, DSIbin needs to provide this information as well. Therefore, DSIbin not only instruments the binary under investigation but also incorporates type information excavated by Howard.

Technically, DSIbin models heap and stack separately to keep track of memory (de)allocations and stack status. All live heap objects are recorded, allowing for sanity checks such as checking for dangling references. The stack is modeled in a way that enables one to keep track of live stack variables and to handle situations like the *red zone optimization* [28]. Additionally, DSIbin monitors registers because registers can temporarily be the only reference to allocated memory; otherwise, DSI would see false positives in its memory leak detection algorithm. The main synchronization point between Howard and DSIbin are the call stacks for heap types and the function addresses for stack types.

When devising our tool chain we slightly extended the implementation of Howard to perform type merging on heap objects. To see why, recall that Howard identifies types based on the call stack. This implies a many-to-one relationship between allocation sites and a type, if objects from that type are allocated at different sites. This is problematic for DSI's strand discovery, because each allocation site is then treated as a separate type by DSI, which in turn leads DSI to miss strands. Thus we modified Howard by adding type merging; it now tracks whether pointers and instructions operate on objects from different allocation sites. If this is the case and the objects are binary compatible, i.e., have the same size and the same layout of primitive data types, then they are merged.

B. Evaluation

We implemented a prototype of our tool chain: DSIbin consists of 3K LOC of C++ and interfaces with DSI and Howard. For our benchmarking, we ran it on a PC with an Intel CORE i7-4800MQ with 2.70GHz and 32GB RAM.

Benchmark. Our prototype has been applied to a benchmark of 30 C/C++ code samples that include a large variety of combinations of dynamic DSs: 4 examples from textbooks (`tb-1`,`tb-2` [36] and `tb-3`,`tb-4` [39]), 5 examples from Forester/Predator [8] (**lit**); 5 (extracted) real-world examples (**(e)r**), namely, the region clipping library of VNC (`hvnc2/libs/libvncsrv/rfbregion.c` found in Carberp [10], `r-3`), the benchmarks treeadd [7] (`r-2`), and binary-trees-debian [4] (`r-1`) and 16 self-written synthetic programs (**syn**). The source code of DSI/DSIbin and the synthetic examples are available from [9].

The benchmark covers many popular DS ingredients (e.g., lists, trees) plus arbitrary interconnections between them (e.g., nesting, overlay). Its examples each contain a single DS, which is convenient for evaluation purposes so as to isolate interesting aspects of our approach for discussion (Sec. V); however, our approach is designed to handle multiple DSs, too.

Our synthetic examples are designed to "stress test" our approach wrt. two axes: firstly they exercise interesting combinations of DSs such as skip lists with nested DLLs (`syn-9`), where we were interested to discover whether the nesting relationship interferes with DSI's skip list detection. Related

TABLE I
DETAILED BENCHMARKING RESULTS OF OUR NOVEL TOOL CHAIN

	Code (ground truth)							Naive Combination (Sec. III)				Sophisticated Combination (Sec. IV)								
expl	**lang**	**DS [LOC / trace length]**	**flat**	**n@h**	**n**	**m**	**h/s**	**DSIbin**	**rec**	**n-d**	**m**	**DSIbin**	**rec**	**n@h-d**	**n-d**	**m**	**nm**	**h/s**	**pr**	**ch hyp**
er-1	C++	**CDLL** [70 / 2009]	n	y	n	y	n	DLL	n	n	y	CDLL	y	y	n	-	y	n	n	DSIref
er-2	C++	**CDLL** [66 / 1474]	n	y	y (p)	y	n	DLL	n	y (p)	y	CDLL	y	y	n (p)	-	y	n	n	DSIref
lit-1	C	**CDLL→2xNo→CDLL,CDLL** [101 / 1326]	n	y	y	y	y	DLL→2xNi→DLL,DLL	n	y	y	CDLL→2xNo→CDLL,CDLL	y	y	y	-	y	y	n	DSIref
lit-2	C	**SLL(→Ni→SLL)x4** [143 / 5667]	n	n	n	n	n	SLL(→Ni→SLL)x4	y	n	n	SLLs with Ni/No	n	n	n	y	-	n	n	(DSIRef)
lit-3	C	**SLo** [179 / 1329]	n	n	n	y	n	SLo	y	n	y	SLo	y	n	n	-	-	n	n	Howard
lit-4	C	**SLL→2xNo→CDLL,CDLL** [90 / 275]	n	y	y	y	n	SLL→2xNi→DLL,DLL	n	y	y	SLL→2xNo→CDLL,CDLL	y	y	y	-	y	n	n	DSIref
lit-5	C	**2 DLLs parallel** [36 / 557]	n	n	n	y	n	4xDLL parallel	y	n	y	2xDLL parallel	n	y	y	-	y	n	n	Howard
r-1	C	**BT** [98 / 5435]	n	n	n	y	n	BT	y	n	y	BT	y	y	n	-	-	n	n	Howard
r-2	C	**BT** [356 / 1635]	n	n	n	y	n	BT	y	n	y	BT	y	n	n	-	-	n	n	Howard
r-3	C	**DLL(5)→Ni→DLL** [712 / 2174]	n	y	y	y	n	DLL (3, no nesting)	n	y	y	DLL (5, with hint on No)	n	y	y	-	y	n	y	(DSIRef)
syn-01	C	**BT** [50 / 1635]	n	n	n	y	n	BT	y	n	y	BT	y	n	n	-	-	n	n	Howard
syn-02	C++	**SLL→No→SLL** [83 / 2298]	n	n	y	y	n	SLL→Ni→SLL	n	y	y	SLL→No→SLL	y	y	y	-	y	n	n	DSIref
syn-03	C++	**SLL (3)** [48 / 161]	n	n	n	y	n	SLL (3)	y	n	y	SLL (3)	y	n	n	-	-	n	n	Howard
syn-04	C	**DLL (10) + I2o+** [111 / 3807]	y	n	y	y	n	DLL (3) + noise	n	n	y	DLL (10) + I2o+	y	n	y	-	y	n	n	DSIref
syn-05	C	**SLL (5)** [64 / 728]	y	n	y	y	n	SLL (2)	n	n	y	SLL (5)	y	n	y	-	-	n	n	DSIref
syn-06	C	**SLL (11)** [72 / 873]	y (pt)	y	y	n	n	nothing	n	y (pt)	n	SLL (11)	y	y	y	-	y	n	n	DSIref
syn-07	C	**CDLL** [214 / 1329]	n	n	y	o	y	DLL	n	y	-	CDLL	y	n	y	-	y	y	n	DSIref
syn-08	C	**DLL→No→DLL** [34 / 204]	y	n	y	y (pt)	n	DLL (only parent)	n	n	y (pt)	DLL→No→DLL	y	y	y	y	y	n	y	DSIref
syn-09	C	**SLo→Ni→DLL** [97 / 761]	n	n	n	y	n	SLo→Ni→DLL	y	n	y	SLo→Ni→DLL	y	n	n	-	-	n	n	Howard
syn-10	C	**SLL→I1o→SLL** [51 / 354]	n	n	n	y (pt)	n	SLL→I1i→SLL	n	n	y (pt)	SLL→No→SLL	n	n	n	y	-	n	n	(DSIRef)
syn-11	C	**SLL (10)** [32 / 585]	n	n	y	n	n	SLL (9)	n	y	n	SLL (10)	y	n	y	-	y	n	n	DSIref
syn-12	C	**SLL (12)** [49 / 191]	n	n	n	y (pt)	n	SLL (11)	n	n	y (pt)	SLL (12)	y	n	n	y	-	n	n	DSIref
syn-13	C	**SLL (6)** [53 / 72]	n	n	n	n	n	nothing	n	n	n	SLL (6)	y	n	n	y	-	n	n	DSIref
syn-14	C	**SLL→No→SLL** [47 / 352]	y	n	y	o	n	SLL→Ni→SLL	n	n	-	SLL→No→SLL	y	n	y	-	y	n	n	DSIref
syn-15	C	**SLL→No→SLL** [45 / 796]	n	n	n	y	n	SLL→No→SLL	y	n	y	SLL→No→SLL	y	n	n	-	-	n	n	Howard
syn-16	C	**SLL (5)** [29 / 329]	n	n	n	o	y	SLL (4)	n	n	-	SLL (5)	y	n	n	-	-	y	n	DSIref
tb-1	C	**SLL (21)** [130 / 925]	n	n	n	n	n	SLL (20)	n	n	n	SLL (21)	y	n	n	y	-	n	y	DSIref
tb-2	C	**SLL (11)** [132 / 367]	n	n	n	n	n	SLL (10)	n	n	n	SLL (11)	y	n	n	y	-	n	y	DSIref
tb-3	C	**DLL** [217 / 1174]	n	n	y (p)	y	n	DLL	y	n (p)	y	DLL	y	n	n (p)	-	-	n	n	Howard
tb-4	C	**SLL (11)** [105 / 869]	n	n	n	y	n	SLL (10)	n	n	n	SLL (10)	n	n	n	n	-	n	n	(Howard)

Symbol explanation: `flat` flattened member access, `n@h` nesting at head, `n` nesting, `m` type merge, `h/s` DS distributed between heap and stack, `rec` DS recognized, `n-d` nesting detected, `n-m` nested types merged, `pr` primitive types refined, `ch hyp` chosen hypothesis.

work in the shape analysis literature contains "plain" skip lists as benchmarking only (`lit-3`), and [15], [21], [22] do not handle skip lists at all. Secondly, our synthetic examples address measures of DS obfuscation, which a malware author could take to circumvent detection by our tool. Note that our interest here is not in *code* obfuscation [19], because our approach is resilient against this by design. Instead we aim at direct *DS* obfuscation [25], by exploring situations that prevent Howard's type merging and nested struct detection, e.g., by avoiding list traversals or performing flattened accesses of nested elements.

Because Howard uses instructions and pointers touching binary compatible objects to merge different allocation sites, we specifically introduce artificial pointers and functions in examples `syn-10`, `syn-12` and `syn-13` to circumvent the merge strategy. The Linux CDLL tests a DS distributed across the heap and stack and the linkage of nodes of different types in the context of a cyclic DLL, e.g., example `syn-07`. With examples `syn-05`, `syn-06` and `syn-16` we test these scenarios with a non cyclic SLL. Additionally, `syn-04`, `syn-05`, `syn-06`, and `syn-14` access their nested elements flattened, i.e., from the base address of the enclosing struct, to prevent their detection.

For compiling the examples we used the default optimization settings for `gcc` and `-O0` for `g++`. Howard is resilient against various compiler optimizations, such as data layout, function frame, and loop optimizations [34], and those are transparent to DSIbin as long as memory is properly allocated and the pointers forming the DS are preserved.

We report the trace length in Table I; it is important for our dynamic analysis as the evidence builds up during the lifetime of a DS. Currently, we do not have a convergence criteria for stopping the analysis as soon as DSI collected overwhelming evidence for a particular data structure interpretation; instead, we let the programs run until termination. Note that, lines of code (LOC) are not an expressive measure for the complexity of a DS, as challenging examples can be constructed with very few lines of code, e.g., `lit-5` of Table I only requires 36 LOC to form two DLLs running in parallel.

General discussion & first successes. In section "Code" of Table I we first give the *ground truth* of the benchmark as revealed by a detailed inspection of the available source code. Because DSIbin can, in contrast to DSI, also handle C++ examples, we distinguish between C and C++ code. The next column reports the shape of the inspected DS; the chosen examples range from skip lists, binary trees and (cyclic) DLLs, to interconnections of these via indirect nesting (Ni) and overlay nesting (No) from parent to child. Additionally, we list various characteristics such as flattened access of struct members (flat), nested struct at the head of the surrounding struct (n@h), nested struct not at the head (n), the possibility to merge allocation sites (m), and whether the DS spawns

across heap *and* stack (h/s); the latter can be seen, e.g., when storing the head of the Linux CDLL on the stack. Some examples only show payload (p) nested structs which does not affect the correct identification of DS shape or only provides partial (pt) merge opportunities. The next columns headed "Naive Combination" firstly describe the discovered DS when Howard's type information is used 'as is' (DSIbin), whether the example's DSs are recognized correctly (rec), whether a nested struct not at the head is detected (n-d), and whether allocation site merging (m) could be performed. Nesting at head, stack/heap merging and nested elements merging is never performed by Howard here and is thus not reported; this is discussed below. Whenever the difference between the naive and sophisticated approach lies in the length of recovered lists, the length is stated in brackets.

Our initial results are quite encouraging as already 10 out of the 30 examples are detectable by DSI when Howard's type information is used. Notably, these examples include situations that are not handled by related work, e.g., skip lists (`lit-3`, `syn-9`) and indirect/overlay nesting (`lit-2`, `syn-9`, `syn-15`). Specifically, one can consult the "Code" section of Table I to see that in any example where (i) Howard is able to fully merge the allocation sites and (ii) the cells forming the DS cover the complete memory chunk, the Howard+DSIbin combination produces the correct result. This emphasizes that Howard, when combined with DSI, performs well even outside its initial use case.

Listing 1. (Flattened) struct member access

```
struct outer {
        int payload;
        struct inner i_struct;
};
struct outer *o_p = malloc(sizeof(*o_p));

// Flattened access relative from outer
o_p->i_struct.a = 2;

// Access relative from inner
struct inner *i_p = &(o_p->i_struct);
i_p->a = 3;
```

Limitations. One obvious limitation that hampers DSIbin is the inability to detect a nested struct at the head of an enclosing struct, which frequently occurs in the real world such as in the VNC example (Fig. 4, left column, `sraSpan front`). This is because Howard tries to find access patterns that apply an offset from a base pointer; when the base pointer of the enclosing struct and the nested struct are the same, the nested access is effectively overridden by that of the enclosing one. The problem is worsened even for nested structs not appearing at the head, if the programmer accesses nested struct elements from the base address of the surrounding struct instead of relatively from the nested struct as seen in Listing 1.

Howard's type merging strategy works well on a struct as a whole, but is not performed between nested elements (Fig. 4, left column) and between nested and non-nested instances, e.g., structs `front` and `back`. This limitation is also coupled with missed nested structs, because Howard would not be able to merge what is not detectable. The consequences for DSI are missed strands as was discussed when we introduced Howard's merging strategy, but now at the level of nested elements. Additionally, the merge operates on the heap exclusively, thereby ignoring type instances found on the stack *and* the heap. Example `lit-1`, a parent CDLL with nesting-on-overlay of two CDLLs, demonstrates these problems. Because the head element of the parent CDLL is placed on the stack, the first consequence is that DSI does not detect the cyclicity property. While Howard indeed detects the two nested head elements of the child CDLLs inside a parent node, it is not able to merge those with the remainder of the list; this again leads to a missed cyclicity property for the child elements. Additionally, the nesting property changes from nesting-on-overlay to nesting-on-indirection.

We illustrate Howard's limitations for our means with two further examples. As can be seen in the middle column of Fig. 4, Howard neither recognizes the nested head element `front` in the VNC example `r-3`, nor does it merge the detected nested struct (corresponding to `back`) and non-nested struct (corresponding to `sraSpan`); all untyped regions are not detected due to the VNC code not accessing them. Further, example `syn-13` shows that type merging is a crucial feature of Howard. If this would be absent, as we simulated in this SLL example by leaving out any list traversal, then the SLL is not even detected partially. This is because each node of the SLL is allocated at a different allocation site, and thus DSI's strand creation fails since type equivalence between cells is never established. Such a situation can in principle occur in practice, e.g., in a skip list where certain regions are never traversed, thereby blurring the shape even in the presence of DS accesses.

Overall, however, the types inferred by Howard make for a good baseline on which DSI is already able to detect certain DSs. Nevertheless, the encountered limitations need to be tackled, as we will do in the following, in order to fully enable DSI's capabilities.

IV. Sophisticated Combination: Howard+DSIbin+DSIref

This section aims to solve the abovementioned problems of nested struct detection and type merging via a more sophisticated combination of Howard and DSIbin. This involves an additional component DSIref (Fig. 3), which implements a novel approach to refining types generated by Howard. Interestingly, this component uses DSI for assessing the quality of inferred type information.

A. Refinement Approach: DSIref

The idea of DSIref is to take Howard's (incompletely) inferred types and use them as a seed for improvement, for which we exploit pointer connections between types to systematically create further (and more complete) type hypotheses. Note that pointer connections reveal much about the layout of a DS; for example, an incoming pointer not at the head of a struct could indicate a nested struct, or pointer linked type objects

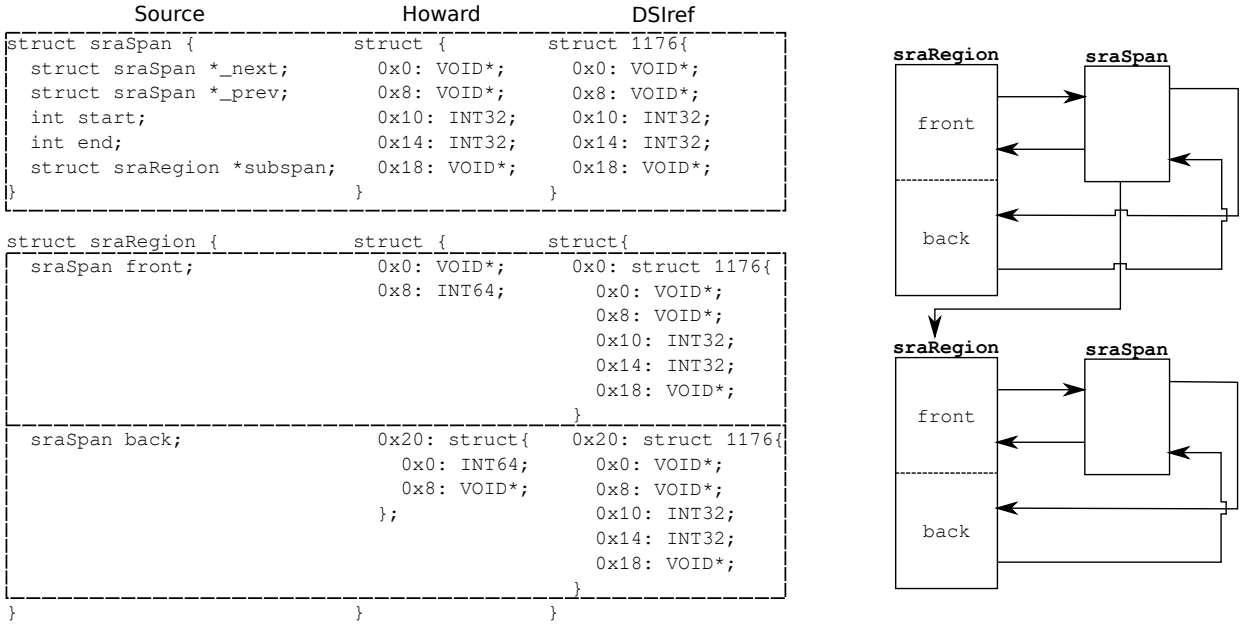

Fig. 4. Refinement result shown for VNC example (`r-3`) (left), VNC DS (right).

of different types could indicate a merge possibility if they are binary compatible, i.e., compatible in size and primitive data types. Our approach thus generates a set of hypotheses, where the best hypothesis needs to be determined. We do this by evaluating each hypothesis with DSI and selecting the hypothesis that results in the most complex DS as identified by DSI. The intuition is that correctly identified nested structs and correctly identified type merges will naturally increase the complexity of the detected DS shape, e.g., revealing the missed cyclic DLL property in example `lit-1`.

We refer to this refinement process as DSIref, which involves eight phases (Phases (a)–(h)) that are illustrated in Fig. 5. DSIref takes the execution trace generated by DSIbin (Phase (a)), which now contains all the 'as is' type information from Howard, and constructs a *merged type graph* where types are vertices and pointers are edges, similar to [11], [15], [31] (Phase (b)). The merged type graph handles heap and stack types transparently, allowing us to merge both, something which is not done in the literature [16].

Phases (c)–(f) utilize the merged type graph to generate new type hypotheses. The first of these phases maps subregions between different types by following pointer connections: two regions are mappable if they are binary compatible. Such mappings may or may not be unique as can be seen in Fig. 6. On the right-hand side, we have a nested linkage struct that is clearly delimited by its surrounding primitive data types, thereby yielding a unique mapping. On the left-hand side, we have pointer-only fields in the source and target of the pointer connection, where it is not clear how to cut the overall struct into nested structs. To solve this problem, we create a multitude of type hypotheses in Phase (d). These hypotheses cover all possible struct sizes and offsets mappable between target and source, as seen by the colored shadings on the left-hand side in Fig. 6. As an important implementation detail, we chose the mappable region to be at least two elements in size; this leaves out single-element pointer chains but decreases the chances of false mappings.

For each detected mappable region, Phase (e) then propagates that region maximally along the pointer connections between different types. This will discover more mappable memory regions and, accordingly, allow us to merge outer and nested structs even when distributed between the heap and stack (in principle, also in stack/stack combinations). More precisely, the propagation proceeds along the pointer offset relative to the starts of the mappable regions under consideration. The propagation stops as soon as there is no pointer found at this specific offset. In case there are memory regions along the path that are left untyped by Howard, these are considered as *don't cares*; they can be typed arbitrarily as long as size boundaries are not violated. This can be seen in Fig. 4, left-hand side, where the right column shows the missing type information revealed after Phase (e). Thus our approach is able to perform fine grained type refinements even across different types, as opposed to ARTISTE [15] which only refines types as a whole and only for allocation sites that it considers type equivalent.

While creating the various mappings, it is possible that inconsistant hypotheses are introduced, which are made consistent in Phase (f), e.g., by discarding overlapping memory regions; note that nested structs cannot overlap. Although this consistency property is local to a type vertex, our approach also considers the consistency of the inferred type hypotheses across the whole merged type graph. This can be seen on the left-hand side in Fig. 7, where two different interpretations are

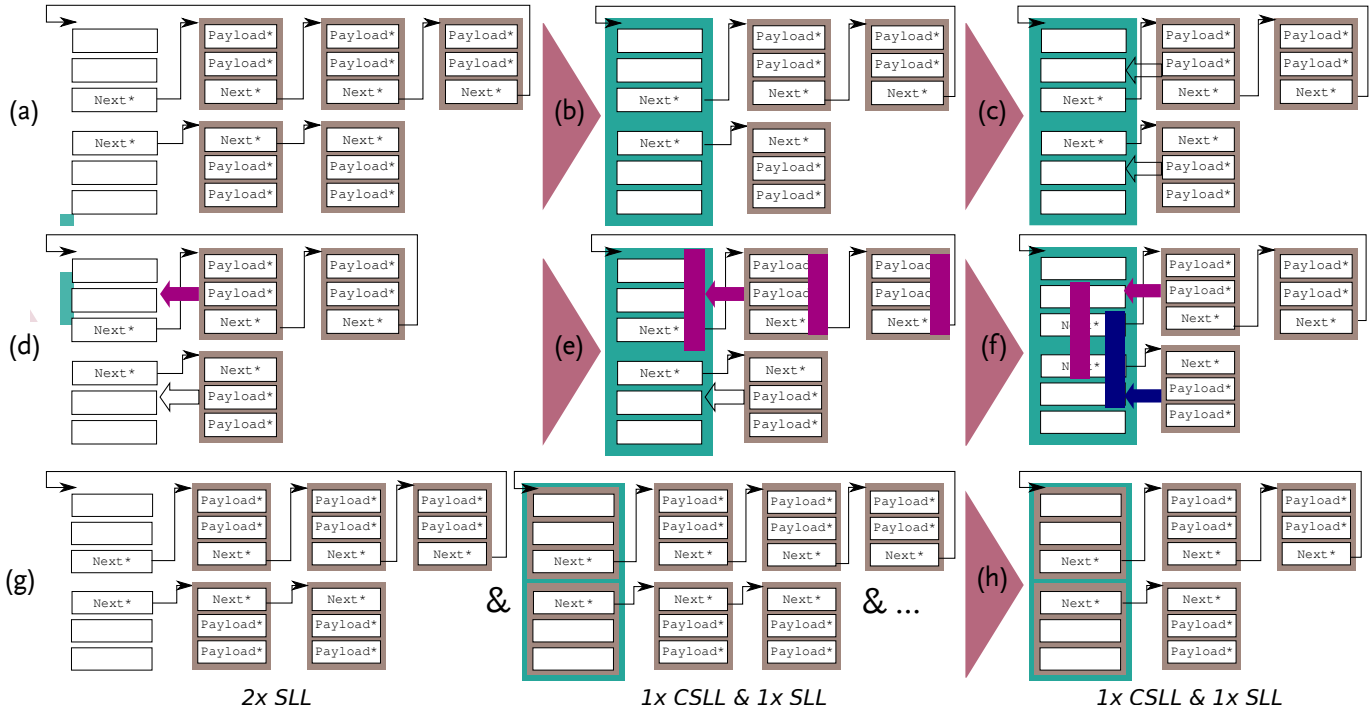

Fig. 5. Overview of the DSIref approach: (a) create sequence of points-to-graphs from program execution (only one shown); (b) construct merged type graph capturing pointer connections between types; (c) exploit pointer connections by mapping type subregions (two possibilities shown); (d) observe that 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 DS complexity (indicated by merged type graph with resulting label *1x CSLL & 1x SLL*).

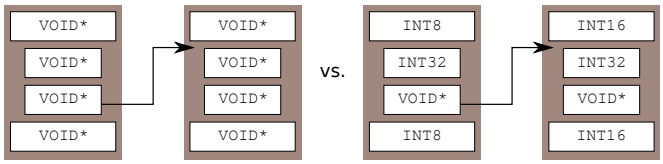

Fig. 6. Illustration of possible mappings.

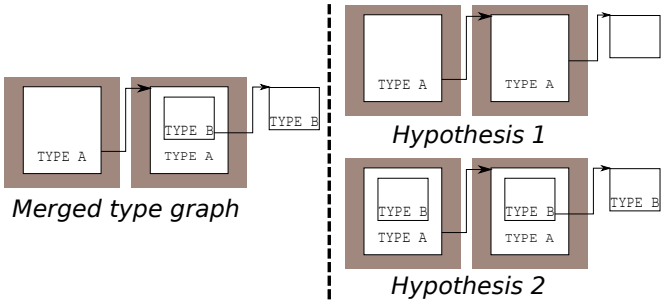

Fig. 7. Motivation for globally consistent types.

available for one enclosing type: in one vertex of Type A, an additional nested vertex of Type B is present, but not in the other. Such a situation can arise when certain parts of the DS behave differently than others, such as the head or tail of a list. This situation needs to be resolved by, again, producing a hypothesis for each different interpretation, leading to the result displayed on the right-hand side in Fig. 7. Howard's inferred type information is simply treated as an additional hypothesis by us (Sec. V).

With a set of hypotheses to hand, we must now determine which one(s) provide(s) the most suitable interpretation of the merged type graph. We use DSI to evaluate the DS that results from the types of a particular hypothesis in Phase (g), ultimately selecting the hypothesis that produces the most complex DS in Phase (h). This is done according to a hierarchy inspired by the DS identification taxonomy [38] used by DSI (*most complex to least complex*): skip list overlay (SLo), binary tree (BT), cyclic doubly-linked list (CDLL), doubly-linked list (DLL), cyclic singly-linked list (CSLL), nesting-on-overlay (No), nesting-on-indirection (Ni). To account for noise in our evidence labeling and counting, we consider hypotheses with evidence counts within 85% of each other as equivalent. To solve situations like example `lit-5`, where the best interpretation reveals multiple instances of the detected DS, our algorithm favors the hypothesis with the highest number of occurrences of the most complex DS. To solve situations in which the number of DS instances does not discriminate between the hypotheses, we apply *Occam's razor* to select the solution using the least amount of refined structs. In cases, where only a SLL is detected, the longest strand length is chosen. This strategy works well, often resulting in a single interpretation (Sec. V).

B. Implementation & Results

We have implemented and integrated the above type refinement approach DSIref into our tool chain (Fig. 3). Because our approach may generate many hypotheses, we have also parallelized DSI [38], specifically the parts of the DS detection: strand-graph creation, the naming, the folded-strand-graph creation, and the aggregation. While the creation of the points-to-graph and the subsequent strand calculation could be parallelized with a producer-consumer pattern, too, this is left to future work.

The tool chain involving DSIref has been applied by us to the above benchmark. The addition of DSIref boosts our correct identification of DSs from 10 of 30 examples with the naive Howard and DSI combination to 26 examples. We summarize our obtained results in Table I under the heading "Sophisticated Combination". Again, the detected DS is reported when DSIref is applied (DSI), whether the true DS is recognized (rec), whether nesting at head (n@h-d) or general nesting (n-d) is detected, whether additional merges when compared to Howard's potentially already merged data types are performed (m), whether merging of nested types (nm) or between heap *and* stack (h/s) occur, and whether an additional type refinement of the primitive data types (pr) is conducted. The final column (ch hyp) in Table I summarizes our results, by stating with what technique an example could be 'solved'; brackets indicate a wrong interpretation. If the example contains only one allocation site, i.e., no type merge is required, it is denoted with "o" in column (m). Interestingly, one example, `lit-5`, could only be solved using the naive but not our sophisticated approach.

Regarding our tool chain's performance, the longest running examples have about 5.5K events. Howard's type inference and the type hypotheses generation of DSIref took in the order of seconds (min: 1s, max: 13s, avg: 3.13s) and consume 565MB RAM on average (min: 0.2GB, max: 2.8GB). Creating an execution trace with Howard's type information or with a refinement hypothesis was also done in seconds (min: 0.7s, max: 17s, avg: 1.8s) and consumes 27MB RAM on average (min: 24MB, max: 33MB). The hypotheses' validation and interpretation with DSI took in the order of (tens of) minutes. On average, a hypothesis was evaluated in 63s (min: 1s, max: 2622s). The two longest running examples required about 50 minutes each. One example is bound by the large number (156) of hypotheses, despite each hypothesis being relatively quick to evaluate. The other instead contains only a few but complex hypotheses that are responsible for almost the entire runtime. On average 13.8 hypotheses were produced per example (min: 1, max: 156). The highest memory consumption for a hypothesis was 3.2GB RAM (min: 0.2GB, avg: 2.4GB). Note that our tool chain is currently a prototype only that runs in large parts on the JVM; we expect that performance can be improved significantly with a reasonable engineering effort.

V. Discussion

This section discusses several observations that we made when applying our DSIref tool chain to our benchmark.

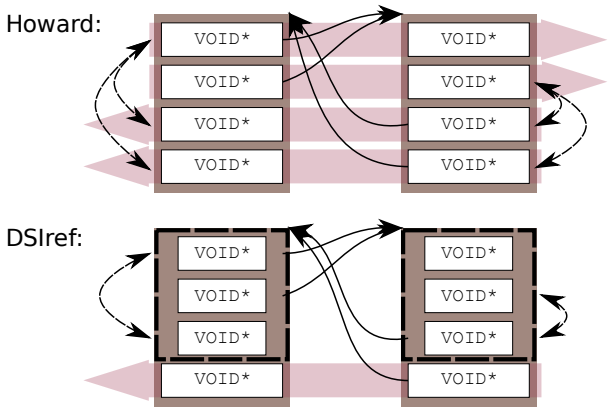

Fig. 8. Two DLLs in parallel.

Sophisticated combination generally outperforms the naive combination. When applying DSIref to our benchmark, we increase the detected DSs from 10 for the naive combination to 26 out of the 30 examples with the sophisticated combination. However, recall that we add a hypothesis from Howard's unmodified type interpretation to those generated by DSIref since, in rare circumstances the best interpretation of a DS can be missed, as is the case with our `lit-5` example. Using Howard's information only, DSI detects two DLLs running in parallel within one struct without nesting; or more precisely, it detects the four possible strand combinations forming the two DLLs (top of Fig. 8). In contrast, DSIref reports nested structs and, thus, prevents the detection of all four combinations (bottom of Fig. 8). This is due to the DLL predicate used by DSI, which enforces the same type for all cells. Hence, the sophisticated combination of Howard and DSI is not always better than the naive combination.

There may also be situations in which both the naive and the sophisticated combination miss the ground-truth interpretation. In example `tb-4`, which consists of an SLL, both combinations cannot merge the list head with the tail, which is allocated at a different allocation site. In addition, the head's string payload is set with function `strcpy` from `string.h`, whereas the string payload of the tail list nodes is a single, immediately assigned character. These different assignments of the payload lead Howard to infer different interpretations and prevents both Howard and DSIref from merging the two allocation sites; the latter is because the primitive data types from Howard are incompatible.

Amount of type merging influences precision. The `syn-08` example shows a flattened access of the nested head node of the child DLL inside the parent DLL, which prevents Howard from detecting the head of the child list. This leads to DSI recognizing an indirect nesting relation between parent and child, instead of an overlay nesting relation. Further, if the child DLL has two elements only, i.e., it consists of the missed nested head node in the parent and one additional element, then the DLL property would not be discovered by DSI. In contrast, DSIref is able to reveal the nested child head by

mapping the child DLL element back to the parent; this also refines the unaccessed previous pointer of the nested DLL head in the parent. For the same example, Howard does not merge the two allocation sites for the child DLL, due to the lack of an iteration pointer. In contrast, DSIref exploits the pointer connections in the merged type graph and fully propagates the type across the child DLL, thus merging all of its nodes. This results in the correct ground-truth interpretation.

However, precision is not always improved by merging types, which can be observed by the `lit-2` example that consists of multiple levels of SLLs that are all of different types and are each connected with nesting-on-indirection. Unfortunately, all types are binary compatible as they all consist of two pointers exclusively: one for connecting the elements of the current level, and one for the downward connection to the next lower level. This leads DSIref to merge type structures in `lit-2` that its programmer clearly wishes to distinguish, thereby misleading DSI by this additional but semantically incorrect information, instead of detecting linked structures with indirect nesting only, DSI recognizes some indirect nesting as overlay nesting. Although our approach also adds the hypothesis created from Howard's non-merged type interpretation, which yields the correct DS interpretation, our selection algorithm does not report this as it favors overlay nesting over indirect nesting.

Limitations inherited by DSI. Other limitations for DSIbin stem from DSI itself. Certain nesting scenarios can currently not be resolved by DSI, such as for `tsort` from coreutils [2] which performs a topological sort. It utilizes a binary tree with SLLs running through it, i.e., pointers are pointing from the tree to the SLLs and vice versa. Consequently, DSI detects parent-child nesting in both directions, i.e., from the tree into the list and reversely. This results in an ambiguous nesting situation because the parent cannot be uniquely determined. We envisage to tackle such situations in the future with certain heuristics, e.g., assuming that child elements are mainly accessed via their parent. Another limitation of DSI/DSIbin is pointer arithmetic that (temporarily) disconnects pointers from memory chunks, e.g., a XOR-list [13], [23] that saves memory by storing the previous and next pointers of a DLL node into one pointer field by XOR-ing them bitwise.

Insights gained into DSI. Our experiments also revealed shortcomings of the DSI core algorithm, which was presented in [38] and is employed by DSIbin as a black box. For example, the DS of the binary `r-3` – a DLL parent with DLL children and taken from Carberp's [10] malware – can in principle be detected by our approach, but DSI does not handle the structures strands underlying the DS in a proper way: nestings via indirection are misinterpreted as strands of length 2, which leads DSI to incorrectly fold parent and child DLL. To remedy this problem, DSI should be more patient until strands have had the opportunity to evolve and only then attempt an interpretation on whether a strand represents indirect nesting.

VI. Conclusions & Future Work

This paper aimed at identifying complex dynamic DSs from C/C++ binaries, so as to aid reverse engineering and program comprehension. We presented a sophisticated combination of the *Data Structure Investigator* tool DSI [38], which operates on source code and exploits type information, and the type excavating tool *Howard* [34]. We demonstrated via benchmarking that even a naive combination of DSI and Howard is better than related work [15], [21], [22]. We then significantly improved on this naive combination via novel ways for merging and refining types that are beyond the capabilities of Howard and similar tools [16]. Interestingly, this also involved the use of DSI's core algorithm to rank the plausibility of typing hypotheses, as the high-level semantic information about dynamic DSs can enhance the lower-level type information recovered by Howard. Our resulting sophisticated combination DSIbin of DSI and Howard was then able to detect the correct DS shapes of most of our benchmark examples. Hence, while there is information lost in binaries that may not be recoverable but may still be crucial for identifying dynamic DSs correctly, much can still be achieved in practice. Most importantly and as a rule of thumb, the more complex a dynamic DS is, the more likely it is that it will be correctly identified by DSIbin.

We have made the source code of DSI/DSIbin and our synthetic examples available online for inspection and use at [9]. Regarding future work, we plan to provide reverse engineers with easy access to DSIbin's results, e.g., by visualizing the results along the lines of [37] or integrate them with the industry-standard disassembler and debugger *IDA Pro* [3]. We also wish to investigate the handling of custom memory allocators that can be found in legacy code. Fortunately, DSI's cell and strand abstraction naturally supports custom memory allocators when source code is available, and approaches like [17] can help in detecting them in binaries. Additionally, our approach currently observes pointer-based queues and stack DSs as lists. To name them precisely would require a semantic analysis, e.g., via recognizing DS operations as proposed by dsOli [37]. This is ongoing research, as is the handling of arrays and hash tables. Finally, DSIbin provides a fine-grained, instruction-precise memory leak detection. Detecting leaks together with DSIbin's ability to replay the execution trace to a leak might be valuable to understand leaks in pre-compiled software, e.g., where hot-patching of the binary is required [30], [32].

VII. Acknowledgments

This work was supported by DFG grant LU 1748/4-1. We thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] A tool to detect bugs in Java and C/C++/Objective-C code before it ships. http://fbinfer.com/. Accessed: 15th August 2017.

[2] GNU core utilities v8.24: `tsort`. https://www.gnu.org/software/coreutils/coreutils.html. Accessed: 8th May 2017.

[3] Hex-Rays: The IDA Pro disassembler and debugger. https://www.hex-rays.com/products/ida/index.shtml. Accessed: 8th May 2017.

[4] Kevin Carson: The Computer Language Benchmarks Game: Binary Tree. https://benchmarksgame.alioth.debian.org/u64q/program.php?test=binarytrees&lang=gcc&id=1. Accessed: 8th May 2017.

[5] libusb 1.0.20. http://www.libusb.info/. Accessed: 8th May 2017.

[6] Linux kernel 4.1 Cyclic DLL (`include/linux/list.h`). http://www.kernel.org/. Accessed: 8th May 2017.

[7] Olden benchmark v1.01. http://www.martincarlisle.com/olden.html. Accessed: 8th May 2017.

[8] Predator/Forester GIT repository. https://github.com/kdudka/predator. Accessed: 8th May 2017. In particular, examples:
`lit-1`: tests/predator-regre/test-0102.c,
`lit-2`: tests/predator-regre/test-0234.c,
`lit-3`: tests/skip-list/jonathan-skip-list.c,
`lit-4`: tests/forester/cav13_tests/sll-listoftwoclists-linux.c,
`lit-5`: tests/forester/dll-duplicate-sels.c.

[9] Supplemental material for DSI. http://www.swt-bamberg.de/dsi. Accessed: 15th August 2017.

[10] theZoo aka Malware DB. https://github.com/ytisf/theZoo. Accessed: 8th May 2017.

[11] 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.

[12] G. Balakrishnan and T. W. Reps. DIVINE: DIscovering Variables IN Executables. In *VMCAI 2007*, volume 4349 of *LNCS*, pages 1–28. Springer, 2007.

[13] J. Berdine, C. Calcagno, and P. W. OHearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[14] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, and L. Wang. On the analysis of the Zeus botnet crimeware toolkit. In *PST 2010*, pages 31–38. IEEE, 2010.

[15] 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.

[16] J. Caballero and Z. Lin. Type Inference on executables. *ACM Computing Surveys*, 48(4):65:1–65, 2016.

[17] X. Chen, A. Slowinska, and H. Bos. Who allocated my memory? Detecting custom memory allocators in C binaries. In *WCRE 2013*, pages 22–31, 2013.

[18] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[19] C. Collberg and J. Nagra. *Surreptitious software: Obfuscation, watermarking, and tamperproofing for software protection*. Pearson, 2009.

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

[21] I. Haller, A. Slowinska, and H. Bos. Scalable data structure detection and classification for C/C++ binaries. *Empirical Software Engineering*, 21(3):778–810, 2016.

[22] 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.

[23] D. Knuth. The Art of Computer Programming 1: Fundamental Algorithms 2: Seminumerical Algorithms 3: Sorting and Searching. *MA: Addison-Wesley*, 1968.

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

[25] Z. Lin, R. D. Riley, and D. Xu. Polymorphing Software by randomizing data structure layout. In *DIMVA 2009*, volume 5587 of *LNCS*, pages 107–126. Springer, 2009.

[26] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *NDSS 2010*. The Internet Society, 2010.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI 2005*, pages 190–200. ACM, 2005.

[28] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. System V Application Binary Interface AMD64 Architecture Processor Supplement (Draft version 0.3). https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf, 2013. Accessed: 8th May 2017.

[29] 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.

[30] M. Payer and T. R. Gross. Hot-patching a web server: A case study of asap code repair. In *PST 2013*, pages 143–150. IEEE, 2013.

[31] E. Raman and D. I. August. Recursive data structure profiling. In *Memory System Performance*, MSP 2005, pages 5–14. ACM, 2005.

[32] A. Ramaswamy, S. Bratus, S. W. Smith, and M. E. Locasto. Katana: A hot patching framework for ELF executables. In *ARES 2010*, pages 507–512. IEEE, 2010.

[33] T. Rupprecht, X. Chen, D. H. White, J. T. Mühlberg, H. Bos, and G. Lüttgen. POSTER: Identifying dynamic data structures in malware. In *CCS 2016*, pages 1772–1774. ACM, 2016.

[34] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS 2011*. The Internet Society, 2011.

[35] D. Urbina, Y. Gu, J. Caballero, and Z. Lin. SigPath: A memory graph based approach for program data introspection and modification. In *ESORICS 2014*, volume 8713 of *LNCS*, pages 237–256. Springer, 2014.

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

[37] D. H. White. dsOli: Data structure operation location and identification. In *ICPC 2014*, pages 48–52. ACM, 2014.

[38] D. H. White, T. Rupprecht, and G. Lüttgen. DSI: An evidence-based approach to identify dynamic data structures in C programs. In *ISSTA 2016*, pages 259–269. ACM, 2016.

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