41

Schriften aus der Fakultät Wirtschaftsinformatik und
Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Data Structure Identification from Executions of Pointer Programs

Thomas Rupprecht

University
of Bamberg
Press

41 Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Contributions of the Faculty Information Systems and Applied Computer Sciences of the Otto-Friedrich-University Bamberg

Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Contributions of the Faculty Information Systems and Applied Computer Sciences of the Otto-Friedrich-University Bamberg

Band 41

University
of Bamberg
Press
2020

Data Structure Identification from Executions of Pointer Programs

Thomas Rupprecht

University
of Bamberg
Press
2020

Bibliographische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Informationen sind im Internet über http://dnb.d-nb.de/ abrufbar.

Diese Arbeit hat der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg unter dem Titel „Data Structure Identification from Executions of Pointer Programs“ als Dissertation vorgelegen.
1. Gutachter: Prof. Dr. Gerald Lüttgen, University of Bamberg
2. Gutachter: Prof. Dr. Thomas Noll, RWTH Aachen University
Tag der mündlichen Prüfung: 7. November 2019

Herstellung und Druck: docupoint, Magdeburg
Umschlaggestaltung: University of Bamberg Press

http://www.uni-bamberg.de/ubp/

ISSN: 1867-7401
ISBN: 978-3-86309-717-2 (Druckausgabe)
eISBN: 978-3-86309-718-9 (Online-Ausgabe)
URN: urn:nbn:de:bvb:473-irb-473250
DOI: http://dx.doi.org/10.20378/irb-47325

Dedicated to my parents Elli and Klaus, my wife Steffi and our son Henrik.

Acknowledgments

I want to thank Dr. David White and Prof. Gerald Lüttgen for making this dissertation thesis possible. Further I am very glad about the collaboration with Prof. Herbert Bos together with Dr. Xi Chen from the VU Amsterdam in the Netherlands and Dr. Tobias Mühlberg from the KU Leuven in Belgium.

However, this work would not have come into existence without the great support and patience throughout the years from my parents and my wife. Thank you!

I also want to thank all my colleagues from the Software Technologies Research Group for lots of interesting discussions and good times, e.g., the country side tour will (mostly) be unforgotten and your wedding gift still resides prominently in our living room. Special thanks to Jan Boockmann from our group, for being an invaluable help on so many occasions.

Additionally, I'm happy about my friends from my home town Hundelshausen and the neighbouring Altmannsdorf, who went with me through good and bad times since I was born. Further, I want to list some great human beings I have met during my life that are worth mentioning for reasons they know best themselves (in order of appearance in my life): Peter Hofmann, Daniel Finster, Christian Wiegand, Leslie Polzer and Tobias Zeck.

Finally, I want to give my appreciation to some outstanding individuals that made or still make music that motivates me: Kevin, Lemmy, James.

Abstract

The reverse engineering of binaries is a tedious and time consuming task, yet mandatory when the need arises to understand the behaviour of a program for which source code is unavailable. Instances of source code loss for old arcade games[1] and the steadily growing amount of malware[2] are prominent use cases requiring reverse engineering. One of the challenges when dealing with binaries is the loss of low level type information, i.e., primitive and compound types, which even state-of-the-art type recovery tools often cannot reconstruct with full accuracy. Further programmers most commonly use high level data structures, such as linked lists, in addition to primitive types. Therefore detection of dynamic data structure shapes is an important aspect of reverse engineering. Though the recognition of dynamic data structure shapes in the presence of tricky programming concepts such as pointer arithmetic and casts – which are both fundamental concepts to enable, e.g., the frequently used Linux kernel list[3] – also bring current shape detection tools to their limits.

A recent approach called Data Structure Investigator (DSI)[4], aims for the detection of dynamic pointer based data structures. While the approach is general in nature, a concrete realization for C programs requiring *source code* is envisioned as programming constructs such as type casts and pointer arithmetic will stress test the approach. Therefore, the first research question addressed in this dissertation is whether DSI can meet its goal in the presence of the sheer multitude of existing data structure implementations. The second research question is whether DSI can be opened up to reverse engineer *C/C++ binaries*, even in the presence of type information loss and the variety of C/C++ programming constructs.

Both questions are answered positively in this dissertation. The first is answered by realizing the DSI source code approach, which requires detailing fundamental aspects of DSI's theory to arrive at a working implementation, e.g., handling the consistency of DSI's memory abstraction and quantifying the interconnections found within a dynamic pointer based data structure, e.g., a parent child nesting scenario, to allow for its detection. DSI's utility is evaluated on an extensive

[1] http://kotaku.com/5028197/sega-cant-find-the-source-code-for-your-favorite-old-school-arcade-games

[2] https://www.av-test.org/en/statistics/malware/

[3] https://github.com/torvalds/linux/blob/master/include/linux/list.h

[4] SWT Research Group, University Bamberg, DFG-Project LU 1748/4-1

benchmark including real world examples (libusb[5], bash[6]) and shape analysis[7,8] examples. The second question is answered through the development of a DSI prototype for binaries (DSIbin). To compensate for the loss of perfect type information found in source code, DSIbin interfaces with the state-of-the-art type recovery tool Howard[9]. Notably, DSIbin improves upon type information recovered by Howard. This is accomplished through a much improved nested struct detection and type merging algorithm, both of which are fundamental aspects for the reverse engineering of binaries. The proposed approach is again evaluated by a diverse benchmark containing real world examples such as, the VNC clipping library, The Computer Language Benchmarks Game and the Olden Benchmark, as well as examples taken from the shape analysis literature.

In summary, this dissertation improves upon the state-of-the-art of shape detection and reverse engineering by (i) realizing and evaluating the DSI approach, which includes contributing to DSI's theory and results in the DSI prototype; (ii) opening up DSI for C/C++ binaries so as to extend DSI to reverse engineering, resulting in the DSIbin prototype; (iii) handling data structures with DSIbin not covered by some related work such as skip lists; (iv) refining the nesting detection and performing type merging for types excavated by Howard. Further, DSIbin's ultimate future use case of malware analysis is hardened by revealing the presence of dynamic data structures in multiple real world malware samples.

In summary, this dissertation advanced the dynamic analysis of data structure shapes with the aforementioned contributions to the DSI approach for source code and further by transferring this new technology to the analysis of binaries. The latter resulted in the additional insight that high level dynamic data structure information can help to infer low level type information.

[5] http://libusb.info/

[6] https://www.gnu.org/software/ bash/

[7] Predator: http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/

[8] Forester: http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/

[9] http://www.cs.vu.nl/ herbertb/papers/dde_ndss11-preprint.pdf

Zusammenfassung

Reverse Engineering von Binärcode ist eine schwierige und zeitaufwändige Tätigkeit, die jedoch unabdingbar ist, wenn das Programmverhalten verstanden werden muss, ohne dass Quelltext zur Verfügung steht. Fälle von Quelltextverlust für alte Computerspiele[10] und die stetig wachsende Anzahl von Schadsoftware[11] sind daher prominente Anwendungsfälle für Reverse Engineering. Eine der Herausforderungen bei der Analyse von Binärcode ist der Verlust von Typinformationen, wie zum Beispiel primitiven und komplexen Datentypen. Oftmals können diese Typinformationen von den aktuellen Werkzeugen zur Typrückgewinnung, die den Stand der Technik repräsentieren, nicht vollumfänglich und korrekt rekonstruiert werden. Weiterhin verwenden Programme zusätzlich zu den primitiven und komplexen Datentypen meist höhere dynamische Datenstrukturen, wie zum Beispiel verkettete Listen. Daher ist die Erkennung der Form von dynamischen Datenstrukturen ein wichtiger Aspekt des Reverse Engineerings. Wobei die Erkennung der Formen dynamischer Datenstrukturen im Kontext von schwierigen Programmierkonzepten, wie Zeigerarithmetik und Typumwandlungen – beides fundamentale Konzepte um zum Beispiel die häufig verwendete Linux Kernel Liste[12] zu implementieren – aktuelle Werkzeuge zur Formenerkennung von dynamischen Datenstrukturen an ihre Grenzen bringen.

Ein aktueller Ansatz (DSI[13]) zielt auf die Erkennung von dynamischen zeigerbasierten Datenstrukturen ab. Der Ansatz ist generell gehalten, wobei eine konkrete Umsetzung für C-Programme unter Verwendung von Quelltext durchgeführt wird, da Typumwandlungen und Zeigerarithmetik als Bestandteil des C-Sprachumfangs einen Stresstest für den Ansatz darstellen. Daher ist die erste Forschungsfrage innerhalb dieser Dissertation, ob DSI seinen eigenen Anforderungen auch unter der schieren Vielzahl an existierenden Datenstrukturimplementierungen gerecht wird. Die zweite Forschungsfrage behandelt, ob Reverse Engineering von C/C++ Binärcode mit DSI erschlossen werden kann, trotz des Verlusts von Typinformationen und der Vielzahl von C/C++ Programmierkonstrukten.

Beide Forschungsfragen werden positiv innerhalb dieser Dissertation beantwortet. Die erste Frage wird durch eine Umsetzung des DSI Quelltext-Ansatzes er-

[10] http://kotaku.com/5028197/sega-cant-find-the-source-code-for-your-favorite-old-school-arcade-games

[11] https://www.av-test.org/en/statistics/malware/

[12] https://github.com/torvalds/linux/blob/master/include/linux/list.h

[13] SWT Research Group, University Bamberg, DFG-Project LU 1748/4-1

forscht. Dies umfasst die Ausdetaillierung von fundamentalen Aspekten der DSI Theorie um eine funktionsfähige Implementierung zu erreichen, zum Beispiel die Erhaltung der Konsistenz der Speicherabstraktion von DSI und die Quantifizierung von Verbindungen innerhalb einer zeigerbasierten dynamischen Datenstruktur, wie zum Beispiel einer Eltern-Kind-Beziehung, um eine Erkennung solcher Verbindungen zu ermöglichen. Die Nützlichkeit von DSI wird an Hand eines umfassenden Testsets untersucht, das unter anderem Praxisbeispiele (libusb[14], bash[15]) und Beispiele der Forschungsrichtung der Shape Analysis[16,17] beinhaltet.

Die zweite Forschungsfrage wird durch die Entwicklung eines DSI Prototypen für Binärcode (DSIbin) beantwortet. Um den Verlust von perfekten Typinformationen, die bei der Verwendung von Quelltext verfügbar sind, zu kompensieren, wird DSIbin mit Howard[18] kombiniert, einem Werkzeug zur Typrückgewinnung, das den aktuellen Stand der Technik in diesem Bereich repräsentiert. Insbesondere verbessert DSIbin zusätzlich die von Howard zur Verfügung gestellten Typinformationen durch eine immens verbesserte Erkennung eingebetteter Strukturen sowie der Typzusammenführung. Beide Problemstellungen sind grundlegende Aspekte für das Reverse Engineering von Binärcode.

Der vorgeschlagene Ansatz wird ebenso durch ein manigfaltiges Testset untersucht, das unter anderem Praxisbeispiele umfasst, wie die “VNC clipping library”, “The Computer Language Benchmarks Game”, den “Olden Benchmark” und Beispiele aus der Shape Analysis Literatur.

Diese Dissertation verbessert den aktuellen Stand der Technik für die Erkennung von Datenstrukturen und des Reverse Engineering durch (i) die Umsetzung und Evaluation des DSI Ansatzes, was Beiträge zur Theorie von DSI beinhaltet und in einem DSI Prototypen resultiert; (ii) die Öffnung von DSI zur Analyse von C/C++ Binärcode um DSI auf das Reverse Engineering zu erweitern, was ebenfalls in einem Prototypen für DSIbin resultiert; (iii) die Behandlung von Datenstrukturen mit DSIbin, die bisher von einiger verwandter Literatur nicht abgedeckt wurden, wie zum Beispiel Skip-Listen; (iv) eine Verfeinerung der Erkennung von eingebetteten Strukturen und die Zusammenführung von Typinformationen die von Howard ermittelt wurden. Weiterhin wird der Anwendungsfall der Analyse von Malware für DSIbin im Bereich des Future Work gestärkt, indem die Verwendung von dynamischen Datenstrukturen in verschiedenen realen Malware Stichproben nachgwiesen wird.

Zusammenfassend erweitert diese Dissertation die dynamische Analyse von dynamischen Datenstrukturen gemäß den zuvor aufgeführten Beiträgen zu dem DSI Ansatz für Quelltext sowie durch den Transfer dieser neuen Technologie auf

[14] http://libusb.info/

[15] https://www.gnu.org/software/ bash/

[16] Predator: http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/

[17] Forester: http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/

[18] http://www.cs.vu.nl/ herbertb/papers/dde_ndss11-preprint.pdf

die Analyse von Binärcode. Letzteres führte zu der zusätzlichen Erkenntnis, dass Informationen von höheren dynamischen Datenstrukturen helfen können primitive Typinformationen abzuleiten.

Contents

List of Tables

List of Figures

List of Algorithms

Acronyms

ABI Application Binary Interface.

AFE Artificial Free Event.

AME Artificial Memory Event.

ASG Aggregated Strand Graph.

AUE Artificial Undef Event.

BT Binary Tree.

CDLL Cyclic Doubly Linked List.

CIL C Intermediate Language.

CMA Custom Memory Allocator.

coreutils GNU Core Utilities.

CPU Central Processing Unit.

CSLL Cyclic Singly Linked List.

DDoS Distributed Denial of Service.

DDS Dynamic Data Structure.

DDS Dynamic Data Structure.

DDT Data-structure Detection Tool.

DFG Deutsche Forschungsgemeinschaft.

DgS Degenerate Shape.

DLL Doubly Linked List.

DSI Data Structure Investigator.

DSIbin Data Structure Investigator for Binaries.

DSIcore Data Structure Investigator Core Algorithm.

DSIref Data Structure Investigator Refinement Component.

DSIsrc Data Structure Investigator for Source Code.

DSItype DSI Type.

dsOli Data Structure Operation Location and Identification.

EP Entry Pointer.

FSG Folded Strand Graph.

FV Formal Verification.

HT Hyper Threading.

JVM Java Virtual Machine.

LKL Linux Kernel List.

LOC Lines of Code.

LT Logical Type.

ME Memory Event.

MER Maximum Enclosing memory sub-Region.

MTG Merged Type Graph.

PC Program Comprehension.

PTG Points-to Graph.

RAM Random Access Memory.

RE Reverse Engineering.

SC Strand Connections.

SG Strand Graph.

SIG Signature Generation.

SL Skip List.

SLL Singly Linked List.

StS Stable Shape.

TT True Type.

VIS Visualization.

VLS Variable Leaving Scope.

X.Org Open source X Window System implementation.

XSD XML Schema Definition.

1 Introduction

This dissertation is primarily concerned with the reverse engineering of pointer based Dynamic Data Structures (DDSs) in *C/C++ binaries*, such as doubly linked lists or trees. The pressing needs to analyse programs for which source code is unavailable are manifold. A company might lose its data – as has happened, e.g., for SEGA's "Magic Knight Rayearth" [1, 37] – but does not want to completely lose the engineering effort put into a software. A customer might be forced to maintain a closed source program after it has reached its end-of-life and need to develop, e.g., security patches [99]. In addition, the constantly growing amount of malware [17] requires security specialists to gain an understanding of malware behaviour.

As DDSs are a fundamental aspect of many programs, it is important to have information about the therein employed DDSs when conducting reverse engineering. When knowing the DDS shape one already has an idea of the corresponding data structure manipulating code sections – e.g., an insert into a Doubly Linked List (DLL) requires to set at least the previous and next pointers instead of only the next pointer for an insert into a Singly Linked List (SLL) – and possibly even about the overall algorithm – e.g., consider a code section performing a search that uses a Binary Tree (BT) versus a Singly Linked List (SLL). The detection of such DDSs is pursued by the recent DSI[1] approach.

Related work. While there already exist Dynamic Data Structure (DDS) identification tools for binaries, with DDT [74], ARTISTE [49] and MemPick [69] being state-of-the-art examples, they have limitations such as regarding recognition precision or due to strong assumptions. For example, DDT relies on the presence of well-defined interface functions, which is suitable when, e.g., the C++ STL library [91] is used in the programs under analysis. But this assumption is not satisfied for customized DDSs or in the presence of low-level optimizations such as inlining or macro based interfaces. ARTISTE looses its precision in the face of degenerate shapes, which occur during DDS manipulation operations. MemPick cuts connections between data structure nodes, which for now correspond to the allocated memory for one element of the DDS, of different types. This prevents MemPick to handle DDSs running through nodes of different types, for which the Linux Kernel List (LKL) [15] is a prominent example. DDT and ARTISTE also do not handle the Linux Kernel List (LKL), and all three tools struggle with complex parent child relations such as arbitrary nesting levels, i.e., where the children of a

[1] SWT Research Group, University Bamberg, DFG-Project LU 1748/4-1

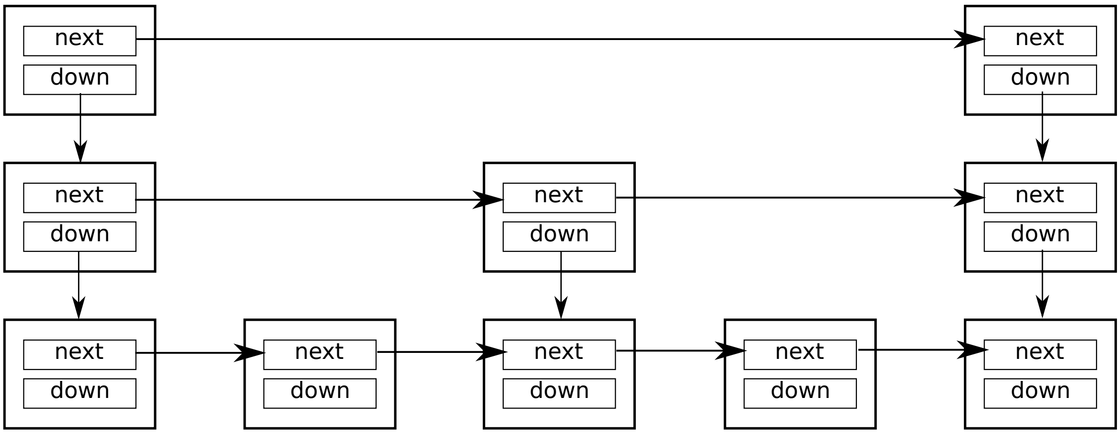


Figure 1.1: Three level skip list with one node per level and all nodes are of the same type.

parent child relation can have children of their own and which is not limited to a certain depth. Additionally, tools based on shape analysis [81] exists, which is concerned with statically inferring and verifying data structure shapes and their invariants. Therefore recent shape analysis tools such as Predator [65] and Forester [72] are of interest to us, though their main difference to DSI is their static analysis approach that is more conservative than a dynamic analysis [74, 81]. Forester handles (cyclic) SLLs and Doubly Linked Lists (DLLs), the LKL, trees and skip lists in sequential non-recursive C programs, whereas DSI also works for recursive C programs. Predator's focus lies on DLLs, especially the LKL, and SLLs. Other recursive data structures like trees, which are supported by DSI, are currently out of scope for Predator.

DSI goals & approach. The limitations of related work are addressed with the novel DSI approach [107], authored by Dr. White in the context of the Deutsche Forschungsgemeinschaft (DFG) project "Learning Data Structure Behaviour from Executions of Pointer Programs" (*LU 1748/4-1*). DSI requires C source code to enable its dynamic analysis, where a concrete execution of a program is precisely analysed; this is in contrast to a static analysis which conservatively reasons about all possible executions of a program [66]. In particular, DSI increases the analysis precision for and the scope of detectable DDSs, such as Skip Lists (SLs) and arbitrary parent child nestings. An SL provides fast search capabilities by providing multiple hierarchical segmented levels on top of a sorted list as shown in Fig. 1.1. The level segments store information about the underlying list elements reachable from a segment, which is used to either skip to the next segment if the searched token cannot be found in the current segment or move down the next lower level to repeat searching segments until the searched token is either found in the bottom list or is not contained at all. Additionally, DSI makes few assumptions, e.g.,

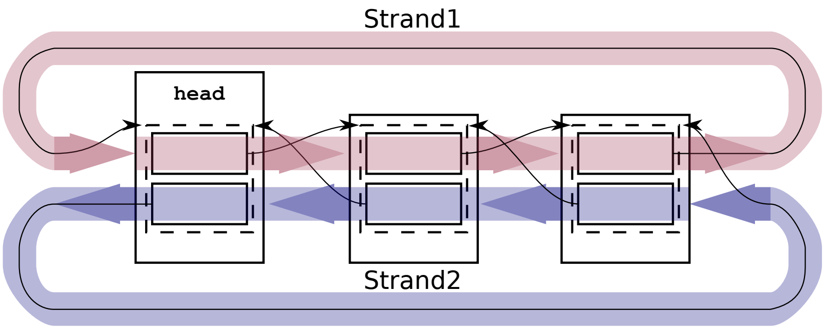


Figure 1.2: DSI's memory abstraction shown for the Linux kernel list. DSI detects two connected (cyclic) singly linked list (atomic building blocks of a DS, shown with colored arrows) that form the cyclic doubly linked list by being connected in reverse order. Additionally, the singly linked lists run through nodes of different types, illustrating that DSI allows that a DS node only covers a subregion (dashed boxes) of the same type of the allocated memory chunk (outermost solid boxes). Figure is reproduced from our publication [94].

it does not require the presence of interface functions or that a memory chunk, i.e., memory allocated on the stack or heap, corresponds to a DDS node as a whole. These properties of DSI have their foundations in the rich type information found in source code and make DSI highly interesting for automated DDS discovery.

DSI functions by first instrumenting the source code with the C Intermediate Language (CIL) framework [88] to capture memory changing events, such as memory (de-)allocations and pointer writes. The instrumented program is then executed and an event trace recorded. Finally the event trace is analysed offline by DSI resulting in a named DDS. DSI sets itself apart from related work by the following novelties:

Novel memory abstraction. DSI uses a novel memory abstraction to represent the heap/stack state, i.e., the allocated memory and the pointers connecting them, which allows us to handle data structures such as the LKL, SLs, and arbitrary nesting scenarios. Our memory abstraction is guided by the observation that a DDS is composed of various SLLs as their atomic building blocks, termed *strands*, and their interconnections, termed *Strand Connections (SC)*, as can be seen in Fig. 1.2. The LKL shown therein is composed of two strands, one for each direction, and the two strands are connected in a reverse order. Additionally, the nodes of an SLL are allowed to cover smaller subregions of memory, termed *cells*, which can either be a complete struct or only a nested struct. This seamlessly covers DDSs in the style of the LKL.

Degenerate shapes. DSI takes Degenerate Shapes (DgSs) into account instead of avoiding them [69, 74, 106]. DgSs occur due to DDS manipulation operations, where the properties of the true Stable Shape (StS) of a DDS are broken, as shown

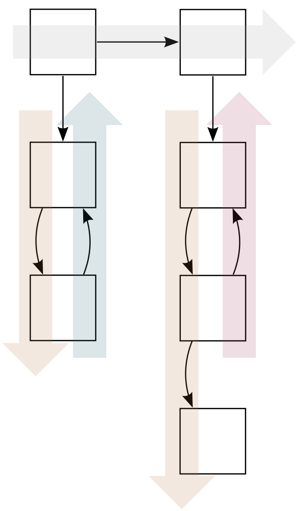


Figure 1.3: A singly linked list with two doubly linked list children. The singly linked lists forming the atomic building blocks of the DS are shown with colored arrows. The right child doubly linked list is in a degenerate shape due to a pending insertion of the last element.

in Fig. 1.3 where one child DLL is in a DgS due to an unfinished insert operation. DSI does so by gathering evidence for the true StS for a DDS during its analysis. **Data structures and their naming.** DSI creates a named output of the DDS pointed to by a given entry pointer of the program under analysis, e.g., "Binary tree with nested singly linked list children". DSI currently supports the following data structures: (cyclic) singly linked lists and doubly linked lists, skip lists, binary trees and arbitrary combinations of those DDSs in the form of parent child nesting. Skip lists and the nesting are not covered by [49, 69, 74].

1.1 Source code: Research question and challenges

This dissertation first contributes to improving upon the state-of-the-art of [49, 69, 74], by realizing the DSI approach on source code. This results in a research tool for automatically detecting and naming DDSs and allows us to answer the first research question:

> **Research Question 1:** *Is the DSI approach adequate to reach its goals of automatically detecting dynamic data structure shapes with high precision in the presence of degenerate shapes and, in particular, how far must the DSI concept be refined in order to deal with the wealth of dynamic data structure implementations employed in real-world software?*

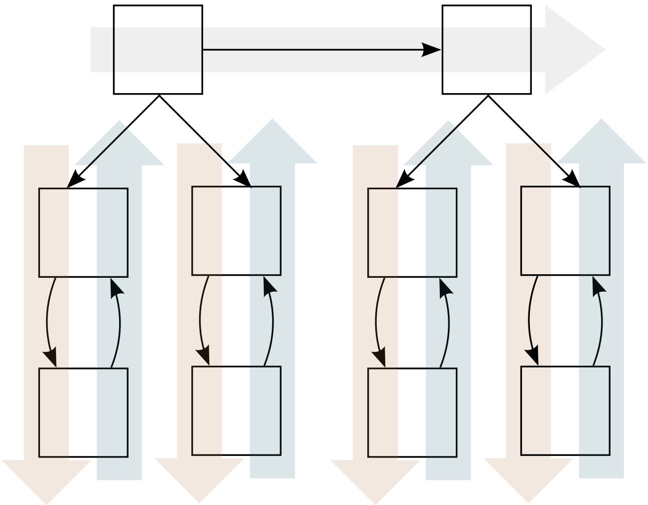


Figure 1.4: A parent singly linked list with two nested doubly linked list children.

The first part of the question will be answered positively by us by carrying out a diverse benchmark including real world examples (libusb [13], bash [10]) and examples taken from the shape analysis literature [65, 72] on the realized DSI approach. Along the way, various problems are resolved within this dissertation, such as quantifying the interconnections of the atomic building blocks of DSI's memory abstraction forming a DDS and guaranteeing the consistency of the chosen abstractions. DSI uses graphs for representing its various employed memory abstractions; it is vital to keep those graphs consistent during the analysis. The foundation for DSI is the current heap state, captured by a Points-to Graph (PTG), where vertices are memory chunks and edges are pointers. Additional abstractions, i.e., as strands, the Strand Graph (SG), the Folded Strand Graph (FSG), and the Aggregated Strand Graph (ASG), are layered upon the Points-to Graph (PTG) and are further discussed in Ch. 2. Therefore a PTG needs to be kept consistent during memory manipulations and programming errors like memory leaks, leading to the first challenge:

> **Challenge 1.1:** *Can DSI's graph abstractions be kept consistent in the face of common memory events such as memory (de-)allocations, pointer writes and even programming errors?*

DSI builds its novel heap abstraction on top of PTGs and deals with programming constructs offered by the C programming language, such as pointer arithmetic, type casts, and self controlled memory management including custom memory allocators [45]. This allows DSI to detect strands as the atomic building blocks of a DDS. However, DSI still needs to correlate the detected strands by finding the SCs between the strands in order to identify the various supported DDSs. When looking at Fig. 1.3, the final interpretation of the DDS as a "SLL with nested DLL children" can only be achieved when knowing that the parent SLL and the child DLLs are connected. More specifically, it becomes evident that the SCs are different for the strands forming the parent child relation, which are pointer based, and the strands of the child DLLs, where both strands reside inside a common memory chunk. Therefore, the descriptions of SCs need to account for their different nature; they also need to handle situations as shown in Fig. 1.4, where there are multiple parent child pointer connections. Each child connection needs to be unique within one node in order to prevent accidental confusion among the children when DSI performs its analysis. Yet the SCs need to be general enough to track the children performing the same role between multiple parent nodes, e.g., find the SC pointing to the first DLL in both nodes of the parent SLL. This leads to the second challenge:

> **Challenge 1.2:** *Can DSI's strand connections be quantified such that connections performing the same role can be identified robustly even accross today's multitude of dynamic data structure implementations?*

In summary, part one of this dissertation resulted in a DSI prototype working on C source code. This requires the realization of the DSI approach, which includes addressing Challenges 1.1-2 and the evaluation of the prototypic implementation. DSI is benchmarked with hand-written, text-book [104, 108], shape-literature [65, 72] and real world examples [10, 13, 38]. The benchmarking of DSI shows that the general approach including our extensions such as the quantification of SCs and the consistency of the PTGs works well on the rich variety of tested data structures. DSI outperforms related work in terms of detectable data structures, such as skip lists and arbitrary combinations of data structures such as those using parent child nestings. Additionally, benchmarking shows that DSI's fine grained memory abstraction enables the seamless handling of data structures running through nodes of different types. Such behaviour is not handled in a general manner by related work, and the Linux kernel list is a prominent example for such a data structure in frequent use today. Some of the benchmarked examples reveal opportunities for future work, e.g. adding arrays into the detectable units for DDS (cf. X.Org [38]) and the handling of even more complex nesting scenarios than currently covered (cf. tsort from coreutils [9]).

While working with the DSI prototype, the algorithm is also inspected for parallelization opportunities, and the resulting parallelized DSI version is also detailed in this dissertation. Our work on DSI has been presented at the *ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA) in 2016 [107].

1.2 Binary code: Research question and challenges

The second part of this dissertation is concerned with opening up the source code DSI version, called *DSIsrc* from now on, to analysing C/C++ binaries, where the initial focus of DSI on program comprehension shifts to reverse engineering [56]. This main part of the dissertation has primarily been executed by the author of this dissertation to answer the second research question:

> **Research Question 2:** *Can DSI's concepts and strengths be preserved when inspecting binaries?*

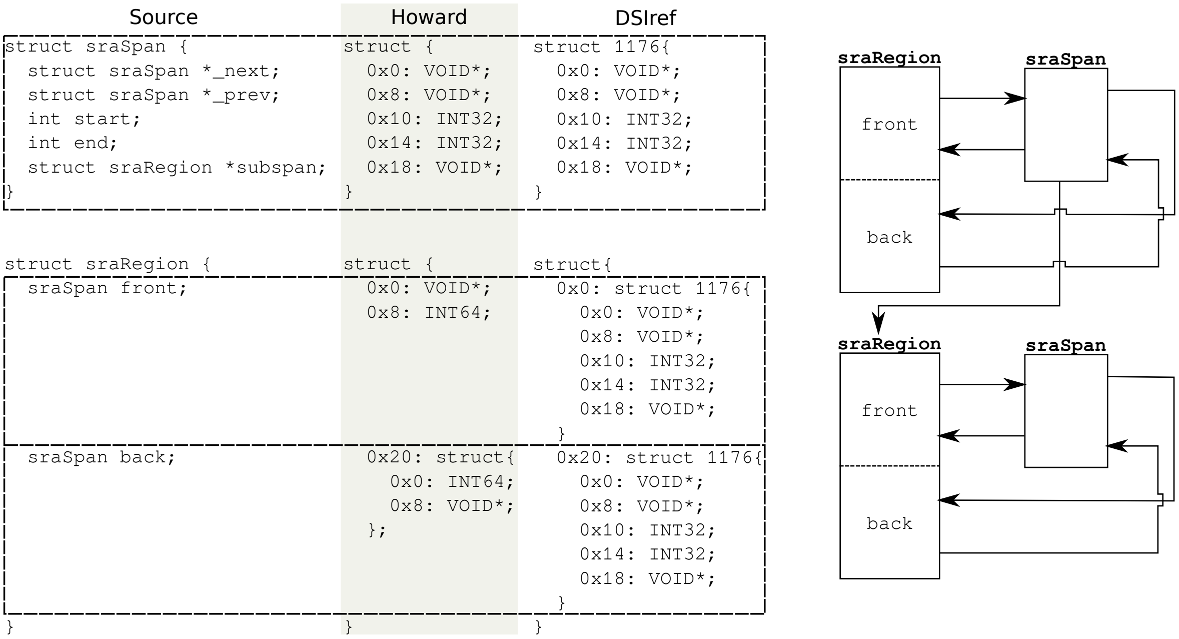
This research question is again answered positively by implementing a DSI version operating on binary code as input. One of the problems when dealing with binaries is the loss of perfect type information found in source code. Therefore, an important task is to analyse DSI's source code dependencies and investigate the implications when type information is lost. Because there exist type recovery tools for binaries that might alleviate the problem [43,78,80], one of the first challenges of RQ2 arises:

> **Challenge 2.1:** *Can external type recovery tools for binaries excavate sufficiently precise type information for DSI to function on binaries?*

A literature survey of type recovery tools leads us to Howard [98]; this state-of-the-art tool is suited for DSI as it handles the detection of nested types, which are important for DSI's memory abstraction. To build a prototypical binary DSI version, as done by us and called *DSIbin*, the recovered information from Howard needs to be included in the event trace recorded by DSI, which leads to the second challenge of capturing the event trace from binaries:

> **Challenge 2.2:** *Can the event trace required by DSI and generated using CIL, be reproduced with a binary instrumentation framework?*

We show that by using Intel's Pin [82] framework, it is possible for us to implement the Data Structure Investigator for Binaries (DSIbin) tool and to incorporate the types recovered by Howard. DSIbin is shown to already outperform related work MemPick [69], DDT [74] and ARTISTE [49] regarding the detection of data structure features such as indirect nesting, and data structures not recognised by the aforementioned tools, such as skip lists. This proves that Howard and DSI work well together; however, DSI's full capacity regarding the fine grained cell-based memory abstraction cannot be unleashed, due to Howard's limitations in detecting nested types and type merging. When considering the LKL shown in Fig. 1.2 it becomes apparent that without this information the DSI algorithm fails



```
Source                             Howard              DSIref
struct sraSpan {                   struct {            struct 1176{
  struct sraSpan *_next;             0x0: VOID*;         0x0: VOID*;
  struct sraSpan *_prev;             0x8: VOID*;         0x8: VOID*;
  int start;                         0x10: INT32;        0x10: INT32;
  int end;                           0x14: INT32;        0x14: INT32;
  struct sraRegion *subspan;         0x18: VOID*;        0x18: VOID*;
}                                  }                   }

struct sraRegion {                 struct {            struct{
  sraSpan front;                     0x0: VOID*;         0x0: struct 1176{
                                     0x8: INT64;           0x0: VOID*;
                                                           0x8: VOID*;
                                                           0x10: INT32;
                                                           0x14: INT32;
                                                           0x18: VOID*;
                                                         }
  sraSpan back;                      0x20: struct{       0x20: struct 1176{
                                       0x0: INT64;         0x0: VOID*;
                                       0x8: VOID*;         0x8: VOID*;
                                     };                    0x10: INT32;
                                                           0x14: INT32;
                                                           0x18: VOID*;
                                                         }
}                                  }                   }
```

Figure 1.5: Types for the VNC clipping library as found in the source code, as recovered by Howard and as refined by DSIbin (left), and the corresponding VNC data structure (right).

at the strand recognition which often results in diminished precision of the analysis. In the concrete example the head node would not be considered as part of the remainder of the list by DSI. So missing nested types and not performing proper type merging might lead to false negatives and positives in the strand creation phase of DSI. However both nesting detection and type merging are fundamental problems when dealing with binary code and a must to enable DSI's rich DDS detection capabilities. This observation leads to the third challenge for RQ2:

> **Challenge 2.3:** *Can the type information recovered by Howard from binaries be refined with the help of DSI itself?*

To address this challenge, a type refinement step is devised into DSI, which improves upon Howard's excavated type information by advanced techniques of type merging, nesting detection and, as a byproduct, primitive type detection. To illustrate the type refinement, the types of the VNC clipping library [14] are shown on the left in Fig. 1.5, there the ground truth for the types is given by the library's source code. The types excavated by Howard are depicted next to the source code, where one can see the different limitations: (i) the nested `sraSpan front` is not detected by Howard; (ii) the nested `sraSpan back` is detected, but the type is not merged with the standalone `struct sraSpan`; (iii) the primitive types of `struct sraRegion` that are not accessed cannot be typed. On the right in Fig. 1.5, the actual shape of the data structure is depicted, which is a "Parent DLL with nested child DLLs".

The main idea behind DSIbin's type refinement is to exploit pointer connections between the allocated memory chunks, with the assumption that incoming pointers to memory always point to the start of a nested struct, i.e., incoming pointers at the middle of a memory chunk hint at a nested struct. Additionally, type information can be propagated along pointer chains as long as both the primitive types along the path and the size of the memory chunks are matched. Untyped memory is treated as a don't care, which matches all primitive types.

Exploiting the resulting pointer information can lead to multiple possible interpretations of the memory, i.e., the types, as will be explained during this dissertation. We term the multiple interpretations *type hypotheses*. Not all of these hypotheses correspond to the actual ground truth, therefore it is required to decide which of the hypotheses are the best interpretations. In this dissertation, it will be shown that DSI itself can be utilized to choose among the possible hypotheses by evaluating these hypotheses with DSI and then selecting the one corresponding to the most complex DDS interpretation found. The intuition is that, e.g., a skip list does not appear by chance, but is a strong hint that the hypothesis resulting in such a DDS interpretation indeed revealed the typing as originally carried out by the software developer. The resulting types refined by DSIbin can be seen on the left in Fig. 1.5, where now the nested `sraSpan front` is detected, all three `sraSpan` instances are merged as indicated by the same name of the structs (`struct 1176`), and the missed primitive types of the recovered `struct sraRegion` are now typed.

Thereby, the nested struct discovery by DSI can be bolstered significantly. The resulting type information lets DSI play off its strength regarding its memory abstraction, especially utilizing cells with the improved nesting detection. This enables DSI to deal with sophisticated data structures such as the Cyclic Doubly Linked List (CDLL) used in the Linux kernel [15] or complicated nesting scenarios including arbitrary parent child nesting.

Summarizing part two of this dissertation, the DSI approach is opened up for binaries in addition to source code. This enables the inspection of not only compiled C but also compiled C++ programs, thus widening DSI's scope. The resulting DSIbin tool is shown to outperform related work [49, 69, 74] in terms of detectable data structures, e.g., skip lists, and arbitrary nesting. Additionally, the type refinement step improves upon the state-of-the-art type recovery tool Howard, making the approach also interesting to software analysts who only require improved primitive and compound type information such as to aid forensics, which is one of Howard's intended use cases. Both binary DSI approaches, i.e. with and without type refinement, are evaluated by hand-written, text-book [104, 108], shape-literature [65, 72] and real world examples [20, 32, 34]. As a side-effect, the benchmarking also testifies the robustness of the initial DSI approach. Further, more insight into DSI's core algorithm and design decisions is gained, which can act as improvements for future work.

1.3 Structure of the dissertation

The structure of this dissertation follows the general division of the DFG project into Pt. I related to source code analysis and Pt. II related to binary analysis. The source code part first describes the detailed preliminary work on DSI by Dr. White in Ch. 2, followed by a discussion on related work in Ch. 3. Subsequently, this dissertation highlights in Ch. 4 the importance of the consistency of the underlying memory abstraction and how this is achieved by DSI. With the established consistency, higher level abstractions can be modelled on top of the memory abstraction. Among those is the quantification of SCs, which is central for DSI's DDS detection capabilities and detailed in Ch. 5. The quantified SCs also enable the analysis of repetitive behaviour of DSs, both structurally and over time. An algorithm for capturing the temporal repetition of a DDS is developed in Ch. 6. To show the feasibility of the whole DSI approach, a working prototype has been implemented. During the work on the DSI prototype an optimized parallel version of the algorithm has been developed, which is explained in Ch. 7. The DSI approach was presented at ISSTA in 2016 [107] and is comprehensively benchmarked in Ch. 8 with a focus on the detectable DDSs instead of performance. Ch. 8 also presents interesting real world examples that are currently out of scope of DSI's capabilities and motivate future work. Finally, the results of the source code DSI approach of Pt. I are summarized in Ch. 9.

The binary part of this dissertation starts out by giving an overview of our approach for opening up DSI for C/C++ binaries in Pt. II, followed by a survey on the usage of DDS in malware as a motivational use case for DSIbin in Ch. 10. With the use case laid out, DSI's dependencies on perfect type information found in source code is investigated in Ch. 11. As this type information is unavailable when dealing with binaries, it needs to be reverse engineered; therefore, existing type recovery tools are surveyed in Ch. 12. In order to interface with a type recovery tool and to extract the runtime information from the binary required for DSI to function, DSI's source code instrumentation with CIL needs to be replaced with a binary instrumentation framework, which is discussed in Ch. 13. A first DSIbin implementation with the state-of-the-art type recovery tool Howard [98] and Intel's instrumentation framework Pin [82] is then detailed in Ch. 14. Our approach is benchmarked in Ch. 15, showing the promising potential of the combination of Howard and DSI. However, it becomes eminent that DSI's full DDS discovery potential cannot be unleashed due to limitations in Howard's type recovery. This is then compensated by a refinement step developed in Ch. 16 and benchmarked in Ch. 17, demonstrating that DSIbin outperforms related work. The results of the binary code DSI approach of Pt. II are summarized in Ch. 18.

1.4 Methodology

In order to answer the previously stated research questions, the methodology shown in Tab. 1.1 was applied. The table shows which **Methodology** was applied to which research question (**RQ**). The methodology is divided into theoretical work (**Theory**) and practical work (**Practical**).

For **RQ 1** a literate review was conducted, to put the DSI algorithm into the context of the related work. As DSI pseudo code is already available from Dr. White, the pseudo code is reviewed to get a deep understanding of DSI and to find opportunities for improvement. Further both challenging real life DDS as well as artificially created DDS that aim for white box testing of DSI are identified by the author of this dissertation. The theoretical work is accompanied by the practical work of implementing and benchmarking a first DSI prototype. Additionally case studies are executed to highlight, e.g., further previously unintended use-cases of DSI. Throughout both challenges the DSI concept gets refined and extended. This includes the identification of weaknesses within DSI and subsequently working out the corresponding pseudo code to improve these points. Additional benchmarks are composed to specifically address the improvements. For all challenges, a prototype is implemented and tested against the composed benchmarks.

The theoretical work for **RQ 2** is again a literature review into various directions. One direction is related work, another direction is about type information and type recovery tools. The latter is in the form of a survey. This lays out the foundation for the theoretical discussion about the loss of type information in binaries compared to source code and the implications for the DSI algorithm. Based on these findings a first architecture of DSI for binaries (DSIbin) is envisioned and the initial benchmark from **RQ 1** gets extended. This is accompanied by conducting a survey on real world malware source code to find interesting DDSs and directions for future work. As is the case with the challenges of **RQ 1** all theoretical work on **RQ 2** and its challenges **Ch 2.1** to **Ch 2.3** is backed up by a prototypical implementation. The prototypical implementations are all benchmarked to have a proof of concept and to identify the positive and negative aspects of the approach. Indeed **Ch 2.3** tackles the weaknesses of **Ch 2.1**, which leads to a whole new approach of binary type refinement.

1.5 Project context and publications

This dissertation has been carried out in the context of DFG funded research project "Learning Data Structure Behaviour from Executions of Pointer Programs" (LU 1748/4-1). As a short historic background, the project ideas are based upon the research program comprehension tool *Data Structure Operation Location and Identification (dsOli)* [105,106] which has been authored and implemented by Dr. White. It allows for the identification of dynamic data structures, e.g., lists, and their

Table 1.1: Methodology for answering the research questions (RQ) and challenges (Ch) stated in Sections 1.1 and 1.2 of this dissertation.

RQ	**Methodology**	
	Theory	**Practical**
RQ 1	Literature review Review DSI pseudo code Identify real life DDS examples Compose challenging artificial DDS examples	Implement DSI prototype Conduct various benchmarks Conduct case studies
Ch 1.1	Identify weaknesses of DSI Refine the DSI pseudo code Setup of benchmark	Refine DSI prototype Conduct benchmark
Ch 1.2	Identify possible strand connections Identify possible quantifications of strand connections Extend the DSI pseudo code	Refine DSI prototype Conduct benchmark
RQ 2	Literature review Create architecture for DSIbin Create pseudo code Discussion of type information loss Compose benchmark Literature and code surveys	Implement various DSIbin prototypes Conduct various benchmarks Conduct case studies
Ch 2.1	Literature review Literature survey on type recovery tools Malware code survey Create architecture for DSIbin Compose benchmark	Implement prototype Conduct benchmark
Ch 2.2	Literature review Identify differences between source and binary instrumentation Create architecture for instrumentation	Implement prototype Conduct benchmark
Ch 2.3	Identify weaknesses of Ch 2.1 Create pseudo code for type refinement	Implement prototype Conduct benchmark

associated operations when C source code is available. However, dsOli does neither support nesting nor recursive data structures such as trees.

The main topics of the DFG project have been extending the scope of detectable data structures, when compared to dsOli and other related work [49, 69, 74], especially regarding nesting and recursive data structures, handling of real world examples and moving towards binaries as input instead of source code. As the research for this dissertation was conducted while working on achieving the DFG project's goals, additional contributions for DSI on source code have been made and a binary version of DSI has been developed. In order to delineate this dissertation from the work done by Dr. White, the contributions from the author of this dissertation for our three jointly authored and peer-reviewed publications within the DFG project are spelled out in the following:

[ASE17] T. Rupprecht, X. Chen, D. White, J. Boockmann, G. Lüttgen and H. Bos. DSIbin: Identifying Dynamic Data Structures in C/C++ Binaries. In 2017 IEEE/ACM International Conference on Automated Software Engineering (ASE'17), pp. 331-341. ACM, 2017.

- Analysis of DSI source code dependencies and the implications of type information loss and weakly robust type information (Ch. 11);
- Survey of type recovery tools and binary instrumentation frameworks suited for DSIbin (Ch. 12);
- Parallelization of the hot-spot parts of the DSI algorithm (Ch. 7);
- Benchmarking of the first naive DSIbin implementation, see **[CCS16]** (Ch. 15);
- Sophisticated DSIbin prototype, which refines Howard's recovered type information (see **[CCS16]**) (Ch. 16);
- Benchmarking of the sophisticated DSIbin prototype on the set of examples used in the naive approach (Ch. 17).

[CCS16] T. Rupprecht, X. Chen, D. White, J. T. Mühlberg, H. Bos and G. Lüttgen. POSTER: Identifying Dynamic Data Structures in Malware. In 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16), pp. 1772-1774. ACM, 2016.

- Survey on the usage of DDSs found in leaked malware source code [34] (Ch. 10);
- Early on evaluation of DSI's robustness wrt. completeness of type information (Ch. 11);

- Survey on type recovery tools for binaries, arriving at Howard as a suitable tool for the combination with DSI (Ch. 12);
- First naive DSIbin implementation with a new binary instrumentation front-end for DSI (Ch. 13), and the incorporation of Howard's excavated types in collaboration with the developers of Howard, Dr. Chen and Professor Bos of VU Amsterdam (Ch. 14);
- Evaluation of Howard's type recovery capabilities and the naive DSIbin implementation via benchmarks of extracted components from examined malware, especially the DLL of child DLLs from HVNC in Carberp and the C++ STL lists in the IRC proxy of AgoBot (Ch. 15);
- Discovery of limitations in Howard's type recovery algorithm, especially regarding nested struct detection and type merging, which hampers DSI's data structure detection capabilities (Ch. 16).

[ISSTA16] D. White, T. Rupprecht and G. Lüttgen. DSI: An Evidence-Based Approach to Identify Dynamic Data Structures in C Programs. In International Symposium on Software Testing and Analysis, (ISSTA'16), pp. 259-269. ACM, 2016.

- Algorithms for artificial memory events and memory leak detection (Ch. 4);
- Quantification of strand connections (Ch. 5);
- Temporal repetition algorithm (Ch. 6);
- Realization of the complete approach resulting in the DSI research tool;
- Extensive benchmarking of DSI including real world (libusb [13], bash [10]) and shape analysis examples (Ch. 8).

Additionally, the following two jointly authored but not peer-reviewed publications resulted through the work on the DSI project:

[KPS17] Rupprecht, T., Boockmann, J. H., White, D. H., and Lüttgen, G. DSI: Automated Detection of Dynamic Data Structures in C Programs and Binary Code. In the 19th Coll. on Programming Languages and Foundations of Programming (Kolloquium Programmiersprachen, KPS'17). 2017.[2]

- Wrap up of project status, including DSI on source code, DSIbin on x86 binaries and various future work topics with early on proof of concepts in the form of bachelor and master theses.

[2] https://www.swt-bamberg.de/luettgen/publications/pdf/KPS2017.pdf. Accessed: 1st September 2018

[KPS15] White, David H., Rupprecht, Thomas and Lüttgen, Gerald. dsOli2: Discovery and Comprehension of Interconnected Lists in C Programs. In the 18th Coll. on Programming Languages and Foundations of Programming (Kolloquium Programmiersprachen, KPS'15). 2015.[3]

- Early on prototypical DSI implementation;
- Benchmark of prototype on a small set of examples.

[3] https://www.swt-bamberg.de/luettgen/publications/pdf/KPS2015.pdf. Accessed: 1st September 2018

Part I

Dynamic data structures in C programs

2 DSI on source code

Data structures are an integral part of a program, making the need to understand them a necessity. However, the task of program comprehension is challenging and might not be straightforward, e.g., when the programmer is confronted with legacy or tricky low-level kernel code. Here code quality and complexity might burden the programmer with, e.g., wrongly chosen variable names, pointer casts, sophisticated memory allocations, or the usage of macros. This is especially true when dealing with C programs, where low level details can distract the programmer from gaining a high level understanding of the program. Unfortunately, the situation worsens with hard to understand data structures such as skip lists and when documentation is lacking. DSI alleviates the problem by automatically detecting dynamic pointer-based data structures when C source code is available. With detailed information about the employed data structures, DSI's use cases are not limited to program comprehension alone; for example, it is possible to interface with formal verification tools, like VeriFast [73] by automatically generating program annotations, or to visualize memory graphs [105].

This chapter first elaborates on the general DSI approach explained in the introduction. Then, specific topics of DSI are highlighted that have been developed within this dissertation. When looking at the strengths of DSI, three main aspects not covered by related work become evident: (i) DSI develops a fine grained heap abstraction surpassing the precision of competing approaches, (ii) DSI describes arbitrary nesting scenarios and data structures not handled by other tools [49, 69, 74], e.g., seamlessly handling the Linux kernel list [15], and previously neglected data structures like skip lists; (iii) DSI includes degenerate shapes into its analysis instead of avoiding them [69, 74, 106], thereby allowing for the inspection of data structure manipulation operations.

DSI is split into a front-end module, performing an online trace recording, and the core DSI algorithm module, performing an offline trace analysis. This modularization allows for adapting DSI to different input formats by exchanging the front-end module, which will be utilized when dealing with binary code in Pt. II of this dissertation. The modularity shows that the DSI approach is by no means tied to a specific programming language but is a general concept for analysing dynamic data structures. The motivation to analyse C source code is C's possibility to create challenging heap states and inclusion of compact but rather hard to read programming constructs. The Linux Kernel List (LKL) [15] is a prominent example for both aspects, as it allows for the connection of arbitrary memory chunks and uses a macro based interface with extensive pointer casts and pointer arithmetic.

The front-end consists of the CIL framework [88], which instruments relevant memory events in the C source code, e.g., heap allocations and deallocations, creation and deletion of stack frames and pointer writes. When running the instrumented program, the runtime events are recorded into an event trace that is passed to the offline analysis once the inspected program has terminated. DSI can only analyse executed parts of the program as is the case with any dynamic analysis. This can be seen as an increased level of precision instead of a limitation, as the characteristics of a particular program run are revealed. Therefore, DSI itself is not concerned with the problem of full path coverage. If need be, dedicated tools like KLEE [52] exist for this purpose. While working with DSI, we guaranteed the proper utilization of the data structures, i.e., creation and deletion, directly within the example programs or within the manually developed drivers for those examples.

We will discuss each step of DSI's analysis with the example depicted in Fig. 2.1. The analysis starts by modeling a Points-to Graph (PTG) to represent the memory layout for each time step, i.e. memory event, within the trace (Fig. 2.1 phase (a)). In a PTG vertices represent memory chunks and edges are pointers. Note that DSI relies on sub-regions of memories, called *cells*, instead of whole memory chunks for the memory abstraction. The cell abstraction is shown in Fig. 1.2, where the cells are depicted by the dashed squares, and which is required to seamlessly identify LKL like data structures. Without the cell abstraction the head node of the LKL would be missed out, as it is of a different type than the list remainder.

Upon this representation, DSI abstracts the memory by establishing *strands*, which are essentially SLLs as shown by S_n. More precisely, a strand requires a common *linkage condition* between the cells forming the strand. The linkage condition enforces that all cells of a strand have the same *linkage offset* for the pointers connecting the cells and that the cells are of the same type. Strands are interconnected with each other either tightly or loosely. The tight connection is termed *overlay* and the loose connection is termed *indirect*. The strands and their connections are represented in a *Strand Graph (SG)*, where strands are vertices and their connections are edges (Fig. 2.1 phase (b)). More specifically, overlay connections are bidirectional edges, as it is possible to get from one strand to the other by an offset calculation, e.g., between strands S_2 and S_3. Indirect connections are directed edges, as they represent the pointer connection from one strand to the other, thus preventing one to get from the pointer target back to the source, e.g., between strands S_1 and S_2. Strand connections are further quantified by describing the offsets between the strands. This is important to enable DSI's data structure detection, which is directly performed on the SG and the follow up phases of structural and temporal repetition aggregation, as discussed in the following.

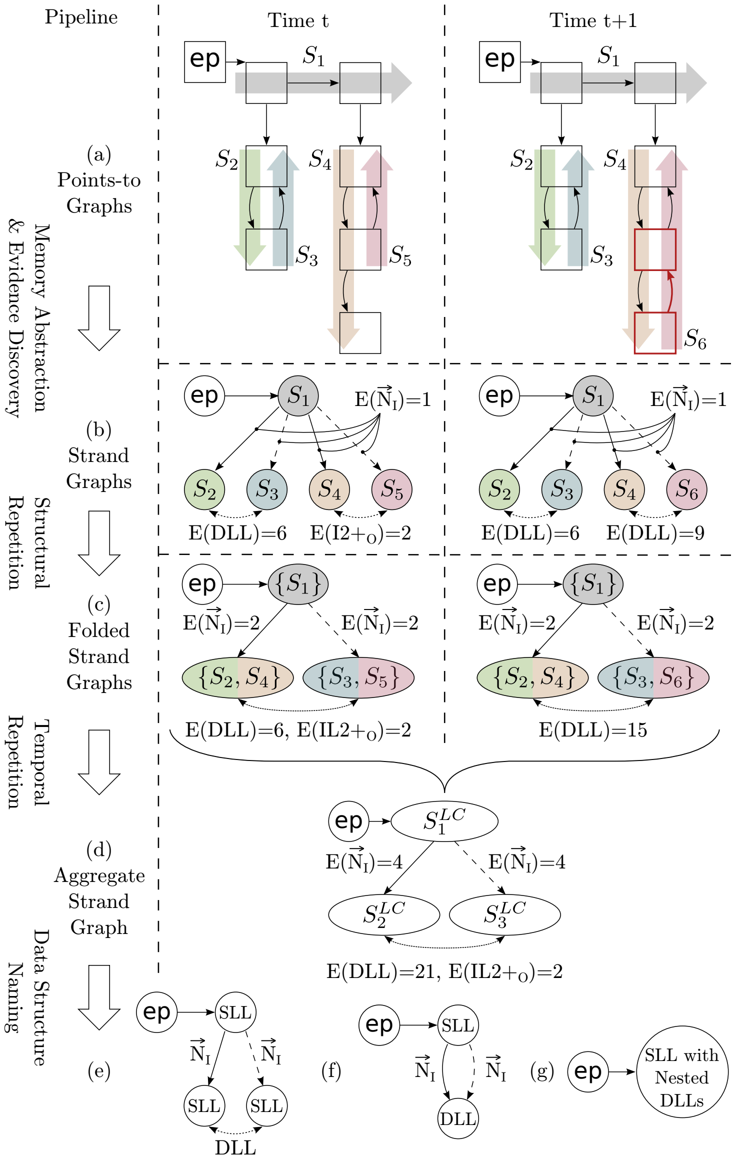


Figure 2.1: Overview of DSI, reproduced from our publication [107].

The data structure detection phase tries to match predicates against the SG, which rely on the characteristics of the strands, i.e., the type of the cells forming a strand and the *linkage offset* that describes the pointer offset of the strand from the start of the cell. Additionally, the predicates depend on the characteristics of the edges, i.e., overlay or indirect edges and their specific offsets. Thereby, it is possible to precisely select strands and strand connections from the graph that form a particular data structure, i.e., fetching a nesting on overlay versus a nesting on indirect parent-child relation. Once a data structure is detected within the SG, all the strand connections are tagged with the label of the detected DDS, e.g., DLL, and an evidence count is applied to the edge, thereby quantifying the observed data structure. The DDS detection is performed exhaustively on the graph. The labeling and evidence count can be seen in phase (b) of Fig. 2.1 when looking at the labels and evidence counts applied at both time steps t and $t + 1$. The DLL label gets applied to strands S_2 and S_3 for both time steps. The evidence count of 6 stems from the DLL predicate, which weighs each cell pair with 1 with and additionally 2 for inspecting both cells, resulting in a count of 3 for each cell pair. The degenerate shape of the second child DLL in time step t results in an evidence count of 2 as only two cells are intersecting, i.e., each intersecting cell pair has a count of 1. At time step $t + 1$ both child elements are in a stable shape of a DLL.

The DDS matching requires detailed knowledge about the different DDSs that DSI can handle and how they are represented by the strand and strand connection abstraction. This leads to a taxonomy that describes precisely in what order the different DDSs need to be matched in order to prevent misclassifications or less precise classifications. This can best be seen with examples. Consider two strands which are intersecting (a) in one node and (b) intersecting on the same head node. In both cases the strand and strand connections are the same, as they only have one intersection point. But one needs to test for (b) first as this is a more precise description of the DDS that is also covered by (a). Another example is overlay nesting (No), e.g., seen when the head node of a child SLL is embedded inside of a parent SLL, and a Binary Tree (BT). A BT actually shows the characteristics of No when considering only one branch of the tree with multiple branches starting from it. Therefore, it is required to test for the BT predicate first to prevent an early No detection which would hinder the BT detection. These specifics are actually present across the whole set of DDSs covered by DSI and are captured in a taxonomy describing the hierarchy for detecting the DDSs. This is however not important until Pt. II of this work, where the hierarchy of the taxonomy will be exploited by DSIbin.

After the DDSs detection phase, we have arrived at a SG that is now decorated with the labels and evidence counts of the detected data structures for one time step. Now the structural repetition phase is executed, where parts of the data structure that perform the same role are aggregated (Fig. 2.1 phase (c)).

The main aspect of the structural repetition is the aggregation of the strands and strand connections that perform the same role and, thereby, the accumulation of the detected data structures and the applied evidence counts. All parts of the SG that perform the same role are folded together, resulting in a *FSG*, as seen in Fig. 2.1. Once the FSG is calculated for each time step of the event trace, the temporal repetition phase can be executed by inspecting the FSGs from the point of view of an entry pointer over its lifetime during program execution. Entry pointers can be either a pointer from the stack into the data structure, or an element of the data structure itself that is stored on the stack. The latter can be seen with, e.g., the Linux kernel list, where the head of the CDLL can reside on the stack. The temporal repetition for an entry pointer is performed by incrementally aggregating the FSGs of each time step until the entry pointer no longer exists (Fig. 2.1 phase (d)). The resulting *ASG* will carry the accumulated evidence counts for the structural and temporal repetitions, where the highest evidence count is considered the correct interpretation for the data structure. This can be seen by the accumulated evidence counts shown in phase (d), where the DLL label clearly outnumbers the I2+$_{\text{O}}$ label. Finally the label information of phase (d) can be used to name the whole DDS by subsequently aggregating all the vertices and replacing them with the attached DDS label (Fig. 2.1 phase (e)).

3 State of the art

This chapter covers closely related work that performs data structure detection either by a dynamic analysis [49, 69, 74, 106] or static analysis [65, 72]. There exist further tools that also analyse dynamic data structures, but for other reasons, e.g., optimization [75, 92] or visualization [39, 85]. Such tools are not discussed extensively here as they are not primarily in line with DSI, but they are referenced whenever required. Type recovery tools for binaries focusing on primitive data types and compound types like structs are presented in Ch. 12 of Pt. II.

In the remainder of this chapter we will discuss the DDS detection tools shown in Table 3.1. The table shows the **main use case** of the tool, i.e., Reverse Engineering (RE), Program Comprehension (PC), Signature Generation (SIG), Formal Verification (FV) or Visualization (VIS). RE and PC are quite similar because, in both cases, the goal is to comprehend the program, but PC points out that the approach works on source code as **input**. The approaches differ in their type of **analysis**, i.e., static or dynamic. Laika [60], Heapviz [39] and HeapDbg [85] are also considered to be dynamic analyses as they take heap snapshots as **input**, which requires program execution. In contrast to the other dynamic analyses they do not track the explicit memory (de-)allocations to construct PTGs; instead, they build the PTGs from concrete heap snapshots as input for their analysis. Other **input** formats are binary code or source code. dsOli and DSI also execute a binary for their analysis, but require the source code to insert their instrumentation routines prior to compilation. The requirement of source code is relaxed for DSI in Pt. II, where DSI is opened up for binaries. All input formats differ in the amount of information they provide, which also correlates with the chosen **language** that the analysis supports. Languages such as Java and C# provide meta information in their bytecode, such as type information including class names and method signatures [12], whereas compiled C code does not provide such information. Source code provides the most information, but it is less likely that source code is always available, e.g., when dealing with third party libraries and malware. Because the Data-structure Detection Tool (DDT) [74] explicitly requires DDSs access through a well defined **interface**, which is a limitation when dealing with, e.g., C macros or inlining, the table below reflects whether an approach *depends* on interfaces for its analysis. Most of the related work does not handle **nested structs**, which is crucial for LKL like DDSs and is also important for detecting situations where parts of a child DDS are embedded inside the parent, e.g., the head node of a child SLL is embedded inside of the parent SLL. In the context of DSI, the latter is referred to as **overlay nesting**. The other possibility of connecting DDSs is simply via pointer connections, termed **indirect nesting**. Some of the tools are able to detect **overlayed** DDSs, which means that multiple DDSs are combined, e.g., a SLL running through a BT. The table explicitly lists the detectable DDS, e.g., Cyclic Singly Linked List (CSLL) and CDLL, and has an additional column **other trees**, which sums up all trees that are not a standard BT, e.g., red black trees and n-ary trees. In the following we will now discuss the different tools shown in the discussed table in more detail.

3.1 MemPick

MemPick [69] is a dynamic analysis for C/C++ binaries for detecting DDS, like (cyclic) singly/doubly linked lists and various trees. It tracks the heap states of the program with the help of a memory graph and performs a classification with a hand crafted decision tree upon the memory graph in order to detect the DDS.

The memory graph is created by tracking memory (de-)allocations and pointer writes, tracked by instrumenting the binary with the PIN [82] framework. MemPick performs type merging of binary types when the same instruction operates on two objects either as source or target operands. Specifically, MemPick aims for instructions that connect two heap buffers through a pointer that always originates at the same offset. It is not fully clear whether this offset is always from the start of a memory region, which would then prevent MemPick to merge multiple nodes of the same type in one memory chunk, as the offset would change.

MemPick is aware of degenerate shapes, but it tries to avoid them instead of including them into the analysis like DSI. MemPick only inspects memory graphs during quiescent periods, when the DDS is not changed. The assumption is that the DDS is in a stable shape during quiescent periods. MemPick actually samples into multiple quiescent periods and interprets the memory graph each time. It only keeps those hypotheses that are true for all samples. This can be considered as a sort of temporal repetition, but not as fine grained as DSI, because the decision for a match is binary whereas partial reinforcements are possible with DSI.

One of the major drawbacks of MemPick is the way it detects DDS. It analyses the memory graph and creates sub-graphs by clustering connected nodes of the same type. All linkages between clusters are cut, thus effectively removing the nesting information. The clusters are then interpreted individually by the aforementioned decision tree. The authors of MemPick never speak about reconnecting the gathered information to identify an indirect nesting scenario. Further, overlay nesting can never be identified with this approach. As an example, consider that the head of a CDLL child is actually embedded inside of the parent. The linkages would be cut in both directions, which would lead to a missed cyclic property. A scenario like this is seen with libusb [13], where the parent LKL contains child LKL. DSI is capable of detecting those scenarios. In addition, MemPick cannot handle a DDS running through differently typed nodes.

3.2 DDT

DDT [74] detects high level DDS, such as lists or trees. It is based on a dynamic analysis of C/C++ binaries, by instrumenting memory events. Data structures constructed on the stack are not considered. DDT (i) creates a memory graph for an initial DDS shape detection; (ii) finds DDS interfaces; (iii) detects invariants on the interfaces; (iv) feeds all the information to a hand crafted *decision tree* in order to arrive at the DDS interpretation.

More specifically, DDT builds up a PTG, termed memory graph, for each DDS in the program by tracking memory events, i.e., (de-)allocations and pointer writes. Therefore, DDT is also confronted with the problem of typing and merging memory chunks as type information is lost in binaries. To do so, it uses allocation sites for initially producing unique types for memory chunks and later on declares and merges chunks to be of the same type if they are accessed via a common interface function. Additionally, the graph edges are annotated with information to where they point, i.e., whether they point to nodes of the same or a different type, or to static data. This allows DDT to apply graph invariants to initially detect the shape of the DDS.

Afterwards, DDT searches for common interface functions, which exclusively manipulate the structure, i.e., no other code sections directly manipulate the DDS. This assumption works well in the presence of, e.g., the C++ STD library. However, it fails for instance with compiler optimizations like inlining or macro based interfaces, as those result in multiple different code sections that are manipulating the DDS. Once the interface functions are detected, DDT uses them to apply invariants before and after function calls to determine, e.g., inserts by observing an additional node in the memory graph when the function returns. With the basic shape information and the additional interface invariants, DDT then applies a hand crafted decision tree to arrive from a high level binary tree to, e.g., a more specific red black tree.

When compared to DSI, DDT makes strong assumptions regarding the presence of well defined interface functions. It has a node based representation of DDSs, as it directly works on the PTG, where DSI instead uses its strand abstraction. Because no notion of cells is present in DDT, the nodes of the DDS need to be of the same type, and one DDS element always corresponds to one memory chunk. This prevents the detection of DDSs running through different nodes, for which the LKL is a prominent example. Additionally, the LKL is usually accessed via a macro based interface, resulting in difficulties for detecting the interface functions. Further, the head node of the LKL can be placed on the stack, which is not considered by DDT. Nested DDSs are handled by DDT as long as the connection is pointer based, e.g., a vector of lists. Arbitrary nesting scenarios are not handled, e.g., nesting on overlay with arbitrary depth.

3.3 ARTISTE

ARTISTE [49] automatically creates data structure signatures for DDS such as (cyclic) lists or trees. The signatures can be used to find instances of these DDSs in binaries; the targeted use cases are reverse engineering, memory forensics or game cheat analysis. ARTISTE functions by creating various kinds of trees (a) to abstract the program heap, (b) to capture different allocation sites for types, termed callsites, and (c) to merge different callsites in order to refine the results.

ARTISTE infers the primitive types that it uses in its various trees. The trees for (a) are called buffer tree and track heap allocations, module loads and stack frames. Callsites are identified by the instruction performing the allocation and are explicitly used to merge buffer trees resulting in callsite trees for (b). This implies that no merging is performed between the heap and the stack. Multiple callsite trees are merged together into type trees if they are considered equivalent by ARTISTE, taking care of one type being allocated at different callsites. With every tree merge performed, the resulting trees are getting more refined, as partial information in each tree can be aggregated. Actually, ARTISTE enables the aggregation of multiple program runs to refine its results further, which is a form of temporal repetition not unlike to that of DSI. The actual shape analysis utilizes the type trees and is performed upon PTGs, termed heap graphs, which are sampled periodically, without avoidance strategies for degenerate shapes. The shapes are matched against a predefined set of predicates, which are then used to form the data structure signature.

When compared to DSI, there is a similar notion of temporal and structural repetition, but the knowledge gained is on the level of the various trees, i.e., the primitive types, and not on the level of the DDSs. This becomes evident in the context of the predicate for a DLL, which requires a forward and backward pointer at the offsets o_f and o_b for each source (src) and destination (dst) memory chunk of the DLL: $\forall(src, dst, o_f), \exists(dst, src, o_b)$. This fails in case of, e.g., insert operations. DSI can outnumber the degenerate shape with stable shapes via structural repetition; in ARTISTE however, the structural repetition treats all DLL children uniformly despite the current shape. Moreover, ARTISTE only considers heap graphs, thus leaving out the stack, though parts of a DDS are sometimes placed on the stack in practice. Additionally, ARTISTE does only consider objects of the same type when detecting a DDS. Both aspects are key when dealing with the LKL, which can place its head on the stack and can run through nodes of different type.

3.4 dsOli

Data Structure Operation Location and Identification (dsOli) [106] is a dynamic analysis operating on C source code, which identifies DDS and their operations with the help of machine learning and pattern matching. Thereby dsOli allows to draw conclusions about DDS semantics, e.g., a list is used as a stack.

More specifically, dsOli instruments the C source code to capture memory events, then compiles and runs the program, resulting in an event trace. The trace is analysed offline by first building a PTG for each time step of the trace to represent the heap state. A time step is defined by various events, such as memory (de-)allocations and pointer writes. For each time step the changes between the PTGs before and after the event are recorded, resulting in a feature trace. Examples for such changes are how many in and out pointers to a vertex exist, or if the pointer points to the same vertex as before. Subsequently, repetition in the feature trace is detected by a machine learning approach that essentially finds regular expressions which capture recurring behaviour in the trace. These repetitions are *potential operations*. To determine if a potential operation is a DDS operation a set of templates are matched against the pre- and post- graphs of the potential operation. The pre-graph represents the layout of a DDS prior to an operation, whereas the post-graph represents the layout of the DDS after an operation. The templates exploit these pre- and post-graphs by describing a DDS operation on behalf of the changes in the layout of a DDS imposed by the operation. If a template match was observed, the potential operation is considered as confirmed. Otherwise it is discarded as being a false positive. After the operations are confirmed, they are used to actually label the DDS by inspecting which operations manipulate a DDS over its lifetime by a majority vote, i.e., counting how often each operation was observed. If the operations are inconsistent, e.g., the same counts for ”tree inserts” and ”SLL inserts into the middle of the list” is observed, this might hint to a programming error.

3.5 Laika

Laika [60] generates DDS signatures for detecting the usage of the same DDSs among different versions of a program, so as to recognize different versions of a polymorphic virus family by their used DDSs. Laika operates by analysing a memory snapshot with unsupervised machine learning to find the DDSs. This is already one fundamental difference to DSI; Laika only analyses one heap state, whereas DSI monitors the complete event trace of a program. Further, Laika does not label the found DDSs, and thus also does not handle degenerate and stable shapes as they are not important for its analysis.

However Laika’s general idea is highly interesting for DSI/DSIbin. Laika creates very coarse grained classification of DDSs for its virus detection, i.e., Laika does not apply concrete DDS labels for the structures it detects. With regards to high level DDS Laika only classifies chunks of memory as being of a certain class and types pointers as being a pointer to a particular class. Therefore DSI's much richer DDS description should allow for more detailed DDS comparisons between different program versions. For example consider the precise differentiation of DSI between indirect and overlay nesting. Without this information two different list of lists implementations, one with indirect and one with overlay nesting, appear to be the same, i.e., as indirect nesting. Such scenarios are discussed in more detail in Ch. 11.

The very coarse grained primitive type recovery performed by Laika, is mainly used to ease the feature creation for the machine learning. There are only four types available: address, zero, string and data. Which is no surprise when only operating on a memory snapshot, i.e., no additional information of the usage of the memory is available. As a forward pointer to the second part of this dissertation, DSIbin also offers more detailed primitive types, which in turn should also be valuable for Laika's machine learning approach. A combination of Laika and DSI/DSIbin is left for future work.

3.6 Predator, Forester and ATTESTOR

Both Predator [65] and Forester [72] are prominent examples of static shape analysis tools. They perform a symbolic execution and target formal verification, in contrast to DSI's dynamic analysis focuses on program comprehension. The main difference between static and dynamic analysis is the generality and precision of information that can be inferred. Static shape analysis is undecidable, leading to conservative approximations [74], while a dynamic analysis can offer high precision regarding a particular execution (or possibly multiple ones as is the case with ARTISTE, see Sec. 3.3) but lacks generality, i.e., the result of the information can vary depending on the analysis input.

Predator checks programs that are using linked lists, where the foundation for its shape analysis are symbolic memory graphs (SMGs), which represent arbitrary heap/stack pointer connections. This allows for the handling of lists spread across both memory regions. Because the central focus of the approach are DLLs, especially the LKL, and SLLs, other recursive DDSs like trees are currently not considered. The focus on DLLs is also reflected in some additional abstractions found in the SMGs, where parts of the lists are represented with (doubly-linked) list segments (DLS). A DLS represents multiple equal list nodes as one node. With the LKL where the head is, e.g., placed on the stack and the remainder of the list runs through equal nodes, this would result in a separated head node connected to a DLS representing the rest of the list. This is one fundamental difference to DSI, where all equal subtypes are captured by the strand abstraction, thus DSI already includes the head and does not need to check for any special nodes when analysing the shape.

Forester relies on forest automata for its shape analysis and allows for the detection of (cyclic) SLLs and DLLs, the LKL, trees and skip lists. The considered programs are sequential non-recursive C programs, whereas DSI can deal with recursive programs as well. Forester abstracts the heap with tree components, by splitting the heap, i.e., the PTG, on so called cut-points that are root nodes of a DDS or nodes with multiple incoming pointers. This results in tuples of trees (forests), where sets of tree components are encoded with tree automata. Sets of forests are represented as tuples of tree automata, resulting in forest automata. Unbounded structures like a DLL can have unbounded cut-points, a problem which is solved by representing multiple edges, i.e., next and previous edges in case of a DLL, with just one edge. DSI's abstraction instead is designed to naturally handle an unbounded number of nodes per strand.

ATTESTOR ATTESTOR [40] verifies Java pointer programs and provides debugging information, including counter examples, in case of validation errors. This is contrary to DSI's main focus of program comprehension. ATTESTOR is based on symbolic execution, in contrast to DSI's dynamic analysis, and uses context-free graph grammars for abstracting the heap. Program specifications, such as memory shape and heap structure preservation, are defined by linear-time temporal logic. The analysis is fully automated, except for the manual definition of DDS shapes on behalf of the graph grammars. The supported DDS shapes are (cyclic) SLLs and DLLs, SLs, (balanced) trees and lists of lists.

3.7 HeapDbg & Heapviz

HeapDbg [85] and Heapviz [39] are two similar tools, which both focus on the graphical representation of heap snapshots rather than a fully automated DDS detection. HeapDbg has more powerful capabilities than Heapviz when abstracting the heap, e.g., being able to correlate multiple heap snapshots.

HeapDbg [85] summarizes heap snapshots for the purpose of debugging and profiling by providing an effective visualization and navigation of the heap. A concrete implementation operating on .Net bytecode is shown, which also allows for the inspection of Java programs when compiled with ikvm [11].

HeapDbg starts out with a concrete heap that gets abstracted by first aggregating connected objects of the same type into a recursive data structure (RDS). It classifies the shape of the RDS as being either *tree* or *any*, by analysing the pointer connections between the aggregated nodes. This is significantly less precise than DSI's set of detectable data structures. However, HeapDbg applies a similar notion of structural repetition, where elements of the same type pointed to by objects from the same parent RDS are considered equal and are merged. The connections between the parent and child elements are not explicitly labeled as in

DSI, but are left for interpretation by the analyst. In addition, HeapDbg allows to merge graph instances, resembling DSI's temporal repetition, but performs an over-approximation which leads to information loss.

HeapDbg only deals with heap objects as it analyses .Net bytecode, where all objects are created on the heap by default[1]. Therefore DDSs that are spread across the heap and the stack are not supported by C# by default and are thus out of scope for HeapDbg. As discussed before, the LKL is a prominent example for such a programming technique. Other LKL like situations, e.g., linking objects of different types, can occur with the .Net framework when using inheritance and casts. When looking at the memory layout in case of inheritance, the memory layout of the base class gets copied into the inherited class. This resembles the embedding of the LKL `struct` into an payload `struct`. Unfortunately, HeapDbg is not extensively tested on such situations. It seems only the `pmd` program, which is a static code analyzer, exposes HeapDbg to inheritance by modelling an abstract syntax tree with inheritance. However, it is not explicitly clear, how HeapDbg represents the inheritance, e.g., which object type is used for HeapDbg's analysis. Using the base class would come closest to DSI's cell abstraction.

Heapviz Heapviz [39] aggregates Java heap snapshots for program understanding and debugging. It functions by creating a graph abstraction where objects are nodes and pointer connections are edges. It performs a similar structural repetition technique as HeapDbg, thus the same limitations apply when compared to DSI. No temporal repetition is executed by Heapviz, nor will the detected DDS be labeled, i.e, named.

[1] Exclusively creating objects on the heap in C# is relaxed by using undocumented C# API calls and manual byte copies for placing objects on the stack [4]. Though this approach is considered a hack by the author of this dissertation, as the technique is neither part of an official specification nor an official documentation. Therefore it is not regarded as common practice.

Table 3.1: Overview of DDS detection tools

Tool	**Input**	**Analysis**	**Interfaces**	**Nested structs**	**Main use case**	**Indirect nesting**	**Overlay nesting**	**Overlayed DDS**	**Language**	**SLL**	**CSLL**	**DLL**	**CDLL**	**SL**	**BT**	**Other trees**
ARTISTE	b	d	✗	✗	RE	✓	✗	✗	C/C++	✓	✓	✓	✓	✗	✓	✓
MemPick	b	d	✗	✗	RE	✗	✗	✓	C/C++	✓	✓	✓	✓	✗	✓	✓
DDT	b	d	✓	✗	RE	✓	✗	✗	C/C++	✓	✓	✓	✓	✗	✓	✓
dsOli	s	d	✗	✗	PC	✗	✗	✗	C	✓	✓	✓	✓	✗	✓	✗
Laika	sn	d	✗	✗	SIG	✓	✗	✗	C++	i	i	i	i	i	i	i
Predator	s	st	✗	✓	FV	✓	✓	✗	C	✓	✓	✓	✓	✗	✗	✗
Forester	s	st	✗	✓	FV	✓	✓	✗	C	✓	✓	✓	✓	✓	✓	✓
ATTESTOR	s/b	st	✗	✗	FV	✓	✓	✗	Java	✓	✓	✓	✓	✓	✓	✓
Heapviz	sn	d	✗	✗	VIS	✓	✗	✓	Java	i	i	i	i	i	i	i
HeapDbg	sn	d	✗	✗	VIS	✓	✗	✓	C#	i	i	i	i	i	✓	i
DSI	s	d	✗	✓	PC	✓	✓	✗	C	✓	✓	✓	✓	✓	✓	✗

Symbol explanation: – different input formats: **b**inary, **s**ource, **sn**apshot – type of analysis: **d**ynamic, **st**atic – capability to handle a DDS: **i**ndirectly, e.g., by displaying the layout of the DDS without automated interpretation

4 Consistency of memory abstractions

DSI's memory abstraction is based on a precise model of the heap, i.e., DSI's PTGs, for every time step of the event trace under analysis. The instrumentation captures all explicit memory manipulating events within the execution trace. However, events such as a `free` might have side effects that are not explicitly captured by the recorded event trace. This is depicted in Fig. 4.1, which shows several allocated chunks of memory, together with their associated entry pointers. When freeing the memory in the middle, all incoming and outgoing pointers to the memory implicitly get broken. The incoming pointers become dangling pointers [61], i.e., point to deallocated memory. Good programming practice takes care of the incoming pointers and explicitly sets them to a defined value, though this practice is not enforced. The outgoing pointers of the freed memory are not of concern to the software developer, as they are inaccessible after the memory is freed and, thus, typically are not set to a default value prior to releasing the memory. As DSI builds its strand abstraction on top of pointer connections, it is vital for DSI to keep track of all pointer manipulations, even those in deallocated heap areas, in order to maintain a precise model of the heap, including the described implicit memory manipulations.

The situation can become worse when programming errors result in memory leaks, where, in the easiest case, a wrongly set pointer might just leak one element, thus having similar semantics as an (unintended) free. In the worst case, the leak might affect all allocated memory, i.e., more than one element becomes unreachable. The rationale regarding implicit pointer manipulations for an explicit free also applies for leaked memory. Therefore, DSI needs to be aware of memory leaks to track implicit pointer manipulations.

It is of course possible to use already existing memory leak detection tools, such as [6, 22, 24, 35, 55, 71, 109], to find and correct memory leaks prior to using DSI, as DSI requires access to source code anyways. The situation changes, when DSI is opened up for binaries, i.e., DSIbin in part II of this dissertation, where it is not necessarily possible to correct the described problems of misbehaving memory handling as this requires patching a binary. Instead DSIbin needs a way to natively detect and deal with such situations. Therefore the required mechanisms are already developed and implemented for DSI to let it profit from the requirements of DSIbin.

With the knowledge about implicitly manipulated pointers, DSI is able to keep its subsequent abstractions, in particular the strands, consistent with the heap.

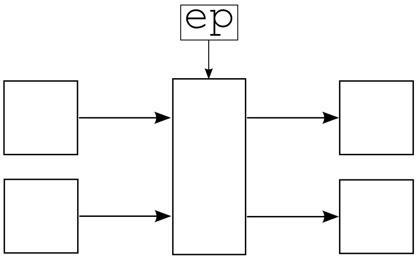


Figure 4.1: Freeing memory with incoming/outgoing pointers, with all entry pointers (ep) omitted except for the freed memory.

This is fundamental for the precision of DSI's analysis. As DSI is a pipelined approach and the heap handling is implemented in the PTG creation phase, this phase becomes the *producer* of this information. The subsequent phases are the *consumer* of this information; thus, a way needs to be devised how to pass the information into DSI's pipeline.

This chapter describes in Sec. 4.1 the concept of an *Artificial Memory Event (AME)* for modeling implicit memory changing events, which are otherwise not perceptible by DSI. Sec. 4.2 discusses why DSI uses its own memory leak detection, including related work with regards to memory leak detection tools and algorithms. Additionally, the integration of the leak detection into the AME concept is presented. A dedicated benchmark with explicit test cases for both the AME concept and our memory leak detection is conducted in Sec. 4.3, though the AMEs and the memory leak detection are also in use in all benchmarks for Data Structure Investigator for Source Code (DSIsrc) and DSIbin.

4.1 Artificial memory events

The creation of AMEs is handled in the PTG creation phase. A PTG reflects the current state of the heap and therefore memory (de-)allocations and pointer writes alter the layout of the PTG. The handling of a memory dealloction event forces DSI to cope with a multitude of affected pointers pointing into and out of the released memory. DSI sets all pointers affected by the event to the value *undef*, even if the software developer later on overwrites this value again. This leaves the heap in a consistent state after the event was executed. Heap memory is usually released with the `free` system call. If the memory is placed on the stack it gets released when ever the stack frame is destroyed. Such an event is called *Variable Leaving Scope (VLS)* event when captured by DSI's instrumentation. When a memory leak gets detected, DSI collects all the leaked memory and artificially adds a `free` event.

To handle both situations within DSI, i.e., restoring pointers to undef and freeing leaked memory, two types of an *AME* are defined (i) an *Artificial Undef Event (AUE)*; (ii) an *Artificial Free Event (AFE)*. The AUE marks a pointer as having an undefined value, the AFE releases a memory chunk. AME are stored inside a Memory Event (ME) E_i of time step i as a *sequence of AMEs*: $E_i.artEvents = < Eart_1, \ldots, Eart_n >$. An ME gets created by memory (de-)allocation events on the heap and the stack. The syntax shown for $E_i.artEvents$, is used in general within this dissertation for expressing a member of an object, similar to the syntax of object oriented programming languages: *object.member*.

The computation of AMEs is described in the following. DSI creates the sequence of AMEs when ever it observes a *free* or *vls* event. In order to process the affected pointers, DSI first collects all edges pointing into and out of the released memory. The resulting edge set is then used to create the sequence of AMEs. In the case of free/VLS the AMEs are exclusively of type *ArtificialUndefEvent* for resetting all incoming pointers to the released memory. In addition to creating the AMEs, DSI keeps the PTG consistent by removing all the collected edges and the released vertex from the the PTG of the current time step. When extending the base case of free/VLS to also handle memory leaks the additional AFE is required, to release the leaked memory chunks as described in Sec. 4.2.

By storing the created AMEs inside of the MEs all subsequent stages of DSI's pipelined architecture are able to access this information. The strand creation phase (cf. Ch. 2) relies on the presence of AMEs to keep their abstraction in sync with the heap. The consuming phases cycle through all MEs and check for the presence of AMEs. If an event has associated AMEs, the consumer cycles through all AMEs and chooses the appropriate action according to the type of the AME. Such an action might for example be the cutting of a strand in case of setting a pointer to undef. After all AMEs are processed, the actual ME itself is consumed.

4.2 Memory leak detection

As the base case of releasing memory on the heap and the stack explained in Sec. 4.1 requires the explicit handling of the incoming and outgoing pointers, the situation becomes more critical in case of programming errors. Specifically, the handling of memory leaks is important for DSI to guarantee that the memory abstraction is in sync with the actual heap state. If this is not the case, the true shape as seen by the program can possibly be not observed. We refer to leaked memory, when all references to allocated memory is lost, also referred to as a *physical leak* [83]. This can happen due to overwriting a pointer or removing a pointer by (i) tearing down a stack frame holding a pointer, or (ii) freeing a memory region that still holds live references to allocated memory. If such leaks are ignored, unreachable memory will blur the actual DDS shape as seen by the program, i.e., the PTG and the heap state start to differ.

Other situations in which reachable but unused memory is simply not freed by the program, are referred to as a *logical leak* [83], and are of no concern to DSI. These memory regions will be taken into account by DSI's analysis. This follows DSI's paradigm to exactly analyse the current heap state: discard unreachable memory and include live memory. While including live memory can also lead to blurring the intended DDS shape, this might very well give a hint to a software engineer that the software has unintended behaviour. DSI could report all memory that was not deallocated at program termination, by listing all heap allocated memory that still resides in the PTG, together with information about the corresponding allocation sites. Thus DSI would provide the same information as memory leak detection tools such as Valgrind [35] and Dr.Memory [6]. However, DSI can even provide more fine grained information such as *where* and *when* memory was leaked, as discussed in the following section. Other forms of memory errors, such as dangling pointers and double frees, are not considered by DSI. These errors will most likely lead to undefined program behaviour resulting in program crashes or are a concern for program security. The related work on DDS tools [49, 69, 74] does not consider memory leaks; hence the tools are exposed to precision loss.

4.2.1 Memory leak detection tools

One can think of choosing an existing memory leak analysis that could be either static, e.g., [55,71,109], or dynamic, e.g.,[6,21,22,24,35,58,83,103,110]. However, most of these tools do not report exactly *when* memory is leaked, i.e., in terms of DSI's event trace the exact event at which a leak occurs. This information is crucial for DSI to execute its analysis up to the point when memory is lost. Static analysis is not capable of delivering such information and therefore is no option for DSI. It is surprising, though that most dynamic analyses also do not offer this information. Even popular tools such as Valgrind [35] and Dr.Memory [6] only report *that* memory was leaked and *where* the leaked memory was *allocated*.

Guided by the need of a software developer to fix bugs fast and because the point of allocation and the cause of the leak can be far apart [58], attempts are made in the literature to find the *location* that is causing the leak. For example, Skiff [103] reports the last valid reference to memory, which narrows down the search space for the leak but still does not report the exact location and, consequently, also cannot do so when the reference to memory is lost. Purify [70], LEAKPOINT [58] and Omega [21] report the exact source code location of the leak, though they still do not report when the leak occurs. To the best of the author's knowledge, the only tool that claims to report when and where memory is lost is Insure++ [22], which instruments source code, i.e., it is not suitable for binaries (cf. Pt. II).

In summary, the available tools either do not expose the required information as they are not trace based as is DSI, or they rely on source code which is not an option, when dealing with binaries. Without suitable tool support being available and given that all information required for leak detection is already available in DSI's execution trace, such as memory (de-)allocations, VLSs and pointer writes, DSI is required and capable of executing its own memory leak detection as described in the following section. As a byproduct, DSI implicitly closes the gap between dynamic data structure detection tools [49, 69, 74] and memory leak detection and debugging tools [21, 22, 58].

4.2.2 Memory leak detection algorithms

As DSI cannot rely on available memory leak detection tools, existing algorithms for memory leak detection might still be suited for DSI's purposes. Prominent examples of such algorithms are reference counting [83] and reachability tests such as the mark-and-sweep and stop-and-copy algorithms [112]. Reference counting is unsuitable for DSI, because it does not handle leaked cyclic structures [41], which are supported by DSI.

The mark-and-sweep and stop-and-copy [112] algorithms used by garbage collectors could in principle be employed by DSI. Due to the periodic nature of garbage collection, where not every instruction is tracked, the algorithms are required to perform a reachability test on memory, i.e., DSI's PTGs, by starting out from root nodes that are guaranteed to be reachable at the point of the analysis, e.g., variables on the stack and global memory. The mark-and-sweep algorithm marks reachable objects in a first pass and subsequently performs a second pass through all live memory to remove all unmarked memory chunks. The stop-and-copy algorithm instead avoids the sweep phase by operating on two memory areas: (i) the current memory to analyse, and (ii) a memory area into which reachable memory is copied. Therefore, all memory from (i) can be deleted after the reachability phase, as live objects now reside in (ii).

Both the mark-and-sweep and stop-and-copy algorithms are applicable to DSI, as DSI's event trace provides all information required by these algorithms. However, DSI would need to execute either of the algorithms on every memory changing event which then exhaustively explores all memory. This would be significantly more often as with the infrequent executions during garbage collection and seems an unnecessary overhead, given that DSI is operating on a contiguous event trace, i.e., exactly one memory changing event occurs between events E_i and E_{i+1}. The affected addresses of an event are also known; therefore, the changes to the PTG can be located to originate from those addresses. This allows us to turn the reachability test around and try to find a path to a root element starting from the altered memory. The memory is considered leaked only if this fails. In this case the algorithm recursively checks all outgoing pointers of the leaked memory to find if further unreachable memory in the graph exists. The algorithm is detailed in the following section.

The advantage of DSI's memory leak detection therefore is that only the linkages of the particular DDS need to be investigated and not the complete live memory reachable by all root nodes. The resulting best case is that a root node is found on the first incoming pointer of the memory that needs to be checked. In the worst case, the complete PTG corresponding to the DDS in which the memory change occurred needs to be investigated. However, this still should be a subset of all allocated memory. This optimization for the memory leak detection comes at the cost of DSI's fine grained analysis that is quite heavy-weight and, as a reminder, performed offline after program termination. However, approaches such as [90] offload their memory leak detection onto a shadow process that is executed on a dedicated core of a multicore system. DSI could well be changed to a similar architecture, which would enable DSI for online monitoring. This topic seems quite promising for future work: the rich information that DSI provides, i.e., the exact location and time a leak occurs, in addition to DSI's ability to replay the program execution might help a lot for understanding the cause of a memory leak. A case study has been conducted within this dissertation that demonstrates the debugging capabilities of DSI in the case of memory leaks in Sec. 4.4.

A limitation of DSI's memory leak detection approach is pointer arithmetic. Changes to a pointer value can leave objects without an explicit reference, as is the case when XOR-ing pointers as used, e.g., with a XOR linked list [44, 76]. These scenarios are out of scope for DSI and, therefore, are also not considered by the memory leak detection, so that these lead to false positives.

4.2.3 DSI's memory leak detection algorithm

This section describes the devised memory leak detection algorithm for DSI and how it is integrated into the AME algorithm of Sec. 4.1. In the Sec. 4.1, only the incoming and outgoing edges to one freed memory chunk have been calculated and linearized as AMEs. Now, the memory leak detection algorithm can result in a set of memory chunks that are unreachable and for which all outgoing pointers need to be cut.

The leak algorithm detection is based on a reachability problem inside a PTG G_i for some time step i. Each vertex in the set of vertices ($v \in \mathcal{V}$) of each PTG in the set of all PTGs of the program execution ($G \in \mathcal{G}$) is annotated as being either heap or stack allocated memory. The algorithm tries to find a path in the PTG of time step i (G_i) from a vertex v to a stack allocated root v_r. The algorithm considers pointer writes ($ptrWrite$s) in addition to $free$ and vls events in line 7 of Alg. 1. This is required, because every pointer write can cut the last reference to allocated memory, resulting in unreachable memory. The memory leak detection operates on three global variables as seen in Alg. 1: (i) $\mathcal{Q}$ contains all vertices that need to be checked for a leak (line 6). The function VERTEX fetches the vertex by looking up the vertex inside of $\mathcal{V}$ with the help of the vertex start address stored inside of the current event (E_i.sAddr); (ii) $\mathcal{P}$ records visited nodes to avoid cycles (line 2), and (iii) $\mathcal{L}$ stores all leaked vertices (line 4). The algorithm starts out with calling function INITCHECKLEAK (line 8), which is shown in Alg. 2.

```
// Bread crumb for all processed vertices following the outgoing nodes
global $\mathcal{P} \leftarrow \{\}$
// All leaked vertices
global $\mathcal{L} \leftarrow \{\}$
// Vertices that still need to be processed (queue)
global $\mathcal{Q} \leftarrow \{\}$
for each $E_i \in E \mid E_i.kind = (ptrWrite|free|vls)$
  InitCheckLeak($E_i$)
  checkLeak()
  linearizeAsEvents(
      calculateEdgesIntoLeaked($E_i$, $i$) $\cup$
      calculateEdgesOutgoingFromLeaked($i$), $E_i, i$)
end
```

Algorithm 1: Main part of memory leak detection.

Function InitCheckLeak first resets the global variables (i)–(iii) and initializes them according to the event: (a) in case of a $free$ and vls event, all vertices reachable from the freed memory are calculated and enqueued into the process queue $\mathcal{Q}$ (line 6) as they are suspect to being also leaked. Additionally, the freed element itself is placed into the leaked set $\mathcal{L}$ (line 8), in order to indicate that this node is definitively lost, and into the processed set $\mathcal{P}$ to avoid cycles (line 10). (b) In case of a $ptrWrite$, the memory, to which the pointer previously pointed to, is put into the process queue $\mathcal{Q}$ (line 18). The pointer write event is only considered, if (i) the pointer write event did not relocate the pointer within same target vertex, i.e., the vertex produced by the previous pointer target (tAddr) is different from the current pointer target (tAddr') (line 15); and (ii) the pointer was not a vertex self reference before, i.e., the vertex holding the source address of the pointer (sAddr) and the vertex of the previous pointer target address (tAddr) are not identical (line 16). As a side note, the sAddr member of the event E_i is overloaded between *free, vls* and *ptrWrite* events.

With the initial set of vertices that require testing now being set up, the memory leak detection takes place in function checkLeak of Alg. 3, which processes the input queue $\mathcal{Q}$ of unprocessed vertices (lines 4 and 5) and then does the recursive reachability test by calling function reachabilityTest (line 8). Note that the reachability test operates on a local bread crumb, i.e., the empty set passed in as a second parameter, for cycle avoidance, i.e., different from the global $\mathcal{P}$ that globally indicates that a particular vertex has already been leak checked. If the reachability test fails, the vertex is marked as such (line 10) and all vertices reachable by its outgoing edges are enqueued for the leak check (line 12).

The recursive reachability test in function reachabilityTest() in Alg. 4 keeps track of visited nodes first (line 3). Then it checks whether the current vertex is annotated as a root vertex (line 4), i.e., the positive base case of the recursion. If

```
function InitCheckLeak($E_i$)
  // Reset all global variables
  $\mathcal{P} \leftarrow \mathcal{L} \leftarrow \mathcal{Q} \leftarrow \{\}$
  if $E_i.kind = (free|vls)$
    // Mark elements reachable by freed element for leak detection
    enqueue(getOutgoingVertices(Vertex($E_i$.sAddr)), $\mathcal{Q}$)
    // Add the freed element to the leaked set
    $\mathcal{L} \leftarrow \mathcal{L} \cup \{$Vertex($E_i$.sAddr)$\}$
    // Add the freed element to the processed set
    $\mathcal{P} \leftarrow \{$Vertex($E_i$.sAddr)$\}$
  else // ptrWrite
    // No need to process:
    // 1) if pointer has been moved inside of same vertex
    // 2) if pointer has been a vertex self reference before
    if (Vertex($E_i$.tAddr') $\neq$ Vertex($E_i$.tAddr)
      $\wedge$Vertex($E_i$.sAddr) $\neq$ Vertex($E_i$.tAddr))
        // Mark the element for leak detection to which
        // the pointer previously pointed to
        enqueue(Vertex($E_i$.tAddr), $\mathcal{Q}$)
    end
  end
```

Algorithm 2: Calculation of the initial set of vertices on which the recursive memory leak detection algorithm operates.

this is not the case, all incoming vertices to the current vertex are selected that are not in the leaked set $\mathcal{L}$ and not in the local bread crumb set $\mathcal{P}_{\text{in}}$ (line 9). Then all incoming vertices are processed recursively. If no vertex is reachable the method returns that the current vertex is also unreachable, i.e., the negative base case.

Once checkLeak returns, all leaked vertices are stored in the global leaked set $\mathcal{L}$ which is now used to calculate the incoming and outgoing edges to and from the leaked set, as shown in Alg. 5 and Alg. 6. As an optimization an empty set for the incoming edges is returned in Alg. 5 in case of a pointer write that causes a leak. In this case all pointers from reachable memory into the leaked set must have already been cut, otherwise no leak would have occurred. The union of the incoming and outgoing edges is passed to function linearizeAsEvents in Alg. 7. The modified linearizeAsEvents also creates .artEvents for all edges incoming and outgoing to the freed/leaked memory, i.e., the memory leak detection is transparent for this part of the algorithm as the edges are calculated as previously discussed. In line 10 of Alg. 7 only in the case of $free$ and vls events the vertex is explicitly removed from the current PTG, i.e., the vertex set of the current time step $\mathcal{V}(t)$, and the leak set $\mathcal{L}$. The freed memory needs to be removed from the leaked set $\mathcal{L}$, as otherwise an artificial free event would be created though the original event E_i already is

```
function checkLeak
  while $v \leftarrow$ dequeue($\mathcal{Q}$)
    // Skip processed elements
    if $v \notin \mathcal{P}$
      $\mathcal{P} \leftarrow \mathcal{P} \cup \{v\}$
      // Check if vertex is reachable.
      // Note the empty bread crumb set {}
      if !reachabilityTest($v, \{\}$)
        // Vertex is not reachable
        $\mathcal{L} \leftarrow \mathcal{L} \cup \{v\}$
        // Store connected vertices for leak detection
        enqueue(getOutgoingVertices($v$), $\mathcal{Q}$)
      end
    end
  end
```

Algorithm 3: Memory leak detection on the outgoing edges.

a free event. In contrast, in case of a memory leak due to a $ptrWrite$ event, all vertices are implicitly removed by the leaked set $\mathcal{L}$.

Complexity wise, the computational costs heavily depend on the layout of the DDS, especially the number of the vertices and edges. As all edges of the graph need to be evaluated in the worst case, the reachability check is a combinatorial problem. As an example we consider a cyclic list, where on average each vertex has o outgoing edges. The number of vertices is n. Therefore, the number of possible pathes through the cyclic graph is o^n. In a cyclic graph one needs to follow n edges on each of the possible pathes to arrive at the the initial vertex where the search started. Without further optimizations the reachability test needs to be conducted for each of the n vertices in the graph, resulting in $O(o^n n^2)$. Thus the computational costs can become expensive, depending on the size of o or n.

However, our benchmark did not suffer from significant high computational costs for the memory leak detection algorithm, even though it is not optimized, yet. This is not surprising, as for most real world DDS the linkages are chosen to cut the search space of the stored data, e.g., a BT or a SL. Such implementations are also beneficial for the memory leak detection algorithm, as not all of the vertices of the DDS need to be inspected. Consider cutting the entry pointer to a BT as an example. When computing the reachability of a node of the BT, the memory leak detection algorithm will only iterate ”upstream” towards the root node, but will never branch into a ”downstream” part of the BT.

If the need arises in the future, the algorithm can be optimized to collect both a set for vertices which are already found to be leaked and one for vertices that are found to be reachable during the exploration of the graph. Those sets can be used to cut the search short in either way. Another possibility is to implement

```
function reachabilityTest($v, \mathcal{P}_{\text{in}}$)
  // Bread crumb to avoid cycles
  $\mathcal{P}_{\text{in}} \leftarrow \mathcal{P}_{\text{in}} \cup \{v\}$
  if isStaticMemory($v$)
    // Vertex is reachable
    return true
  end
  // Fetch incoming pointers for reachability test
  $\mathcal{V}_{\text{in}} \leftarrow \{v_i \in \text{getIncomingVertices}(v) \mid v_i \notin \mathcal{L} \wedge v_i \notin \mathcal{P}_{\text{in}}\}$
  for each $v_i \in \mathcal{V}_{\text{in}}$
    // Recurse on surrounding vertices
    // Stop on the first reachable vertex
    if reachabilityTest($v_i, \mathcal{P}_{\text{in}}$)
      return true
    end
  end
  // Vertex is not reachable
  return false
```

Algorithm 4: Search for a static vertex on incoming edges.

```
function calculateEdgesIntoLeaked($E_i, i$)
  if $E_i.kind = (free|vls)$
    return $\{(v_s, \_, v_t, \_) \in \mathcal{E}(t) \mid v_s \notin \mathcal{L} \wedge v_t \in \mathcal{L}\}$
  else
    // Pointer write: All incoming edges must
    // already have been cut
    return $\{\}$
  end
```

Algorithm 5: Calculation of all edges that point into the leaked memory and that need to be removed.

a breadth first search, i.e., check through all immediately neighbouring vertices first, which point into the potentially leaked vertex, in order to find, e.g., a stack allocated pointer. This aims for the best case, where the reachability is verified by only checking one incoming pointer and the corresponding vertex.

4.2.4 CIL: Creation of temporary pointers

While implementing our memory leak detection, a severe design decision of the CIL framework was observed. CIL splits direct assignments of function calls to variables into two statements if the types of the assignment differ, i.e., when a cast is required. To illustrate the problem the C code as given by the programmer is shown in Fig. 4.2 and the resulting code generated by CIL is shown in Fig. 4.3. The initial code shown in Fig. 4.2 allocates an integer on the heap with `malloc`. A

```
function calculateEdgesOutgoingFromLeaked(i)
    return $\{(v_s, \_, v_t, \_) \in \mathcal{E}(t) \mid v_s \in \mathcal{L} \land v_t \notin \mathcal{L}\}$
```

Algorithm 6: Calculation of all edges that point outwards from the leaked memory.

```
function linearizeAsEvents($\mathcal{E}, E_i, t$)
    for each $e \in \mathcal{E}$
        $E_i.artEvents \leftarrow E_i.artEvents$ ++ $ArtificialUndefEvent(e.$sAddr$)$
        // Remove the edge from the PTG
        $\mathcal{E}(t) \leftarrow \mathcal{E}(t) - \{e\}$
    end
    // In case of free:
    // 1) Remove the freed vertex from the PTG
    // 2) Remove the freed vertex from the leaked set
    if $E_i.kind = (free|vls)$
        $\mathcal{V}(t) \leftarrow \mathcal{V}(t) - \{$Vertex$(E_i.$sAddr$)\}$
        $\mathcal{L} \leftarrow \mathcal{L} - \{$Vertex$(E_i.$sAddr$)\}$
    end
    // Remove all vertices of the leaked set from the PTG
    // and record an artificial free event
    for each $v \in \mathcal{L}$
        $\mathcal{V}(t) \leftarrow \mathcal{V}(t) - \{v\}$
        $E_i.artEvents \leftarrow E_i.artEvents + +ArtificialFreeEvent(v.$bAddr$)$
    end
```

Algorithm 7: Creation of artificial memory events to cut incoming and outgoing edges to a freed/leaked memory and to remove leaked vertices.

cast is required, because the return value of `malloc` is `void*` and the type of the assigned variable is `int*`. The resulting CIL code shown in Fig. 4.3 thus generates a temporary variable `tmp` to which the result of `malloc` is assigned. Next, the assignment takes place, including the pointer cast. Note that `tmp` is never reset to, e.g., `NULL`.

The CIL source code contains the variable `doCollapseCallCast`[1] which can be only manipulated directly in CIL's source code. Setting the variable should change CIL's behaviour to a direct assignment without an intermediate pointer. However, setting the variable does not trigger the desired effect, at least for CIL version 1.7.3 employed by us. This problem is also reported on a CIL mailing list [2] for CIL version 1.3.6. The suggested workaround in the mailing list is apparently not implemented in CIL version 1.7.3. The discussed patch in the mailing list did also not work when patched manually by the author of this dissertation.

With full access to the (instrumented) source code, this situation might still be solvable by preventing the instrumentation of these void pointers; however, when dealing with just the binary produced when compiling with CIL, false negatives

[1] Variable located in file `src/frontc/cabs2cil.ml` in CIL's repository [3]

```
#include<stdlib.h>

void main(void) {
    int *ptr = malloc(sizeof(int));
}
```

Figure 4.2: C code performing an allocation of an integer on the heap.

```
/* Generated by CIL v. 1.7.3 */
/* print_CIL_Input is true */

typedef unsigned long size_t;
extern  __attribute__((__nothrow__)) void *( ←
   __attribute__((__leaf__)) malloc)(size_t __size )  ←
   __attribute__((__malloc__)) ;
void main(void)
{
  int *ptr ;
  void *tmp ;

  {
  tmp = malloc(sizeof(int ));
  ptr = (int *)tmp;
  return;
}
}
```

Figure 4.3: CIL: Artificial temporary pointer.

are inevitable. Therefore, CIL should either avoid the usage of additional void pointers or reset them to NULL immediately after use, as every dynamic memory leak detection algorithm should suffer from CILs artificially created pointers.

For memory leak detection, CIL's artificial void pointer thus imposes a problem, because it is inaccessible by the programmer but still is a valid reference to the memory. This leads to false negatives for the memory leak detection if the artificial pointer is the only reference to memory. Especially when dealing with DLLs, memory leak can thus be missed, as both directions of the memory graph are usually accessible in a DLL by following the `next` and `prev` pointers. Thus, if one element of the DLL has an artificially created pointer by CIL attached, a potential memory leak is missed. This problem is not only limited to DSI, but to any memory leak detection algorithm that follows pointer chains to reach stack allocated memory. Hence, the design decision taken by CIL is severe.

4.3 Benchmark

In this section the benchmark for the AME and memory leak detection algorithms is discussed, which consists of 6 synthetic self written examples (**leak[0-5]**), 1 memory leak example from [18] (**leak6**), and 2 examples (**leak7** and **leak8**) taken from threads on stackoverflow.com [27,28]. Those examples are specific for benchmarking the AME and memory leak detection, and are not used in other benchmarks of this dissertation. Additionally, some selected examples from the benchmarks used to test DSI are shown here, in order to testify that the previously described scenarios exist and must be handled. These examples are from the literature (**lit[1,5,8]**) [23,65,72], real world (**rosetta-dll**) [5] and a text book (**tb4**) [108].

All benchmarks are listed in Tab. 4.1. The (**ID**) gives a unique name for the examples which is consistent throughout this dissertation. The column (**strands**) indicates, whether strands are created within the example, i.e., if AMEs will be consumed by the strand creation phase. The columns (**free**), (**vls**) and (**ptrWrite**) are used to list the events which occur in the example that trigger the AME creation and memory leak detection. Column (**leak**) shows if a leak occurs or not. The columns (**incoming ptr**) and (**outgoing ptr**) indicate if there are incoming and, resp., outgoing pointers to and from the freed/leaked memory area, because some of the examples only test the AMEs and some test the memory leak detection together with AMEs creation. The column (**cyclic**) states, whether the pointer connections are cyclic. The remaining columns give information, whether the test succeeded in detecting a memory leak (**leak detected**), whether all pointers incoming and outgoing to freed or leaked memory were detected and cut (**all in/out pointers cut**), whether all freed/leaked vertices were removed from the graph (**all vertices removed**) and if there were any **false positives** or **false negatives**.

leak0, leak1 and tb4. Both examples consist of an SLL as the main data structure and both trigger a memory leak by setting the `next` pointer of one element inside of the SLL to `NULL`, while no stack based references exist for the elements in the cut off list segment. In case of **leak0**, the last two elements of the list are cut off. No pointer exists that points from the leaked area back into reachable memory. Example **leak1** has an additional SLL child element hanging off from the leaked element, where one element of the child SLL points back into reachable memory, i.e., has an outgoing pointer. This pointer forms a cycle, and thus tests that our algorithm terminates. For both examples, the leaks are detected, all incoming and outgoing pointers are cut, and all leaked vertices are removed from the graph. Another SLL implementation is taken from a text book implementation (**tb4**) [108], which does not expose a memory leak, in order to vary the implementation techniques for SLLs. Again, the test passed by detecting the freed memory together with the incoming and outgoing pointers.

Table 4.1: Results for artificial memory event and memory leak detection benchmark

ID	strands	free	VLS	ptrWrite	leak	incoming ptr	outgoing ptr	cyclic	leak detected	all in/out pointers cut	all vertices removed	false positives	false negatives
leak0	✓	✓	✗	✓	✓	✓	✗	✗	✓	✓	✓	✗	✗
leak1	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
leak2	✓	✓	✗	✗	✓	✓	✓	✗	✓	✓	✓	✗	✗
leak3	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗
leak4	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✗
leak5	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗
leak6 [18]	✗	✗	✗	✓	✓	✓	✗	✗	✓	✓	✓	✗	✗
leak7 [27]	✗	✓	✗	✗	✓	✓	✗	✗	✓	✓	✓	✗	✗
leak8 [28]	✗	✗	✗	✓	✓	✓	✓	✗	✓	✓	✓	✗	✗
lit1 [23]	✓	✓	✗	✗	✗	✓	✓	✓	✗	✓	✓	✗	✗
lit5 [23]	✓	✓	✗	✗	✗	✓	✓	✓	✗	✓	✓	✗	✗
lit8 [23]	✓	✓	✗	✗	✗	✓	✓	✓	✗	✓	✓	✗	✗
rosetta-dll [5]	✓	✓	✗	✗	✗	✓	✓	✓	✗	✓	✓	✗	✗
tb4 [108]	✓	✓	✗	✗	✗	✓	✓	✗	✗	✓	✓	✗	✗

leak2 and leak3. Both examples are identical to **leak0** and **leak1**, respectively, with the only difference that the leak is now caused by a free instead of overwriting the element. Again, all pointers that need to be cut are detected, and the freed and leaked vertices are removed from the graph.

leak4. This example consists of an SLL and tests various VLS situations by inserting two static SLL nodes into the list. The last static SLL node is again connected to an allocated SLL node. Here, the specifics of the CIL framework of setting an explicit pointer to the allocated memory is utilized to check that the memory leak detection is triggered when this pointer goes out of scope. The test consists of three VLSs events, two for the SLL nodes and one for the artificial pointer inserted by CIL. At first, the last element allocated is explicitly cut off by setting the corresponding `next` pointer to `NULL`. However, this does not result in a leak, because there exists the artificial CIL pointer as discussed previously. This can be considered a false negative, due to the CIL pointer being out of reach of the programmer, which makes the memory effectively lost. However, the memory leak detection al-

```
typedef struct sample_help {
    int *value;
    void **pointers;
} *sample, sample_node;

sample foo(void)
{
    sample ABC=NULL;
    sample XYZ=NULL;
    sample kanchi = NULL;

    ABC = malloc(sizeof(sample_node));
    XYZ = malloc(sizeof(sample_node));
    ABC->pointers = malloc(5*sizeof(void *));
    XYZ->pointers = malloc(5*sizeof(void *));
    ABC->value = malloc(5*sizeof(int));
    XYZ->value = malloc(5*sizeof(int));

    ABC->value[0] = 10;
    ABC->value[1] = 20;

    XYZ->pointers[0] = ABC;
    kanchi = XYZ->pointers[0];

    printf("::::%d\n",XYZ->pointers[0]);
    printf("kanchi1:::::%d\n",kanchi->value[0]);

    return XYZ;
}
```

Figure 4.4: Source code excerpt of `leak7` [27].

gorithm has no information about this situation, and therefore correctly reports no leak. The leak detection will again be triggered as soon as the CIL pointer gets out of scope.

All three static nodes are teared down when the stack frame is removed. At first, the two static SLL nodes are removed in the order of creation, i.e., the first inserted element is removed first. This tests that the algorithm correctly identifies the static successor SLL node, i.e., it immediately stops after removing the first node and cutting the corresponding `next` pointer. Subsequently, the second static SLL node is removed, which has no further connections; thus, the algorithm terminates immediately. Finally, the artificial CIL pointer is removed, which still holds the reference to the allocated memory leading to a leak. The leak is detected, demonstrating that the memory leak detection also works in case of VLS events.

leak5. With this example, the cycle detection is tested more extensively with a DLL, as all nodes in the list expose a cycle with their predecessor and/or successor. Note that no CDLL is used as the cyclic nature prevents a leak. The DLL tests both cycle detection algorithms, i.e., the global cyclic test that keeps track of nodes that have already been classified as leaked or not, and the local cyclic test that is used to prevent cycles in finding the path to the root nodes, as described in Sec. 4.2.

leak6 and leak7. Both examples do not create strands. Example leak6 was taken from [18] and simply allocates memory in a loop where the receiving pointer is constantly overwritten. **leak7** was taken from [27] and modified to trigger a leak via a call to `free`. Both examples are also used to rule out bias in the test set, as they are taken from the Internet. Especially the latter example makes use of arbitrary `mallocs` as seen in Fig. 4.4. The returned reference `XYZ` is simply freed, without freeing the references to `ABC`, `value` and `pointers`. Again, the leak is detected and all leaked vertices are removed from the PTG.

lit1, lit5, lit8 and rosetta-dll. All examples are DLLs. Examples **lit1**, **lit5** and **lit8** are taken from the shape analysis literature [65, 72], while the **rosetta-dll** example is taken from [5]. Example **lit1** exposes no memory leak, but it has an extensive amount of incoming and outgoing pointers to the freed memory because two DLLs run in parallel through each node, i.e., two `next` and `previous` pointers exist per node. Again, the cyclic nature of the DLL tests, whether the algorithm terminates, but in this case with the presence of multiple cycles between nodes. The remaining dll implementations, i.e., **lit5** and **rosetta-dll**, are used to apply the algorithms against different DLL implementations techniques. In both cases, no false positives and false negatives are created, all edges are cut and all vertices are removed from the points-to graph. Both examples also expose the already stated fact that the programmer typically does not explicitly cut any references of the freed memory, i.e., making the handling of those implicitly cut pointer connections mandatory for DSI. The same is true for **lit8** that is employed to test a LKL implementation, i.e., a CDLL. The points-to graph created by the example is quite challenging, because it is comprised of a CDLL parent element, with two CDLL child elements each. Interestingly, this code does not explicitly reset pointers *into* freed memory, which highlights that even a proper cleanup of pointers that are exposed to potential use-after-free cannot be expected.

4.4 Case study

In this section, we discuss example **leak8** in a case study like fashion, as it demonstrates the debugging capabilities of DSI with regards to memory leak detection. The example is taken from a real world thread posted on stackoverflow.com [28], where a custom SLL implementation is given in which a memory leak has been identified by Valgrind as reported by the thread starter. Interestingly despite the reported leak by Valgrind, the cause of the leak could not be found by the thread starter. When using DSI the leak can be precisely correlated with the line `temp->next = ptr;`, see line 10 in Fig. 4.5, and even the precise time step within the concrete execution of the program is identifiable, when the leak occurs. Addi-

```
 1 node *deletemultiples(node *head, int a){
 2   node *ptr = head, *temp = head;
 3   while (ptr != NULL) {
 4       if(ptr->info % a > 0){
 5          ptr = ptr->next;
 6          temp = temp->next;
 7       }
 8       else{
 9          ptr = ptr->next;
10          temp->next = ptr;
11       }
12    }
13   return(head);
14 }
```

Figure 4.5: Source code excerpt of `leak8` [28].

tionally, DSI can play back the steps prior to the leak, which helps the developers in understanding the problem.

This can be seen throughout Figs. 4.6– 4.12, which show the steps leading to the leak within method `deletemultiples` of Fig. 4.5. The figures show stack variables in blue, the special purpose target `UNDEF` indicating that a pointer has an undefined value, and heap allocated nodes in orange. The method `deletemultiples` has two parameters: (i) the `head` pointer, which points to the head of the SLL and (ii) the integer `a`, which is used to select if a node of the SLL should be skipped or deleted by computing the modulus on the `info` property of each SLL node (line 4). Fig. 4.6 shows the state after initializing the two variables `ptr` and `temp` to the head of the list (line 2), followed by iterating variable `ptr` to the next element (line 9) in Fig. 4.7 and setting `temp->next = ptr;` (line 10) in Fig. 4.8. The last event is interesting: it executes line 10 that actually exposes the leak, but not within the current iteration, i.e., DSI's timing information is valuable in narrowing down the situation of the leak. In the following two events, both `ptr` and `temp` are iterated one element further (lines 5 and 6) as seen in Fig. 4.9 and Fig. 4.10, resp. The following event shown in Fig. 4.11 now is the predecessor step of the actual leak, where `ptr` is again iterated (line 9), leaving vertex 17 with only the incoming pointer of the SLL. The subsequent execution of line 10 now unlinks vertex 17 from the SLL, resulting in the leak of the vertex. The solution for this problem is to perform a `free` on `temp->next` prior to the assignment of the variable `ptr` to it.

This stepwise debugging demonstrates that DSI's fine grained analysis does not only help in detecting a data structure, but also acts as a debugging tool for memory leaks. DSI has a higher precision than other memory leak detection tools [6, 21, 24, 35] with regards to when and where exactly a leak occurs. Tools like [22] that supposedly also provide the time step when the leak occurs, still do not record a trace to replay the execution and visualize the PTG as is possible with DSI. The precision of DSI comes with the fine grained trace recording, where each memory manipulating event gets recorded. This of course makes DSI's analysis more resource consuming in terms of memory and runtime consumption.

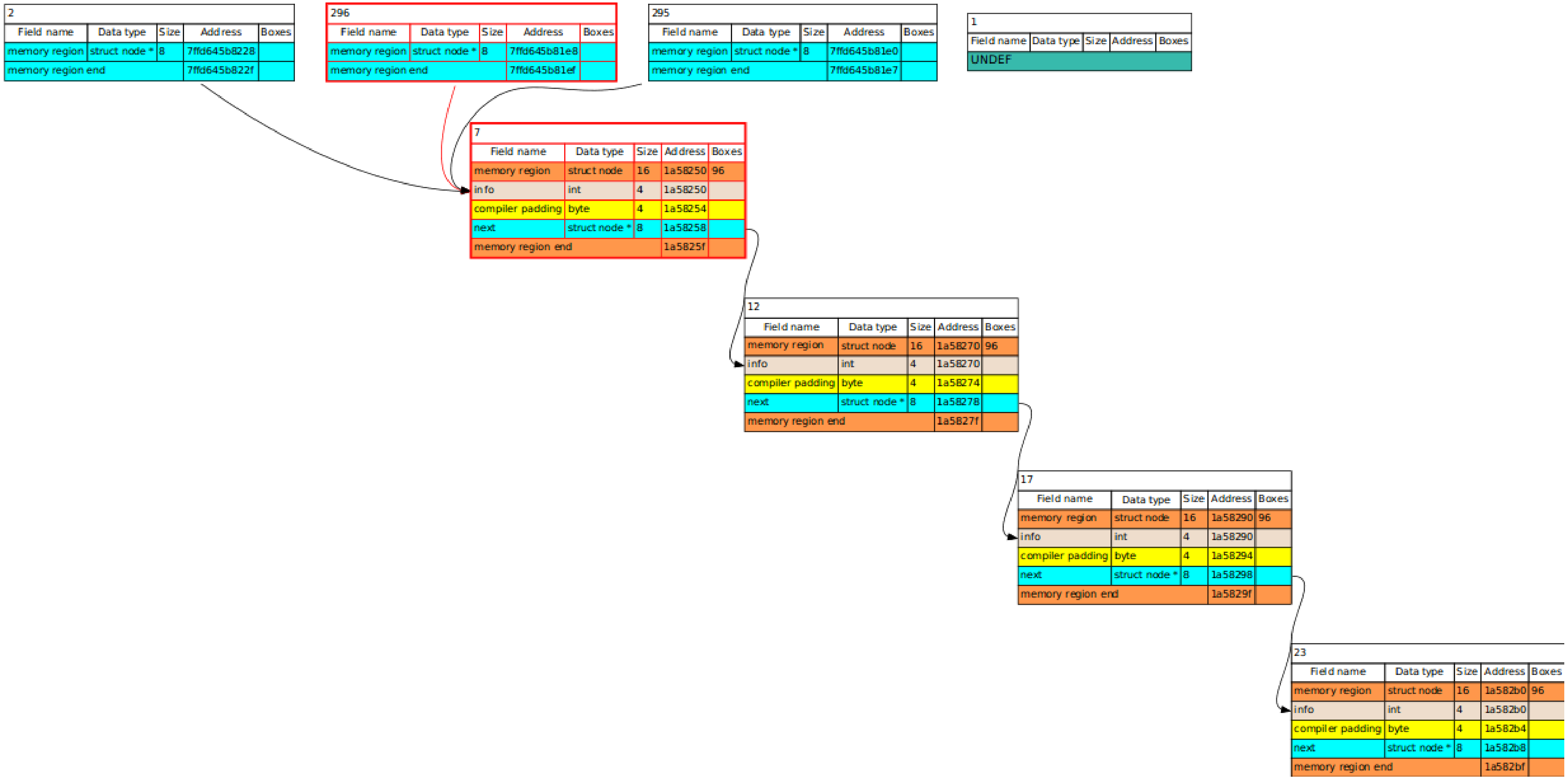

2
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b8228
memory region end | 7ffd645b822f
296
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b81e8
memory region end | 7ffd645b81ef
295
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b81e0
memory region end | 7ffd645b81e7
1
Field name | Data type | Size | Address | Boxes
UNDEF
7
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58250 | 96
info | int | 4 | 1a58250
compiler padding | byte | 4 | 1a58254
next | struct node * | 8 | 1a58258
memory region end | 1a5825f
12
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58270 | 96
info | int | 4 | 1a58270
compiler padding | byte | 4 | 1a58274
next | struct node * | 8 | 1a58278
memory region end | 1a5827f
17
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58290 | 96
info | int | 4 | 1a58290
compiler padding | byte | 4 | 1a58294
next | struct node * | 8 | 1a58298
memory region end | 1a5829f
23
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a582b0 | 96
info | int | 4 | 1a582b0
compiler padding | byte | 4 | 1a582b4
next | struct node * | 8 | 1a582b8
memory region end | 1a582bf

Figure 4.6: stackoverflow.com example, step 6826.

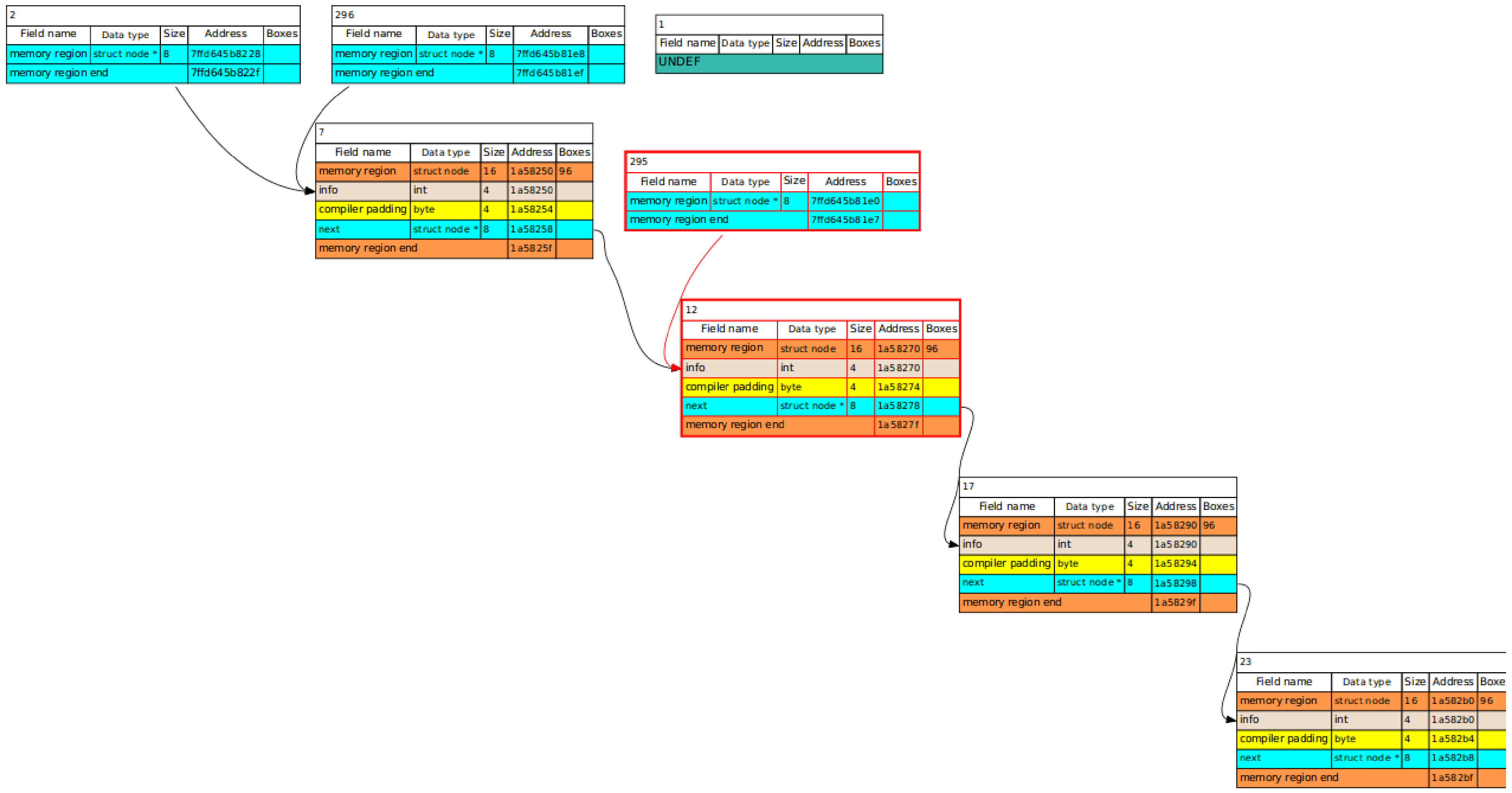

2
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b8228
memory region end | 7ffd645b822f
296
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b81e8
memory region end | 7ffd645b81ef
1
Field name | Data type | Size | Address | Boxes
UNDEF
7
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58250 | 96
info | int | 4 | 1a58250
compiler padding | byte | 4 | 1a58254
next | struct node * | 8 | 1a58258
memory region end | 1a5825f
295
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b81e0
memory region end | 7ffd645b81e7
12
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58270 | 96
info | int | 4 | 1a58270
compiler padding | byte | 4 | 1a58274
next | struct node * | 8 | 1a58278
memory region end | 1a5827f
17
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58290 | 96
info | int | 4 | 1a58290
compiler padding | byte | 4 | 1a58294
next | struct node * | 8 | 1a58298
memory region end | 1a5829f
23
Field name | Data type | Size | Address | Boxe
memory region | struct node | 16 | 1a582b0 | 96
info | int | 4 | 1a582b0
compiler padding | byte | 4 | 1a582b4
next | struct node * | 8 | 1a582b8
memory region end | 1a582bf

Figure 4.7: stackoverflow.com example, step 6830.

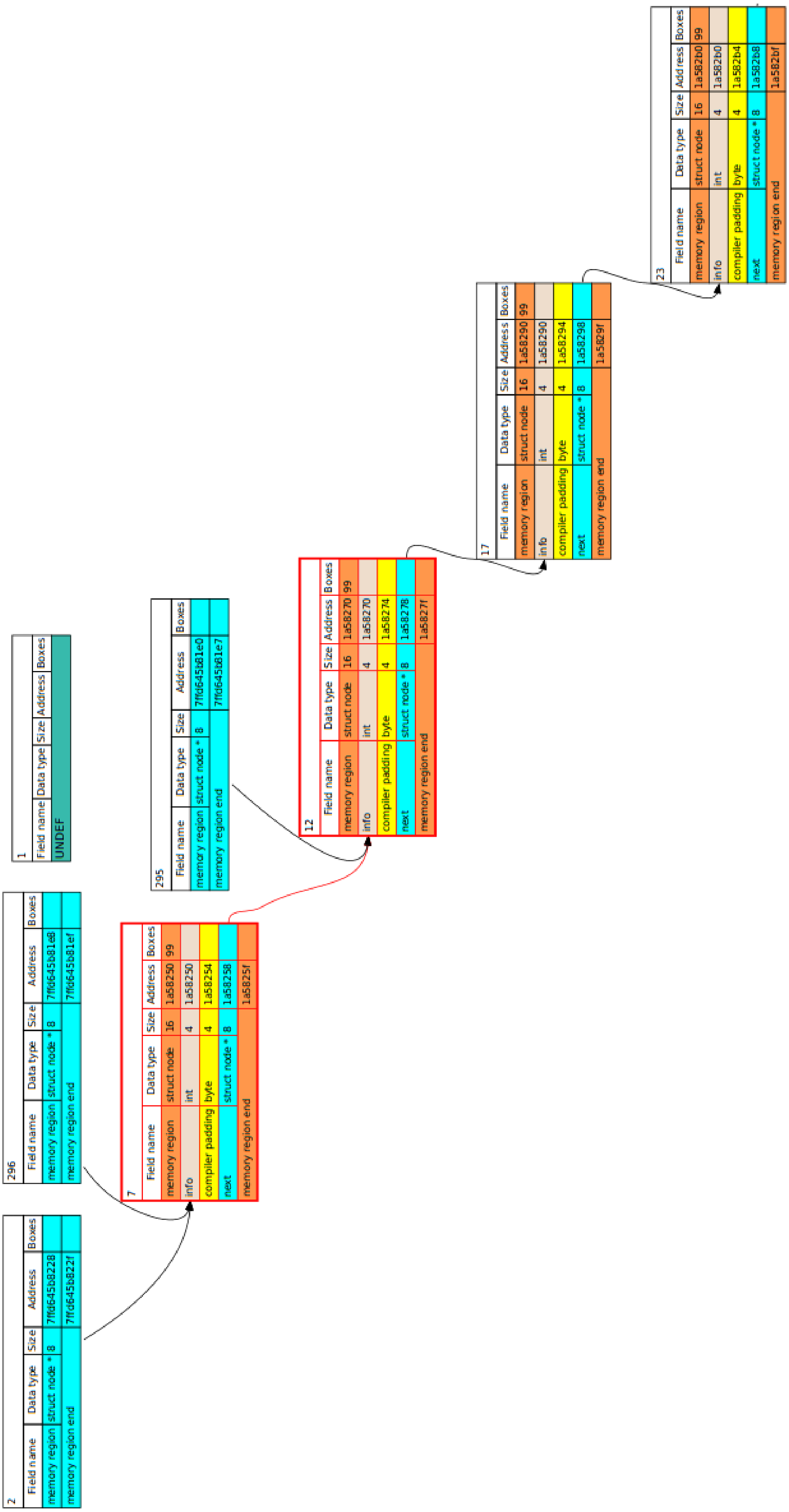
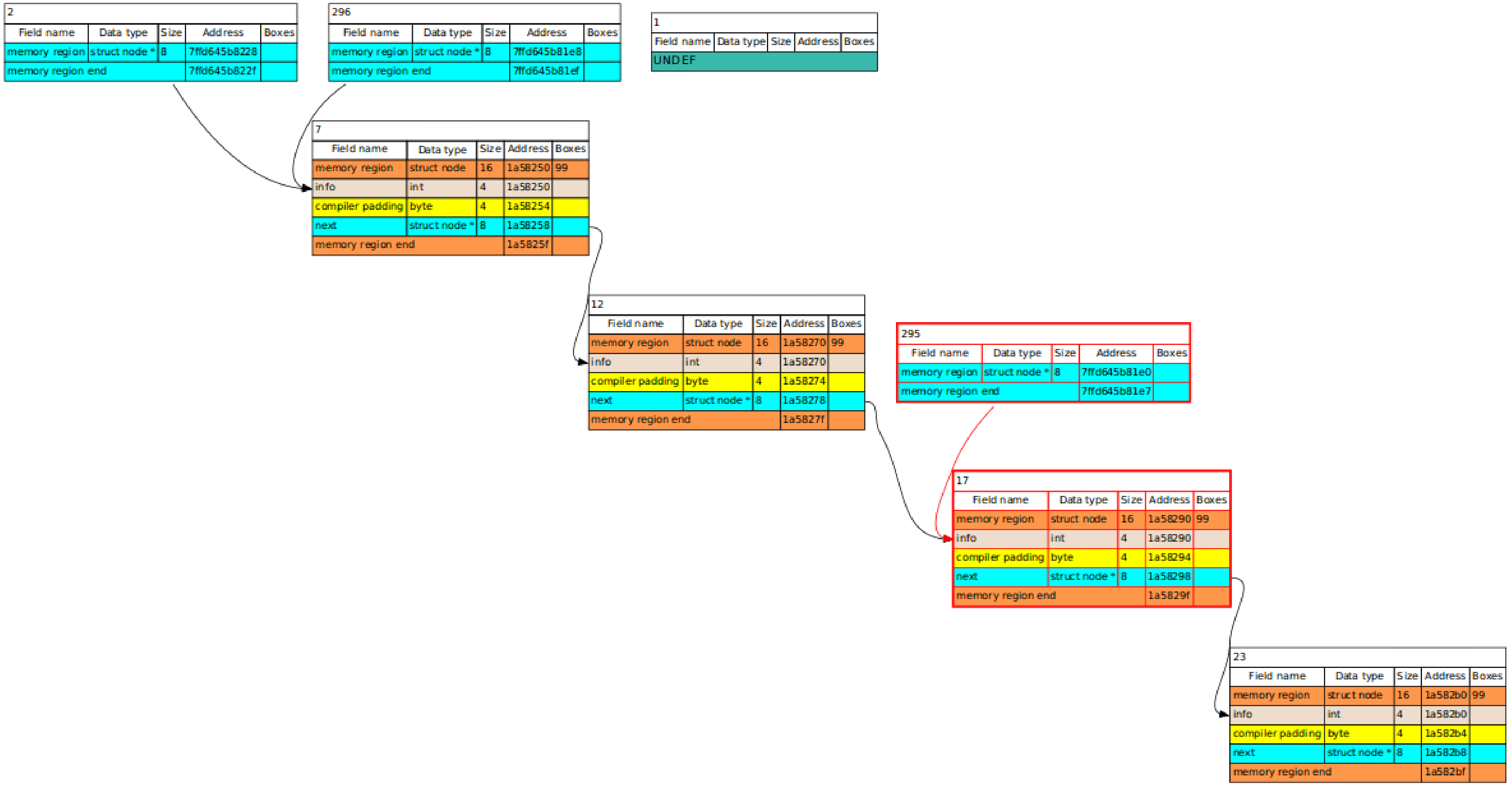

2
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b8228
memory region end | 7ffd645b822f
296
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b81e8
memory region end | 7ffd645b81ef
1
Field name | Data type | Size | Address | Boxes
UNDEF
295
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b81e0
memory region end | 7ffd645b81e7
7
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58250 | 99
info | int | 4 | 1a58250
compiler padding | byte | 4 | 1a58254
next | struct node * | 8 | 1a58258
memory region end | 1a5825f
12
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58270 | 99
info | int | 4 | 1a58270
compiler padding | byte | 4 | 1a58274
next | struct node * | 8 | 1a58278
memory region end | 1a5827f
17
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58290 | 99
info | int | 4 | 1a58290
compiler padding | byte | 4 | 1a58294
next | struct node * | 8 | 1a58298
memory region end | 1a5829f
23
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a582b0 | 99
info | int | 4 | 1a582b0
compiler padding | byte | 4 | 1a582b4
next | struct node * | 8 | 1a582b8
memory region end | 1a582bf

Figure 4.8: stackoverflow.com example, step 6831.



2

Field name	Data type	Size	Address	Boxes
memory region	struct node *	8	7ffd645b8228	
memory region end			7ffd645b822f	

296

Field name	Data type	Size	Address	Boxes
memory region	struct node *	8	7ffd645b81e8	
memory region end			7ffd645b81ef	

1

Field name	Data type	Size	Address	Boxes
UNDEF				

7

Field name	Data type	Size	Address	Boxes
memory region	struct node	16	1a58250	99
info	int	4	1a58250	
compiler padding	byte	4	1a58254	
next	struct node *	8	1a58258	
memory region end			1a5825f	

12

Field name	Data type	Size	Address	Boxes
memory region	struct node	16	1a58270	99
info	int	4	1a58270	
compiler padding	byte	4	1a58274	
next	struct node *	8	1a58278	
memory region end			1a5827f	

295

Field name	Data type	Size	Address	Boxes
memory region	struct node *	8	7ffd645b81e0	
memory region end			7ffd645b81e7	

17

Field name	Data type	Size	Address	Boxes
memory region	struct node	16	1a58290	99
info	int	4	1a58290	
compiler padding	byte	4	1a58294	
next	struct node *	8	1a58298	
memory region end			1a5829f	

23

Field name	Data type	Size	Address	Boxes
memory region	struct node	16	1a582b0	99
info	int	4	1a582b0	
compiler padding	byte	4	1a582b4	
next	struct node *	8	1a582b8	
memory region end			1a582bf	

Figure 4.9: stackoverflow.com example, step 6836.

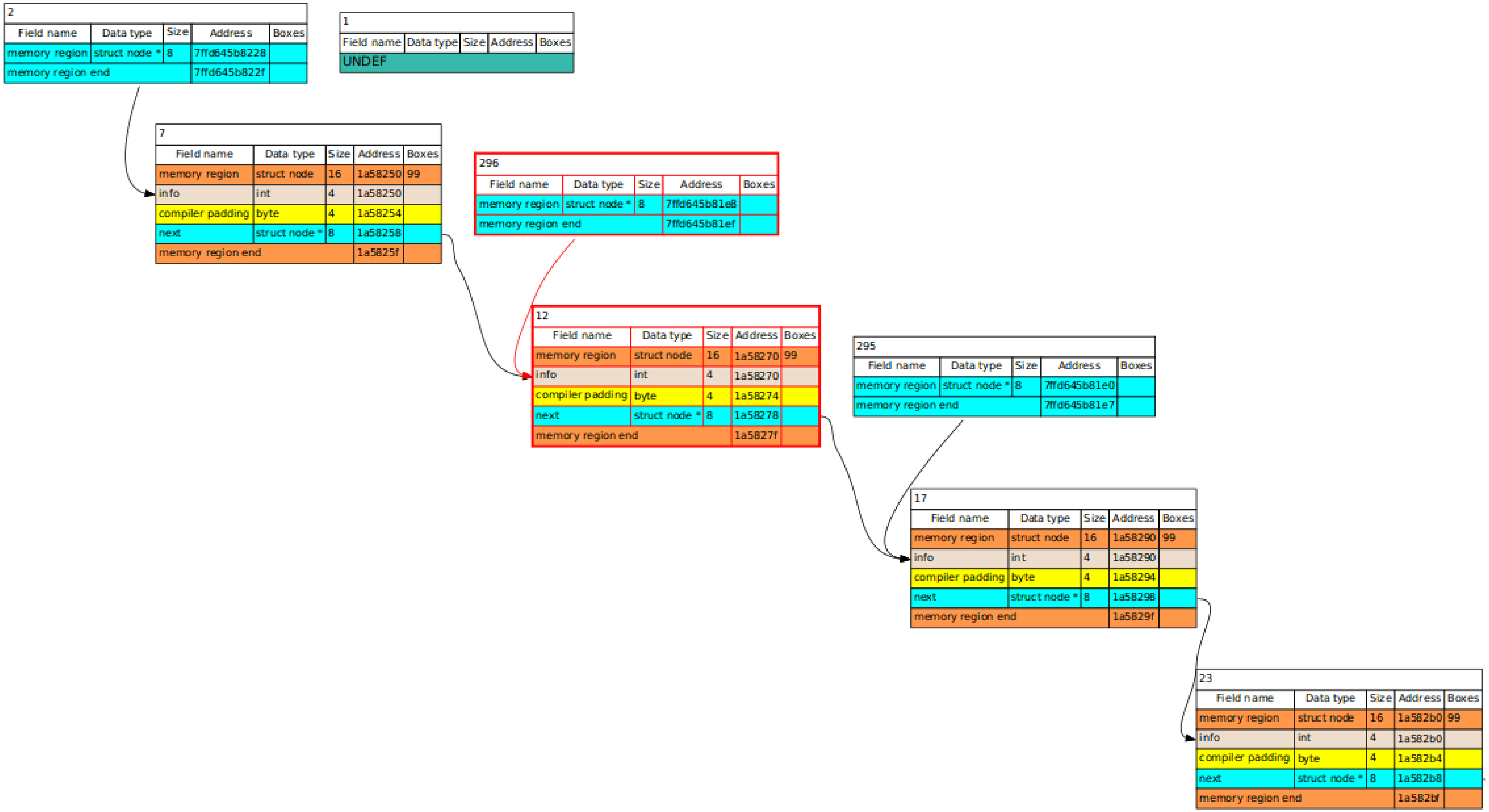

2
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b8228
memory region end | 7ffd645b822f
1
Field name | Data type | Size | Address | Boxes
UNDEF
7
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58250 | 99
info | int | 4 | 1a58250
compiler padding | byte | 4 | 1a58254
next | struct node * | 8 | 1a58258
memory region end | 1a5825f
296
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b81e8
memory region end | 7ffd645b81ef
12
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58270 | 99
info | int | 4 | 1a58270
compiler padding | byte | 4 | 1a58274
next | struct node * | 8 | 1a58278
memory region end | 1a5827f
295
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b81e0
memory region end | 7ffd645b81e7
17
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58290 | 99
info | int | 4 | 1a58290
compiler padding | byte | 4 | 1a58294
next | struct node * | 8 | 1a58298
memory region end | 1a5829f
23
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a582b0 | 99
info | int | 4 | 1a582b0
compiler padding | byte | 4 | 1a582b4
next | struct node * | 8 | 1a582b8
memory region end | 1a582bf

Figure 4.10: stackoverflow.com example, step 6837.

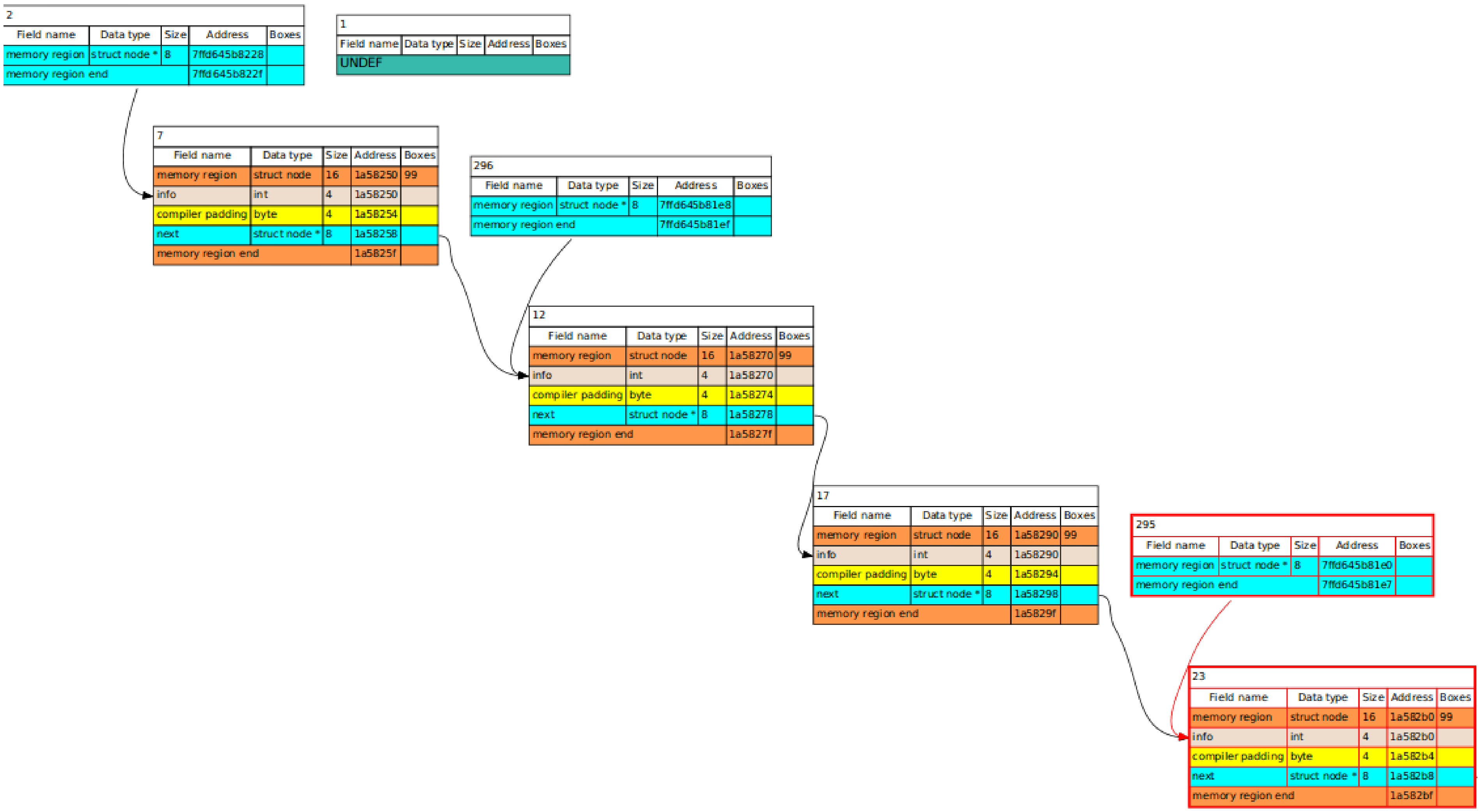
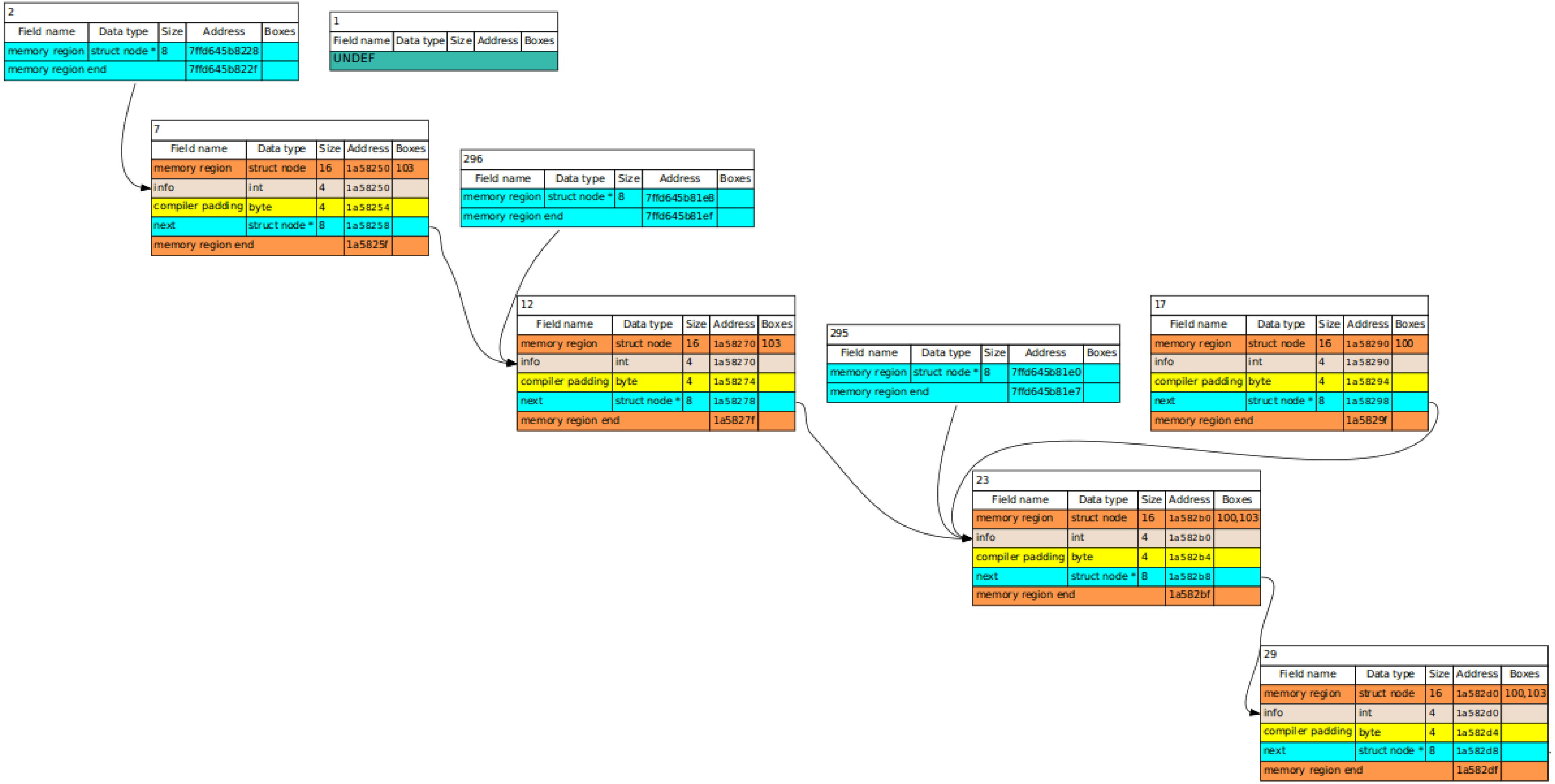

2
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b8228
memory region end | 7ffd645b822f
1
Field name | Data type | Size | Address | Boxes
UNDEF
7
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58250 | 99
info | int | 4 | 1a58250
compiler padding | byte | 4 | 1a58254
next | struct node * | 8 | 1a58258
memory region end | 1a5825f
296
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b81e8
memory region end | 7ffd645b81ef
12
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58270 | 99
info | int | 4 | 1a58270
compiler padding | byte | 4 | 1a58274
next | struct node * | 8 | 1a58278
memory region end | 1a5827f
17
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a58290 | 99
info | int | 4 | 1a58290
compiler padding | byte | 4 | 1a58294
next | struct node * | 8 | 1a58298
memory region end | 1a5829f
295
Field name | Data type | Size | Address | Boxes
memory region | struct node * | 8 | 7ffd645b81e0
memory region end | 7ffd645b81e7
23
Field name | Data type | Size | Address | Boxes
memory region | struct node | 16 | 1a582b0 | 99
info | int | 4 | 1a582b0
compiler padding | byte | 4 | 1a582b4
next | struct node * | 8 | 1a582b8
memory region end | 1a582bf

Figure 4.11: stackoverflow.com example, step 6842.



2

Field name	Data type	Size	Address	Boxes
memory region	struct node *	8	7ffd645b8228	
memory region end			7ffd645b822f	

1

Field name	Data type	Size	Address	Boxes
UNDEF				

7

Field name	Data type	Size	Address	Boxes
memory region	struct node	16	1a58250	103
info	int	4	1a58250	
compiler padding	byte	4	1a58254	
next	struct node *	8	1a58258	
memory region end			1a5825f	

296

Field name	Data type	Size	Address	Boxes
memory region	struct node *	8	7ffd645b81e8	
memory region end			7ffd645b81ef	

12

Field name	Data type	Size	Address	Boxes
memory region	struct node	16	1a58270	103
info	int	4	1a58270	
compiler padding	byte	4	1a58274	
next	struct node *	8	1a58278	
memory region end			1a5827f	

295

Field name	Data type	Size	Address	Boxes
memory region	struct node *	8	7ffd645b81e0	
memory region end			7ffd645b81e7	

17

Field name	Data type	Size	Address	Boxes
memory region	struct node	16	1a58290	100
info	int	4	1a58290	
compiler padding	byte	4	1a58294	
next	struct node *	8	1a58298	
memory region end			1a5829f	

23

Field name	Data type	Size	Address	Boxes
memory region	struct node	16	1a582b0	100,103
info	int	4	1a582b0	
compiler padding	byte	4	1a582b4	
next	struct node *	8	1a582b8	
memory region end			1a582bf	

29

Field name	Data type	Size	Address	Boxes
memory region	struct node	16	1a582d0	100,103
info	int	4	1a582d0	
compiler padding	byte	4	1a582d4	
next	struct node *	8	1a582d8	
memory region end			1a582df	

Figure 4.12: stackoverflow.com example, step 6843.

5 Strand connections

The connections between the memory comprising DDSs is a fundamental property of a DDS, which will be both identified and quantified in this chapter. The memory connections are both important within one particular DDS, such as the connections of the left and right pointers of a binary tree, as well as for the composition of various DDSs, such as in parent child nestings. Being able to precisely describe those connections plays an important role with regards to the expressiveness of the detectable DDSs. Typically, related work only considers pointers for those connections as discussed in Sec. 5.1. This does not support parent child nestings as shown in Fig. 5.1, where each head node of both child elements is embedded inside of the parent. Therefore, DSI considers *two* types of connections instead of just pointers: (i) classical pointer connections, which are termed *indirect* connections, and (ii) tight connections, where parts of the DDS are reachable by offset calculation with a memory vertex, termed *overlay* connections.

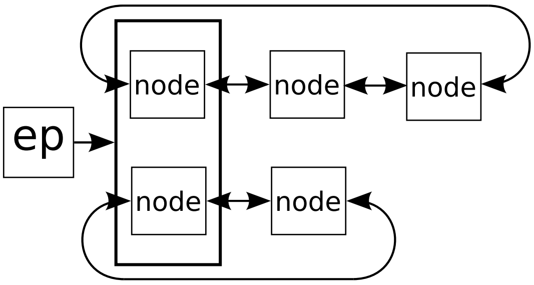


Figure 5.1: Parent child nesting, with two child Linux kernel lists running through a common parent node. Figure reproduced from our publication [107].

As the memory abstraction used by DSI are cells, i.e., (sub-)regions of memory, which subsequently form strands, it is required to express the connections between strands. This is done by calculating the connections between cells, which naturally quantifies the connections between strands as strands are composed of cell sequences. The interconnection between strands are called *SCs*, which are used in all of DSI's strand graphs, i.e., SG, FSG and ASG. Thus, the expressiveness of the SCs developed in this chapter are mandatory for DSI's concepts such as structural and temporal repetition.

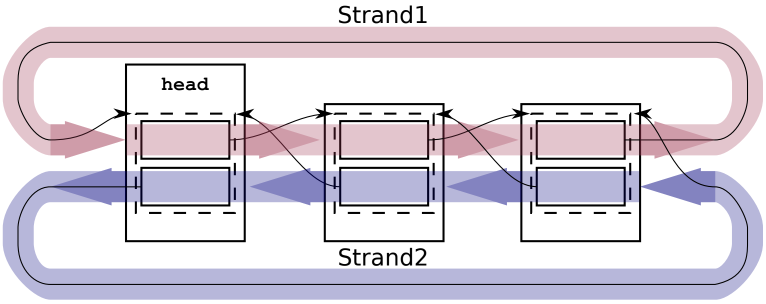


Figure 5.2: DSI's memory abstraction shown for the Linux kernel list. DSI detects two connected (cyclic) singly linked lists that form the cyclic doubly linked list by being connected in reverse order. Figure is reproduced from our publication [94].

The next step in quantifying the connections is to identify the requirements of both overlay and indirect connections. The requirements for overlay connections are discussed first by analysing the LKL example, already shown in the introduction, and consisting of two strands, `Strand1` and `Strand2` running in opposite directions, see Fig. 5.2. The LKL runs through nodes of different types, leading to the important requirement that the relative connection between `Strand1` and `Strand2` is independent of the placement of those strands inside of the surrounding struct. This situation is shown in isolation in Fig. 5.5. An additional requirement is to capture changes in the connection between two strands, as shown in Fig. 5.6 and Fig. 5.9, as soon as their relative offset changes. This is mandatory because the predicates used by DSI for detecting DDSs requires elements performing the same role to have identical connections between them. For example, a DLL requires all connections between the cells of the strands to be the same, e.g., it is not allowed to form a DLL in cases where the distance between the two strands vary. This is a design decision, which could be further investigated in future work, whether relaxing this requirement leads to higher abstractions in the detected DDS, i.e., finding data structure patterns not intended originally by the programmer, or to a loss of precision.

The same requirements hold for overlay parent child nesting scenarios, where the same connections between the parent and the children allows one to unify children performing the same role. When looking at Fig. 5.5 again and imagining a parent child nesting relation, such a relation needs to be unaffected of the placement of a nested element inside the surrounding memory region, as long as their relative offsets stay the same. In Fig. 5.8, a situation is shown, where the parent is connected to two different child elements, each with another offset within the surrounding memory region. Again, both scenarios need to be expressible and detectable as discussed in the previous section.

When looking at indirect connections instead of the overlay connections, it becomes clear that the same requirements apply as well. The only difference is that, in case of overlay connections, one deals with nested structs, whereas with indirection connections, one deals with pointer connections. In case of indirect parent child nesting, it is also mandatory to be able to unify over child connections that all originate at the same offset relative to the parent strand. Thus, the placement of a nested struct inside of the outer memory region should not affect the connections, as is shown in Fig. 5.7. If changes in the connections occur, they need to be recognized, as seen in Fig. 5.8, but which also applies to indirect connections. These requirements apply to both indirect and overlay connections and allow us to detect repetitive behaviour in (parts of) the DDS, exposing connections that perform the same role within the DDS. At the same time the abstraction is fine grained enough to detect differences in the SCs, e.g., when multiple different child elements are present simultaneously.

Regarding the implementation, the above requirements cannot be achieved immediately. One option to precisely describe connections is by using absolute start addresses of cells. However, this would be too specific as addresses are unique within a program, thus preventing the unification of connections that perform the same role. This would effectively disable the structural and temporal repetition detection. Another option would be to classify connections between cells as either overlay or indirect, without additional information. However, this would lead to an over-generalization; it would not be possible to differentiate, e.g., between different children of a parent, which diminishes the precision of the analysis. Therefore, the solution needs to be precise enough to distinguish between different roles of connections, yet uniformly enough to be able to detect commonalities between identical connections. Additionally, the abstraction needs to be chosen in such a way that DSI's capabilities to let strands run arbitrarily through memory vertices are still preserved. This imposes problems such as having multiple cells of one strand within one vertex, thus rising the need to choose which cells are considered when computing the connections.

With these prerequisites in mind, the remainder of this chapter is structured as follows. First the techniques to describe connections used by related work are discussed in Sec. 5.1. Thus, the discussion of how DSI describes the overlay connections found in DDSs in Sec. 5.2 follows. This includes how overlay connections are detected (Sec. 5.2.1), how they are quantified to meet DSI's requirements (Sec. 5.2.2) and presents pseudocode for the actual implementation (Sec. 5.2.3). Subsequently, indirect strand connections are discussed in Sec. 5.3.1, including pseudocode for the actual implementation (Sec. 5.3.2). The chapter concludes with Sec. 5.4 that describes entry point connections, which are the handles into the DDS, i.e., they are the only way how a programmer can access the DDS.

5.1 Related work

The dynamic analysis tools [49, 69, 74] and heap snapshot tools [39, 85], discussed in Ch. 3, also deal with connections between (parts of) the DDSs they support. In the following, the tools will be analyzed according to this specific aspect.

Heapviz. Heapviz [39] uses connections between objects located on the heap to fold the graph, so as to ease understanding of the heap. The stack is not taken into consideration. Most interestingly, Heapviz uses pointer connections to fold two objects (o_1 and o_2) of the same type having a common set of parent objects. It is sufficient for the folding that a pointer connection exists between the objects. It is not required that both objects o_1 and o_2 are connected by the same pointer type. Thus, the two objects are merged, even if they are connected by multiple different pointers, say `child1` and `child2`, which generalizes multiple different parent child connections to just one. DSI instead wants to be precise enough to be able to explicitly speak about all child elements. Further, nested objects are not supported by Heapviz, in contrast to DSI.

HeapDbg. HeapDbg [85] also folds the PTG of the heap as does Heapviz. The major difference is that HeapDbg is more precise with regards to the parent child relation, as they require the connections between the parent and the child elements to be of the same label and both children of the same type. Again, the connections are only pointer based, i.e., no overlay connections are considered. The label of a connection is specified as being either a variable name, a field name inside an object, or an array index. This information requires access to the names of the variables, e.g., source code or Java bytecode. Binary code does not provide this information, which is discussed when opening up DSI for binaries in Pt. II of this dissertation.

Mempick. Mempick [69] cuts pointers between nodes of different types, which is not desired by DSI as this prevents us to reason about parent child nestings, both in the nesting scenario and with pointer connections. MemPick handles PTGs comprising of nodes of the same type; the pointer connections are identified by their offset within an allocated memory chunk.

ARTISTE. ARTISTE [49] has the same notion for merging PTGs as HeapDbg, but the pointer edges between objects are now described by their offset relative from the start of the allocated memory chunk, as does Mempick, rather than a label as in HeapDbg. Besides that, the merging is comparable to HeapDbg. Again, the nesting case is not considered and the offsets are calculated from the beginning of the enclosing memory, i.e., ARTISTE has no notion of nested structs.

DDT. DDT [74] creates a PTG similar to the tool RDS [92], but does not clearly state, if it stores pointer offsets on the edges. In either case, the offset mentioned by the authors of DDT is given relative from the start of the enclosing memory region. DDT decorates the edges with three high level labels (i) the *child* label for edges between nodes of the same DDS; (ii) the *foreign* label for marking edges between nodes of different data structures; (iii) the *data* label for identifying static data associated with a data structure node. If the pointer offset is indeed missing, the high level labels will result in the same limitations as for Heapviz. If the offset is present, the same limitations apply as with ARTISTE and HeapDbg.

Summary. All related tools do not explicitly consider the nesting case, where part of a data structure is nested inside another, for which the LKL is a prominent example. In such scenarios, the nested element would just be ignored by related work, as all of the approaches only consider pointer connections between data structures. Heapviz and (supposedly) DDT only use a high level description of pointer connections, which, e.g., does not allow one to distinguish between multiple parent child relations where the child elements all have the same type. The remaining approaches [49, 69, 85] are more descriptive on the edges by using either labels [85], which are unavailable when dealing with binaries, or offsets [69, 85], which are always from the start of the enclosing memory chunk, i.e., offsets alone do not support the fine grained cell abstraction of DSI. To tackle these limitations overlay connections between strands are discussed further in the following section. Subsequently, indirect connections between strands are introduced in Sec. 5.3.1. For both type of connections, the key insight are *relative offsets* between the involved strands within a memory region.

5.2 Overlay strand connections

The process of forming an overlay connection is twofold. Firstly, a decision needs to be made during the build up phase of the SG, whether two strands are actually connected via overlay and which cells are involved. Secondly, the connection needs to be quantified. The first aspect will be described in Sec. 5.2.1, the second will be discussed in Sec. 5.2.2.

5.2.1 Detecting an overlay connection

As discussed in Sec. 5.1, the related work does not explicitly model connections in case of nesting scenarios that occur, e.g., in certain parent child relations, where parts of a child are directly embedded inside of the parent. Such a situation is shown in Fig. 5.1, where a parent node contains two embedded head nodes of the LKL children. The connection between those two children cannot be captured when only pointer connections are considered. A connection between two cells

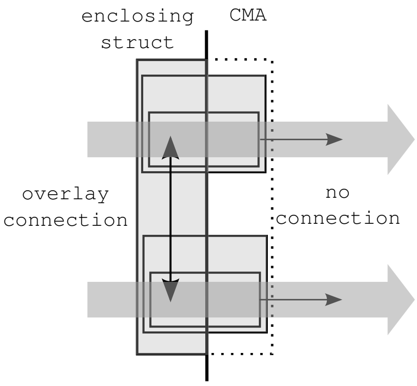


Figure 5.3: The maximum enclosing memory sub-region in case of a surrounding struct (solid line on the left) and a custom memory allocator (dotted line on the right) highlighted in grey. The strands are indicated as block arrows and the resulting overlay connection as a bidirectional arrow.

residing in the same memory chunk is termed overlay connection, i.e., they are reachable from each other through offset calculations within a common memory region without following pointers. As DSI is based on strands, which are made out of cell sequences, the connection between strands can be calculated by considering connections between cells. Therefore, cell connections also describe strand connections. Overlay connections are always computed pairwise between cells of the strands residing in the same memory chunk with respect to a common *Maximum Enclosing memory sub-Region (MER)* for both cells. In terms of the C programming language, this is the maximal enclosing struct containing both cells.

This notion establishes connections between all cells of different strands within one enclosing struct, but prevents connections between cells of unrelated structs within a common memory region, which might be the case with Custom Memory Allocators (CMAs) as depicted in Fig. 5.3. Here two structs are seen, each containing one cell. The inner structs are once surrounded by an enclosing struct (left) and once embedded inside a chunk of memory allocated by a CMA (right). Therefore, the Maximum Enclosing memory sub-Region (MER) will find a common struct containing both cells in case of the surrounding struct, but none in case of a CMA. The result is an overlay connection in case of the enclosing struct and no connection in case of the CMA. This semantics prevents the creation of arbitrary connections between strands in case of a CMA.

As strands are designed to run through memory regions in arbitrary fashions, it is possible that multiple cells of one strand reside within the same memory region. To avoid clutter of possible connection combinations between cells in such a scenario, the most upstream cell ($\mathbf{A}_{\text{up}}$) of each strand still within this memory region is used, see Def. 1. An example requiring the selection of the most upstream cell is shown in Fig. 5.4, where the most upstream cell of the lower strand

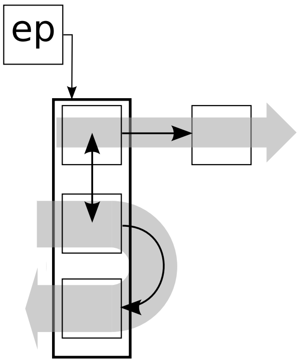


Figure 5.4: Overlay connection between the most upstream cell of a strand with multiple cells per vertex and another strand with only one cell per vertex.

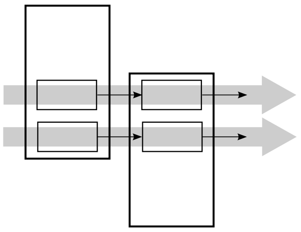


Figure 5.5: Strand position changes within vertices, with unchanged relative offsets of the overlay connections.

is selected to form the overlay connection with the only cell of the upper strand present inside the vertex.

Definition 1. *The most upstream cell* $\boldsymbol{A_{up}}$ *of a strand* $\mathcal{S}$ *inside a vertex* $\mathcal{V}$ *is defined as* $\boldsymbol{A_{up}} \leftarrow minIndex(\{cell \in \mathcal{S} \mid cell \in \mathcal{V}\})$
Where $minIndex$ *returns the cell with the minimum index within the sequence of cells as defined by* $\mathcal{S}$*, where the first cell of the strand starts out with the lowest index.*

5.2.2 Quantifying an overlay connection

Once the cells forming the overlay connection are found, the connection needs to be quantified. This is important to be able to unify or differentiate connections. To quantify the connections multiple possibilities exist:
i) One could use the start addresses of cells to quantify their positions in memory. However, this will prevent any unification of cell positions, as addresses are unique within the address space of the program.
ii) Another possibility is by using the offsets of the cells relative from the surrounding memory region. This will produce the same offsets as long as the memory regions and the position of the cells within the memory region are the same.

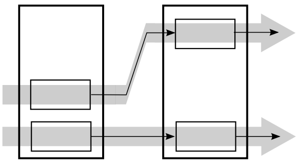


Figure 5.6: Strands with changed offsets between them.

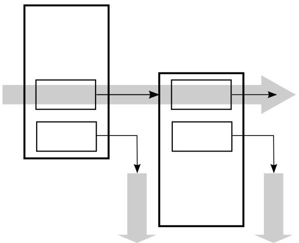


Figure 5.7: Parent child relation on indirect nesting, with parent strand changing position within vertices. Child strands are indicated with two vertical grey arrows only, without cells or vertices.

Therefore, it is possible to unify over connections to a certain extent. As soon, as strands run through different positions of a memory region, this approach produces different start offsets, even though the relative distance between the strands is actually the same, as is seen in Fig. 5.5.

iii) This problem is similar to the problem of the linkage offset for strands, which is solved by calculating the linkage offset from the start of a cell and not from the surrounding memory region. This insight leads to describing the relative distance, or offset, between cells. This will give the same connections even when the position of the strands inside of the memory region change, as can be seen in Fig. 5.5, and capture changes between the strand distance as shown in Fig. 5.6.

For these reasons, the third option is chosen as the solution. However, there are three different possibilities how to express the relative distance between the cells, as shown in Fig. 5.10. The same vertex is shown three times with two cells and with two different linkage offsets, resulting in four strands ($\mathcal{S}_{1-4}$, $\mathcal{S}_{5-8}$ and $\mathcal{S}_{9-11}$). For each distance possibility, the corresponding offsets are shown. The possibilities are as follows: (left) relative from the start of the cells, (middle) relative from the linkage offsets of the cells, (right) relative from the start of the source cell to the linkage offset of the target cell. Though not all three approaches do fulfill the requirement of capturing the relative changes between two strands, as shown in Figs. 5.5, 5.6 and 5.9. The difference lies, e.g., in the precision of quantifying the parent child relations, as seen in Fig. 5.8, and the switching of strand connections within a vertex, as see in Fig. 5.9. This can be seen, when looking at Table 5.1, which shows the possible strand combinations of the connections of Fig. 5.10. The columns and rows of Table 5.1 labelled $\mathcal{S}_x$ identify the strands of Fig. 5.10.

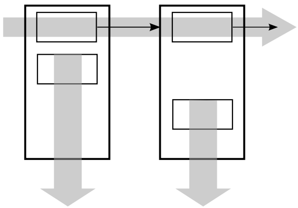


Figure 5.8: Parent child relation on overlay nesting, with child strands changing position within vertices. Child strands are only indicated with two vertical grey arrows, without further cells besides the head cell.

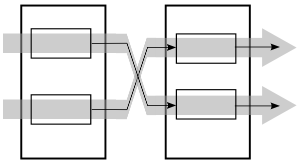


Figure 5.9: Strands switching positions within vertices.

The offset between the strand pairs are listed as tuples consisting of the offset from the start strand to the target strand as the first element of the tuple, and the backwards direct as the second element of the tuple, e.g., $(a, -a)$. The tuples below the diagonal of the matrix is left empty (−), as those tuples will just have flipped elements as the source and the target are reversed.

The table reveals that the information density varies between the differently chosen offset calculations. Starting out on the left, it becomes eminent that this is the least precise solution. The offsets are either $(0, 0)$ for the strands residing inside of the same cell, or $(a, -a)$ for the other combinations. Especially strand connections within the same cell are problematic, because the offset will always be $(0, 0)$, thus making DSI agnostic of the situation shown in Figs. 5.9 and 5.6, where the positions of the strands are changed. The abstraction also imposes the problem of not being able to detect different parent child relations as shown in Fig. 5.8, as the tuple for both connections would again be $(0, 0)$. This in turn would lead DSI to consider both parent child relations as performing the same role, which is not true because clearly different elements are participating in the parent child nesting. Thus this solution works only when the strands participating in the overlay connection reside in different cells. And even in this case, multiple strands with different offsets within the source and target cells would not be distinguishable, leading to a loss of precision. This solution is not an option, because we want to overcome this limitation, from which related work suffers as well.

The approaches in the (middle) and the (right) of Fig. 5.10 are more descriptive; they can both handle all situations described previously. The (middle) solution works by directly calculating the offsets between the strand pairs, whereas the (right) solution takes the surrounding cells into consideration as well. In both

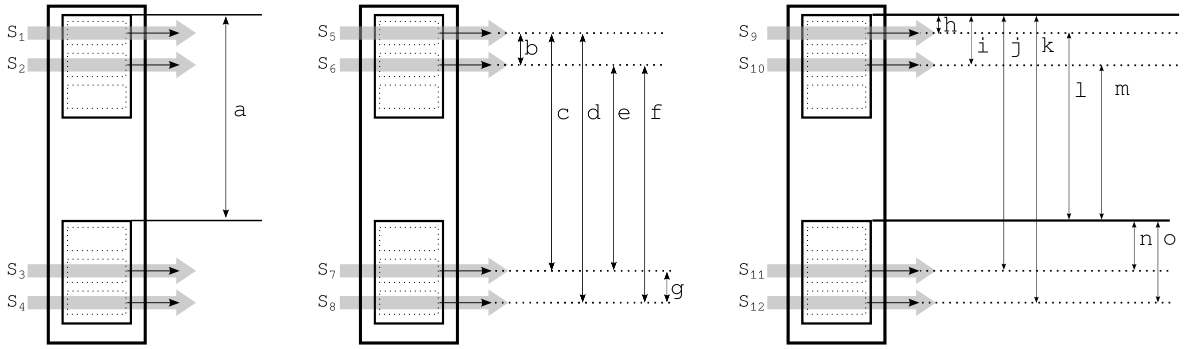


Figure 5.10: Relative offset calculations between strand connections: beginning of cells (left), between linkage offsets (middle), beginning of cells to linkage offsets (right).

cases, information about the positions of the strand pairs relative to each other are encoded through the sign of the offsets within the tuples. Though in Table 5.1 no signs show up explicitly for the right most solution, as all of the variables are assumed to be unique in this case, i.e., no variables are the same when considering their absolute values. Further, the variables shown for the (right) solution follow the pattern, seen on the right in Fig. 5.11, whereby the offset from the source (upper) strand to the target (lower) strand are always calculated from the start of the enclosing cell to the linkage offset of the target strand. However, other possibilities for calculating the offsets are conceivable, e.g, the left side of Fig. 5.11 where the offsets are always calculated from the start address of the source cell to both the source and target strand linkage offsets. The difference to favour the right solution becomes eminent when considering how the offsets will change in case of alternations in either the strands within the cells or the distance between the cells. As an example consider that the lower strand changes its linkage condition within its surrounding cell. This might happen in a parent child nesting situation with different children hanging off different pointers. Indeed, both approaches again detect the differences in the connections, but the right solution is more descriptive in what exactly changed. In this particular example, the offset from the start of the upper cell to the linkage offset of the target strand would change, the backward offset would remain equal. This indicates that the distance between the cells has remained stable and only the linkage offset of the target has changed. With the left solution, the distance from the start of the source cell to the linkage offset of the target cell would also change, but this change cannot only happen due to a change in the linkage offset, but also due to a change in the relative distance of the surrounding cells. This makes the left solution more ambiguous. For the right solution, both of the relative offsets change when the relative distance between cells changes. Though the (right) solution is also not completely immune to ambiguity, e.g., when it comes to changing both linkage offsets and cell posi-

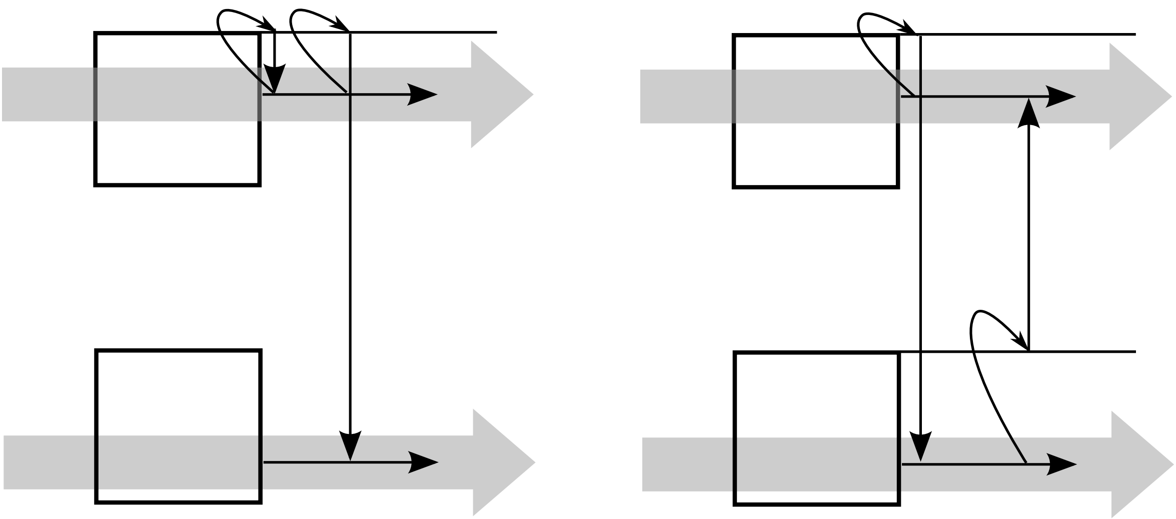


Figure 5.11: Two possible implementations shown for calculating the relative offsets between two strands in an overlay connection. On the left the offset calculation always starts from the start address of the cell of the top strand to the linkage offset of both strands. On the right the calculation is performed once from the start address of the cell of the top strand and once from the start address of the lower strand.

tions simultaneously. However, the increased precision of the right most solution leads us to choosing this offset calculation strategy.

While constructing the strand connections, all connections of the same configuration are combined to just one connection in the SG, FSG and ASG. However, the set of source and target cells for each connection between strand pairs are stored so that they are available for further calculations.

The general definition for overlay and indirect strand connections is given in Def. 2. The definition of an overlay connection is given in Def. 3 (both definitions are reproduced from our publication [107]).

Definition 2. *A* ***strand connection*** *(SC)* $S_1 \overset{\alpha}{\cdots\cdots} S_2$ *between two strands,* S_1 *and* S_2*, describes exactly one way in which a subset of the cells of* S_1 *are related to a subset of the cells of* S_2*. An SC is defined by the set of* ***cell pairs****, consisting of cell* c_1 *of* S_1 *and cell* c_2 *of* S_2 *that establish the relationship:* $\textsc{pairs}(S_1 \overset{\alpha}{\cdots\cdots} S_2) = \{(c_1, c_2) \in \textsc{cells}(S_1) \times \textsc{cells}(S_2) : c_1 \overset{\alpha}{\cdots\cdots} c_2\}$.

Definition 3. *The* ***overlay*** *relationship between two cells* c_1 *and* $_2$*, connecting two strands* S_1 *and* S_2 *resp., form a cell pair:* $c_1 \overset{xw}{\longleftrightarrow} c_2$ *if* $S_1 \neq S_2 \wedge \textsc{mer}(c_1) = \textsc{mer}(c_2)$*, where*
$x = (c_1.\textit{bAddr} + \textsc{linkageOffset}(S_1)) - c_2.\textit{bAddr}$ *and*
$w = (c_2.\textit{bAddr} + \textsc{linkageOffset}(S_2)) - c_1.\textit{bAddr}$

Table 5.1: Tabular view of relative offset calculations between strand connections as seen in Fig. 5.10: beginning of cells (starting with column $\mathcal{S}_1$), between linkage offsets (starting with column $\mathcal{S}_5$), beginning of cells to linkage offsets (starting with column $\mathcal{S}_9$).

	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_3$	$\mathcal{S}_4$		$\mathcal{S}_5$	$\mathcal{S}_6$	$\mathcal{S}_7$	$\mathcal{S}_8$		$\mathcal{S}_9$	$\mathcal{S}_{10}$	$\mathcal{S}_{11}$	$\mathcal{S}_{12}$
$\mathcal{S}_1$	–	$(0,0)$	$(a,-a)$	$(a,-a)$	$\mathcal{S}_5$	–	$(b,-b)$	$(c,-c)$	$(d,-d)$	$\mathcal{S}_9$	–	(i,h)	(j,l)	(k,l)
$\mathcal{S}_2$	–	–	$(a,-a)$	$(a,-a)$	$\mathcal{S}_6$	–	–	$(e,-e)$	$(f,-f)$	$\mathcal{S}_{10}$	–	–	(j,m)	(k,m)
$\mathcal{S}_3$	–	–	–	$(0,0)$	$\mathcal{S}_7$	–	–	–	$(g,-g)$	$\mathcal{S}_{11}$	–	–	–	(o,n)
$\mathcal{S}_4$	–	–	–	–	$\mathcal{S}_8$	–	–	–	–	$\mathcal{S}_{12}$	–	–	–	–

5.2.3 Pseudocode for calculating overlay strand connections

Because the SCs are created when building up the SG, the pseudocode contains some parts of the build up phase of the SG as well, though they were developed by Dr. White. These parts are greyed out in function CALCULATESTRANDGRAPH, see Alg. 8. This function calculates the SG for one timestep t of the event trace, including both overlay connections and indirect connections. Overlay connections are discussed within this section, while indirect connections are discussed in Sec. 5.3.1.

```
function CALCULATESTRANDGRAPH($t$)
  // Create strand vertices
  $\mathcal{V}_{\text{SG}}(t) \leftarrow \{\{S\} \mid S \in \mathcal{S}(t)\}$
  // Set of edges, i.e., strand connections
  $\mathcal{E}_{\text{SG}}(t) \leftarrow \emptyset$
  // Add entry point vertices
  ADDEPVERTICES($t$)
  // Cycle through all pairwise strand combinations
  for each $(S_a, S_b) \in \mathcal{S}(t) \times \mathcal{S}(t)$
    // Calculate vertices for strands
    $\mathcal{V}_a \leftarrow \{v \in \mathcal{V}(t) \mid \exists \mathbf{C} \in S_a \mid \text{VERTEX}(\mathbf{C}) = v\}$
    $\mathcal{V}_b \leftarrow \{v \in \mathcal{V}(t) \mid \exists \mathbf{C} \in S_b \mid \text{VERTEX}(\mathbf{C}) = v\}$
    // Indirect strand connections
    // Get edges between source and target strands
    $\mathcal{E}_{\text{ab}} \leftarrow \{(v_s, \text{sAddr}, v_t, \text{tAddr}) \in \mathcal{E}(t) \mid (v_s \in \mathcal{V}_a \wedge v_t \in \mathcal{V}_b)$
        // No linkage condition between the strands
        $\wedge !\text{LINKCOND}(\text{sAddr}, \text{tAddr}, \mathcal{G}(t))\}$
    ADDINDIRECTSCS($\mathcal{E}_{\text{ab}}$,$S_a$,$S_b$,$t$)
    // Overlay based strand connections
    ADDOVERLAYSCS($\{\mathcal{V}_a \cap \mathcal{V}_b\}, S_a, S_b, t$)
  end
  return $(\mathcal{V}_{\text{SG}}(t), \mathcal{E}_{\text{SG}}(t))$
```

Algorithm 8: Calculation of the strand graph vertices and edges for a time step

Initialize the SG. The SG algorithm starts out by creating the set of vertices $\mathcal{V}_{\text{SG}}(t)$ of the SG, each consisting of a set of strands and initially containing one strand per vertex. When merging elements of the SG later on, i.e., creating the FSG or subsequently the ASG, the strands of the merged vertices are unioned, leading to more elements in the set. The edge set $\mathcal{E}_{\text{SG}}(t)$, i.e., the strand connections, is initially empty. Then, the entry point vertices and the offsets for those are calculated, as explained in Sec. 5.4. Entry point connections can be seen as specialized strand connections.

Strand combinations and connections. Following the previous step one needs to process all pairwise strand combinations of the strands for the current time step $\mathcal{S}(t)$ from the strand creation phase and calculate the strand connection between each source and target strand. Note that the algorithm always calculates directed strand connections from the source to the target, but as all pairwise strand connections are processed, the backward direction of overlay strand connections is also created, thus forming the bidirectional property of the connection. The SG is calculated on top of the PTG; thus, it requires information about the vertices and edges of the PTG. Therefore, the set of vertices through which each strand is running is determined by fetching the vertices from the memory vertices $\mathcal{V}(t)$ of the PTG in which a cell of the strand resides. The vertices are used to define the edges of the PTG, i.e., pointer connections, with the source and target vertices and addresses $(v_s, \text{sAddr}, v_t, \text{tAddr})$. Thus, all edges where the source and target vertices are in the set of vertices through which the strands are running are selected. An additional restriction being that the edge does not fulfill a linkage condition itself, as this would mean both strands are connected via a strand instead of a pointer. This is computed with the function LINKCOND, which takes the source (sAddr) and target (tAddr) address of the pointer and the PTG of the current time step ($\mathcal{G}(t)$) as parameters. The edges can be extended to arbitrary pointer chains without forming a linkage condition. This is left for future work. The edges $\mathcal{E}_{\text{ab}}$ found are used to calculate the indirect strand connections, while the intersection of memory vertices ($\mathcal{V}_a$ and $\mathcal{V}_b$) select all vertices that potentially have an overlay connection, i.e., at least both strands run through the same vertex. The MER function later on makes the final decision if an overlay connection is actually present.

Overlay strand connection. Function ADDOVERLAYSCS in Alg. 9 calculates the actual connection configuration as discussed previously, by cycling through each vertex containing both the source and target strands and first calculating the most upstream cell of each strand to handle CMA-like situations. Then, the MER of both cells need to match. If this is the case, the forward and backward distances of the cells are calculated as described in Sec. 5.2.2. The result is stored in a key-value set ($\mathcal{KV}$), termed *connection configuration*. The key value is composed of the *overlay* label and the offsets. The value consists of a set of tuples of source and target cells forming the connection. Thus, all equal connections between strands are unified to one connection configuration that carries a set with the corresponding cell pairs. Then, the actual strand connections are created in function ADDSTRANDCONNECTIONS in Alg. 10. The algorithm fetches all key values by calling the function DOM$(\mathcal{KV})$ and iterates over them to create a set of strand connection edges.

```
function ADDOVERLAYSCS($\mathcal{V}_{\text{ab}}, S_a, S_b, t$)
  for each $v \in \mathcal{V}_{\text{ab}}$
    // Vertices might contain multiple cells of a strand,
    // thus the most upstream cell needs to be fetched
    $\mathbf{C}_a \leftarrow$ GETTOPMOSTSTRANDCELL$(S_a, v)$
    $\mathbf{C}_b \leftarrow$ GETTOPMOSTSTRANDCELL$(S_b, v)$
    // Maximum Enclosing memory Region needs to be equal
    if (MER$(\mathbf{C}_a)$ = MER$(\mathbf{C}_b)$)
      // Offset calculation
      $c_{\text{a off}} \leftarrow (\mathbf{C}_b.\text{bAddr} + S_b.\text{linkageOffset}) - \mathbf{C}_a.\text{bAddr}$
      $c_{\text{b off}} \leftarrow (\mathbf{C}_a.\text{bAddr} + S_a.\text{linkageOffset}) - \mathbf{C}_b.\text{bAddr}$
      // Record the overlay connection configuration
      $cc \leftarrow (\text{``}overlay\text{''}, (c_{\text{a off}}, c_{\text{b off}}))$
      $\mathcal{KV}(cc) \leftarrow \mathcal{KV}(cc) \cup \{(\mathbf{C}_a, \mathbf{C}_b)\}$
    end
  end
  ADDSTRANDCONNECTIONS($\mathcal{KV}, S_a, S_b, t$)
```

Algorithm 9: Calculation of the overlay connection configurations

A strand connection edge is a five-tuple, which holds a set of source strands S_a, a set of target strands S_b, connection configurations cc, a set of the source and target cells forming the strand connection, and an empty set that holds the different classifications for this particular connection during the lifetime of the data structure used by the final naming algorithm of DSI. The classification and naming algorithm is not part of this dissertation.

```
function ADDSTRANDCONNECTIONS($\mathcal{KV}, S_a, S_b, t$)
  for each $cc \in \text{Dom}(\mathcal{KV})$
    // Add the edge, and store the connection configuration and
    // an empty set for storing the classification information
    // used during the naming process. Additionally, the set of
    // cell tuples forming the connection is stored.
    $\mathcal{E}_{\text{SG}}(t) \leftarrow \mathcal{E}_{\text{SG}}(t) \cup \{(\{S_a\}, \{S_b\}, cc, \mathcal{KV}(cc), \{\})\}$
  end
```

Algorithm 10: Calling the classification routine on configuration sets. Additionally, the edges are being stored

5.3 Indirect strand connections

In the following the detection of an indirect strand connection is discussed in Sec. 5.3.1, the pseudocode is given in Sec. 5.3.2.

5.3.1 Detecting an indirect strand connection

Indirect strand connections are formed by pointer connections between strands that do not fulfill a linkage condition as this would lead again to an overlay connection. The pointer does not need to originate directly from the source cell, nor does it directly need to link to the target cell. It is only required that the source and target cells have the same MER. Indirect connections face the same problem of calculating the connection offsets as is the case with the overlay connections, see Sec. 5.2. With the knowledge gained from overlay connections, indirect connections are also expressed using relative offsets. The difference between overlay and indirect connections is that, for overlays, two cells form the connection and, for indirect connections the source and target cells are put in relation with the connecting pointer to form the connection. More precisely, indirect connections form the offset on the source side of the pointer connection between the start address (bAddr) of the source cell and the start address of the outgoing pointer (sAddr). On the target side, the start address of the target cell plus the linkage offset is correlated with the target address of the outgoing pointer (tAddr). This is formalized in Def. 4 (reproduced from our publication [107]), and the memory layout and addresses are shown in Fig. 5.12.

Definition 4. *The **indirect** relationship between cells c_1 and $_2$, connecting two strands S_1 and S_2 resp., form a cell pair: $c_1 \xrightarrow{yz} c_2$ if $\exists e = (\_, sAddr, \_, tAddr) \in \mathcal{E} : a_s \mathrel{\bar{\in}} \textsc{mer}(c_1) \wedge a_t \mathrel{\bar{\in}} \textsc{mer}(c_2)$ and there is no linkage condition on edge e, where*
$y = sAddr - c_1.\mathit{bAddr}$ and
$z = (c_2.\mathit{bAddr} + \textsc{linkage}\textsc{Offset}(S_2)) - tAddr.$

Implications for connections. As is the case with overlay connections, the relative offsets allow changes of the position of the strands and pointers as seen in Figs. 5.7 and 5.13. In the latter, a more detailed view of the source strand switching the position within the vertex is given. Additionally, the dotted lines indicate where the relative offsets are calculated. As the layout of the smaller parent vertices on the left and the right is duplicated within the bigger middle one, the corresponding pointer resides at the same relative offset as seen in the smaller vertices. Thereby, the source side computes the same relative offset for all three parent vertices. The target side is also shown in Fig. 5.13, where the distance between the address of the incoming pointer to the linkage offset of the child element is shown. The same logic as on the source side applies, i.e., relative offset changes can be captured.

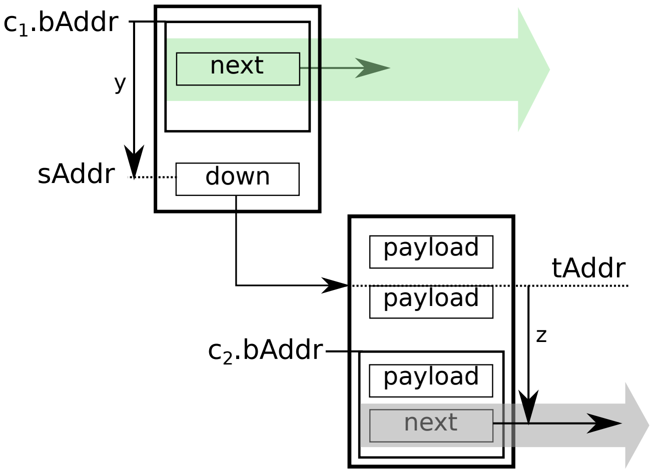


Figure 5.12: Memory offsets used for computing an indirect strand connection. c_1**.bAddr** marks the start address of the the first cell, and c_2**.bAddr** the start address of the second cell. **sAddr** is the source address of the outgoing pointer and **tAddr** is the target address where the pointer points to.

As is the case with overlay connections, all possible combinations between the pointers and the strands residing inside both the source and the target vertex are computed. In contrast to the strand creation, it is not mandatory that the pointer on the target side is incoming to the head of a (nested) struct. This relaxation makes the detection of parent child relations more generic.

In Def. 4 it can further be seen that the offset y on the source side is between the start of the cell of the source strand and the start of the pointer connection. This implies that, on the source side, all strand connections running through the same cell, i.e., with different linkage offsets, still have the same offset y. This is not a problem as we do not have a situation like with the overlay connections as seen on the left in Fig. 5.10 where this leads to ambiguous connections in parent child nesting scenarios, as seen in Fig. 5.8.

On the target side an ambiguity can arise if both the cell position and the linkage condition of the target strand change simultaneously in such a way that the relative offset z stays the same for different parent child relations. This problem is solved by taking the linkage offset into the predicate for matching the same parent child relations, i.e., the connection configuration needs to match among the children and, additionally, all child elements need to be of the same type and have the same linkage offset.

5.3.2 Pseudocode for calculating indirect strand connections

The function addIndirectSCs of Alg. 11 processes all pointers ($\mathcal{E}_{\text{ab}}$) found earlier between two strands S_a and S_b for a timestep t. Again, the top most upstream cell,

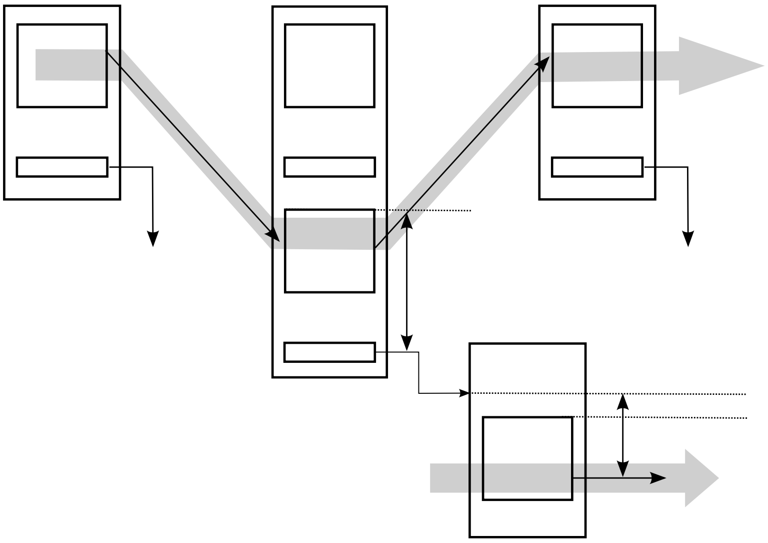


Figure 5.13: Relative offsets for an indirect strand connection both on the source (top vertices) and the target side (bottom vertex). The parent strand (source) switches position within a vertex without changes in the relative offsets (between parent strand and outgoing pointer).

$\mathbf{C}_a$ and $\mathbf{C}_b$, of each strand still residing in the source vertex v_s and target vertex v_t is chosen for the offset calculation to account for CMA cases. The MER is also obeyed, and connections of the same type are unified by creating a key-value store $\mathcal{KV}$, where the key is a connection configuration cc with the label *indirect* and the corresponding offsets. The value is a set of tuples with the source and target cell, as seen with the overlay connection. In the end, the actual strand connections are recorded by calling addStrandConnections of Alg. 10, i.e., the same function can be used for both overlay and indirect connections.

5.4 Entry point connections

The chapter on strand connections is concluded with a specialized case of connections, namely from entry points to strands. An entry point is a handle into a data structure, which can either be on the stack or in global memory and holds a heap reference pointing to a strand or contains a cell participating in a strand. Therefore, entry points also have connections to the strands. Entry points are the stable view point into a DDS, with the intuition being that the longest running entry point reports the correct interpretation of a DDS. As with the overlay and indirect connections before, the entry point connections are quantified by offset calculations. The big difference to the connections before is the now absolute offset calculation instead of relative offsets. For entry point connects, the offset is given from the start address of the memory chunk. In the following both cases forming an entry point, i.e., a pointer and a cell, are discussed.

```
function ADDINDIRECTSCs($\mathcal{E}_{\text{ab}}, S_a, S_b, t$)
  for each ($v_s$, sAddr, $v_t$, tAddr) $\in \mathcal{E}_{\text{ab}}$
    // Calculate source and target cell
    $\mathbf{C}_a \leftarrow$ GETTOPMOSTSTRANDCELL($S_a, v_s$)
    $\mathbf{C}_b \leftarrow$ GETTOPMOSTSTRANDCELL($S_b, v_t$)
    // Maximum Enclosing memory Region needs to be equal
    if (MER($\mathbf{C}_a$) = MER($\mathbf{C}_b$))
      // Target offset: relative to incoming pointer
      $c_{\text{off}} \leftarrow (\mathbf{C}_b.\text{bAddr} + S_b.\text{linkageOffset}) - \text{tAddr}$
      // Record the pointer connection configuration
      // Source offset: relative to cell
      $cc \leftarrow$ ("indirect", (sAddr $- \mathbf{C}_a$.bAddr, $c_{\text{off}}$))
      $\mathcal{KV}(cc) \leftarrow \mathcal{KV}(cc) \cup \{(\mathbf{C}_a, \mathbf{C}_b)\}$
    end
  end
  ADDSTRANDCONNECTIONS($\mathcal{KV}$, $S_a$, $S_b$, $t$)
```

Algorithm 11: Calculation of the indirect strand connections.

Entry point: pointer. The pointer based entry point as described in Def. 5 (reproduced from our publication [107]), has two offsets. The first offset x is given as the offset between the entry point vertex v_{ep} start address and the start address of the pointer, resulting in an absolute offset for this particular v_{ep}. The second offset y is given analogous to the second parameter of the indirect strand connections in Def. 4, thus making the offset relative again.

Definition 5. *A **pointer based entry point connection** $v_{ep} \xrightarrow{xy} S$ from an entry point $v_{ep} \in \mathcal{V}$ to a cell $c \in$ CELLS(S) via a non-linkage condition edge $(v_{ep}, a_s, v_t, a_t) \in \mathcal{E}$ is defined by two parameters $x = a_s - v_{ep}.\textit{bAddr}$ and $y = (c.\textit{bAddr} + \text{LINKAGEOFFSET}(S)) - a_t$.*

Entry point: cell. The cell based entry point as described in Def. 6 (reproduced from our publication [107]) only has one absolute offset z from the start of v_{ep} to the linkage offset of the cell.

Definition 6. *A **cell based entry point connection** $v_{ep} \xrightarrow{z} S$ from an entry point $v_{ep} \in \mathcal{V}$ to a cell $c \in$ CELLS(S) such that $c \mathrel{\bar{\subseteq}} v_{ep}$ is defined by one parameter $z = (c.\textit{bAddr} + \text{LINKAGEOFFSET}(S)) - v_{ep}.\textit{bAddr}$.*

6 Temporal repetition

DSI reinforces the shape of a DDS by taking temporal information into account for its analysis, i.e., it analysis the DDS over its lifetime. DSI does this by inspecting a DDS at each time step of the execution trace and aggregating the results over time to arrive at a final DDS interpretation at the end of DSI's analysis.

More specifically, the stepwise inspection of the DDSs is enabled by DSI's evidence based DDS detection, where each heap state is interpreted individually for each time step according to DSI's DDS taxonomy. Each detected DDS label is accompanied by an evidence count to quantify the interpretation. Then two properties of a DDS are exploited: (i) structural repetition, where certain parts of a data structure perform the same role, e.g., all children of a parent-child list (ii) temporal repetition, as a DDS typically exists over a longer time period during program execution and thus can be monitored continuously. The previously described quantification of strand connections in Ch. 5, is mandatory to find parts of a DDS that perform the same role, both structurally and temporally.

With DSI's ability to inspect each time step of the DDS evolution, no countermeasures to avoid degenerate shapes need to be taken, e.g., finding operation boundaries [74, 106] or quiescent periods where a DDS is not changed [69]. DSI also does not become more conservative when including degenerate shapes as is the case for other tools, e.g. [49, 67]. Additionally, each time step of a DDS can be inspected without missing interesting behaviour during data structure manipulations. This chapter describes the temporal repetition algorithm.

6.1 Algorithm

The temporal repetition is part of DSI's data structure detection algorithm; hence, it relies on intermediate results calculated in previous steps, i.e., the PTGs, strands, SGs and FSGs. The overview of the DDS detection is given in Alg. 12, where the first `for`-loop shows the previous analysis steps, which are not discussed in this dissertation. The second `for`-loop performs the actual temporal repetition for each entry pointer (ep) beginning from its time of creation (t_{start}). Both are artifacts from the previous analysis steps, i.e., the trace generation and strand creation.

Table 6.1: Symbols used in the temporal repetition algorithm

Symbol/Term	**Explanation**
eps	Set of entry pointer tuples: (start time, entry pointer vertex)
$asgs$	Set of aggregated strand graphs
$\mathcal{F}$	Sequence of folded strand graphs
$\mathcal{F}_{\text{i}}$	Folded strand graph for time step i
fsg	Reference to a folded strand graph
$\mathcal{S}_i$	Set of strands for time step i
$\mathcal{G}$	Sequence of points-to graphs
G_{i}	Points-to graph for time step i
E	Sequence of events
E_i	Event for time step i
e_{fsg}	Edge of the folded strand graph
e_{asg}	Edge of the aggregated strand graph
asg	Aggregated strand graph
$subgraph_{\text{com}}$	Graph containing the common elements of a folded strand graph and an aggregated strand graph
$subgraph_{\text{new}}$	Graph containing the new elements when comparing a folded strand graph and an aggregated strand graph
v	Vertex of a strand graph
cc	The quantified strand connection
$cc_{\text{classified}}$	Label and evidence count for a strand connection
bc	Set of visited vertices when iterating a graph, i.e. a bread crumb
SG	Strand graph

```
// Global variables
// Sequences of points-to graphs, folded strand graphs and events
$\mathcal{G}, \mathcal{F}, E$
// Initialize sets of entry pointer tuples and aggregated strand graphs
$eps \leftarrow \emptyset$
$asgs \leftarrow \emptyset$
// Processing of all events
for each $i \in 0 \ldots E.length$
    // Strand graph creation, including entry point calculation
    $\mathrm{SG} \leftarrow$ calculateSG$(G_{\mathrm{i}}, \mathcal{S}_i, eps)$
    // Naming of the data structure
    detectDS$(\mathrm{SG}, G_{\mathrm{i}}, i, E_i)$
    // Folded strand graph creation
    // Store the resulting folded strand graph in a sequence
    $\mathcal{F} \leftarrow \mathcal{F}$++calculateFSG$(\mathrm{SG}, i)$
end
// Aggregated strand graph calculation from the point of view
// of each entry pointer
for each $(t_{\text{start}}, ep) \in eps$
    $asgs \leftarrow asgs \cup \{$calculateASGforEP$(t_{\text{start}}, ep)\}$
end
```

Algorithm 12: Main part of DSI's data structure detection algorithm, including the strand graph creation, the naming of the data structures, and the structural and temporal repetitions

```
function calculateASGforEP($t_{\text{start}}, ep$)
  // Initialize ASG and store entry pointer as start point
  $asg \leftarrow$ newGraph()
  // $asg.\mathcal{V}$ is analogous to the notation of object
  // oriented programming languages: *object.property*
  $asg.\mathcal{V} \leftarrow asg.\mathcal{V} \cup \{ep\}$
  $i \leftarrow t_{\text{start}}$
  while ptgContainsEP($G_{\text{i}}, ep$)
    if vertexIsEP($\mathcal{F}_{\text{i}}, ep$) $\wedge E_i.kind = memoryWriteEvent$
      // Find common subgraph between ASG and FSG
      $subgraph_{\text{com}} \leftarrow$ newGraph()
      $subgraph_{\text{com}}.\mathcal{V} \leftarrow subgraph_{\text{com}}.\mathcal{V} \cup \{ep\}$
      // Note that the alignment starts out on $ep$ for both
      // graphs. Hence, $ep$ gets passed twice initially
      alignASGandFSG($asg, ep, \mathcal{F}_{\text{i}}, ep, subgraph_{\text{com}}$)
      // Find new elements between ASG and FSG
      $subgraph_{\text{new}} \leftarrow$ newGraph()
      // Initialize a bread crumb set to detect cycles
      $bc \leftarrow \emptyset$
      // The computed difference gets stored in $subgraph_{\text{new}}$
      graphDiffRec($\mathcal{F}_{\text{i}}, ep, subgraph_{\text{com}}, subgraph_{\text{new}}, bc$)
      // Add new elements to ASG
      calculateASG($asg, subgraph_{\text{new}}$)
    end
    $i \leftarrow i + 1$
  end
  return $asg$
```

Algorithm 13: Computation of an aggregated strand graph from the point of view of a given entry pointer (ep)

6.1.1 Main algorithm for temporal repetition

The main part of the temporal repetition calculation is shown in Alg. 13, where the high level overview is as follows: iterate over all time steps where the entry pointer vertex is alive and actually acts as an entry pointer and aggregate all elements of all inspected FSGs during program execution into one ASG. But the ASG is not simply the ”union” of all FSGs, as only the parts that are reachable from a given entry point are aggregated. Hence, the ASG contains all elements reachable by the given entry pointer over its life-time. Note that elements are only *added* to the ASG but never removed, because the ASG represents what is observed over one entry pointer’s lifetime. Hence, the method combines the three phases of the temporal repetition algorithm, namely (i) computing commonalities between the FSG and the ASG (line 11), (ii) computing differences between the FSG and the ASG (line 15), and (iii) extending the ASG with newly found elements (line 21).

When discussing Alg. 13 in more detail, first a new and empty ASG gets created (line 3). As the aggregation of all FSGs is always from the point of view of a given entry pointer (ep), the created ASG gets initialized with the current ep (line 6). Recall that entry pointers are the stable anchors into a DDS; thus, they are used to perform the temporal repetition of the DDS. The aggregation phase is performed iteratively starting at the given time step (t_{start}), which is the creation time of the entry pointer. The iteration continues until the entry pointer is not present in the current PTG (G_{i}) in line 8. As an optimization step, the temporal repetition is only performed, when the ep is currently connected to the data structure (VERTEXIsEP()) and the current event (E_i) is a memory write event (line 9). The latter avoids events that are recorded by the instrumentation, e.g., entering or leaving a function, but are not actually considered an event that (potentially) interferes with the DDS, such as pointer writes or memory (de-)allocations.

The specific functions of the above mentioned steps (i), (ii) and (iii) are described in Secs. 6.1.2, 6.1.3 and 6.1.4,respectively.

6.1.2 Aligning ASG and FSG

In this section, Alg. 14 is discussed, which recursively compares the FSG of the current time step (fsg) and the ASG aggregated so far, starting out with an ASG that only contains the current ep. The algorithm operates from the point of view of the given ASG, as each FSG potentially carries changes to the previous time step, i.e., the FSG is unstable. This could result in missing elements in the FSG that would make an alignment of the ASG and the FSG impossible. The current entry pointer is used as an anchor point into both the ASG and the FSG, as this vertex is the initial commonality between both graphs. This can be seen in line 15 of Alg. 13, where ep is passed into the recursive function twice, once for the ASG and once for the FSG. Because the algorithm recursively explores both graphs, the

```
function alignASGandFSGrec($asg, v_{asg}, fsg, v_{fsg}, subgraph_{\text{com}}$)
  // Inspect outgoing edges
  for each $e_{\text{asg}} \in asg.\mathcal{E} \mid e_{\text{asg}}.source = v_{asg}$
    if $\exists e_{\text{fsg}} \in fsg.\mathcal{E} \mid (e_{\text{asg}}.cc = e_{\text{fsg}}.cc)$
       $\wedge(e_{\text{fsg}}.source = v_{fsg})$
       $\wedge($strandProperties$(e_{\text{asg}}.target) =$
         strandProperties$(e_{\text{fsg}}.target))$
      // 1) Does target exist in the common subgraph
      $targetExists \leftarrow (\exists v \in subgraph_{\text{com}}.\mathcal{V} \mid v = e_{\text{asg}}.target)$
      // 2) Does the edge exist in the common subgraph
      if $\nexists e_{\text{com}} \in subgraph_{\text{com}}.\mathcal{E} \mid e_{\text{com}} = e_{\text{asg}}$
        $e_{\text{asg}}.cc_{\text{classified}} \leftarrow$
          aggregateLabelEvidence$(e_{\text{asg}}.cc_{\text{classified}}, e_{\text{fsg}}.cc_{\text{classified}})$
        if $\neg targetExists$
          $subgraph_{\text{com}}.\mathcal{V} \leftarrow subgraph_{\text{com}}.\mathcal{V} \cup \{e_{\text{asg}}.target\}$
        end
        $subgraph_{\text{com}}.\mathcal{E} \leftarrow subgraph_{\text{com}}.\mathcal{E} \cup \{e_{\text{asg}}\}$
      end
      // If target was not present, recurse
      if $\neg targetExists$
        alignASGandFSGrec($asg$,$e_{\text{asg}}.target$,$fsg$,$e_{\text{fsg}}.target$,$subgraph_{\text{com}}$)
      end
  end
```

Algorithm 14: Recursive alignment of the aggregated strand graph and folded strand graph for computing a common subset of both graphs

vertices forming the anchor points into the ASG (v_{asg}) and FSG (v_{fsg}) are selected from both graphs.

The algorithm follows the chain of *outgoing* edges inside the ASG, starting from the given entry pointer, to explore both graphs (line 3). Whenever an edge from the ASG is inspected, a lookup of a corresponding edge in the FSG is performed (line 4). The first test ensures that the connection configurations of both edges in the ASG and FSG are equal. Thus the algorithm again relies on the quantification of strand connections as discussed in Ch. 5. Further, the target vertices of the edges need to have the same strand properties, i.e., the same linkage offset and cell types.

If the criteria are matched, line 6 tests whether the target vertex already exists inside the common subgraph ($subgraph_{\text{com}}$) that contains the commonalities between the FSG and the ASG. In line 8, the presence of the edge inside the common subgraph is tested, in order to prevent multiple aggregations of the same edge. If the edge does not exist, yet, the evidences on the ASG edge is aggregated with the new evidence from the FSG edge and gets stored inside the ASG (line 9). Hence, line 9 computes the actual temporal repetition. Then, the target and the edge are recorded in the common subgraph (lines 11 and 13). Finally, if the target of the edge is not yet present inside of the common subgraph, i.e., has not been visited before, the algorithm recurses to continue its exploration from the target in line 17. Thus, the algorithm performs a depth first search. Note that the search is continued on the current target vertices of the ASG and FSG. The difference between the ASG and FSG is computed upon the result of the previous calculation. The difference corresponds to the new elements in the FSG, as described in Sec. 6.1.3.

6.1.3 Finding differences between ASG and FSG

In Alg. 15 the difference between the ASG and the FSG is recursively calculated upon the common subgraph ($subgraph_{\text{com}}$), which represents the common elements in both graphs. By comparing the FSG with the common subgraph, the elements new to the FSG are detected and stored into a subgraph containing the new elements ($subgraph_{\text{new}}$).

More precisely, the exploration of the FSG starts from the ep and follows only *outgoing* edges, analogously to the previous alignment between ASG and FSG (line 2). This can be seen in line 17 of Alg. 13, where ep is passed into the function as a starting point together with the current FSG ($\mathcal{F}_{\text{i}}$). Additionally the previously discussed common subgraph and graph for holding the new elements is computed. Additionally a set is passed into the function, which contains the visited vertices to account for cyclicity (bc).

In lines 3-11, the source and target vertices and the edge of the FSG as seen by the ep are added to the subgraph containing the new elements, provided they where not yet present. In line 12-15, the algorithm recurses if no cyclicity is de-

```
function graphDiffRec($fsg, v_{fsg}, subgraph_{\text{com}}, subgraph_{\text{new}}, bc$)
  for each $e_{\text{fsg}} \in fsg.\mathcal{E} \mid e_{\text{fsg}}.source = v_{fsg}$
    if $\neg$graphContainsVertex($subgraph_{\text{com}}, e_{\text{fsg}}.source$)
      $subgraph_{\text{new}}.\mathcal{V} \leftarrow subgraph_{\text{new}}.\mathcal{V} \cup \{e_{\text{fsg}}.source\}$
    end
    if $\neg$graphContainsVertex($subgraph_{\text{com}}, e_{\text{fsg}}.target$)
      $subgraph_{\text{new}}.\mathcal{V} \leftarrow subgraph_{\text{new}}.\mathcal{V} \cup \{e_{\text{fsg}}.target\}$
    end
    if $\nexists e_{\text{sub}} \in subgraph_{\text{com}}.\mathcal{E} \mid e_{\text{sub}} = e_{\text{fsg}}$
      $subgraph_{\text{new}}.\mathcal{E} \leftarrow subgraph_{\text{new}}.\mathcal{E} \cup \{e_{\text{fsg}}\}$
    end
    if $\nexists v \in bc \mid v = e_{\text{fsg}}.target$
      $bc \leftarrow bc \cup \{e_{\text{fsg}}.target\}$
      graphDiffRec($fsg$, $e_{\text{fsg}}.target$, $subgraph_{\text{com}}$, $subgraph_{\text{new}}$, $bc$)
    end
  end
```

Algorithm 15: Computation of new elements inside of the folded strand graph

tected. The exploration continues on the target vertex of the currently processed outgoing edge ($e_{\text{fsg}}.target$).

Because the FSG can potentially contain more elements than are visible from the point of view of the ep, a simple difference between the FSG and the common subgraph might result in too many elements being considered as being new. The resulting graph containing the new elements is subsequently used to extend the ASG. The last step is discussed in Sec. 6.1.4.

6.1.4 Extending the ASG with new elements

The ASG is extended with the new elements by merging both the vertices ($asg.\mathcal{V} \leftarrow asg.\mathcal{V} \cup subgraph_{\text{new}}.\mathcal{V}$) and edges ($asg.\mathcal{E} \leftarrow asg.\mathcal{E} \cup subgraph_{\text{new}}.\mathcal{E}$) of the two graphs. This is shown in Alg. 16, where each edge of the subgraph containing the new elements is iterated in line 2. The source and target vertices are only added, if they are not present inside of the ASG before (lines 3-8). Because the subgraph only contains edges that have not been present before, each edge (e_{new}) can always be added to the ASG (line 9).

6.2 Summary

In this chapter, the temporal repetition algorithm was discussed. The algorithm explores how a data structure, as seen by a particular entry pointer, evolves over time by aggregating all FSGs seen by an entry pointer into an ASG. In addition, the evidence counts of the FSGs are accumulated over time, thus reinforcing the evidences. The benchmarking of the algorithm is done both in the benchmark of

```
function CALCULATEASG($asg, subgraph_{\text{new}}$)
  for each $e_{\text{new}} \in subgraph_{\text{new}}.\mathcal{E}$
    if $\neg$GRAPHCONTAINSVERTEX($asg, e_{\text{new}}.source$)
      $asg.\mathcal{V} \leftarrow asg.\mathcal{V} \cup \{e_{\text{new}}.source\}$
    end
    if $\neg$GRAPHCONTAINSVERTEX($asg, e_{\text{new}}.target$)
      $asg.\mathcal{V} \leftarrow asg.\mathcal{V} \cup \{e_{\text{new}}.target\}$
    end
    $asg.\mathcal{E} \leftarrow asg.\mathcal{E} \cup \{e_{\text{new}}\}$
  end
```

Algorithm 16: Calculation of additions to the aggregated strand graph

DSI working on source code in Ch. 8 and DSIbin working on binaries in Chs. 15 and 17 in Pt. II of this dissertation. The pseudo code in the current chapter was more driven by a programmers notation, resembling the DSI source code published online [31].

7 Parallelization

The initial DSI approach is designed as a pipeline, i.e. it is sequential. Therefore, the first DSI implementation has also been sequential, in order to focus on the correctness of the approach and avoid being side-tracked by premature optimization [77]. With the correctness of the results validated in [107] (see also Ch. 8), the focus has been shifted to cover speed improvements.

This chapter first analyses the pipelined DSI approach for parallelization potential. Then, a hot-spot analysis is discussed to evaluate which parts of the theoretically discussed computation steps take longest to execute. Based on the outcome of the hot-spot analysis, a parallel implementation of DSI is presented in Sec. 7.2 and benchmarked in Sec. 7.3.

7.1 Possibilities for parallelization

This section discusses the individual steps of the DSI pipeline with regards to speed improvement potential. This section is a theoretical discussion of parallelization possibilities, together with empirical data which steps of the pipeline are actually worth while to parallelize.

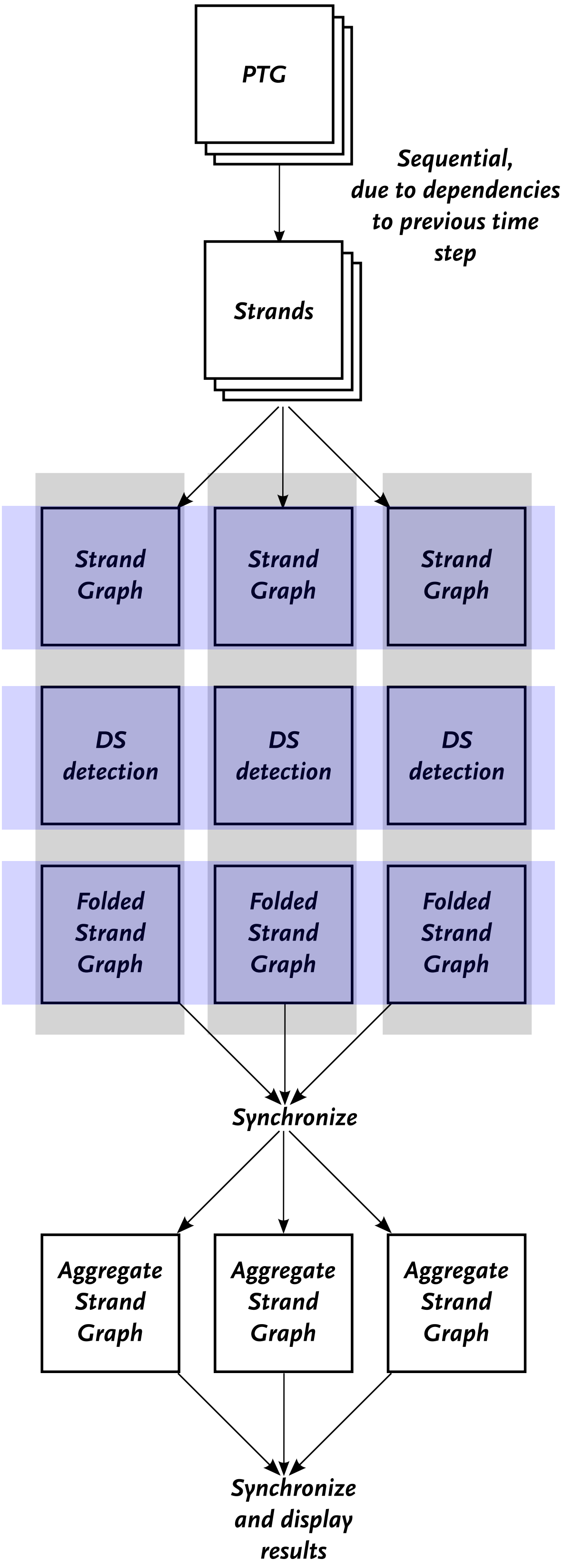


Figure 7.1: Overview of DSI with respect to sequential/parallel execution.

Event trace parsing. DSI starts out by parsing the event trace stored as an XML file. Because traces can be long, parallel parsing of the XML file could be an option. This is inspired by techniques such reported in [111], which parallelizes XML parsing at arbitrary points inside an XML file. Due to the simplicity of the XML Schema Definition (XSD) used by DSI, i.e., just storing a sequence of `event` elements, parallelization could be achieved by jumping to arbitrary locations inside the XML file and then search for the next begin/end of an `event` element. Thus, it would be possible to split the XML into individual *valid* XML file chunks for parallel processing, which is different from [111] where the parsing of the individual chunks can start at an arbitrary position. The latter does not guarantee that the parser starts at a valid element boundary, requiring additional complexity for the parser. As `event` elements are short, the preprocessing for creating valid XML slices that are parsable by a standard XML parser is not too time consuming.

PTG creation. The next step is the creation of a sequence of PTGs, with one PTG representing the memory state per event. The algorithm for calculating the memory state depends on the previous time step. This does not hinder the parallelization in principle, but requires an extensive sequential post-processing step where the information of each parallelized part needs to be distributed to its successor.

Strand creation. On top of the PTGs, the strands are computed for each time step, where information from time step $t_{\text{n-1}}$ is required for time step t_n. Thus, the strand creation suffers from the same problem as PTG creation regarding parallelization. However, it is possible to turn strand creation into a producer consumer problem, where PTG creation is the producer and strand creation is the consumer. This allows us to parallelize these two processing steps, where the strand creation will, at best, be only one time step behind PTG calculation.

SG creation, DDS detection and FSG computation. The computation of the following three steps of the analysis have in common that they only require the information about the current time step on which they are operating. In particular, these steps are: (i) the creation of the SG, where the strands and their interconnections are calculated; (ii) the detection of the DDS based on the SG; (iii) the structural repetition performed on the decorated SG from (ii). The last step, i.e., Step (iii), creates the FSG. Note that Steps (ii) and (iii) require information from their respective predecessor step. This either requires a sequential execution of the steps or a producer consumer pattern among them.

The parallelization for Steps (i)-(iii) can primarily be done along two axes which are discussed in the following. Both options are shown in Fig. 7.1, with the horizontal parallelization shown with blue and the vertical parallelization shown with gray shaded boxes. The first option parallelizes each step individually. The second option parallelizes each Step (i)-(iii) as one computational unit. As already mentioned, it is also possible to add a producer consumer pattern into the parallelization, which allows us to parallelize on both axes, but also increases the complexity of the scheduling and the implementation.

The vertical parallelization seems to be the most promising, as the whole pipeline for one time step can be computed independently with no need for further synchronization or additional setup of threads.

When looking at the horizontal parallelization, i.e., the parallelization of each of the individual steps, Steps (i)-(iii) need to be discussed individually. This is done in the following.

Step (i): SG computation. For computing the SG, strand combinations need to be inspected for connections between them. This problem is embarrassingly parallel, because all the strand combinations can be computed independently. The results only need to be collected with a barrier and aggregated into the final SG.

Step (ii): Naming phase. For detecting the DDS the SG from Step (i) is explored exhaustively, starting from one edge. Once all edges belonging together are found, those are sequentially tested against a DDS taxonomy. All processed edges are removed from the graph once they are classified. Thus, a parallel processing would need to synchronize the exploration of the graph in order to avoid conflicts when processing the edges.

The classification of the edges according to the taxonomy could indeed be run in parallel. However, this would require a speculative processing of all edges in parallel, as described in the following with the best and worst case scenarios. For both cases the classification runs in parallel, i.e., each label of the DDS taxonomy is tested against the SG by an individual process. In the best case, the result for computing the label with the highest priority is finished first and gives a match. In this case the algorithm could discard the remaining parallel computations. In the worst case, one would need to wait for all levels of the hierarchy to finish before being able to make a decision. Both collecting the edges and classifying them thus requires substantial synchronization effort, which makes the naming phase less attractive for parallelization.

Step (iii): FSG computation. Folding takes place when two child elements of the SG have the same type and have a common parent. The folding is done exhaustively until no more options for folding are present in the SG. The exploration of finding folding opportunities can be carried out in parallel, because the graph only needs to be read in parallel, i.e., no synchronization is required. This allows for parallel graph exploration. When it comes to the actual folding, things become more complicated, as the graph then would be manipulated in parallel. This requires higher overhead for keeping the graph in sync, similar to Step (ii).

In summary Step (i) is suited for parallelization, as the parallel distribution of the computations is straightforward. The result only needs to be collected with the help of a barrier. Steps (ii) and (iii) are parallelizable in principle, though the overhead for a parallel version of the algorithms is much more complex and requires more synchronization between threads to allow for parallel manipulation of the graph. Thus, the vertical parallelization is favoured over the horizontal implementation. This allows for an embarrassingly parallel implementation of (i)-(iii) because one computational block is executed with one thread, i.e., each time step can be run in parallel. The final result of computing all three steps is then collected with the help of a barrier and passed to the final ASG computation.

ASG computation. The final phase of DSI's analysis is the ASG computation, i.e., the temporal repetition. This phase inspects the DDS from the point of view of an entry point over the lifetime of the entry pointer, i.e., all time steps of the analysis where the entry pointer is alive are processed iteratively. In each time step, the information of the FSG is accumulated into the current ASG, to arrive at the final ASG that represents the DDS.

The inspection of multiple entry points is embarrassingly parallel as no dependencies exist between the entry pointers. The previously discussed steps, e.g., SG computation, naming phase and FSG computation, conduct their computations on individual time steps and are independent from the other time steps. Instead, the temporal aggregation naturally depends on the results of all other time steps. Therefore, the ASG requires the results of the previous computation steps for all time steps. In particular, the ASG process the results sequentially and in ascending order according to the time steps.

Thus, one can put a synchronization point, i.e., a barrier, in front of the ASG computation that ensures that all results are available. Alternatively, a producer consumer pattern can be applied to process available time steps required for an individual ASG computation as soon as they are available. In this case, each thread computing the ASG for an entry pointer would wait until the required time step arrives. This requires more complex synchronization mechanisms to inform all waiting threads, which in turn need to decide if they newly arrived time step is the one the thread is waiting for, obeying the ascending order of the time steps. Additionally, the producer consumer pattern is only beneficial when enough processing units are available, i.e., enough threads and free Central Processing Unit (CPU) cores both for the producer *and* the consumer side.

Inspired by the synchronization overhead required by the ASG for ordering the computed results, it might be interesting to study the behaviour of the ASG, both conceptually and empirically, when it is applied on non sequential ordering. As the aggregation of the FSGs over time only ever adds elements to the ASG, it might be possible to aggregate the FSGs out of order and arrive at the same result as the in order processing. If out of order aggregation is possible, it could speed up the analysis because the individual results from the previous time step would not be required as one chunk, i.e., be collected with a barrier, but could be processed as they are finished by the individual threads. Of course, synchronization overhead increases as all ASG threads would need to be informed of newly arriving FSGs.

General parallelization. As the discussion shows, the ways to parallelize DSI are manifold in principle. However, it only makes sense to parallelize if enough idle threads are available to perform the tasks. Thus, overly complex parallelization approaches might simply suffer from the limited resources and are thus not worth the added complexity and synchronization efforts.

When looking at producer consumer patterns, threads that are not needed anymore by the producer can subsequently be used by the consumer, which would indeed be beneficial, e.g., the PTG creation phase with the subsequent strand creation phase. In other situations like with the individual parallelization of the SG creation, it is questionable if there are actually free threads available because the number of time steps for which the SG needs to be computed exceeds the number of available cores by far.

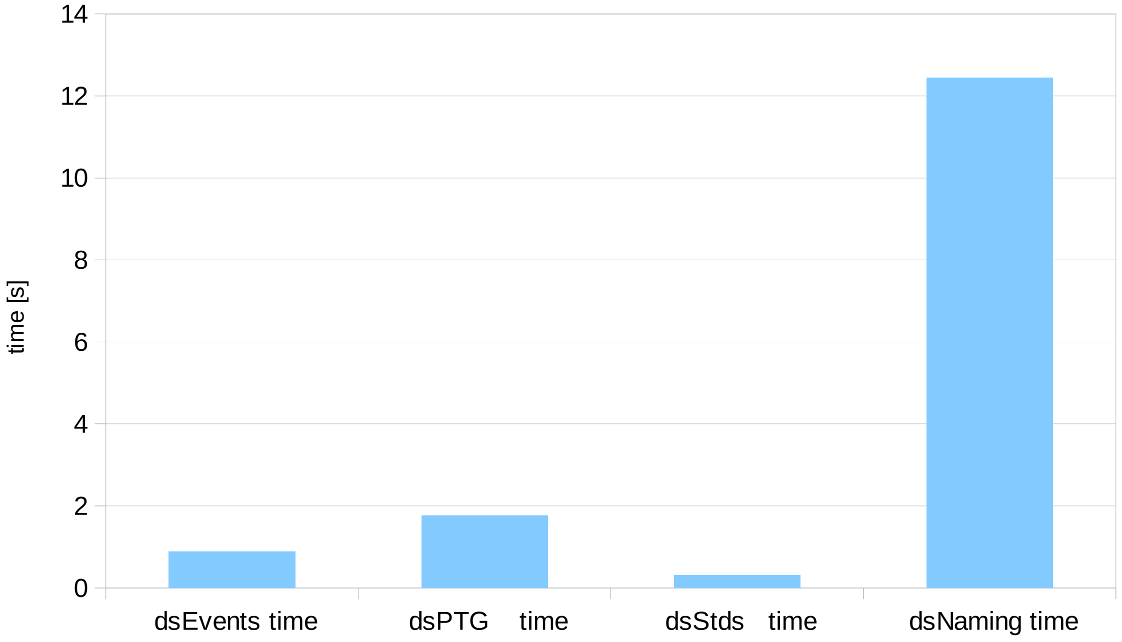


Figure 7.2: Hot-spot analysis of example binary-trees-debian (sequential execution).

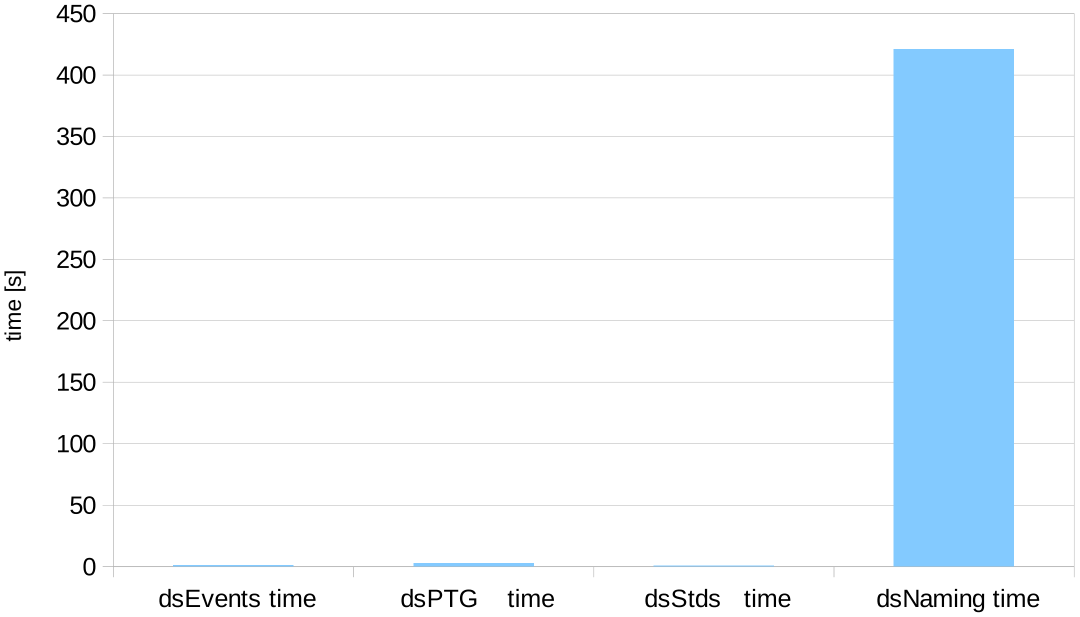


Figure 7.3: Hot-spot analysis of example five-level-sll-destroyed-top-down (sequential execution).

In the following, the considerations up to this point are accompanied by an empirical hot-spot analysis that quantifies which parts are worthwhile for a parallel implementation. The hot-spot analysis is conducted on the sequential implementation of DSI. The timings are done from a high level perspective, i.e., the total execution time of each computation step of DSI's pipeline was measured. Two examples were measured and averaged over five runs each. The timing results for the individual computation steps of the `binary-trees-debian` example are shown in Fig. 7.2 and those for the `five-level-destroyed-top-down` in Fig. 7.3. The timings are (i) for the parsing of the XML trace file (`dsEvents`), (ii) the creation of the PTGs (`dsPTG`), (iii) the creation of the strands (`dsStds`) and, (iv) the SG, detection of the DDS, FSG and ASG computation (`dsNaming`) as one timing block. Timing block (iv) is chosen as one block as all the steps together form the core of DSI's analysis. The steps are later on split up timing wise when conducting the benchmark of the parallelization in Sec. 7.3. The results reveal that the time consuming part is indeed part (iv). This part of the analysis contains the most complex calculations, e.g., the creation of the various graphs (SG, FSG and ASG) and the labeling of the DDS. As part (iv) consumes over 80% of the computation time in both examples, it is the hot-spot of DSI's analysis and thus is parallelized first by us.

7.2 Parallel implementation

The hot-spot analysis, together with the aforementioned possibilities how to parallelize DSI's pipeline approach, result in the described parallel implementation. DSI is written in Scala, which runs on top of a Java Virtual Machine (JVM). Scala offers *Futures* for parallelization. The code sections that should be parallelized are encapsulated by a Future. All data required by the Future can be passed, e.g., by reference. The created Futures are then processed by a set of worker threads. The amount of workers can be controlled both programmatically and via command line parameters to the JVM.

As has been stated previously, the pipeline approach of DSI is well suited for parallelization after the strands are created, i.e., starting from the SG computation. From this point onwards, each time step can be processed independently up to the point of the ASG creation, which requires all results of the previous time steps. To avoid the overhead of generating and synchronizing a set of Futures three times, i.e., the horizontal parallelization, the vertical parallelization is implemented. This allows a worker thread to execute all three steps within one Future. The results of the Futures are then collected by setting up a barrier. Once all time steps are collected, a new Future is spawned off for each entry point to compute the ASG. The results are again collected with a barrier to have them available simultaneously once all ASG computations have finished.

All data required by the Futures is held in program memory, thus no disk-I/O is performed. The Futures also don't write their results onto the disk, or perform other blocking I/O operations. Thus it is best to utilize as many threads as cores are available [7].

7.3 Benchmarking

To show the effect of the implemented parallelization, a benchmark is conducted on two machines: (i) *Laptop*: DELL Precision M4800 with one Intel(R) Core(TM) i7-4800MQ CPU @ 2.70GHz and 32GB of RAM; (ii) *Compute Box*: HP ProLiant DL580 with 8 x E7520 Processor CPU @ 1.86GHz and 128GB of RAM. The tests are conducted on two different machines to compare the various hardware characteristics and in how far DSI benefits from them. For the laptop, the employed software is Ubuntu 14.04, Scala in version 2.11.0 and OpenJDK version 7. The compute box uses the same Scala and OpenJDK version, but runs on Debian 7.

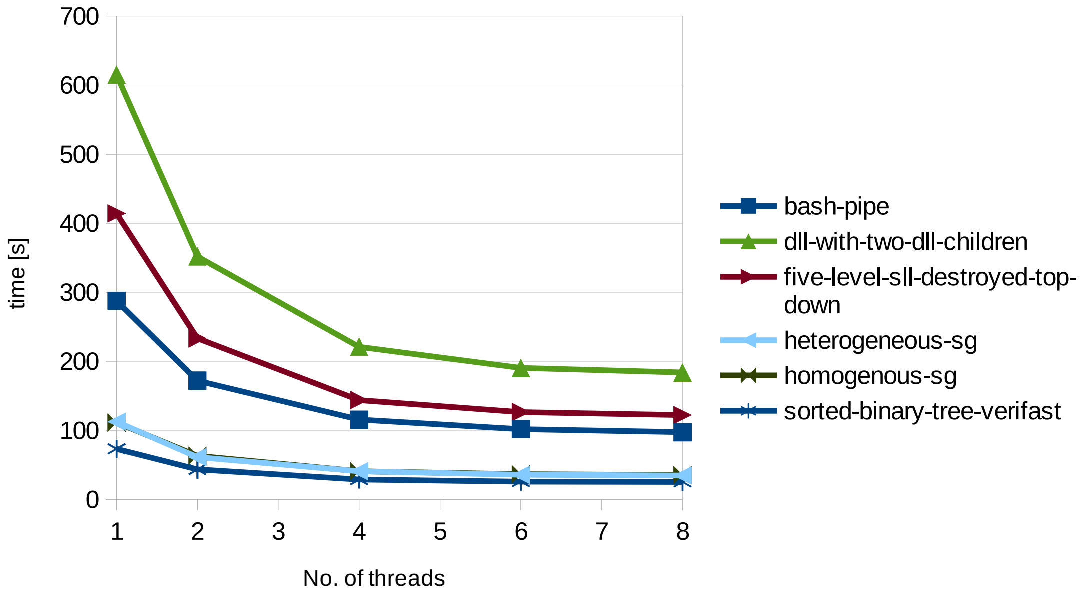


Figure 7.4: Laptop: Total runtimes for long running examples.

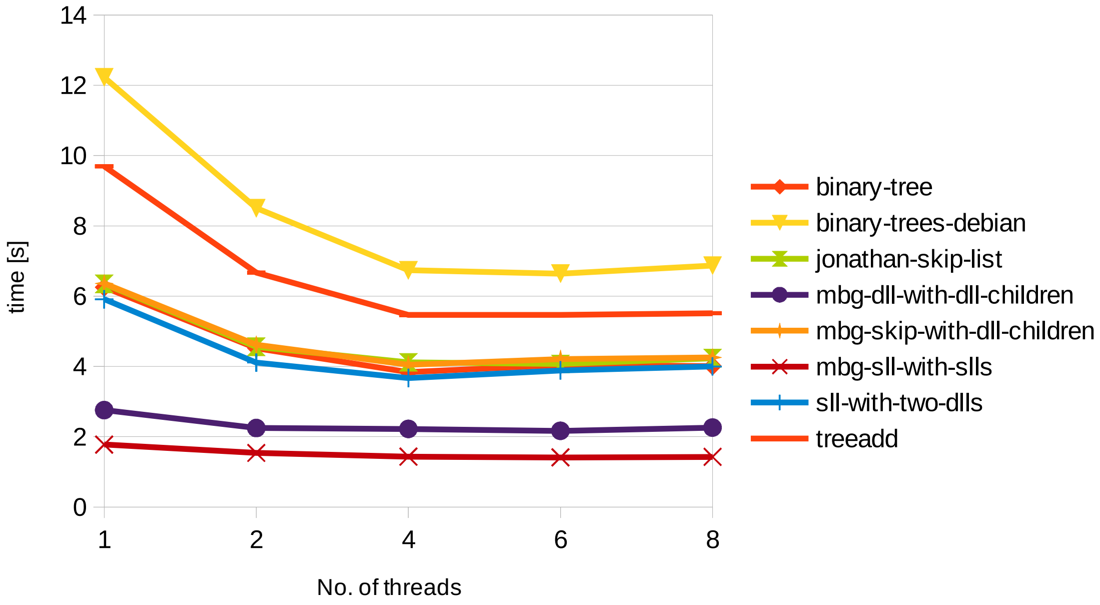


Figure 7.5: Laptop: Total runtimes for short running examples.

The benchmark is composed of long running examples (total execution time >90s) and short running examples (total execution time <12s). The long running examples include a real world example CDLL (`bash-pipe`), two examples from the literature: a five level parent child nesting with SLLs (`five-level-destroyed-top-down`) and a sorted BT (`sorted-binary-tree-verifast`), and three synthetic examples. The three synthetical examples include a DLL with two nested DLL child elements (`dll-with-two-dll-children`) and two examples where the SGs are homogeneous, i.e., the strands are all of the same type, (`homogeneous-sg`) and heterogeneous, i.e., the strands are of different types, (`heterogeneous-sg`).

The short running examples include BTs implementations taken from the real world (`binary-trees-debian`), a benchmark (`treeadd`) and a synthetic handwritten example (`binary-tree`). The SLs implementations are composed of an example from the literature (`jonathan-skip-list`) and a synthetic example (`mbg-skip-with-dll-children`). Additionally, synthetic examples comprised of various combinations of (nested) SLLs and DLLs are part of the short running examples. Thus, the selection of the benchmark covers realistic DDSs both with and without nesting scenarios. The different runtimes of the examples demonstrate, whether the parallelization scales for both situations.

7.3.1 Laptop

This section discusses the timing results computed on the laptop. All examples have been averaged over ten runs. For all examples, the disk-I/O is disabled to measure the computational time only. The disk-I/O of the results is done sequentially, but is embarrassingly parallel, given parallel storage devices.

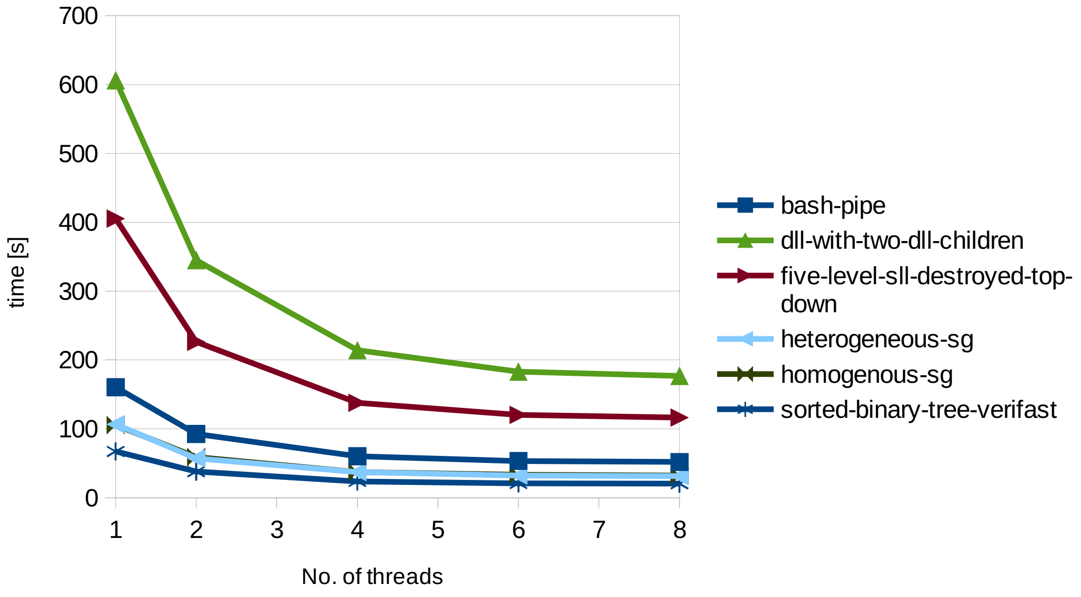


Figure 7.6: Laptop: Runtimes for strand graph, DS detection and folded strand graph creation for the long running examples.

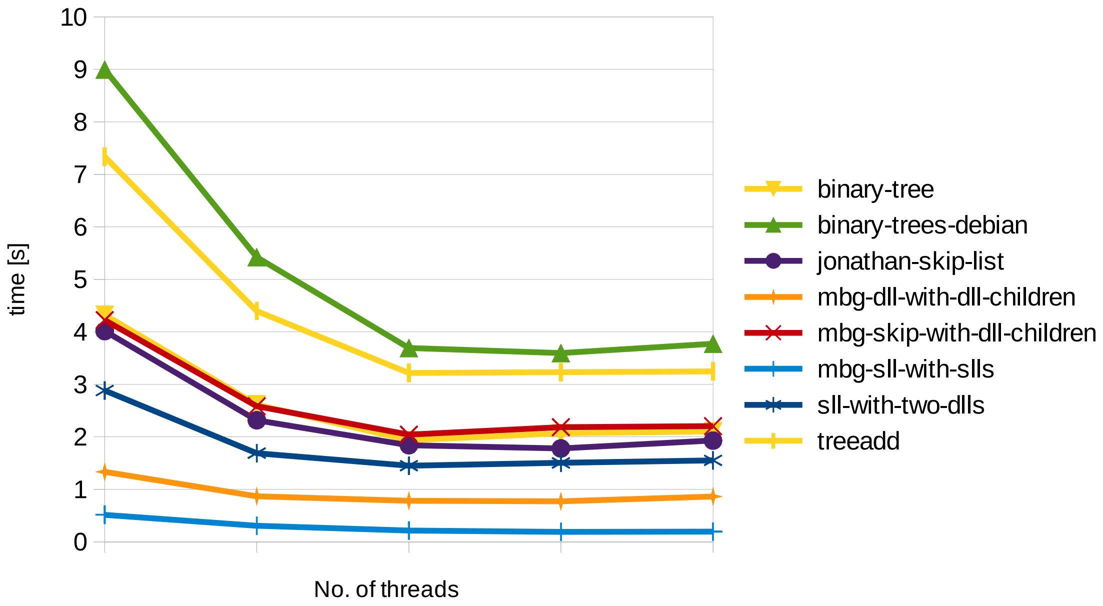


Figure 7.7: Laptop: Runtimes for strand graph, DS detection and folded strand graph creation for the short running examples.

The benchmark on the laptop shows that the total runtime of the examples indeed scales well for the long running examples as seen in Fig. 7.4. The figure shows the number of threads used for the `ForkJoinPool` on the x-axis and the runtime in seconds on the y-axis. As the laptop has a quad-core CPU with Hyper Threading (HT), up to eight threads are evaluated, i.e., the maximum number of available cores for the operating system including HT.

The Futures of Scala are mapped onto the threads available via the `ForkJoinPool` of the JVM, which has been set statically to the number of desired threads[1]. That the specified number of threads is obeyed by the JVM is manually verified with `htop`, which shows the number of running threads.

[1] This can be done via a command-line parameter passed to the JVM. The parameter used to set worker threads is a follows: `-Dscala.concurrent.context.numThreads`

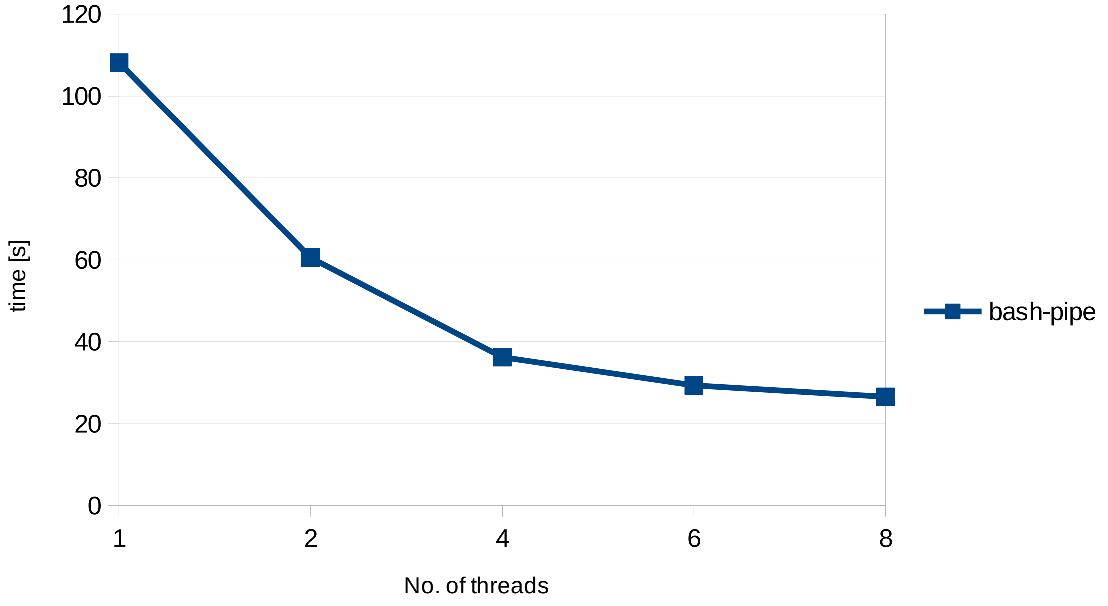


Figure 7.8: Laptop: Runtime for the longest running aggregated strand graph creation of the benchmark (`bash-pipe` example).

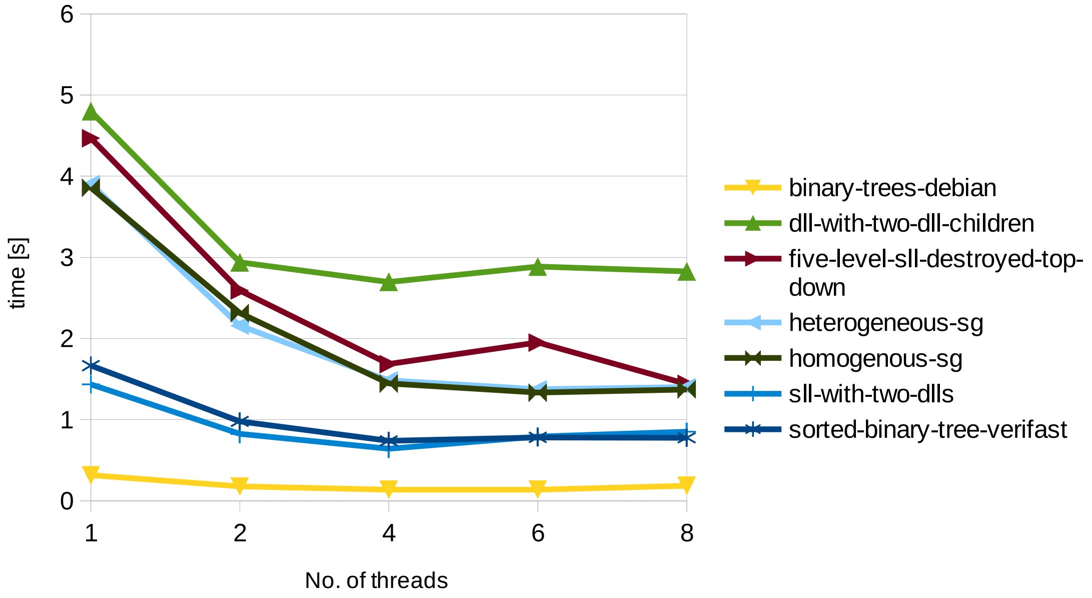


Figure 7.9: Laptop: Runtimes for short running aggregated strand graph creations. This chart combines both long and short running examples of Figs. 7.4 and 7.5; note that the aggregated strand graph runtimes are equally short for most of the examples.

The charts for the long running examples in Fig. 7.4 show performance gains. This indicates that the JVM indeed follows the thread requests as listed by `htop`. The performance gain is higher when operating on the real cores, which are four. The execution time is nearly cut in half, when doubling the amount of used cores. There still is a performance gain when HT [84] comes into play, but the improvements are naturally less than with a dedicated physical CPU core.

When considering the short running examples shown in Fig. 7.5, the same as for the long running examples is true for the physical cores. However the performance either stagnates or even gets worse when using HT. This is not surprising, as the run times are quite short; thus, the overhead for setting up the parallel execution and considering the overhead for scheduling the threads cannot be compensated with the HT approach.

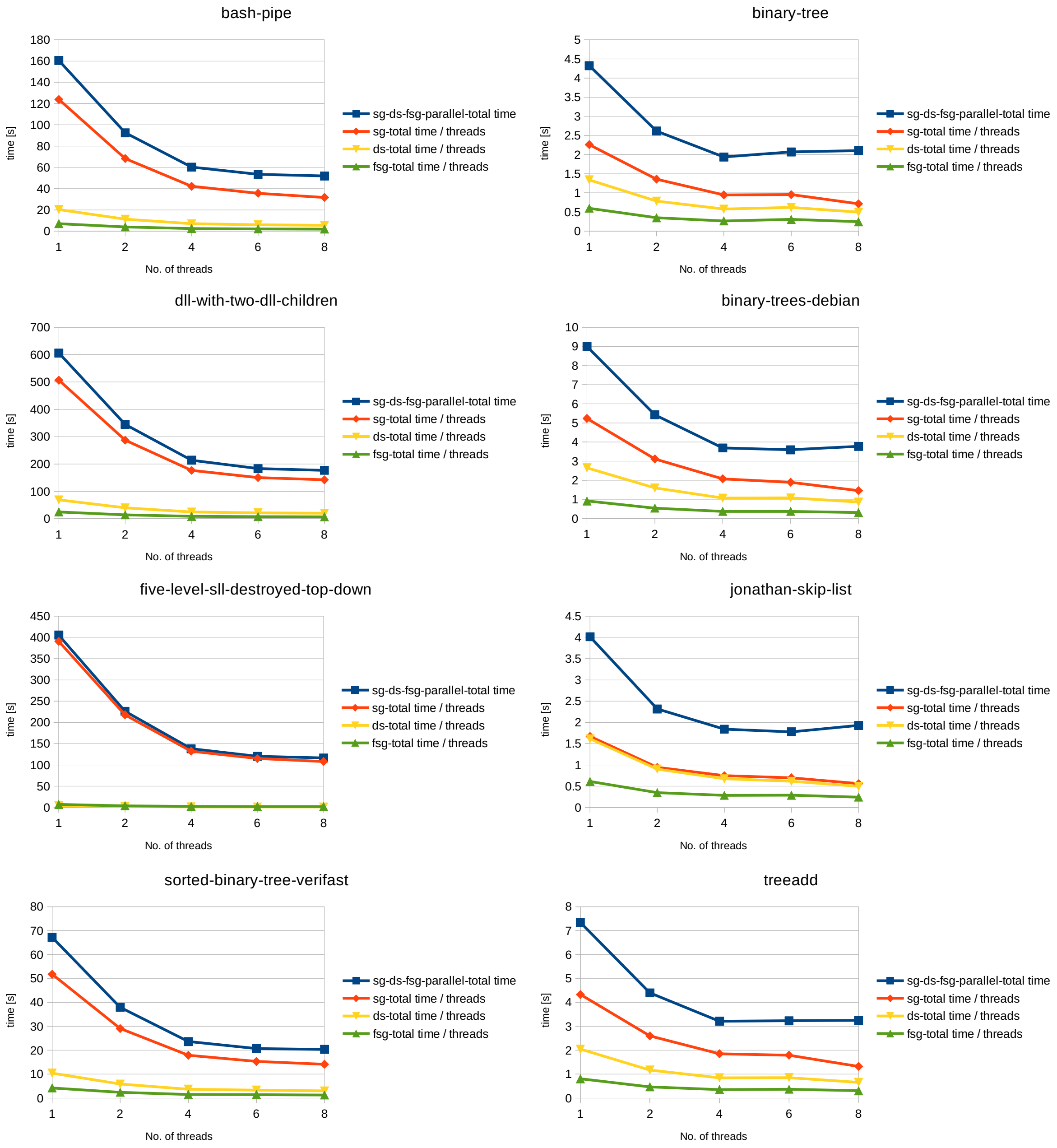


Figure 7.10: Laptop: Details of runtimes for the strand graph, DS detection and folded strand graph creation, both for a selection of long (left) and short (right) running examples.

The actual parallelized parts of the DSI approach are shown in Figs. 7.6- 7.9. Again the figures are split between long running and short running examples. Specifically the two parallelized blocks are shown as discussed previously, i.e., the parallelization of the SG creation, the DDS detection and the FSG creation as one parallelized block (see Figs. 7.6 and 7.7), and the ASG creation as the second parallelized block (see Figs. 7.8 and 7.9). Again, in both cases, the long running examples benefit from the parallelization even when HT sets in, but the short running examples stagnate and some even suffer from a performance loss. Note that only the `bash-pipe` example takes a significant amount of time for the creation of the ASG, which stems from the multitude of entry points in this example and the long trace. In general, the ASG creation is not too time consuming.

In order to get an even more fine grained view into the first parallelized block, the SG creation, the DDS detection and the FSG creation is timed individually. The run time consumed by each thread per computation step gets collected at the barrier, where the times are summed up and then divided by the number of threads, in order to arrive at the mean time for completing each step. The results are shown in Fig. 7.10. One can observe a couple of things from these graphs: (i) the most time consuming step is the creation of the strand graph where all strand combinations need to be processed in order to calculate the connections between the strands; (ii) the overall runtime of the short running examples does not improve or even gets worse when using more than four threads. However, the individual steps of the short running examples benefit from parallelization when eight threads are used, although the overall runtime increases. This leads to the assumption that the increased overhead for the scheduling between six and eight threads cannot be countered by the performance gains. Note, however, that the differences of the runtimes are in the range of half a second.

In order to show scalability, a second machine (HP ProLiant DL580) has additionally been used. The scalability of the implementation can be seen in Fig. 7.11, which shows the `bash-pipe` example with a stepwise increasing number of threads up to 64 threads. The limit of 64 threads correlates with the 32 physical cores of the machine multiplied by two for HT. The test in Fig. 7.11 is executed twice, as is explained below.

The timings shows that the approach also scales on the second machine, at least for the first 16 threads. After this, the performance does not improve, although 32 real cores are available and 64 cores with HT. When comparing the total run time of the example on the compute-box in Fig. 7.11 and on the laptop in Fig. 7.4, it is surprising that the laptop indeed outperforms the compute-box. This phenomenon has been observed independently from the measurements for the DSI project by other researchers using this machine. Besides the effects of the scheduling overhead, which will eventually slow down every parallelized application, the bus speed might be the bottle neck.

7.3.2 Computebox

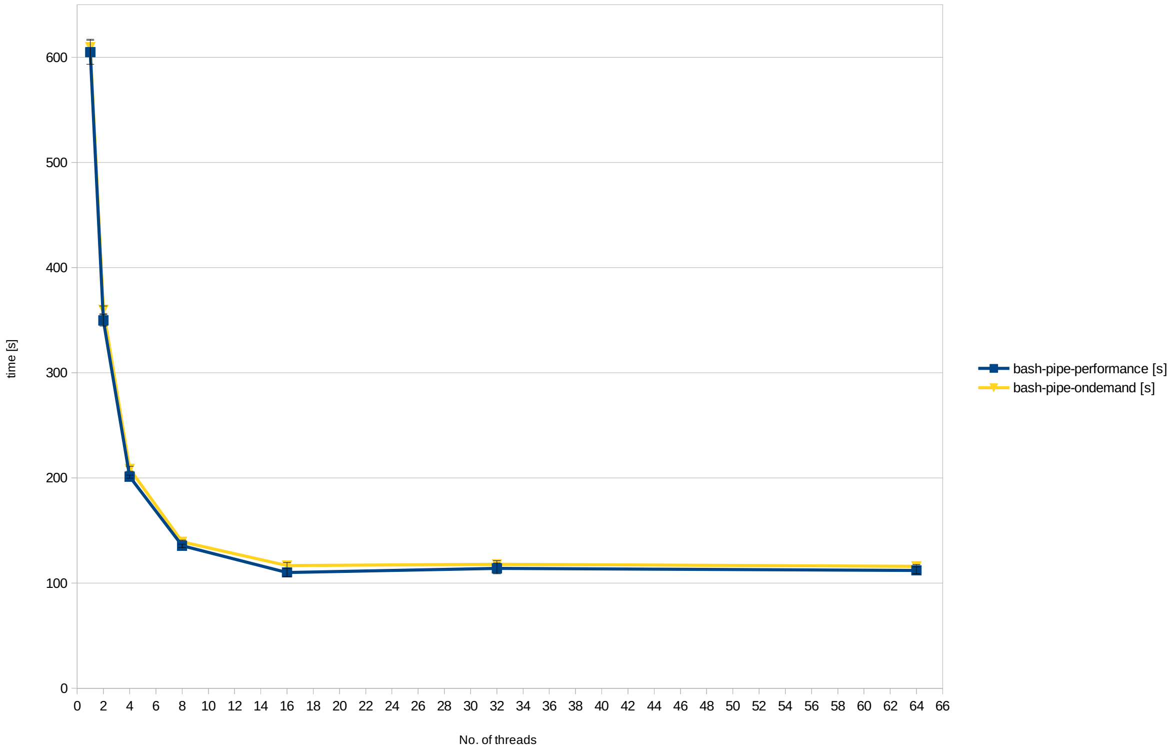


Figure 7.11: Computebox: Example `bash-pipe` with two governor strategies for scaling the CPU frequency: ondemand and performance.

Some attempts have been undertaken by the author of this dissertation to improve the performance of the computebox in the context of the DSI project, to counter the measured unimpressive performance of the computebox give the hardware of the machine (observed independently by others researchers as well, as discussed in the previous section). One was to set the governor of the Linux kernel that controls the CPU frequency from on-demand frequency scaling to full clock speed permanently[2]. Indeed, this gave some performance gains as seen in Fig. 7.11 which shows the execution times for the `bash-pipe` example with the Linux kernel governor set to `performance` and `ondemand`. The speed-up is not much, with the best performance gain of nearly 6% being at 16 cores, as seen in Fig. 7.12 where the execution times of the `ondemand` and `performance` strategy are correlated. Additional tests of pinning[3] the worker threads onto dedicated cores to

[2] Scheduling strategies can be influenced by setting either the `ondemand` or `performance` option in the Linux system configuration file: `/sys/devices/system/cpu/cpu*/cpufreq/scaling_governor`

[3] Information about the threads: `jstack -l `pgrep --exact java``; Pin threads to a CPU core: `taskset -c -p cpuid threadpid`

avoid scheduling overhead between the cores has also been performed, but did not reveal any performance gains; thus, no figures are shown.

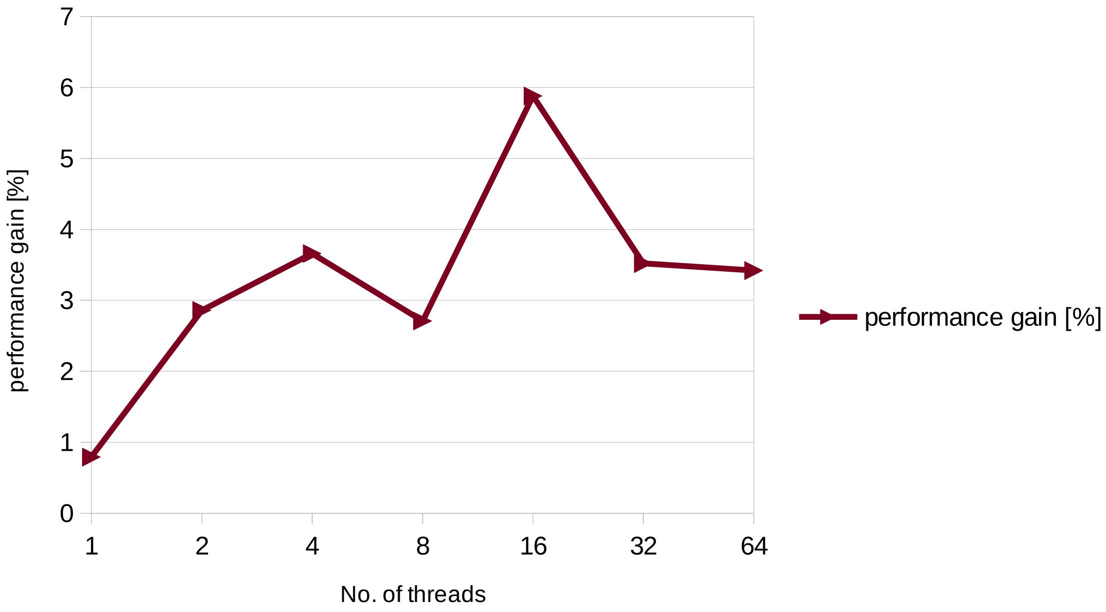


Figure 7.12: Computebox: Performance gains (in percent) for the `bash-pipe` example between the Linux kernel `performance` governor and the `ondemand` governor.

7.3.3 Optimizations

During the analysis of parallelization opportunities, it has become evident that the current FSG creation implementation can be optimized as it considers Entry Pointers (EPs) when performing the folding. In short, the folding is performed whenever two vertices are of the same type and are connected with the same strand connection configuration to a common parent vertex (called *three-vertices-condition*). The implementation checks for the first condition, i.e., equality of the vertices, and if true performs the check of the three-vertices-condition. When comparing two EPs, they are considered equal and thus trigger the three-vertices-condition check. However, two EPs can never be folded and therefore can be excluded from the analysis.

8 Benchmarking

With the theory of the DSI approach explained in the previous chapters, this chapter describes the benchmarking of the DSI prototype, which is implemented in Scala comprising ~10k LOC. The DSI benchmark is composed of real world examples, e.g., `bash`, `X.org` and `tsort`, examples taken from other benchmarks, e.g., [20], examples taken from the literature, e.g., from Predator[65] and Forester [72] and synthetic hand written examples. The benchmarking assembled for this dissertation contains examples from our previous publication [107], which have been conducted by the author of this dissertation. The benchmark is discussed in more detail than in our publication, and additional examples are added to the benchmark that where also evaluated by the author of this dissertation. The chosen set of examples covers a good variety of data structures and implementation techniques. In order to give an overview of how the benchmark is set up, the examples are categorized in Sec. 8.4 which also gives an overview of the additional examples added to the benchmark that where not part of [107].

The benchmark itself is discussed in Sec. 8.2, where certain aspects of the examples are highlighted. This includes more high level discussions of the usefulness of DSI when the readability of the source code is poor or the code base is simply huge and thus hard to comprehend. However, low level details are explained as well, such as comparing connections between strands and how they evolve over time in various nesting scenarios.

Further, the diversity of the benchmark also revealed limitations of DSI. Hence, these examples are discussed in more detail to motivate future work. Specifically, these real world examples are `X.org` [38] and `tsort` from `coreutils` [9].

Interestingly, one literature example shows that DSI cannot only be used as a program comprehension tool, but can be used as a tool for debugging as well. This turns DSI's idea of pure program comprehension of unknown code around. Instead, in the debugging scenario a priori knowledge about the example is available, but which does not match with the outcome of DSI. Thus, DSI is used to find the mismatch between intended and actual program behaviour, instead of comprehending unknown code. The exploration of the example is a case study how DSI performs in such a scenario and is detailed in Sec. 8.3.

All experiments were computed on an Intel i7-4800MQ with 32GB of RAM. The results take in the order of tens of minutes to compute. For a more in depth analysis regarding the timing behaviour of DSI, see Ch. 7.

8.1 Compiling the DSI benchmark

This section first discusses in Sec. 8.1.1 the various categories of the examples comprising our benchmark and, subsequently, gives a brief overview of the examples in Sec. 8.1.2 that are added within this dissertation but were not part of [107].

8.1.1 Categorization of examples

The benchmark for DSI is set up to cover a wide variety of DDSs and to maximize the variety of implementation techniques employed in practice. This is achieved by either testing the DDSs exclusively or by creating benchmarks with arbitrary combinations of the DDSs with multiple nesting scenarios. Hence, the benchmark comprises real world examples (e.g., libusb, bash, rosetta-dll, xorg, tsort and benchmarks: Olden/Debian), examples from the shape analysis literature (Predator/Forester [65, 72]) (**lit**) and from textbooks [104, 108] (**tb**) and synthetic self-written examples that test certain features of our approach not covered by others [65,69,72], e.g., nesting on overlay or skip lists with additional nested payload. The examples and results are shown in Tabs. 8.4, 8.5 and 8.6.

To minimize the bias in the synthetic examples, various implementation techniques are covered by also adding synthetic examples from other authors, namely from a master thesis [62] (`mt`) and a bachelor thesis [46] (`bt`) which are both conducted in the context of DSI and supervised by the author of this dissertation.

All examples can be classified first by their source (real world, `lit`, `tb`, `syn`, `mt` and `bt`), then the contained DDS, and finally by the categories of Tables 8.1 and 8.2. In the following, the purpose of the categories is discussed. The benchmark examples are classified according to the categories given in Tables. 8.1 and 8.2.

Nesting. This category aims at various nesting scenarios, such as arbitrary parent child combinations. The nesting is tested both for indirect and overlay nesting scenarios and with multiple numbers of children. Additionally, the parent and child DDS are varied.

(C)/(DS) implementation sophisticated. Two implementation aspects are shown by this category: (i) whether sophisticated C programming constructs are used (`C`), such as pointer casts, e.g., employed by the LKL, or placing multiple parts of one DDS inside a struct, simulating a CMA like behaviour; (ii) whether the DDS implementation itself is sophisticated (`DS`), such as a complex SL implementation.

Cyclicity. Cyclicity, states whether the example contains cyclic strands. This is important to study the strand handling of cyclic DDSs over their lifetime, as cyclic strands require dedicated handling in the DSI implementation. Additionally, DSI's nesting predicates are tested in case of cyclicity.

Table 8.1: Benchmarks (Part I) (see Table 8.2 for Part II of the benchmarks)

Example	**DS**	**Nesting**	**(C)/(DS) impl. sophisticated**	**Cyclicity**	**Limitations**	**(S)trands winding through vertex / Requires (C)ells**	**Base case strands**	**Short running**	**(M)acros/[no] (I)nterfaces**	**Stack only**
lit1	2 x DLL		DS						no I	
syn2	BT								I	
lit6	BT								I	
treeadd	BT								I	
treebnh	BT								I	
lit7	BT nst. ind. CSLL	✓	DS	✓					no I	
tsort	BT nst. ind./ovl. SLL	✓			✓				I	
syn1	CDLL		C/DS	✓		C		✓	M/I	✓
syn11	CDLL		C/DS	✓		C			M/I	✓
bash	CDLL			✓					I	
libusb	CDLL (nst. ovl. CDLL x 2)	✓	C/DS	✓		C			M/I	
tb2	DLL								I	
syn7	DLL								I	
lit5	DLL								I	
rosetta-dll	DLL		C						I	
mt1	DLL		DS					✓	I	
mt2	DLL		DS					✓	I	
bt11	DLL					C		✓	I	
bt12	DLL					C		✓	I	
bt13	DLL					C		✓	I	
bt14	DLL					C		✓	I	
bt15	DLL					C			I	
syn5	DLL nest. ovl. DLL	✓				C			no I	
xorg	Hash Map	✓			✓				I	

Table 8.2: Benchmarks (Part II) (see Table 8.1 for Part I of the benchmarks)

Example	**DS**	**Nesting**	**(C)/(DS) impl. sophisticated**	**Cyclicity**	**Limitations**	**(S)trands winding through vertex / Requires (C)ells**	**Base case strands**	**Short running**	**(M)acros/[no] (I)nterfaces**	**Stack only**
tb1	SLL						✓		I	
tb3	(SLL)						✓		I	
tb4	(SLL)						✓		I	
syn12	SLL					S/C	✓		no I	
bt1	SLL					C	✓	✓	I	
bt2	SLL					C	✓	✓	I	
bt3	SLL						✓	✓	I	
bt4	SLL						✓	✓	I	
bt5	SLL					C	✓	✓	I	
bt6	SLL					C	✓	✓	I	
bt7	SLL					C	✓	✓	I	
bt8	SLL						✓	✓	I	
bt9	SLL					C	✓	✓	I	
bt10	SLL						✓	✓	I	
lit4	SLL (3 x nst. ind. SLL)	✓							I	
lit3	SLL (nst. ovl. CDLL x 2)	✓	C	✓		C			M/I	
syn10	SLL (nst. ovl. SLL x 2)	✓							no I	
mt3	SLL ind. ovl. DLL	✓	DS					✓	I	
syn3	SLL nst. ind. SLL	✓	C			S/C		✓	no I	
syn8	SLL nst. ind. SLL	✓							no I	
syn4	SLL nst. ovl. SLL	✓	C			S/C		✓	no I	
syn9	SLL nst. ovl. SLL	✓							no I	
lit2	SLo2		DS						I	
syn6	SLo2 nst. ovl. DLL	✓							I	

Table 8.3: Additional benchmarks that test either previously untested DS or different programming constructs (tests where DSI detects the underlying DS but not it's semantics, e.g., stack or queue, are denoted with ✓)

Type/ID	**Named DS**	**Detected**	**Reason for test**
`lit5`	DLL	✓	break strands in the middle of DLL
`rosetta-dll`	DLL	✓	casts enclosing node to DLL node
`syn7`	DLL	✓	break strands in the middle of DLL
`syn8`	SLL + nest. SLL	✓	base case for nesting on indirection one child
`syn9`	SLL + nest. SLL	✓	base case for nesting on overlay one child
`syn10`	SLL + 2 x nest. SLL	✓	base case for nesting on overlay two children
`syn11`	CDLL	✓	stack based LKL through different sized nodes
`syn12`	SLL	✓	SLL (strand) with two nodes (cells) per vertex
`tb3`	SLL	(✓)	tests SLL with stack usage pattern
`tb4`	SLL	(✓)	tests SLL with queue usage pattern
`mt1`	DLL	✓	create different EPs
`mt2`	DLL	✓	handover DDS between EPs
`mt3`	SLL + nest. DLL	✓	create EP with partial view upon DDS
`bt1-10`	SLL	✓	various SLL implementations
`bt11-15`	DLL	✓	various DLL implementations
`lit6`	BT	✓	alternative BT implementation
`tsort`	BT + nest. SLLs	✗	BT implementation with sophisticated nesting showing limitations of DSI
`xorg`	Hash Map	✗	Hash Map implementation with arrays showing limitations of DSI

Table 8.4: Results obtained from the prototypical DSI implementation (Part I)

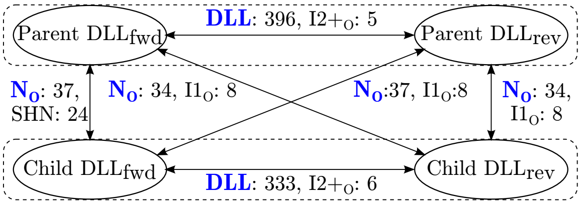
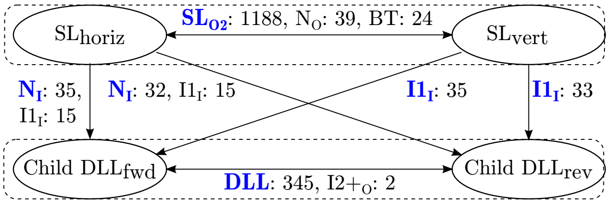
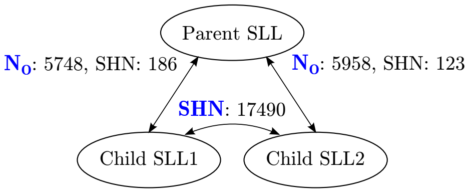
ID	Named DDS	Evidence count	% Supporting evidence
tb1 [104]	SLL	None/no SCs present	
tb2 [108]	DLL	**DLL**: 1440, I2+$_{\text{O}}$: 220	87%
tb3 [104]	SLL	**I2+$_{\text{O}}$**: 38, I1$_{\text{O}}$: 3	93%
tb4 [108]	SLL	**I2+$_{\text{O}}$**: 36, I1$_{\text{O}}$: 1	97%
syn1	CDLL	**CDLL**: 15, I2+$_{\text{O}}$: 10, DLL: 6	48%
syn2	Binary Tree	**BT**: 248, N$_{\text{O}}$: 6, I1$_{\text{O}}$: 3	97%
syn3	SLL + nest. SLL	**N$_{\text{I}}$**: 6, I1$_{\text{I}}$: 2	75%
syn4	SLL + nest. SLL	**N$_{\text{O}}$**: 10, SHN: 6	63%
syn5	DLL + nest. DLL	 Parent DLL$_{\text{fwd}}$ **DLL**: 396, I2+$_{\text{O}}$: 5 Parent DLL$_{\text{rev}}$ **N$_{\text{O}}$**: 37, SHN: 24 **N$_{\text{O}}$**: 34, I1$_{\text{O}}$: 8 **N$_{\text{O}}$**:37, I1$_{\text{O}}$:8 **N$_{\text{O}}$**: 34, I1$_{\text{O}}$: 8 Child DLL$_{\text{fwd}}$ **DLL**: 333, I2+$_{\text{O}}$: 6 Child DLL$_{\text{rev}}$	Avg. 84%
syn6	Skip List + nest. DLL	 SL$_{\text{horiz}}$ **SL$_{\text{O2}}$**: 1188, N$_{\text{O}}$: 39, BT: 24 SL$_{\text{vert}}$ **N$_{\text{I}}$**: 35, I1$_{\text{I}}$: 15 **N$_{\text{I}}$**: 32, I1$_{\text{I}}$: 15 **I1$_{\text{I}}$**: 35 **I1$_{\text{I}}$**: 33 Child DLL$_{\text{fwd}}$ **DLL**: 345, I2+$_{\text{O}}$: 2 Child DLL$_{\text{rev}}$	Avg. 89%
syn7	DLL	**DLL**: 273, I2+$_{\text{O}}$: 22, N$_{\text{O}}$: 2	92%
syn8	SLL + nest. SLL	**N$_{\text{I}}$**: 21, I1$_{\text{I}}$: 6	79%
syn9	SLL + nest. SLL	**N$_{\text{O}}$**: 236, SHN: 27	90%
syn10	SLL + 2 x nest. Same-head-node SLL	 Parent SLL **N$_{\text{O}}$**: 5748, SHN: 186 **N$_{\text{O}}$**: 5958, SHN: 123 **SHN**: 17490 Child SLL1 Child SLL2	Avg. 99%
syn11	CDLL	**CDLL**: 27, I2+$_{\text{O}}$: 21, DLL: 6	50%
syn12	SLL	None/no SCs present	

Table 8.5: Results obtained from the prototypical DSI implementation (Part II)

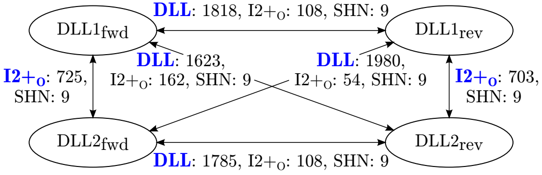
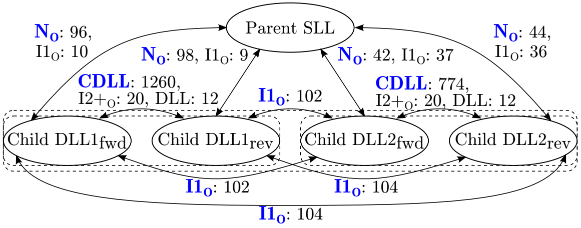
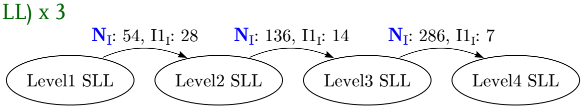
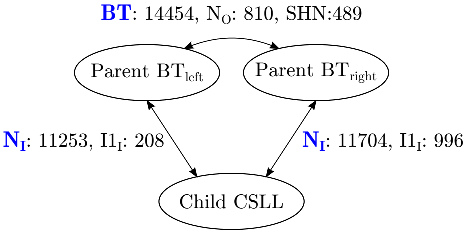
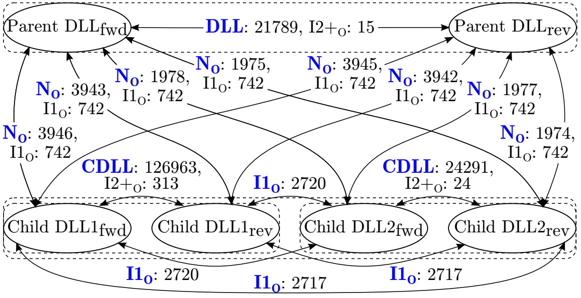
ID	Named DDS	Evidence count	% Supporting evidence
`lit1` [23]	DLL of (DLL, DLL) OR Intersecting(2xDLL)	 **DLL**: 1818, I2+$_O$: 108, SHN: 9 DLL1$_{fwd}$ DLL1$_{rev}$ **DLL**: 1623, I2+$_O$: 162, SHN: 9 **DLL**: 1980, I2+$_O$: 54, SHN: 9 **I2+$_O$**: 725, SHN: 9 **I2+$_O$**: 703, SHN: 9 DLL2$_{fwd}$ DLL2$_{rev}$ **DLL**: 1785, I2+$_O$: 108, SHN: 9	Avg. 96%
`lit2` [23]	SL	**SL$_{02}$**: 1242, BT: 72, N_O: 58, I1$_O$: 8, SHN: 3	90%
`lit3` [23]	SLL + nest. Intersecting(2x CDLL)	 Parent SLL **N$_O$**: 96, I1$_O$: 10 **N$_O$**: 98, I1$_O$: 9 **N$_O$**: 42, I1$_O$: 37 **N$_O$**: 44, I1$_O$: 36 **CDLL**: 1260, I2+$_O$: 20, DLL: 12 **I1$_O$**: 102 **CDLL**: 774, I2+$_O$: 20, DLL: 12 Child DLL1$_{fwd}$ Child DLL1$_{rev}$ Child DLL2$_{fwd}$ Child DLL2$_{rev}$ **I1$_O$**: 102 **I1$_O$**: 104 **I1$_O$**: 104	Avg. 88%
`lit4` [23]	SLL (+ nest. SLL) x 3	 **N$_I$**: 54, I1$_I$: 28 **N$_I$**: 136, I1$_I$: 14 **N$_I$**: 286, I1$_I$: 7 Level1 SLL Level2 SLL Level3 SLL Level4 SLL	Avg. 85%
`lit5` [23]	DLL	**DLL**: 22422, I2+$_O$: 1974, SHN: 30, N_O: 24, I1$_O$: 1	92%
`lit6`	BT	**BT**: 180, N_O: 96, I1$_O$: 15, SHN: 9	60%
`lit7` [23]	BT + nest. CSLL	 **BT**: 14454, N$_O$: 810, SHN:489 Parent BT$_{left}$ Parent BT$_{right}$ **N$_I$**: 11253, I1$_I$: 208 **N$_I$**: 11704, I1$_I$: 996 Child CSLL	Avg. 94%

Table 8.6: Results obtained from the prototypical DSI implementation (Part III)

ID	**Named DDS**	**Evidence count**	**% Supporting evidence**
`bash` [10]	CDLL	**CDLL**: 68529, I2+$_\mathrm{O}$: 72, DLL: 6	∼100%
`treeadd` [20]	BT	**BT**: 256	100%
`treebnh` [32]	BT	**BT**: 930	100%
`libusb` [13]	CDLL + nest. Intersecting(2x CDLL)		Avg. 88%
		 Parent DLL$_\mathrm{fwd}$ **DLL**: 21789, I2+$_\mathrm{O}$: 15 Parent DLL$_\mathrm{rev}$ **N$_\mathrm{O}$**: 3943, I1$_\mathrm{O}$: 742 **N$_\mathrm{O}$**: 1978, I1$_\mathrm{O}$: 742 **N$_\mathrm{O}$**: 1975, I1$_\mathrm{O}$: 742 **N$_\mathrm{O}$**: 3945, I1$_\mathrm{O}$: 742 **N$_\mathrm{O}$**: 3942, I1$_\mathrm{O}$: 742 **N$_\mathrm{O}$**: 1977, I1$_\mathrm{O}$: 742 **N$_\mathrm{O}$**: 3946, I1$_\mathrm{O}$: 742 **N$_\mathrm{O}$**: 1974, I1$_\mathrm{O}$: 742 **CDLL**: 126963, I2+$_\mathrm{O}$: 313 **I1$_\mathrm{O}$**: 2720 **CDLL**: 24291, I2+$_\mathrm{O}$: 24 Child DLL1$_\mathrm{fwd}$ Child DLL1$_\mathrm{rev}$ Child DLL2$_\mathrm{fwd}$ Child DLL2$_\mathrm{rev}$ **I1$_\mathrm{O}$**: 2720 **I1$_\mathrm{O}$**: 2717 **I1$_\mathrm{O}$**: 2717	
`rosetta-dll` [5]	DLL	**DLL**: 276, I2+$_\mathrm{O}$: 27, N$_\mathrm{O}$: 26, SHN: 12	81%
`bt1-10`	SLL	None/no SCs present	
`bt11`	DLL	**DLL**: 138, I2+$_\mathrm{O}$: 18	88%
`bt12`	DLL	**DLL**: 84, I2+$_\mathrm{O}$: 9	90%
`bt13`	DLL	**DLL**: 246, I2+$_\mathrm{O}$: 9	96%
`bt14`	DLL	**DLL**: 165, I2+$_\mathrm{O}$: 9	95%
`bt15`	DLL	**DLL**: 165, I2+$_\mathrm{O}$: 9	95%
`mt1`	DLL	**DLL**: 30	100%
`mt2`	DLL	**DLL**: 636, I2+$_\mathrm{O}$: 14	98%
`tsort`	BT with nest. ind. bidirectional SLL ambiguous nesting detected		-
`xorg`	Hash Map not detectable by DSI		-

Limitations. This category explicitly marks examples that show the limitations of DSI. By researching these examples, directions for future work are opened up, and ways of solving these limitations are discussed theoretically.

(S)trands winding through vertex / (C)ell abstraction. Two aspects are covered by this category: (i) the cell abstraction is required for detecting the DDS (`C`); (ii) the strand runs through one vertex multiple times (`S`). Both (i) and (ii) are tightly coupled as (ii) requires (i) to function.

Base case strands. This category covers the base cases for the strands, i.e., the tests only contain an SLL which is the basic building block of DSI. These examples show various usage patterns for the SLLs, such as different insertion patterns.

Short running. Short running examples are interesting for verifying whether the correct label is still detected in such situations, i.e., if the evidence reinforcement strategies of DSI allow for a quick convergence of the accumulated evidences.

(M)acros/ [no] (I)nterfaces. This category shows, whether the example uses macros (`M`) and whether the DDS is accessed via an interface (`[no] I`), e.g., functions for insertion/removal of elements into lists. Both aspects are a problem for related work such as DDT [74], which requires well defined interfaces.

Stack only. This category indicates, whether the DDS is placed completely on the stack, which mainly acts as a corner case to show that the allocation site is transparent to DSI.

Master and bachelor theses. This section offers some insight into the master [62] and the bachelor [46] theses that emerged within the DSI project and from which examples have been compiled in our benchmark. The examples used in both theses, i.e., `bt1-15` and `mt1-3`, were run through DSI. The DDS shape was given to the authors of the theses as a requirement, e.g., SLL, DLL or the specific shape of `mt3`, while the actual implementation was not enforced. This includes both the coding style and the way operations are implemented. The results obtained by running the examples through DSI were analysed by the author of this dissertation with regards to the correctness of the identified DDS. The results of DSI were then passed to the authors of both theses for input into their tool chain.

The master thesis [62], for which tests `mt1-3` were created, generalizes the temporal repetition of Ch. 6 to be able to match arbitrary FSGs and does not require FSGs to be consecutive. The main intention for the examples are situations where there exist multiple EPs that do not show the correct DDS label, but where it is possible to transfer the DDSs over time between various EPs to arrive at the correct

DDS interpretation. This is accomplished by applying a generalized graph matching between the various FSG seen by the different EPs.

Such a matching algorithm might be useful for increasing the performance of DSI by not considering every event, but instead sample at non-contiguous time points of an execution. For the core DSI implementation such tests are interesting, because they add more variety to the benchmark (`mt1-3`), which further reduces any bias in the test set and shows the generality of the approach, e.g., it is not bound to any specific ordering of DDS operations.

The bachelor thesis [46], for which tests `bt1-15` were created, automates the creation of annotations for data structures that are verifiable by VeriFast [73], a formal verification tool. Examples `bt1-10` employ different implementation techniques for an SLL. As SLLs are DSI's atomic building blocks, i.e., strands, those examples act as a base case benchmarks for DSI. The examples include, e.g., insertion of new nodes into an SLL at different positions, e.g., head or tail, which investigates DSI's strand detection. Further, DSI's cell detection is required by some of the examples, by placing the linkage struct inside a wrapper struct, making DSI's cell abstraction mandatory. Additionally, multiple DLL implementations are evaluated, which also use different implementation techniques such as a global pointer to the DLL head and single or double pointers, e.g., passed as references to the insertion function. In summary, the examples employ different DDS usage patterns and C implementation techniques, showing DSI's robustness to handle these situations.

8.1.2 Discussion of additional benchmarks

This section introduces additional examples added to the benchmark of this dissertation. It is mainly intended to clearly separate what has already been published, though the examples are discussed in more depth than in [107]. The examples are listed in Table 8.3, together with a short description for each example. This table accompanies the classification of the examples presented before.

8.2 The DSI benchmark

A full instrumentation is performed upon all examples, with the exception of `bash`, where only `array.c` and `xmalloc.c` are instrumented, and `xorg`, where only the subcomponent `div/resource.c` is analysed. The `bash` real world example uses a CDLL that is not a LKL. All examples focus on one main data structure for evaluation purposes; however, our approach is not limited to the detection of only one DDS.

We provide self-written drivers for the DDSs whenever required. The `bash` example is triggered with a piped command in the fashion of `ls | grep <pattern>` and the `libusb` example uses the `listdevs` utility provided with `libusb`. The examples exercise a great variety, e.g., SLs, (cyclic) lists, BTs and combinations of those with various nesting scenarios of the 18 memory structures of DSI's taxonomy, as shown in [107].

The results are summarized in Table 8.4, where **ID** gives a unique identifier for each test. `tb` stands for text-book, `syn` for synthetic and `lit` for literature used before. The real world examples each contain their name in the **ID** column. The column **Named DDS** shows the name of the DDS, according to the naming module as described in [107]; the naming module is not part of this dissertation. The column **Evidence count** gives the label together with the observed evidence counts for the strand connections over program execution. The highest ranked label found for the DDS is highlighted in **blue**. Whenever the structure is too complex to be described textually, the resulting ASG is shown, where vertices are strands and edges are SCs. Finally, column **% Supporting evidence** states the percentage between the correct interpretation, i.e., the stable shape, and the interpretations of the Degenerate Shape (DgS). In the following we will step through the table and discuss selected aspects of the benchmark.

8.2.1 First SLL and DLL examples

Example `tb1` is a SLL, and thus, corresponds to a strand. This is the minimal example that DSI can detect. As there are no other strands, no SCs and therefore also no evidence counts exist. More specifically, the insertions into the strand are always at the end of the strand, i.e., the strand is only extended without creating multiple overlapping strands during the operation. This can be observed when looking at example `tb2` that shows a DLL. The shape is correctly identified by DSI, as can be seen by the blue DLL label. However, the $I2+_O$ label is observed, which indicates that the DLL is manipulated, i.e., is in a DgS during DDS operations, where two strands exist that overlap at multiple intersection points but do not fulfill the DLL label.

8.2.2 Short running example

The lowest evidence count is encountered with example `syn1`, which is a self-written example utilizing the LKL. It only creates a short CDLL and then stops, thus lacking any usage of the DDS that is not the common case, as DDSs normally exist for a longer period and are not constantly in a DgS. However, it is still possible to detect the true CDLL shape of the LKL despite its short lifetime. Additionally, the example consists mainly of DDS operations, which force the DDS into DgSs but which does not hinder DSI to detect the correct DDS shape.

One might wonder, why a DLL label is observed without the cyclic property. This stems from the implementation of the insert operation, where the previous pointer of the first element is set first and the next pointer of the second element is set next. So when inserting into the empty list, i.e., only the head exists, the newly added element results in a DLL that subsequently is turned into a CDLL when setting the remaining previous and next pointers.

8.2.3 Skip list implementations

Example `syn6` exercises the skip list detection of DSI in combination with nesting. Skip lists are not covered by related dynamic analysis approaches [49, 69, 74], and the shape analysis literature that covers skip lists [65, 72] does not benchmark skip lists with nested payload. The example shows that the nesting relation does not interfere with the detection of the skip list and the DLL detection of the nested child elements. It also demonstrates that the quantification of the strand connections allows us to detect the nesting.

When considering the results of the `lit2` skip list example, one observes a multitude of DDS interpretations (BT, N_O, $I1_O$, SHN) that are all ruled out by the SL_{02} label. This stems from the complexity of the SL_{02} label according to DSI's taxonomy. Hence, the skip list label outnumbers the DgS interpretations by magnitudes when evidence counts are accumulated. The same behaviour is encountered with the `syn6` example.

8.2.4 Revealing unintended data structure semantics

`lit1` is interesting, because it is comprised of two DLLs that run in parallel inside one struct. As can be seen in the ASG, the DLL label is detected four times, which can be interpreted as a DLL of two DLLs. Interestingly it is possible to detect the DLLs in the various combinations of the forward and backward strands of the two DLLs. Additionally, an interpretation as two intersecting DLLs is possible. The multitude of interpretations are revealed by DSI but might not be immediately visible to the programmer.

8.2.5 Structurally complex examples

Both `libusb` and `lit3` use the LKL; they are the most challenging examples, as can be seen by the complex ASGs in Table. 8.4. Example `libusb` has an LKL as its parent list, while `lit3` has a SLL. Both examples have two LKL children, where the list heads are embedded inside the parent struct. Both examples are discussed in more detail in the following. In Sec. 8.2.5 DSI's usefulness is shown when the readability of the code is poor, as is the case with example `lit3`. Subsequently, in Sec. 8.2.5 DSI is discussed in the context of a huge code base, as is observed with example `libusb`.

Usefulness of DSI: Poor code readability

The code for `lit3` is shown in Fig. 8.1. We consider even this small example from [65, 72] hard to read, thus emphasizing the need for DSI's program comprehension capabilities. When using DSI the overall DDS shape becomes clear immediately, without studying the source code, just by looking at the graph for `lit3` in Table 8.4.

```c
/*
 * Linux kernel doubly linked lists
 *
 * boxes: test-0098.boxes
 */

struct my_item {
    struct list_head link;
    void             *data;
};

#define append_one(head)\
{\
    now = malloc(sizeof *now);\
    now->data = NULL;\
    lh2 = &now->link;\
    lh3 = (head)->prev;\
    __list_add(lh2, lh3, head);\
    lh2 = NULL;\
    lh3 = NULL;\
    now = NULL;\
}

struct master_item {
    struct master_item *next;
    struct list_head   nested1;
    struct list_head   nested2;
};


#define create_dll(dll)\
{\
    (dll)->prev = dll;\
    (dll)->next = dll;\
    append_one(dll);\
/* Manually controlled child elements */
    append_one(dll);\
\
}

#define create_sll_item(sll) {\
    pm = malloc(sizeof *pm);\
    pm->next = sll;\
    lh = &pm->nested1;\
    create_dll(lh);\
    lh = &pm->nested2;\
    create_dll(lh);\
    lh = NULL;\
    sll = pm;\
    pm = NULL;\
}

#define destroy_sll_item(sll) {\
    pm = sll;\
    sll = sll->next;\
    lh = &pm->nested1;\
    destroy(lh);\
    lh = &pm->nested2;\
    destroy(lh);\
    lh = NULL;\
    free(pm);\
    pm = NULL;\
}

int main()
{
    struct master_item *sll = NULL;
    struct master_item *pm;
    struct list_head *lh,*lh2,*lh3;
    struct my_item *now,*next;

/* Manually controlled number of sll items */
    create_sll_item(sll);
    create_sll_item(sll);

     return 0;
}
```

Figure 8.1: Source code excerpt of `lit3` [23] with the Linux kernel list macros and dead code sections removed. Manually adjusted number of elements are indicated by comments.

When consulting the code, this shape is much harder to infer manually. The inner workings of the macros are dominated by cryptic variable names, where the semantics can only be guessed, e.g., pm (line 67) or lh (line 68), and the lack of documentation. The (nested) usage of macros additionally refers to the variables similarly to global variables, because all of them are placed inside of main. All macros are also used inside of main, which makes reading the code even harder. One general problem when dealing with the LKL becomes evident with the structs my_item (lines 7-10) and master_item (lines 24-28), as it is not clear how the nested LKL structs are meant to be connected without looking at the code. This stems from the generality of the LKL that can link arbitrary elements. Therefore, this example also shows where the cell and strand abstraction is required to find the commonality when linking elements of different types, i.e., master_item elements with my_item child elements. For example, MemPick [69] cuts connections between nodes of different types, which will diminish the precision of its analysis in scenarios where the DDS is composed of nodes of different types.

The heavy macro usage of Lst. 8.1 is not only hard to read for the programmer, but even is a problem for related work, such as DDT [74], because it relies on well defined interface functions. For DSI, this is not a problem as such strong assumptions are not present. The topic of interface functions is also addressed within this dissertation when dealing with binaries in Pt. II (see Sec. 12.2); inlining performed by a compiler might prevent well defined interface functions.

Usefulness of DSI: Huge code base

Example libusb is a C user space library that provides access to USB devices [13]. Contrarily to example lit3 regarding code quality: the code is well structured and documented, but overwhelms the developer by its sheer length of about 7k LOC.

libusb allows for multiple simultaneous usages within one executable, by organizing multiple struct libusb_contexts in a LKL parent CDLL. This behaviour is simulated by us by modifying the listdevs example, contained in the libusb source code, to use three contexts instead of one. In our scenario, each libusb_context records an LKL of USB devices and an LKL of associated file descriptors, which corresponds to the ASG displayed in Table 8.4. The knowledge of the underlying DDS representation should be of much help when diving into the thousands of lines of C code.

Both libusb and lit3 also show the $I1_O$ connections between the parent and child strands, because each child strand has exactly one cell inside the parent, i.e., namely the child CDLL's head node. The cyclicity of the children results in the $I1_O$ label; otherwise, the same-head-node label (SHN) would have been applied.

Additional benchmarking is conducted in Sec. 8.2.7 to prove that DSI correctly identifies the SHN label in the absence of cyclicity.

8.2.6 DLL implementations

Examples lit5, rosetta-dll and syn7 all implement a DLL. The difference among them is the way in which the DLL is implemented, e.g., where new nodes are added into the list. Example syn7 inserts an element in the middle of the DLL, thus breaking the strands instead of just extending them as is the case when inserting to the head or the tail of a DLL. Example lit5 is chosen, although it also inserts into the middle of the DLL; however, it is an example from the literature and thus eliminates bias. Additionally, the insertion takes place behind and in front of a fixed element of the DLL, which resembles an insert after the head element of the list but in two directions. The third DLL example rosetta-dll is chosen because it also embeds the linkage struct inside of the payload element, as does the LKL. Therefore an alternative implementation to the LKL technique is tested but without cyclicity. The results show that DSI is still able to detect the correct shape for all three examples, which emphasises the robustness of DSI, including temporal and structural repetition that are based on the quantification of strand connections, see Ch. 5.

Additionally to the above examples, a real world CDLL example has been taken from the bash source code. The example is chosen as it exposes a non LKL-like CDLL implementation. Again, the labels show intersecting on multiple nodes overlay, as example tb2, as well as a standard DLL label. The latter is introduced when connecting the first two nodes of the CDLL without the cyclic links being established. After the cyclic links are set, further list manipulating operations only show intersecting on multiple nodes. The overall evidence for the CDLL is overwhelming and outnumbers both degenerate shapes.

8.2.7 Nesting scenarios

The examples sny8-10 all target the nesting behaviour, with only SLLs for the parent and the children. The difference lies in the nesting relation, which is either overlay or indirection. Further, the number of children per parent node is either one or two. The examples run longer than examples syn2 and syn3, where the DDS is placed inside of one enclosing vertex, i.e., inside a struct. Example syn8 tests the indirect nesting relation to one child and syn9 the overlay nesting to one child, where the parent node type is also used for the child nodes, thus resulting in nesting on overlay. Example syn10 creates a parent SLL with two SLLs nodes embedded inside the parent, which function as the head of the child lists. This test again aims at the nesting-on-overlay scenario, but it also shows the relation between the two child SLLs as being of type SHN (same-head-node), which is correctly identified by DSI. This demonstrates the precision between the different connection types that are detectable, as a comparison with the examples lit3 and libusb from Table 8.4 reveals that the connections for those examples are detected as $\mathrm{I1_O}$. This difference stems from the cyclicity of the child elements in the latter examples, which eliminates the notion of a head node. Again, syn8-10 are correctly identified by the DSI tool.

With example syn5, the nesting on overlay case is tested with DLL child elements connected via nesting on overlay to the parent DLL. For both the parent and child elements, the DLL connections are discovered by DSI. Both connections record the I2+$_O$ label, indicating the degenerate shape during list manipulation, where the forward and backward strands of the DLL intersect on more than one node, but not in all nodes, e.g., during insertion where one strand is temporarily longer than the other. When considering the parent child connections, three different labels are recorded: (i) nesting on overlay, (ii) intersecting on one node, and (iii) same-head-node. The last label is only present on the first connection shown on the left of the corresponding ASG in Table 8.4. This connection is between the first element of the forward strand of the parent and the first element of the forward strand of the child strand, which indeed share the same enclosing struct. The remaining connections are correctly classified as intersecting on one node overlay. Overall, the correct nesting on overlay relation dominates.

Example lit4 tests multiple nesting levels, where each parent and child relation has a different data type, i.e., there only exists a nesting on indirection relation. When observing example lit4 in Table 8.5 indeed the SLL strands can be seen for the various nesting levels. Additionally, the nesting on indirection relation dominates at all levels. The evidence counts for nesting on indirection increase top down, while the intersecting on one node indirection decreases top down. This reveals two things: (i) the longest running EP is not attached to the finished DDS but has observed both the stable and degenerate shapes; (ii) the nesting relations for the child elements accumulate faster due to more child elements being folded during FSG creation, i.e., the child elements show more nesting evidence when compared to the top-level parent strand.

8.2.8 Data structures on the stack

Two corner cases are tested with examples syn11 and syn12. The first creates an LKL CDLL on the stack, similar to syn1 from Table 8.4 but with differently typed payload nodes, which results in three differently typed nodes in total: the struct list_head plus the two payload nodes. In fact, those scenarios are transparent to DSI due to its cell abstraction.

The example shows that the placement on the stack imposes no problem for DSI and that the cell abstraction allows for the detection of the CDLL even when more than two differently typed elements are present. Two differently typed elements are tested with examples syn1, lit3, libusb from Table 8.4. Example syn12 creates a short SLL that is composed of four cells, where the first two cells reside in one vertex and the last two cells reside in another vertex. The setup is chosen to focus on the correct cell detection and subsequently the strand creation.

8.2.9 Binary tree implementations

Both examples `treeadd` and `treebnh` exclusively show the binary tree label, which happens when the longest running EP gets attached to the BT after it is constructed, i.e., there are no more DDS operations. This is in contrast to, e.g., the previously discussed example `lit4`, where the EP observes both stable and degenerate shapes.

Test `lit6` is another BT implementation taken from the VeriFast [73] test cases. It adds to the number of different binary implementations tested by DSI to demonstrate that DSI can cope with a wide variety of BT implementations.

8.2.10 Limitations of DSI revealed by X.Org

The Open source X Window System implementation (X.Org) has over 380k lines of C code. Therefore, only an interesting subcomponent is analysed (`div/resource.c`) that handles client resources with a hash map. Clients are represented as an array of `struct _ClientResource` pointers, with one entry for each client. In turn, each `struct _ClientResource` struct has a pointer to an array of `struct _Resources`, which functions as the bucket array of the actual hash map. The `struct _Resources` are organized as anSLL in order to handle key collisions.

This is a prominent example of how a DDS not only comprises strands but a combination of arrays and strands. The case study shows that DSI is capable of detecting both the bucket array and the bucket SLLs as strands. The final detection of the hash map however fails, as DSI only recognizes the SLL strands but not the array as part of the DDS. This hash map implementation is thus a motivation for modeling arrays into the strand abstraction, although they are not an SLL.

8.2.11 Discussion of arrays and strands

As pointed out by the `xorg` example discussed in the previous section, DSI does not consider arrays for building a DDS. A straightforward implementation of this would be to model each array as a strand. This requires the detection of arrays by the instrumentation. Then, the array could be transformed into a strand by treating each element of the array as a cell.

One fundamental difference to the current strand implementation would be that a strand of arbitrary length can be created within one time step. When tracking pointer writes, the strand develops over time, where the initial length is always two. For arrays, the linkage offset would need to be simulated, as no pointer connections between the elements are present. This can be simulated by choosing an arbitrary yet consistent offset among arrays of the same type, e.g., zero. One needs to pay attention to special cases, where the elements of the array get linked, as is the case with the cache optimized singly linked list [47]. Here, a design decision would need to be made, whether arrays are always treated as strand entities, leading to intersecting list interpretations when such additional linkages within the array occur. Alternatively, the array strand could be discarded, when linkage conditions inside the array are observed that lead to the creation of strands. The latter choice can be seen as treating the array creation as a custom memory allocator. With this abstraction, it would be possible to also handle situations where an array is directly embedded into a struct, as is the case with a *flexible array member*[1], which requires one to find the first cell element of a strand. When modeling each element of the array as one cell, the calculation of the SC as described in Ch. 5 is also transparent to DSI.

8.2.12 Requiring semantics to reveal true shape

The two textbook examples `tb3` and `tb4` are similar to `tb1` from Table 8.4 in that the underlying DDS is an SLL. They differ in the way the SLL is used: `tb3` is a stack and `tb4` a queue. Because the semantics are not detectable by DSI, the results in Table 8.3 are in brackets, as DSI indeed detects the underlying SLL but does not report a stack/queue. The semantics detection could be enabled by adding operation detection in the future. It would then be possible to take pre/post-conditions for the operations into consideration. However, both examples show that the way in which the DDS is used (stack vs. queue in this case) does not influence the detection of the correct underlying data structure, i.e., the SLL.

Another example is the hash map implementation of X.Org. In this case, given that pointer arrays would be turned into strands as discussed previously, the resulting interpretation of the DDS would be parent child nesting on indirection. The array would be the parent strand and the bucket SLLs are the child elements. Thus the interpretation of a hash map in the X.Org example and the general parent child nesting scenario is ambiguous. Hence, the shape would be detectable by DSI, but the semantics, i.e., the usage as a hash map, would escape DSI. This problem is analogous to the stack (`tb3`) and queue (`tb4`) examples, where the underlying DDS is detectable but the final interpretation as stack/queue cannot be made by DSI. However, the problem of observing a hash map with DSI is even harder than detecting an SLL functioning as a queue or stack. For the hash map, the detection of the DDS operations is insufficient, because the insertion/removal characteristics are not as revealing as is the case with a queue, i.e., the insertions and removals into/from the hash map are more vague. One might need to fall back to less revealing patterns such as inserts following the pattern of indexing into the array, first and then iterating the bucket to find the place for inserting the element. Also array iterations should occur less frequently, because the buckets are indexed by the hash value which makes the accesses non-sequential. Only special cases dealing with the consistency of the hash map, e.g., rebuilding the hash

[1] https://en.wikipedia.org/wiki/Flexible_array_member last accessed 4th of November 2018, http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf last accessed 4th of November 2018

map, might fall back to iterative accesses. This type of analysis is not considered in this dissertation and is left for future work.

8.2.13 Limitations of DSI revealed by `tsort`

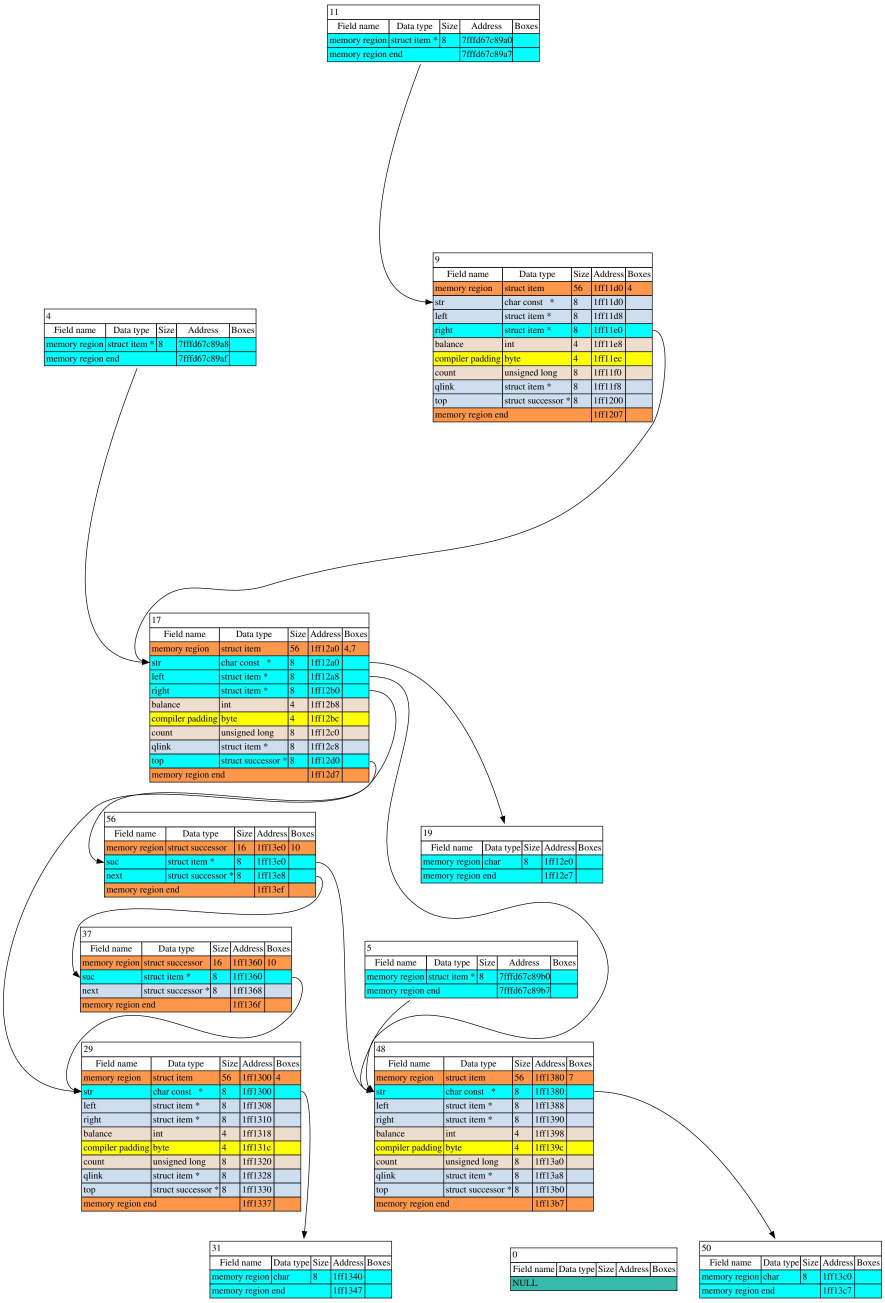
Another case study is that of `tsort` from the GNU Core Utilities (coreutils) [9] which performs a topological sort. The underlying data structure consists of a binary tree of type `struct item`, and SLLs of type `struct successor` running between the nodes of the tree. Additionally, an SLL is directly traversing the binary tree nodes on the `struct item *qlink` pointer. This layout can be seen in Fig. 8.2.

DSI correctly identifies all strands and strand connections as shown by the ASG in Fig. 8.3. This leads to the correct identification of the following two parts of the overall DDS: (i) The two strands for the binary tree (`struct item` field `left` and `right`) are detected and DSI infers the BT label between them; the two additional SLLs are also detected, with strand `struct item` on field `qlink` and strand `struct successor` on field `next` (ii) The connection between the `qlink` and the `next` field is indeed on indirection, as there exists the `top` pointer inside the `struct item`, which connects to the `struct successor`. Therefore, the `qlink` strand has a pointer connection to the `next` strand, where the `qlink` acts as the parent and the `next` strands act as children, resulting in the indirect nesting detection.

Ambiguous nesting cases

However, the following two problems arise with the current DSI approach: (i) The connections between the `qlink` strand and the `left` and `right` strands are identified as N_O, because all three fields run through the binary tree nodes of type `struct item`. This results in overlay connections where the `qlink` acts as the parent, from which the `left` and `right` strands are hanging off as children. As we consider the main DDS to be the binary tree, this makes a nesting interpretation where elements of the binary tree act as children of the `qlink` strand misleading; (ii) An ambiguous nesting on indirection (N_I) is detected between the `right` and `next` strand (and in principle between the `left` and `next` strand, but this situation is discussed separately below). Here, both interpretations are valid, as the `right` strands act as a parent from which multiple `next` strands are hanging off. But since the `next` strand points back to multiple `right` strands, it also acts as a parent for the `right` strands.

The ambiguities of (i) and (ii) stem from the initial assumption that the nesting direction automatically infers the correct parent child relation. While this works for many examples, the assumption does not hold for `tsort`. Therefore, the main data part of the data structure, i.e., the parent, needs to be detected separately from the DDS detection, in order to arrive at a consistent naming. In this case, we would consider the binary tree as the parent, thus choosing the nesting from the binary tree to the SLLs.



11

Field name	Data type	Size	Address	Boxes
memory region	struct item *	8	7fffd67c89a0	
memory region end			7fffd67c89a7	

9

Field name	Data type	Size	Address	Boxes
memory region	struct item	56	1ff11d0	4
str	char const *	8	1ff11d0	
left	struct item *	8	1ff11d8	
right	struct item *	8	1ff11e0	
balance	int	4	1ff11e8	
compiler padding	byte	4	1ff11ec	
count	unsigned long	8	1ff11f0	
qlink	struct item *	8	1ff11f8	
top	struct successor *	8	1ff1200	
memory region end			1ff1207	

4

Field name	Data type	Size	Address	Boxes
memory region	struct item *	8	7fffd67c89a8	
memory region end			7fffd67c89af	

17

Field name	Data type	Size	Address	Boxes
memory region	struct item	56	1ff12a0	4,7
str	char const *	8	1ff12a0	
left	struct item *	8	1ff12a8	
right	struct item *	8	1ff12b0	
balance	int	4	1ff12b8	
compiler padding	byte	4	1ff12bc	
count	unsigned long	8	1ff12c0	
qlink	struct item *	8	1ff12c8	
top	struct successor *	8	1ff12d0	
memory region end			1ff12d7	

56

Field name	Data type	Size	Address	Boxes
memory region	struct successor	16	1ff13e0	10
suc	struct item *	8	1ff13e0	
next	struct successor *	8	1ff13e8	
memory region end			1ff13ef	

19

Field name	Data type	Size	Address	Boxes
memory region	char	8	1ff12e0	
memory region end			1ff12e7	

37

Field name	Data type	Size	Address	Boxes
memory region	struct successor	16	1ff1360	10
suc	struct item *	8	1ff1360	
next	struct successor *	8	1ff1368	
memory region end			1ff136f	

5

Field name	Data type	Size	Address	Boxes
memory region	struct item *	8	7fffd67c89b0	
memory region end			7fffd67c89b7	

29

Field name	Data type	Size	Address	Boxes
memory region	struct item	56	1ff1300	4
str	char const *	8	1ff1300	
left	struct item *	8	1ff1308	
right	struct item *	8	1ff1310	
balance	int	4	1ff1318	
compiler padding	byte	4	1ff131c	
count	unsigned long	8	1ff1320	
qlink	struct item *	8	1ff1328	
top	struct successor *	8	1ff1330	
memory region end			1ff1337	

48

Field name	Data type	Size	Address	Boxes
memory region	struct item	56	1ff1380	7
str	char const *	8	1ff1380	
left	struct item *	8	1ff1388	
right	struct item *	8	1ff1390	
balance	int	4	1ff1398	
compiler padding	byte	4	1ff139c	
count	unsigned long	8	1ff13a0	
qlink	struct item *	8	1ff13a8	
top	struct successor *	8	1ff13b0	
memory region end			1ff13b7	

31

Field name	Data type	Size	Address	Boxes
memory region	char	8	1ff1340	
memory region end			1ff1347	

0

Field name	Data type	Size	Address	Boxes
NULL				

50

Field name	Data type	Size	Address	Boxes
memory region	char	8	1ff13c0	
memory region end			1ff13c7	

Figure 8.2: A points-to graph for one time step, when running `tsort`. The binary tree nodes are formed by `struct item` nodes, the singly linked list is formed on `struct successor` nodes. Blue nodes are pointers into the data structure. Moreover, blue nodes referenced by the data structure (by field `str` of `struct item`) are abstracted `char` arrays. `struct`s are indicated by orange fields marking the start and end of the `struct` memory region. Pointer members of a `struct` are colored as follows: (i) uninitialized pointers are grey; (ii) pointers pointing to valid memory are blue. Paddings added by the compiler are yellow. The remaining primitive types are colored pinkish.

A way to tackle this situation might be to track the access patterns to the data structure, where traversal patterns should start inside the parent and then continue to the child elements. Such a traversal detection might also help one to perform a hot-spot analysis within the DDS, for finding which areas are frequently accessed. This might enable performance bug detection similar to Travioli [89]. In order to increase the confidence level for the parent, one could additionally monitor to which parts of the DDS the EP is connected first; the intuition is that the parent part of the DDS is created prior to the child elements.

8.2.14 Segmentation of event trace and convergence of evidence counts

The `tsort` example shows one additional aspect for future work, which is choosing the most complex DDS interpretation even though it has not accumulated the highest evidence count. This can be seen when looking at the connection between the `left` pointer field back to `next` pointer. This setting shows the ”intersecting on one node indirect” ($I1_I$) label as the highest ranked label. Nevertheless the structurally more complex N_I label is present as well, but is not able to outperform the $I1_I$ label. One could think of segmenting the trace into subparts, in which another label than DSI’s final interpretation dominates. This could be helpful in situations where DDSs are short lived, i.e., where the buildup phase showing a degenerate shape likely outnumbers the Stable Shape (StS).

The additional information about time periods in which the DDS is in a structurally more complex state might help the developer to gain a deeper understanding of the DDS, by inspecting those time steps. This information should be seen as an addition to DSI’s final interpretation that reports the highest ranked label, as only the overall interpretation of the DDS reflects that the DDS was in a DgS most of the time.

The segmentation could be performed by observing each label of the final interpretation that contains all labels collected by means of temporal and structural repetition, individually over time. The idea is to start monitoring each label from the time step of its first occurrence in order to see how the ranking of the observed label develops. The problem lies in the question when to stop this intermediate analysis. If one lets the analysis run until the last occurrence of the inspected label, the segmentationwise interpretation might suffer from the same problem as the global analysis, i.e., a long period where the DDS is in a DgS again outnumbers the StS. When considering short time periods that show a structurally more complex label than the final interpretation, might result in noise during the creation of the DDS and cause too much attention. Here, one could think of parameterizing the inspected interval similar to the quiescent period detection done by MemPick [69], where the DDS is only inspected during certain intervals.

This future work topic is somewhat connected to the question when DSI will converge to the correct DDS label. By allowing timewise segmentation with different interpretations of the DDS, the problem of when the evidence collected by DSI converges to the true DDS shape might be alleviated to a certain extend. The general problem of not knowing when to stop the analysis still remains, as changes can happen at any time to the DDS. For example, consider a DLL where the forward linkage is created first, i.e., only one SLL exists. The backward linkage is created subsequently and thus, when stopping the analysis short, the true DDS might never be observed. However, with the finer grained examination of how labels evolve over time, one could visualize the rise of a label that is just not able to ultimately outweigh the other interpretations, e.g., due to cutting the trace short.

The previous discussion and the rather short running syn1 example also raises the question, whether one can make a statement about how fast a DDS converges to its StS. This question cannot be answered in general. The characteristics of a dynamic analysis is to analyse exactly what it records. Therefore, one always gets additional approximations when trying to cut the analysis short and stopping at a random point of the recorded trace, because arbitrary changes to the DDS can happen at every time step. Consider the creation of a parent DLL with child SLLs. It may well be the case that the parent DLL is created first, without any children. Thus the evidence for the DLL might already be overwhelming, which could be used as an indicator to stop the analysis. This would then lead to missing the nesting relation to the child SLLs, which are created and/or connected later. Even when exploring the complete execution trace, one cannot generalize that the observed data structure is the only possible shape exposed by the particular program. It is only a precise representation of the recorded trace.

One aspect that could be used as a hint to make an educated guess, whether a DDS has reached a StS is the monitoring of the available pointers within a detected DDS. If all pointers are assigned and an overwhelming evidence for a particular DDS was found, one might consider to stop the analysis short. Again, this is only an approximation as restructuring of the DDS can happen at any time during further program execution.

8.2.15 Removing parts from a data structure

The discussion of the previous section must also be considered when elements are being removed from a DDS. This additional aspect is currently not covered by DSI, i.e., the removal of elements from the ASG is not supported. The current implementation only aggregates elements into the ASG, but never removes parts from the ASG. This is beneficial as the DDS is covered completely over its lifetime, as we have seen with the multitude of examples discussed in this chapter. However, this neglects the collection of evidence for the *absence* of parts of the DDS once they have been recorded. Consider the example where a parent child relation is present for sufficiently many time steps for DSI to accumulate enough evidence for the detection of nesting on overlay. This relation will thus be reflected by the ASG, however, if the majority of the lifetime of the DDS does not have this particular parent child relation, e.g., because the child elements are removed, this fact is neglected.

Therefore for future work, it will be interesting to also gather evidence for the absence of elements within the ASG. This will give the analyst an even more fine grained insight into the usage patterns of the DDS.

8.3 Case study: Debugging with DSI

This section shows how a programmer is supported by DSI to gain insight into the behaviour of a program, and how this can help in debugging source code. The example is taken from Predator [23] and is described within the test case (called `lit7`) as "A tree of circular singly linked lists". The intuitive expected behaviour would be that each element of the binary tree has a CSLL child as payload. Therefore, the test case is of interest as it targets DSI's nesting detection in a previously untested combination, i.e., a binary tree with nested CSLLs. When executing the test with DSI, the expected DDS as stated by the test is not observed. Instead, the reported DDS is a binary tree with *one* CSLL child element, as can be seen by DSI's ASG in Fig. 8.4. Here, the `left` and `right` strands of type `struct TTreeNode` are connected with a BT label, but the `next` strand of type `struct TListNode` forming the CSLL is connected to the binary tree strands only via $I1_I$. Further investigation reveals that this is indeed the correct interpretation of the DDS as produced by the example. Because the unexpected behaviour occurs with the creation of the child elements, the programmer is pointed to the code sections dealing with the child creation, which are lines 20-26 and 35-46 of Figure. 8.6, (Note that this listing includes both versions of the test, the original and the modified).

When inspecting the source code in more detail, one observes that the `tree` pointer points to the first element of the tree, and the first element of the CSLL is allocated and assigned to the `tree` node in line 20. Line 21 initializes the child CSLL to point to itself. Note that DSI does not recognize a strand for an element pointing to itself, because a strand requires at least two distinct cells. Then, lines 22-26 insert more nodes into the CSLL, where the first insertion results in a strand creation by DSI, which gets extended with subsequent inserts.

In order to avoid side tracking of the reader, some code sections for building the tree is truncated in Figure 8.6 as indicated by line 31. The skipped routines include, e.g., iteration of the tree and decision routines where to insert a new tree element, are not of interest for the child creation. The child CSLL of the new node `newNode` is initialized in lines 35 and 36, which is analogous to lines 20 and 21. Then, the remaining elements are inserted inside a loop (line 38, analogous to line 22). The original implementation is shown in lines 41 and 42, which are identical to lines 24 and 25; thus, the `tree` pointer is used instead of the `newNode` pointer. The `tree` pointer is not updated elsewhere; therefore, the CSLL elements are inserted into the tree element instead of the newNode. Whether this behaviour is intentional or a bug, cannot be said.

However, the usage of DSI quickly reveals the specifics of the code, which in this case is not exactly the behaviour expected when reading the test case description "A tree of circular singly linked lists". The analysis of the code with DSI leads to the code changes as seen in line 44 and 45, which actually uses the newNode pointer for inserting the new CSLL elements. The resulting ASG for the modified code is shown in Fig. 8.5, which now shows the N_I label from the left and right strands pointing to the next strand.

In conclusion, this example shows how DSI helps programmers to quickly comprehend program behaviour. It also points out that DSI is not only helpful for a programmer who is confronted with unknown code, but also for a programmer who is looking to verify program behaviour that he or she expects. Because the ASG represents the DDS the misbehaving part can in this case be narrowed down to the child creation, which guides the programmer to the corresponding code sections. This can either be done by reading the code, or by using the information stored inside of the event trace generated by DSI, as the instrumentation also records line numbers for pointer manipulations.

8.4 Summary of benchmarking

The benchmarking results show that DSI can cope with a large variety of DDS shapes and implementations. We have seen various implementation techniques of data structures, ranging from real world examples, examples from the literature, textbook and synthetic hand-written examples from the author of this dissertation and authors of two student theses. This underlines the robustness of the DSI approach. Especially, the limitations of related work, e.g., [49, 69, 74], with regards to the handling of nested structs as used by the Linux kernel list implementation were verified. Additionally overlay nesting was successfully identified by DSI, which is also not covered by related work. DSI was exposed to difficult examples, such as very short running examples, in order to prove that the evidence reinforcement, i.e., structural and temporal repetition, works.

Additionally the benchmark revealed open areas where DSI can be improved. These became evident, when benchmarking DSI on prominent examples from the open source community such as X.Org [38], which creates a hash map out of an array with SLLs. DSI does not handle arrays and, thus, is not able to identify the hash map. A possible way to handle arrays as strands was discussed as a direction for future work. With the tsort example from the coreutils [9], which performs a topological sort, the shortcoming of DSI not being able to handle ambiguous nesting scenarios, was exposed. A possible solution for future work was discussed to tackle this problem by, e.g., tracking the usage patterns of the data structure.

Overall, it was shown that DSI's novel memory abstraction in the form of cells and strands works well in practice. Multiple different data structures such as (cyclic) DLLs, SLs and BTs and various combinations in the form of overlay and indirect nesting are correctly identified by DSI. Additionally, the discovered shortcomings seem to be not a general problem of the approach, but they are rather an interesting topic for future work to extend DSI's capabilities.

Further, it was shown by a case study that DSI cannot only be used to gain insight into an *unknown* code base, but can also be used to analyse code with *a priori* knowledge, so as to find unexpected program behaviour. This opens perspectives for DSI being used as a debugging/manual verification tool by a developer during program development.

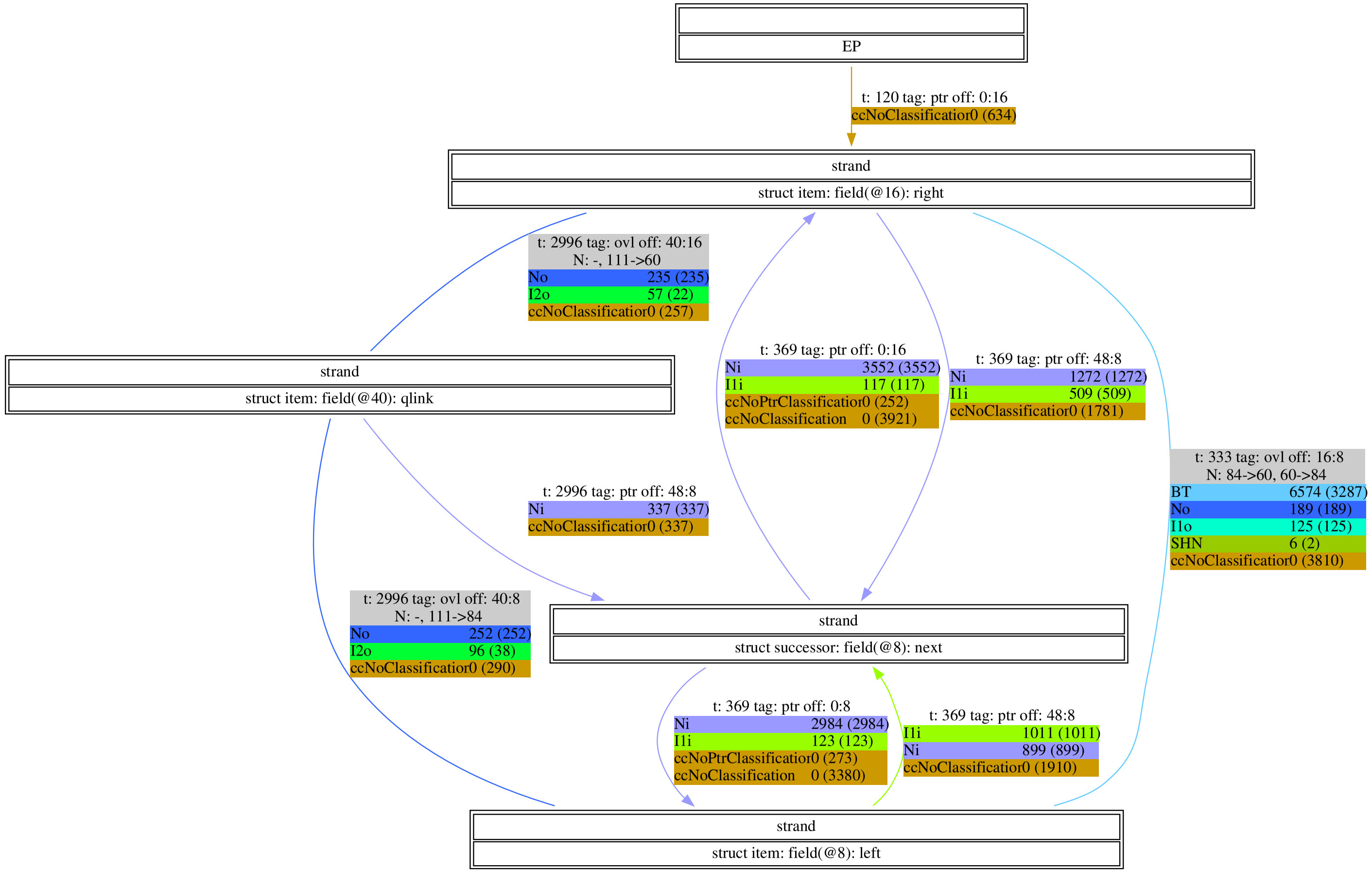

EP
t: 120 tag: ptr off: 0:16
ccNoClassificatior0 (634)
strand
struct item: field(@16): right
t: 2996 tag: ovl off: 40:16
N: -, 111->60
No 235 (235)
I2o 57 (22)
ccNoClassificatior0 (257)
t: 369 tag: ptr off: 0:16
Ni 3552 (3552)
I1i 117 (117)
ccNoPtrClassificatior0 (252)
ccNoClassification 0 (3921)
t: 369 tag: ptr off: 48:8
Ni 1272 (1272)
I1i 509 (509)
ccNoClassificatior0 (1781)
strand
struct item: field(@40): qlink
t: 333 tag: ovl off: 16:8
N: 84->60, 60->84
BT 6574 (3287)
No 189 (189)
I1o 125 (125)
SHN 6 (2)
ccNoClassificatior0 (3810)
t: 2996 tag: ptr off: 48:8
Ni 337 (337)
ccNoClassificatior0 (337)
t: 2996 tag: ovl off: 40:8
N: -, 111->84
No 252 (252)
I2o 96 (38)
ccNoClassificatior0 (290)
strand
struct successor: field(@8): next
t: 369 tag: ptr off: 0:8
Ni 2984 (2984)
I1i 123 (123)
ccNoPtrClassificatior0 (273)
ccNoClassification 0 (3380)
t: 369 tag: ptr off: 48:8
I1i 1011 (1011)
Ni 899 (899)
ccNoClassificatior0 (1910)
strand
struct item: field(@8): left

Figure 8.3: The aggregated strand graph for `tsort`, with an ambiguous nesting relation between the binary tree strands (`left`/`right` fields of type `struct item`) and the `struct successor` singly linked list.

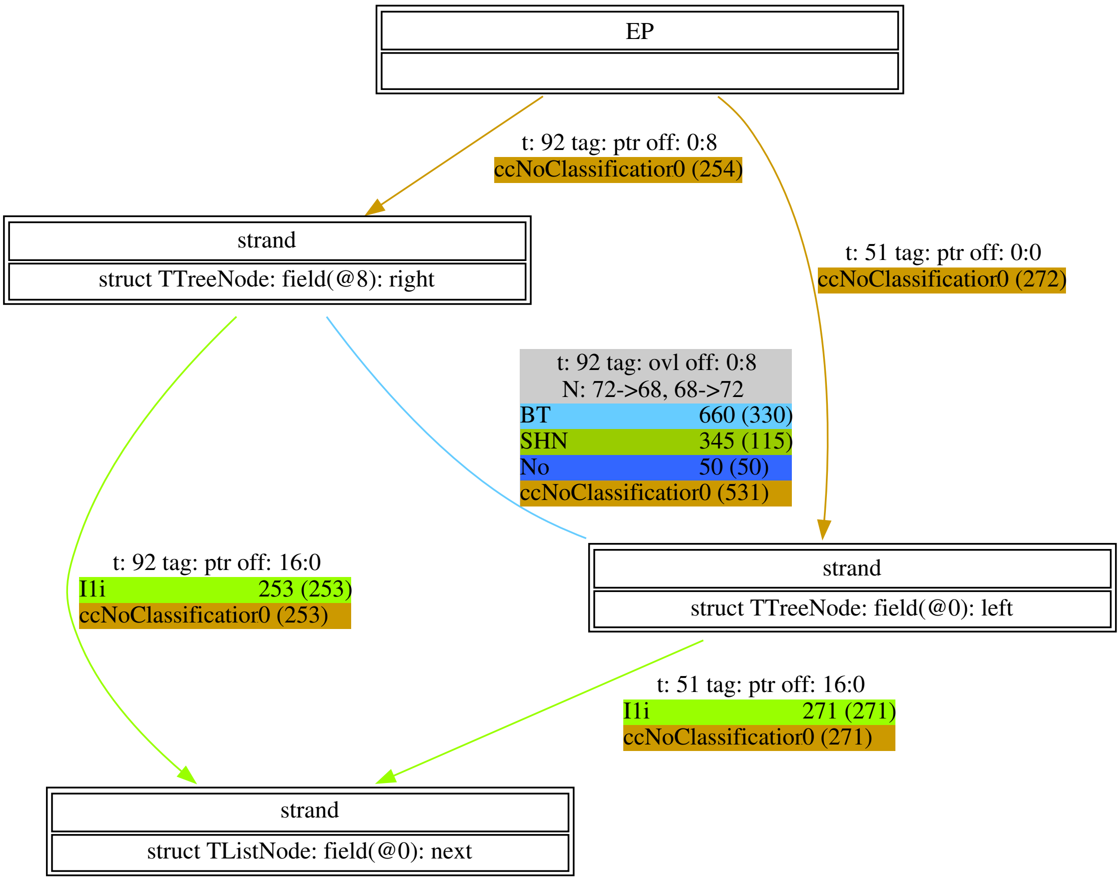

EP
t: 92 tag: ptr off: 0:8
ccNoClassificatior0 (254)
t: 51 tag: ptr off: 0:0
ccNoClassificatior0 (272)
strand
struct TTreeNode: field(@8): right
t: 92 tag: ovl off: 0:8
N: 72->68, 68->72
BT 660 (330)
SHN 345 (115)
No 50 (50)
ccNoClassificatior0 (531)
t: 92 tag: ptr off: 16:0
I1i 253 (253)
ccNoClassificatior0 (253)
strand
struct TTreeNode: field(@0): left
t: 51 tag: ptr off: 16:0
I1i 271 (271)
ccNoClassificatior0 (271)
strand
struct TListNode: field(@0): next

Figure 8.4: `lit7` [23]: DSI's aggregated strand graph showing the BT label for the parent and the $I1_I$ label for the child element, which hints at a different implementation of the test case than has been expected: not every binary tree node has a nested cyclic singly linked list.

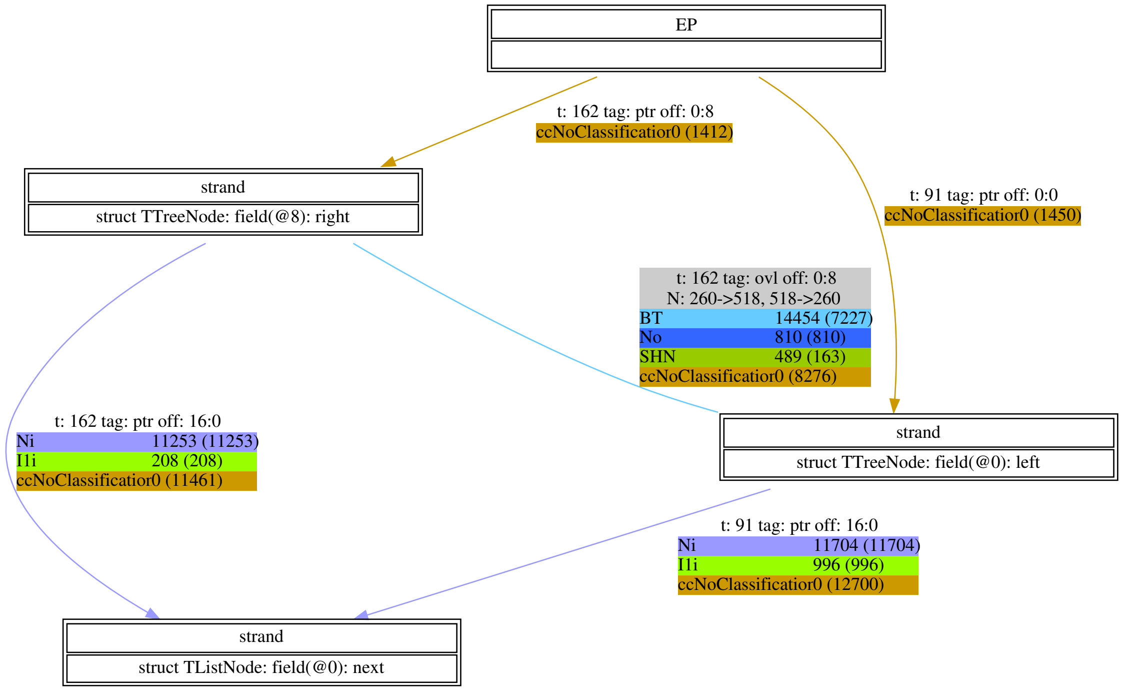

EP
t: 162 tag: ptr off: 0:8
ccNoClassificatior0 (1412)
t: 91 tag: ptr off: 0:0
ccNoClassificatior0 (1450)
strand
struct TTreeNode: field(@8): right
t: 162 tag: ovl off: 0:8
N: 260->518, 518->260
BT 14454 (7227)
No 810 (810)
SHN 489 (163)
ccNoClassificatior0 (8276)
t: 162 tag: ptr off: 16:0
Ni 11253 (11253)
I1i 208 (208)
ccNoClassificatior0 (11461)
strand
struct TTreeNode: field(@0): left
t: 91 tag: ptr off: 16:0
Ni 11704 (11704)
I1i 996 (996)
ccNoClassificatior0 (12700)
strand
struct TListNode: field(@0): next

Figure 8.5: `lit7` [23]: DSI's aggregated strand graph showing the BT label for the parent and the expected N_I label for the child elements, after modifying the initial test case.

```
/* A tree of circular singly linked lists */
typedef struct TListNode {
    struct TListNode* next;
} ListNode;
typedef struct TTreeNode {
    struct TTreeNode* left;
    struct TTreeNode* right;
    ListNode* list;
} TreeNode;
#define DSI_TREE_ELEMS  20 // DSI
#define DSI_LIST_LENGTH 7  // DSI
int main() {
  int i,e; // DSI
  TreeNode* tree = malloc(sizeof(*tree));
  TreeNode* tmp;
  ListNode* tmpList;

  tree->left  = NULL;
  tree->right = NULL;
  tree->list = malloc(sizeof(ListNode));
  tree->list->next = tree->list;
  for(e = 0; e < DSI_LIST_LENGTH; e++){//DSI
    tmpList = malloc(sizeof(ListNode));
    tmpList->next = tree->list->next;
    tree->list->next = tmpList;
  }
  while(i < DSI_TREE_ELEMS){//DSI
    tmp = tree;

    TreeNode* newNode;
    /* Skipped tree iteration for newNode insertion */

    newNode->left = NULL;
    newNode->right = NULL;
    newNode->list = malloc(sizeof(*newNode->list));
    newNode->list->next = newNode->list;

    for(e = 0; e < DSI_LIST_LENGTH;e++){//DSI
      tmpList = malloc(sizeof(ListNode));
      // Original: insertion into first node
      tmpList->next = tree->list->next;
      tree->list->next = tmpList;
      // Modified: insertion into newNode
      tmpList->next = newNode->list->next;
      newNode->list->next = tmpList;
    }
  }
}
```

Figure 8.6: Source code excerpt from example `tree-with-cslls` taken from Predator [23], with modifications done to control the number of created elements. In addition the code section responsible for the creation of a child cyclic singly linked list for each binary tree node has been changed from the original example which created a child cyclic singly linked list only for the first node in the tree.

9 Conclusions

In Pt. I of this dissertation, our first research question was addressed:

> **Research Question 1:** *Is the DSI approach adequate to reach its goals of automatically detecting dynamic data structure shapes with high precision in the presence of degenerate shapes and, in particular, how far must the DSI concept be refined in order to deal with the wealth of dynamic data structure implementations employed in real-world software?*

This question was answered positively by deepening and realizing the DSI approach to arrive at a research tool comprised of ~10k LOC of Scala. The tool was evaluated by a diverse benchmark including real world examples (libusb, bash), hand-written and text-book examples, and examples taken from the shape analysis literature. The performed benchmark showed that DSI outperforms related work [49, 69, 74] regarding, various nesting scenarios including arbitrary nesting combinations of DSs, previously unhandled DSs such as skip lists, and when running through nodes of different types such as in the Linux kernel list.

On the way of realizing DSI, Challenges 1.1–1.2 were solved:

> **Challenge 1.1:** *Can DSI's graph abstractions be kept consistent in the face of common memory events such as memory (de-)allocations, pointer writes and even programming errors?*

This first challenge was tackled by the introduction of artificial memory events, which are injected into the event trace if (i) an event implicitly cuts pointers to/from a memory region or (ii) memory is leaked as observed by our memory leakage detection algorithm that is able to report the exact instruction that caused the leak. Both (i) and (ii) together guarantee the consistency of DSI's memory abstractions, i.e., keeping DSI's points-to graphs and the associated strand graphs in sync with the heap state. This is fundamental for DSI's analysis in order to handle common programming situations such as frees, and to be resilient against programming errors. During the memory leakage analysis, design decision of the CIL framework became evident, which introduces temporary pointers inaccessible by the programmer when allocating memory. This behaviour can lead to false negatives of the memory leakage detection algorithm because the temporary reference might still be in place when performing the analysis. This is a severe side effect introduced by the CIL compiler writers. Except for this aspect that is out of reach for DSI, the memory consistency can be guaranteed, leading to our second challenge:

> **Challenge 1.2:** *Can DSI's strand connections be quantified such that connections performing the same role can be identified robustly even accross today's multitude of dynamic data structure implementations?*

This question was answered positively by using the distance between strands if both reside in one memory chunk or, otherwise, by using the offset of a pointer connection between strands. The idea is that the same offsets are found repetitively across strands performing the same role, as is the case when studying a list of lists, i.e., parent child nesting. The robustness of the abstraction is given by DSI's structural and temporal repetition, which is implemented on top of the chosen abstraction. DSI is able to inspect DDSs even during degenerate shapes, which occur when a DDS looses its shape due to manipulation operations such as list inserts or deletions, and is still able to detect the true shape. This approach separates DSI from related work which either avoids degenerate shapes [69, 74] or looses precision by being conservative when observing degenerate shapes [49].

This dissertation contains a benchmark comprised of 49 examples including real world examples (libusb, bash, tsort, X.org, Olden benchmark, and The Computer Language Benchmarks Game: Binary Tree), examples taken from the shape analysis literature [65, 72], textbook examples [104, 108], and synthetic self written examples both from students in the context of a bachelor thesis [46] and a master thesis [62] and from the author of this dissertation. The variety of the benchmark shows that DSI detects a large variety of data structures and can cope with a lot of different DDS implementation techniques. To highlight the program comprehension capabilities of DSI, it was shown that an unexpected behaviour of one of the literature examples (`lit7` [23]) was quickly revealed by DSI's analysis. The corresponding code section could then easily be fixed to show the desired semantics. This example points out that DSI is not only useful for comprehending unknown code but can also help a programmer to test intended program behaviour.

As an aside, the DSI approach was parallelized, which decreased its execution time significantly and showed that the approach does indeed scale. The parallel version is used in Pt. II to speed up the analysis of binary code, which makes the therein developed DSIbin approach far more usable.

Future work. With the foundations of DSI being settled, DSI's capabilities could be extended in various ways. At the moment, the analysis does not consider any payload information within DDSs, which would be necessary to detect the semantics of a DDS, e.g., sortedness. Such an analysis could be devised by taking non-pointer manipulating events into consideration when performing the instrumentation and, thus, the event trace generation. With this information, executed instructions prior to DDS manipulations could give a hint on the inspected payload and the operations performed on the payload.

Another path to explore relates to the precision of DSI in nesting scenarios; it is not always clear how a nesting situation needs to be interpreted, because the role of the parent and child are not straightforward to detect. An example for this is `tsort` from coreutils [9], which performs a topological sort. The underlying data structure comprises a tree with various SLLs running through it. DSI already detects the strands and the interconnections for this scenario, as was demonstrated in this dissertation via a small case study. However, the final interpretation is ambiguous: is the nesting from the lists to the tree or vice versa? Situations like these should be solvable by tracking access patterns to a DDS, such as traversals through the DDS or to which strand the longest running entry point is attached. Such in-depth information may also help performing a hot-spot analysis of the DDS, e.g., finding areas of a DDS that encounter a lot of insertions, analogous to TRAVIOLI [89], which operates on JavaScript instead of C source code.

Further, DSI currently cannot handle arrays, which are used in sophisticated DDSs such as hash maps. For example, the classical hash map implementation found in `dix/resource.c` from the X.org server [38] stores buckets as an array of pointers to SLLs for handling key collisions. A case study of the particular DDS showed that the current instrumentation, trace generation and analysis by DSI already detects the strands for the SLLs, and even the array for the buckets is detected. However, because DSI only analyses interconnections between strands, the final DDS interpretation is impossible. A straightforward solution would be to model arrays as strands and to simulate the linkage offset with a predefined value common to all array-strands, so as to seamlessly handle strand connections, both indirect and overlay.

Part II

Dynamic data structures in object code

DSIbin

As the IT industry starts to age, today's developers might be confronted with situations like SEGA's loss of source code for old arcade games [26], where corporate know-how is only available in a binary format. Additionally, the amount of malware is steadily growing [17], where analysts need to understand their behaviour without source code being available. Both situations are prominent use cases requiring reverse engineering [56]: (i) to not loose the previously performed engineering effort completely, and (ii) to understand the behaviour of an unknown binary file.

Most prominently, a hard challenge is the recognition of dynamic data structure shapes in the presence of tricky programming concepts like pointer arithmetic and pointer casts. Both are fundamental concepts to enable, e.g., the frequently used LKL [15], and bring current shape detection tools to their limits. To tackle the challenge of recovering DDSs in binaries, one needs to solve the loss of low level type information, like primitive and compound types, which even state-of-the-art type recovery tools, e.g., [43, 78, 80, 98], cannot reconstruct with full accuracy.

With DSI developed in Pt. I and working on source code, which already improves upon the related work [49, 69, 74], the second research question follows naturally:

Research Question 2: *Can DSI's concepts and strengths be preserved when inspecting binaries?*

With the limited information available in binaries, the rich DDS analysis of DSI would help in gaining a more precise insight into the program behaviour for the reverse engineer. For example, automatically detecting the cyclicity of a DLL that was previously missed reveals much information about the underlying DDS manipulating code, and relieves the reverse engineer from manually trying to comprehend the binary. Thus, the second part of this dissertation opens up DSI for binaries. The resulting approach/tooling is called DSIbin. DSIbin still relies on the core algorithms of DSI, which we call *Data Structure Investigator Core Algorithm (DSIcore)* from now on. As the ultimate use case for DSIbin might be the reverse engineering of malware, a survey on leaked malware source code is conducted to answer the question if current malware actually uses complex DDSs. The survey presented in Ch. 10 reveals that malware indeed uses DDSs.

The process of opening up DSIbin for binaries comes with quite some challenges. One of the first obstacles is the loss of perfect type information found in source code, such as information about (nested) structs that correspond to DSIcore's cell abstraction and is thus vital for the precision of the DSIbin approach. If such information is unavailable or is imprecise, DSIbin fails with different outcomes, e.g., DSIbin cannot establish the linkage condition between cells and thus misses strands, i.e., false negatives, or DSIbin establishes strand connections where it should not, i.e., false positives. Those source code dependencies are discussed in Ch. 11.

To remedy the type loss, one can fall back to an existing dedicated type recovery tool for binaries such as [43,78,80,98]. Therefore a literature survey on type recovery tools is conducted in Ch. 12, which yields Howard [98] as a suitable candidate for integration into DSIbin as it detects (nested) structs that are vital for DSIcore. In order to perform DSIcore's analysis and to incorporate the information from Howard, DSIbin needs to capture the same events as the CIL instrumentation does for source code but now for a binary. The task of performing such a binary instrumentation is shown in Ch. 13 and, once accomplished, enables the integration of Howard's type information into the event trace, i.e., combining Howard and DSIcore in Ch. 14.

With the development of a first DSIbin tool, it is possible to benchmark the new tool-chain, which is conducted in Ch. 15 for examples taken from publicly available malware samples. The results of this naive approach already show that DDS can indeed be detected out of the box. This includes DDSs not covered by related work [49,69,74] such as skip lists and nesting relations. However, some of Howard's limitations become evident, like missing nested structs and not merging types, which prevent DSIbin to fully utilize its fine grained cell abstraction and leads to the aforementioned precision loss when detecting strands. To tackle these limitations, an additional type refinement step was incorporated into DSIbin, which exploits pointer connections in memory to find nested types or to find candidates for type merging. More specifically, type merging aims at unifying allocated memory regions to the same type as long as some predefined merge conditions hold. As an example, consider the static and dynamic allocation of a `struct`. With source code available it is immediately clear that both memory regions on heap and stack are of the same type. Within a binary, this information gets lost and needs to be recovered. To unify both memory regions an exemplary merge condition can be the size and binary compatibility of the allocated memory.

Both the detection of nested types and type merging are fundamental problems when dealing with binaries. As the lack of information in the binary is insufficient to draw definitive conclusions from pointer connections at this point, it is necessary to create a set of type refinement hypotheses that all need to be evaluated according to their plausibility. As we are interested in DDSs, this plausibility check can be performed with DSIcore itself, by interpreting the resulting DDS of each type hypothesis with DSIcore. The creation of the hypotheses is explained in Ch. 16, and the benchmarking of the sophisticated tool-chain including the refinement is done in Ch. 17. Interestingly, the refinement not only improves upon the nesting detection and type merging of Howard, but also improves upon the primitive type detection. This increased precision, which could be seen as an improved version of Howard, should be of interest even to a reverse engineer whose primary focus is not the detection of DDSs. The exploration of DSIbin is concluded in Ch. 18 which also gives an outlook on future work.

10 Dynamic data structures in malware

To give a motivating use case for the analysis of binaries regarding DDSs, publicly available source code (C/C++) of real world malware [34] was inspected within this dissertation. This determines the ground truth that DDSs are indeed used in malware, and further that DDSs are reused in different malware. Both aspects stem from the increasing complexity seen with malware, where (i) DDSs are required to handle the growing amount of data that is processed and (ii) malware also starts to reuse components and libraries among different malware, both within one family and across families, as is the case with benign software projects.

The fact that components are reused in different versions of one particular malware, in malware families or even across malware families gives rise to interesting use cases that exploit information about the used DDSs. This is in the line with Laika [60], which uses DDS information to create malware signatures used by a virus scanner. While the aim of DSIbin is the detection of the DDS in binaries, the resulting information could be used to enrich malware signatures.

The malware samples are shown in Tab. 10.1, together with a short description of the malware and its relation to other malware. Additionally, the components that use potentially interesting DDSs for DSIcore/DSIbin are shown and where they are used. In the following, the analyzed malware of Table 10.1 is discussed together with the DDSs that we have found. A short summary of the analysis is published in [95].

10.1 Carberp & Rovnix

Carberp is malware designed to target bank accounts and can be extended with various features such as *bitcoin mining* and *backdoors* [63, 64]. The latter employs the component HVNC, as found in [34], which constructs a hidden desktop to which one may connect via VNC [14]. The VNC code appears to be taken as-is from the VNC repository [14], as the server contains a region clipping library also found in the VNC repository. The clipping library relies on a complex DDS consisting of a DLL parent with indirect nesting on DLL children, which are of the same type as the parent, as shown in Fig. 10.1. Interestingly, the head (`front`) and tail (`back`) of the list are embedded inside the `struct sraRegion`. This makes the detection of the DLL challenging, as it requires DSIcore's notion of cells to be able to recognize the head and tail node of the list, which is missed by type recovery tools such

Table 10.1: Overview of the analysed malware, including which malware is based upon each other (if applicable) and which components are reused (only with regards to the DS of interest to DSI/DSIbin)

Malware	**Description**	**lwIP**	**HVNC**	**massmail.c**	**struct mx_list_t**	**IRC bouncer (BNC)**
Rovnix	scareware and bootkit used by Carberp	✓				
Carberp	banking trojan	✓	✓			
MyDoom	worm with backdoor for DNS attack			✓	✓	
HellBot3	worm with backdoor partially based on MyDoom			✓	✓	
Grum	spam botnet				✓	
Agobot	modular IRC bot					✓

as [49, 69, 74]. The detection of this particular DLL with DSIbin is discussed in more detail later in this dissertation in Chs. 15 and 17.

Another component used by Carberp is lwIP [16], which is an independent TCP/IP stack implementation written in C with a focus on resource usage. In the context of Carberp lwIP is used to construct a hidden TCP/IP stack for hiding the communication of the malware. lwIP makes use of multiple independent lists, such as SLLs and DLLs. lwIP also forms a component of the scareware malware *Rovnix*, which in turn is used as a bootkit by Carberp, see Tab. 10.1.

10.2 MyDoom, HellBot3 & Grum

MyDoom is a worm that initiated a *Distributed Denial of Service (DDoS)* attack against the website of the SCO Group [68]. Additionally, MyDoom offers a backdoor for remote control [42]. MyDoom uses an SLL that implements a priority queue to store and process mail addresses, see `massmail.c` in Tab. 10.1. The DNS MX records are stored in an SLL of child SLLs, see `struct mx_list_t` in Tab. 10.1. Identical functionality is also present in *HellBot3*, which has its roots in MyDoom, as MyDoom was reused in MYTOB that was subsequently used in HellBot3 [19, 29, 36]. Additionally, `struct mx_list_t` was partially reused in the spam botnet *Grum* [50].

Source

```
struct sraSpan {
  struct sraSpan *_next;
  struct sraSpan *_prev;
  int start;
  int end;
  struct sraRegion *subspan;
}

struct sraRegion {
  sraSpan front;
  sraSpan back;
}
```

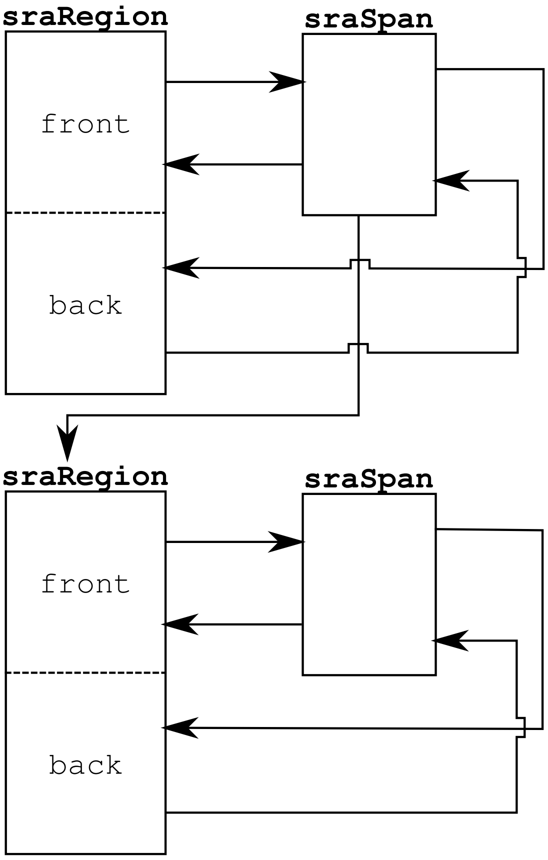


Figure 10.1: The source code of the VNC clipping library doubly linked list on the left and an example instantiation on the right (figure adopted from our publication [94]).

10.3 AgoBot

AgoBot [87] is a modular IRC [97] bot. It obfuscates its connections with an IRC proxy component, which employs C++ std::lists that are CDLLs. The implementation [30] is similar to that of the LKL, but uses C++ programming constructs, i.e., the DLL linkage is encapsulated inside a base struct from which the specialized list node carrying the payload is inherited. The actual head node is of the base struct type and, therefore, requires DSIcore's cell-based heap abstraction to detect the list as a whole.

10.4 Conclusions

We have seen that malware indeed uses DDSs and that it gets re-used due to the component based structuring of malware projects. The implementations range from standard linked list implementations, where the linkage pointers are of the type of the payload struct, e.g., lwIP, to standard containers such as C++ std::list implementation in Agobot, to highly customized parent child lists as seen with the

VNC component in Carberp. As the last two examples require the heap abstraction introduced by DSIcore, those are examined further during the benchmarking of DSIbin in Chs. 15 and 17.

11 DSI source code dependencies

This section covers the implications of the loss of perfect type information found in source code and the implications for DSIcore, which relies on this detailed information to function.

DSIcore requires type information to establish a linkage condition between cells, i.e., two cells connected by a pointer need to be of the same type and the pointer needs to be incoming at the start of the cell (see Ch. 2). As cells can either occupy a whole memory chunk or a sub-chunk, DSIcore requires knowledge about structs and nested structs. The information about nested structs can also be applied to CMAs implementations to detect a DDS running through a memory chunk allocated by a CMA. When this type information is unavailable, DSIcore is not able to reliably establish the linkage condition.

This can be seen in Fig. 11.1 that shows a LKL-like linked list implementation with type information present at the top. Only the `next` pointer is set, and the linkage offset o is represented relative from the start of the enclosing nested linkage struct shown in blue. As the available type information allows one to identify the commonality between the linkage structs even if they are embedded inside of differently typed list elements, e.g., the head node and the remainder of the list, the full list length can be identified, including the cyclicity of the list.

The same list as shown above but with the type information removed, can be seen at the bottom of Fig. 11.1. The only type information available is the size of the memory chunks. This prevents DSIcore from detecting any linkage condition, as neither type equivalence can be established, e.g., between the head node and the remainder of the list, nor does any incoming pointer point to the start of a cell. This prevents a linkage condition between the head node and the nodes of the remainder of the list.

In this example, the loss of type information leads to *false negatives* for the linkage condition, i.e., less strands are detected than with type information available. The same is true for the example shown in Fig. 11.2, where a list runs through one memory chunk by means of several nested structs. Again, with the knowledge of the nested structs, the strands can be established as DSIcore can verify the type commonality, the linkage offset and that the incoming pointer enters at the start of the nested struct. This is shown on the left of Fig. 11.2. Without the type information of the nested structs, the list cannot be detected by DSIcore, as seen on the right of Fig. 11.2

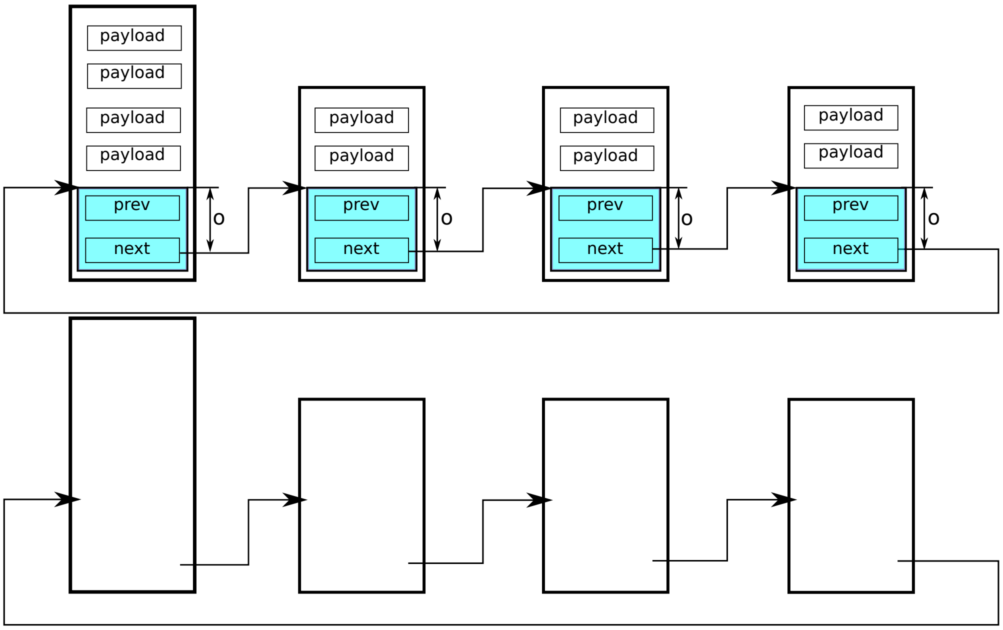


Figure 11.1: A Linux kernel list implementation with type information (top) and without type information (bottom).

Other than false negatives, it is also possible to encounter *false positives* when perfect type information is lost. This is due to the fact that the missing type information cannot be guaranteed to be recovered with full accuracy, as shown in Fig. 11.3. Here, the ground truth as given by source code shows a different *True Type (TT)*, i.e., True Type (TT)1 and TT2, for the two memory chunks respectively. This implies DSIcore's linkage condition is not fulfilled and no strand gets detected. Suppose that a type recovery tool is able to recover the type information; this recovered type information is called a *Logical Type (LT)*. As perfect type information of the TT is lost, it might be possible that the Logical Type (LT) is inferred incorrectly, e.g., the LT is identified to be the same for both memory chunks, i.e., LT1 in Fig. 11.3. DSIcore now detects a fulfilled linkage condition and, thus, creates a strand, resulting in a false positive.

Similarly to Fig. 11.3, it is also possible that the inferred types are considered differently, though the ground truth is not, as shown in Fig. 11.4. This would again lead to a false negative, as was the case with the missed nested structs in Figs. 11.1 and 11.2.

The implications for DSIcore's analysis are manifold. For one, the precision of the strands gets diminished, as strands might become shorter or longer than their true length, depending whether false negatives or false positives occur. Strands can also suffer from both effects at the same time, where a strand is cut short at one side and extended at the other. The consequences are a loss in precision of DSIcore's analysis. This can result in either the creation of noise that can lead to misinterpretations, e.g., detecting nested child elements where none are present,

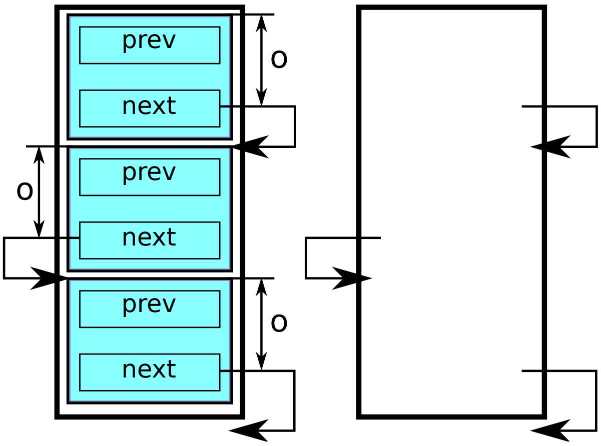


Figure 11.2: A linked list running through multiple nested structs inside an enclosing struct, both with type information (left) and without type information (right).

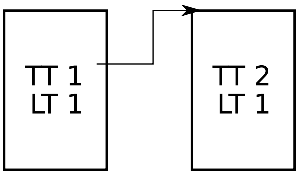


Figure 11.3: Different true type as given by source code versus identically inferred logical type.

and blurring the shape such that the true shape is not recognizable as DSIcore's predicates for the shape fail.

Further, some characteristics of a strand can get lost, such as cyclicity if the head node of the strand is cut off. The opposite is less likely to occur, i.e., introducing cyclicity by wrongly extending a strand. The reason for this is that, in order to detect cyclicity, an explicit pointer would need to be present from the last element to the head element, which is less likely to occur if cyclicity is not explicitly desired by the programmer.

The problem might get worse, if false positives lead to the detection of more strands or, vice versa, if false negatives lead to the detection of less strands. In the case of missed strands, the analysis might be rendered useless in the worst case as no DDS might be detected, or that at least parts of a DDS are missing, leading again to misinterpretations. Examples of such a loss of precision are, e.g., missed parent-child relations or wrong classifications, such as detecting only a nesting relation instead of a Binary Tree (BT).

The final aspect that heavily suffers from imprecise type information is the detection of the connections between strands. Consider a typical parent-child relation on indirect nesting as the ground truth, as seen on the left in Fig. 11.5. Here

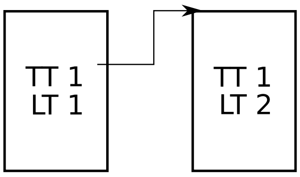


Figure 11.4: Identical true type as given by source code versus differently inferred logical type.

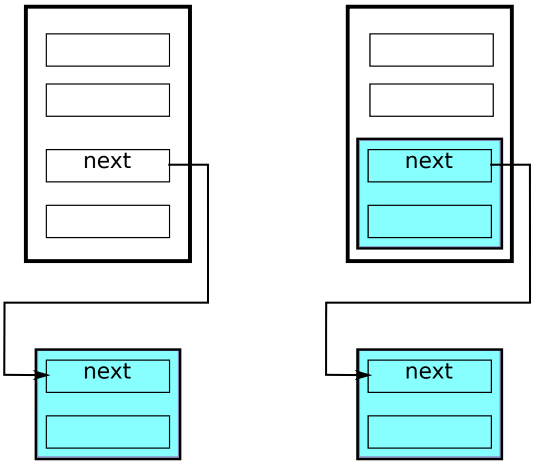


Figure 11.5: Memory chunks with a parent (big chunks at the top) child (blue chunks) relation. On the left, no nested child element inside the parent is present; on the right, a nested child element inside the parent is present.

the head element of the child list lies outside the parent. If a wrong LT is inferred inside of the parent, which happens to be of the same type as the child element with which the child list is extended to start inside of the parent, as shown in Fig. 11.5 on the right. This not only wrongly extends the child strand by one element, but also changes the nesting relation from indirect to overlay. The same can also happen vice versa, resulting in indirect instead of overlay nesting.

In summary, lost type information can have a severe impact on DSIcore's analysis. The effects can have different severity levels. In the least severe case, the strand length is wrong, but the overall data structure gets detected, e.g., only extending or shortening the SLL in a parent-child relation without other side effects such as influencing the parent child connection. Therefore, the analyst would still get the right high-level view upon the data structure and draw the right conclusions regarding the implementation details of the DDS operations. As soon as properties such as cyclicity of the strands or the connections between strands are affected, the impact becomes more severe, as this sets an analyst on the wrong track not only with the high-level interpretation of the DDS, but also when drawing conclusions about the operations performed on the DDS. For example, consider an insert into a DLL, which most likely differs from that into a CDLL. In the worst case, the analysis either shows no DDS at all or a plain wrong interpretation of the DDS. The effects of lost type information have been confirmed during the benchmark of DSIbin and, thus, are also discussed in the benchmarking chapters, i.e., Chs. 15 and 17.

12 Data type recovery tools

As laid out in Ch. 11, perfect type information is crucial for DSIcore's analysis and, thus, also for the analysis of binaries with DSIbin. As no type information is available in binaries, the type information needs to be inferred separately. This chapter surveys some of the available data type recovery tools, which can potentially assist DSIbin to recover the type information needed by DSIcore. The tools differ in various aspects, e.g., on which input they operate, what kind of analysis they perform and what kind of type information is recovered, as shown in Tab. 12.1. There are two factors when choosing a type recovery tool which are important to DSIcore's analysis. The first is the quality of the recovered type information and the second is the input format, as the latter has implications on the quality of the analysis that can be performed by DSIbin. This means that even if the recovered type information is acceptable, the tool might not be suited if it operates on the wrong input format. This aspect is analysed in Sec. 12.1. The quality of the recovered type information is discussed in Sec. 12.2.

12.1 Input formats of the type recovery tools

There are various possible input formats used by the type recovery tools. As the input format required by the analysis tools should be used by DSIbin as well, it is important to analyze which limitations upon DSIcore's analysis are imposed by the different input formats. The reason for choosing the same input format for both the type recovery tool and DSIbin lies in the fact that it is not easily possible to mix the different levels of information given by different input formats. A concrete example for such an information mismatch is the absence of information about allocation sites in memory snapshots, in contrast to the availability of this information when instrumenting binary code. Allocation sites are the places during program execution where chunks of memory get allocated. The allocation site can for example be expressed by the call stack leading to a `malloc`.

The implications imposed upon DSIcore's analysis given a concrete input format concern either engineering aspects or conceptual limitations specific to DSIcore's analysis, which are explained in this section. The input format used by a type recovery tool is shown in Tab. 12.1 under column **Input format**. The input can either be a memory snapshot (**m**), which is discussed in Sec. 12.1.1, or binary code (**b**), discussed in Sec. 12.1.2.

Table 12.1: Overview of type recovery tools: the analysis can be **s**tatic, **d**ynamic or both; the input is either **b**inary code or a **m**emory snapshot; features marked with a ? are not clear from the literature

Tool	Analysis	Memory chunk size	Pointer detection	Outermost struct	Inner struct	Type of struct members	Semantics	CMA	Input format
Divine [43]	s	✓	✓	✓	✓	✓	✗	✗	b
Howard [98]	d	✓	✓	✓	✓	✓	✓	✗	b
ARTISTE [49]	d	✓	✓	✓	✗	✓	✓	✗	b
REWARDS [80]	d	✓	✓	✓	✗	✓	✓	✗	b
TIE [78]	s/d	✓	✓	✓	✗	✓	✗	✗	b
SigPath [101]	s	✓	✓	✓	✗	✗	✗	?	m
MemPick [69]	d	✓	✓	✓	✗	✗	✗	✗	b
DDT [74]	d	✓	✓	✓	✗	✗	✗	✗	b
HeapViz [39]	s	✓	✓	✓	✗	✗	✓	?	m
RDS [92]	d	✓	✓	✓	✗	✗	✗	✗	b
Laika [60]	s	✓	✓	✓	?	✗	✗	?	m
HeapMD [57]	d	✓	✓	✗	✗	✗	✗	✗	b

12.1.1 Memory snapshots

This section discusses the challenges for DSIbin when considering type recovery tools that operate on memory snapshots as an input for the analysis, such as [39,60, 101] of Table 12.1. Those analysis techniques take a snapshot of the live memory, i.e. the Random Access Memory (RAM), during runtime. Afterwards the memory layout is inspected to infer the data types and DDSs. The advantage of taking snapshots is that it is even possible to recover volatile memory to a certain extend after a warm reboot [102]. This renders countermeasures of malware useless to detect running anti-malware software.

In order to discuss the implications of using memory snapshots for DSIbin, let us briefly recap the structural and temporal repetition as applied by DSIcore, because especially temporal repetition is affected by the limitations imposed by using memory snapshots. We do this by looking at a SLL parent with indirect nested child DLLs as seen in Fig. 12.1. Here the heap is shown at time step t and $t+1$. At time step t, one child DLL is in a degenerate shape. At time step t, DSIcore performs structural repetition over all child DLLs. But only with the knowledge about the heap state at step $t+1$, DSIcore is able to perform the temporal repetition to outweigh the degenerate shape of the child DLL. Without this contiguous heap

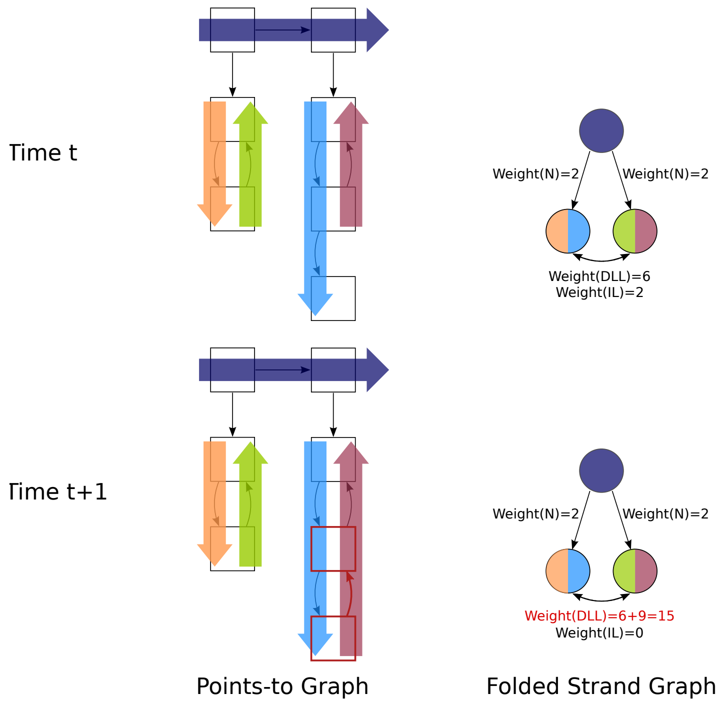


Figure 12.1: Temporal repetition between time step t and $t+1$ used to outnumber the degenerate shape of the child double linked list at time step t.

state information the chances increase that misinterpretations of the DDSs occurs due to information sampled during periods of degenerate shapes. We can now discuss the various aspects when using memory snapshots as input for DSIbin.

One of the key challenges is that DSIcore operates on a trace of contiguous memory changing events, where at every such event the type information is required. Thus, using memory snapshots can be viewed from two possible directions when using them with DSIcore. One would be to make the type information available to DSIcore for each instruction within the event trace, to be able to perform DSIcore's analysis ”as-is”, i.e., to analyse the complete instruction trace. However, this is contrary to the coarse-grained sampling rate of memory snapshots, which normally are only taken once or at most sparsely during program execution. Therefore, the type information is unavailable for each memory changing instruction as required by DSIcore. This imposes a problem for DSIcore's temporal repetition approach to accumulate evidence for the true DDS shape, as explained with Fig. 12.1. To remedy this limitation of memory snapshots, one could simulate the contiguous trace generation by taking memory snapshots analogous to DSIcore's instrumented source code, i.e., at every memory manipulating event. This imposes a huge overhead in terms of execution time and memory consumption, which seems impractical. Further, a literature review could not answer the question, whether the type information would be stable across all taken snapshots. This could lead to false positives and negatives, which might even alternate over the program analysis. Additionally, memory snapshot tools normally operate on WinDbg memory dumps [60, 101], TEMU snapshots [101] and Sun's HPROF tool format [39]. Relying on special memory snapshot formats is a strong assumption and limits the generality of DSIbin. The only advantage is that more (meta) information is available within those snapshots than within DSIcore's trace, e.g., HeapViz uses the rich meta information found in Java heap snapshots.

The other possible direction is to let DSIcore operate exclusively on the taken snapshots for which type information is available. The problem here is that the PTG represented by a snapshot is not tracked over time, i.e., the instruction trace leading to the current PTG layout is unknown. Thus, in order to let DSIcore operate on the PTG, an event trace would need to be created artificially by recording a malloc event for every identified memory chunk within the snapshot and a pointer write for every pointer that connects those memory chunks. The sequence of such events can only be chosen arbitrarily, as timing information is unavailable within one memory snapshot. Therefore, the evidence accumulation could be led astray by the arbitrarily chosen sequence of events; for example, the cyclicity connection of a CDLL is only established with the last event of the trace, which would then lead DSIcore to outweigh the CDLL with a DLL. This comes to no surprise, as sequentializing a memory snapshot removes all in-between states of the DDS.

In summary, both available directions for utilizing memory snapshots with DSIbin challenge DSIcore's temporal repetition approach. The possibilities to remedy this problem as discussed are not an option for DSIbin, as they are either impractical or could easily lead to falsely detected DDS. Relying on structural repetition alone, which is still available with both approaches, is insufficient to reliably detect the true DDS shape, e.g., when sampling only degenerate shapes.

Therefore, the disadvantages of memory snapshots outweigh the advantage of malware forensics; thus, memory snapshots are not considered as a suitable input format for a first implementation of DSIbin. This rules out the tools that operate on memory snapshots as seen in Tab. 12.1, i.e., [39, 60, 101]. However, snapshots could be used as input to DSIbin as future work, where the shortcomings could be investigated more thoroughly in order to see if they can be mitigated.

12.1.2 Binary code & byte code

Binary code can be instrumented directly and executed to record a trace of memory events, similar to the source code instrumentation performed by DSI. This can be achieved with instrumentation frameworks such as Intel's Pin framework [82] and DynamoRIO [48]. Binaries can provide different levels of information: (i) debug information, e.g., variables and functions, (ii) symbol tables where function names are still present, and (iii) stripped binaries, where unneeded symbols are removed from the symbol table of the binary as well. DSIbin aims for stripped binaries to be more generally applicable, as (i) and (ii) cannot be taken for granted when dealing with binaries that are created outside of the control of an analyst.

For completeness, byte code is also considered as an input format, though it is basically subsumed in the already discussed memory snapshots and binary code, as both techniques can be applied to byte code programs. Byte code has the advantage that it carries more meta information, e.g., type information, class and method names, similar to debug information in binary code [53]. However, relying on such information would hinder the general applicability of the approach and is thus avoided for DSIbin. Actually, none of the listed tools of Tab. 12.1 explicitly uses byte code as input, with the exception of HeapViz [39] that operates on Java-based heap snapshots. In this case, the usage of such additional information found in the snapshot might be imprecise, as a DLL is labeled `LinkedList` as found in the class name of the objects [39].

12.2 Recovered type information

As seen in Tab. 12.1, the different tools provide different granularity of type information. As discussed in Ch. 11, this has implications on the precision of DSIcore's analysis. In the following, we discuss the type information provided by the tools of Tab. 12.1 and how this influences the precision of the analysis. First, we have the **Memory chunk size**, which means that allocated memory is tracked and at least the size information is available, i.e., no type information within the allocated memory is available. As can be seen in Tab. 12.1 all tools provide this information. However, no linkage condition can be established by DSIcore with this information alone, as there is no information about how memory is linked together. This information comes with knowledge about pointers (**Pointer detection**). As soon as the linkages between memory are present, it is possible to correlate the memory chunks accordingly. However, for DSIcore, these two kinds of information are not necessarily enough to establish a linkage condition leading to a detected DDS. This comes down to a design decision whether one wants to consider two chunks of memory as being of the same type because they have the same size and they are linked by a pointer. Due to the already discussed aspects of false positives and negatives in Ch. 11 that come with such an assumption, this level of information is not considered to be sufficient for an analysis by DSIcore.

The next step is to determine a logical type of the **Outermost struct**. This can be done by various techniques, e.g., applying a Logical Type (LT) to all memory chunks allocated at the same allocation site, i.e., the unique position of the `malloc` as given by the callstack leading to the allocation [49, 69, 74, 98]. Another possibility is the usage of type sinks, where well known interface functions, e.g., from libraries or system calls, are used to type operands [49, 80, 98]. It is also possible to use meta information directly [39], but this is not desired by DSIbin as it restricts its generality. With the information about outer structs being available, DSIcore is now able to perform a useful DDS analysis, though it can still only deal with DDS that occupy the memory chunk as a whole, i.e., DSIcore's notion of cells is not yet fully supported. In order to exploit DSIcore's fine grained cell abstraction, the type recovery tools are required to detect **Inner struct**s. As can be seen in Tab. 12.1, only Divine and Howard reveal information about inner structs.

Additional information about the **Type of the struct members**, such as the primitive types, is important when considering binary compatibility of memory chunks. This can be an indication for type equivalence between memory chunks, if it cannot be determined by other means as discussed previously. Such low level information can be inferred by the type of operands when looking at CPU instructions, or again by using type sinks. The latter can also reveal the **Semantics** of the recovered type information, such as revealing that a struct is used for establishing a network connection. The primitive types are indeed already utilized by the actual DSIbin implementation, as discussed in Ch. 16. The information about the semantics can easily be added to decorate the created SGs of DSIbin to further enhance their usability, though this is out of scope of this dissertation.

As DSIcore's memory abstraction allows us to handle CMAs transparently, it would be beneficial if the recovery tool would support CMAs. However, we leave this for future work. As shown in Tab. 12.1, none of the tools explicitly mentions CMAs. However, in conjunction with tools, such as [54] that explicitly detect CMAs, type recovery tools could be made aware of the presence of CMAs, which might help in performing their analysis.

When looking at Tab. 12.1, almost all type recovery tools can provide the type of the outermost struct. From those candidates, the ones operating on memory snapshots have already been eliminated, as discussed previously. The remaining candidates do not deal with inner structs. Only two recovery tools mention them explicitly, namely Howard and Divine. As the detection of inner structs is highly desired by DSIbin to get as close as possible to the precision of DSIcore, those two tools are in the main focus for DSIbin.

12.3 Comparison between Howard and Divine

Due to Divine's [43] and Howard's [98] capability to deal with nested structs, they are the first choice for combining them with DSIbin. In this section, we discuss which alternative is most suited for DSIbin.

We start with Divine, which statically analyses Windows binaries. It is able to type memory regions both on the heap and the stack. Types can be either primitive or complex data types, e.g., arbitrarily nested structs. Divine employs a value-set analysis [43] in combination with an algorithm for aggregate structure identification, and exploits how memory is accessed in a binary to infer how the data is laid out in memory. One can think of a code instruction that accesses eight bytes in a sequence at a particular offset within a memory region. This information would lead Divine to assume that this eight byte sequence corresponds to a variable or field within the memory region. However, according to Howard [98], memcpy-like functions can hinder Divine's utility, as these bulk memory accesses used, e.g., for initialization, blur the actual usage patterns of the memory.

Howard instead handles memcpy-like functions in C/C++ binaries when performing its dynamic analysis. It infers types for local variables in stack frames and monitors heap allocations for typing dynamically allocated memory. For this, Howard collects: (i) pointer information that is also used subsequently to reveal nested structs information, and (ii) infers primitive types by monitoring instructions and type sinks. Howard identifies allocation sites to apply LTs to heap memory and uses function addresses to identify stack frames. The results of the analysis are reported for each allocation site and stack frame.

While both tools offer a similar level of type information to exploit DSIcore's fine grained heap abstraction, the following engineering aspects favored Howard: it (i) operates under Linux as does DSIbin, which avoids the time-consuming porting of Divine or DSIbin to the corresponding platform, and (ii) deals with memcpy-like functions in C/C++ binaries, where Divine seems to focus on C++.

In the following chapters we show Howard and DSIbin can be combined. It however becomes eminent that Howard still has some limitations relevant for DSIbin. Additionally, we demonstrate how DSIcore's algorithm itself can be employed to improve Howard's type information.

13 Binary code instrumentation

In the previous chapter it was laid out that the type recovery tool Howard is being used to infer the missing type information from binaries. Howard works by conducting a dynamic analysis on binary code; thus, DSIbin also has to operate on binary code so that it can be combined with Howard. This chapter discusses the technical aspects of the binary instrumentation used to generate an event trace for DSIbin, similar to that created by the source code instrumentation used by DSI. While DSI employs CIL for instrumenting the source code, DSIbin uses the industrial strength Intel Pin framework [82] for dynamically instrumenting x86 C/C++ binaries. In the context of DSIbin, pointer reads and writes and memory (de-)allocations both on the heap and the stack are monitored in x86 C/C++ binaries. For this purpose, a socalled *Pintool*, i.e., a program that uses the Pin framework, is created within the context of this dissertation. The Pintool is publicly available at https://github.com/uniba-swt/DSIbin-inst. In the next section, a short overview of Pin is given and, subsequently in Sec. 13.2, instrumentation details relevant for DSIbin are discussed.

13.1 General overview of Pin

The architecture of Pin is divided into two parts: (i) the general Pin framework which provides routines for instrumenting the binary, and (ii) the part for analysing the executed binary. The instrumentation routines are used to decide where to insert an instrumentation into the binary. The actual instrumentation code that gets executed are the analysis routines. As the name implies, they perform the analysis logic during execution of the instrumented program.

An actual tool implementation is done in a *Pintool*. Pin performs a dynamic instrumentation, i.e., no recompilation of the code under analysis is necessary. This is crucial for DSIbin; otherwise, source code would again be required. The availability of source code is a very strong assumption, and this case is already covered by DSI.

13.1.1 Instrumentation and analysis routines

The instrumentation routines are registered for different events during execution of a binary. For example, one can register an instrumentation function that is called whenever a binary image, e.g., a library, is loaded into the main executable during execution of the binary. The registered instrumentation routine then inspects what type of image is loaded, and, e.g., searches for special routines such as `main` to find the entry point of a program. Further instrumentation routines are available to inspect each instruction of the binary. Again, the instrumentation routine can then implement the logic to decide which instruction to instrument.

In general, the instrumentation routines are not called periodically, but only once for each event, e.g., only once for each instruction. Each instrumentation routine then installs the call back, i.e., the analysis routine, which will subsequently be called whenever an event for which it was registered is processed during execution of the binary. The instrumentation routines can pass an arbitrary amount of parameters to the analysis functions. This can be Pin-specific information, such as the value of the stack pointer, or custom parameters.

The registered analysis routines can be configured to happen before or after an event occurred. The analysis routines implement the actual logic of performing the desired functionality when analysing a binary. As the analysis routine is called during execution of the binary the execution time of the analysis routine has an impact on the overall slowdown of the binary under analysis.

13.2 Instrumentation details for DSIbin

This section presents some of the specific implementation details for the instrumentation used by DSIbin, such as the tracking of heap allocated memory and registers. The latter is unique to DSIbin, as DSI operates on source code level, where such low level details are abstracted away by instrumenting the high-level C code. The instrumentation part for DSIbin consists of 2800 lines of C++ code.

13.2.1 Instrumentation of malloc and free

In order to trace the `malloc` and `free` routines, an instrumentation function is installed to monitor the loading of images such as a the `libc` standard C library. The instrumentation function is registered with the method call `IMG_AddInstrumentFunction`. For every loaded image, the specific keywords `malloc` and `free` are searched and the appropriate callback methods are installed. Thus, more standard memory management functions can easily be added, if the need arises. It should even be possible to use MemBrush [54] to detect CMAs inside the binary and hook specifically into those routines to deal with non-standard allocators, but this has not been further investigated.

Once the callbacks are installed, they are used to monitor the dynamic memory behaviour of the binary.

13.2.2 Startup and shutdown of the program under analysis

The startup and shutdown phase of the program is skipped from the analysis, as a lot of boiler plate code gets executed, which obviously includes memory operations such as background data structures for bookkeeping purposes. With DSIbin we are interested in the DDSs as used by the programmer. Therefore, the analysis starts with the call to `main`, as found by searching for the specific string within the loaded images, as described in Sec. 13.1.1. Additionally, the analysis stops after the `main` function terminates.

13.2.3 Tracking the stack and the heap

DDSs can be distributed across the heap and the stack, e.g., the head node of a LKL can be placed on the stack with the `LIST_HEAD` macro found in list.h [15]. Therefore, the stack needs to be monitored by the Pintool, too. The stack frames are also shadowed and contain information about the function address to which a particular stack frame belongs, the start and end address and the size of a stack frame. Additionally, the inferred type from Howard is stored. Thus, the shadow stack allows one to keep track of the live stack variables; and handle situations like the *red zone optimization* [86].

The red zone is a 128 byte area beyond the current stack pointer. It is required by the x86-64 Application Binary Interface (ABI) [86], and it s not modified by signal or interrupt handlers. This optimization is used in cases where a callee is a leaf function. In this case, the red zone can be used as the local stack of the callee without the need to adjust the stack pointer. As this technique is required by the x86-64 ABI, compilers such as GCC [8] adhere to the standard and enable the red zone by default, which makes handling such situation mandatory for DSIbin.

The heap is tracked by recording each allocated heap memory. The memory is identified by the call stack leading to its allocation site. The call stack is constructed by pushing the current stack pointer value when a `call` instruction gets executed and, conversely by popping the last value when a `return` statement is executed.

For this, the heap memory is shadowed, as all allocated live memory chunks are being tracked by the Pintool. The shadow heap carries the information about the start and end addresses, the chunk size, the call stack where the memory has been allocated and the type inferred by Howard. Whenever memory is accessed, the shadow heap is queried to ensure that live heap memory is addressed and manipulated. If memory inside the live heap is manipulated, the shadow heap is used to get information about the field type that was accessed, i.e., the type information retrieved by Howard. Additionally, the shadow heap allows sanity checks, such as avoiding double frees and the usage of unallocated memory.

13.2.4 Modeling registers

When inspecting a program at binary level, the only valid reference to live memory might be inside of a register. This differs slightly from inspecting source code, where such low-level details are abstracted away by dealing with pointer variables directly. Therefore, registers need to be treated analogously to variables; otherwise, the memory leak detection algorithm of DSIcore, as described in Ch. 4, would report a false positive on leaked memory and remove the memory.

Again, the registers need to be shadowed to keep track of their state, i.e., whether they are currently storing a pointer value or not. Therefore, each register gets the attribute `POINTER` or `NOPOINTER` attached to it, depending on its current value. Then, each register manipulation is monitored by checking, whether a register stores a live heap address. The live heap address is determined with the help of the shadow heap as described in Sec. 13.2.1. Whenever a live heap address is stored, the pointer connection is recorded, otherwise the pointer connection is removed.

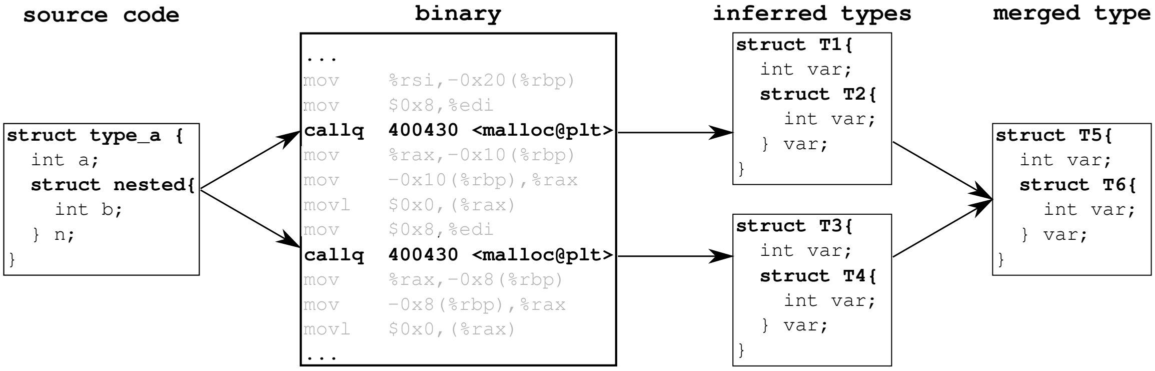
14 Combination of Howard and DSIcore

This chapter brings together the discussion of the last three chapters by actually combining DSIcore and Howard [98] to create a first naive implementation of DSIbin. In this implementation, CIL is replaced by Intel's Pin framework [82] to execute the instrumentation as discussed in Ch. 13. The type information demands of DSIcore are contained in Ch. 11 and are provided, to a certain extend, by Howard.

One of the fundamental problems when recovering type information in binaries is that of *type merging*, which occurs when the same type of memory is allocated at different allocation sites, i.e., the allocation sites have different call stacks. This results in a many-to-one relationship between allocation sites and a type, which imposes a problem for DSIcore's strand detection, as each allocation site would be treated as a separate type leading to missed strands as discussed in Ch. 11. The problem is shown in Fig. 14.1, where `struct type_a` is given by source code, which then gets allocated twice inside the program by the two `callq`s to `malloc`. When using a type recovery tool that does not support type merging, the resulting types might look like the `inferred types`, i.e., `structs` `T1-4`. Only with type merging, the four types are combined again as seen on the right in Fig. 14.1. Because there is no explicit type information available in a binary that explicitly indicates type equivalence, a separate inference step is required.

Despite all advantages of Howard as discussed in Ch. 12, type merging was not performed in the version of Howard described in [98]. Therefore, the authors of Howard thankfully modified the initial version of Howard mildly for us to perform type merging between structs identified by Howard. This is done by tracking instructions and pointers with a *taint analysis algorithm* [96] to monitor which heap memory chunks they access. If memory chunks allocated at different allocation sites are touched by the same instructions or pointers and if they are binary compatible, i.e., have the same size and the same inferred primitive types, then they get merged. Note, however that this does not cover all situations, e.g., Howard does not merge nested types and misses nested structs if they are placed at the head of the surrounding struct. How these limitations are overcome by DSIbin is discussed in Ch. 16.

This chapter first presents in Sec. 14.1 the overall architecture of DSIbin, as shown in Fig. 14.2, and subsequently zooms into the binary frontend Pintool, called *DSI binary frontend*, in Sec. 14.2.



```
struct type_a {
  int a;
  struct nested{
    int b;
  } n;
}
```

```
...
mov     %rsi,-0x20(%rbp)
mov     $0x8,%edi
callq   400430 <malloc@plt>
mov     %rax,-0x10(%rbp)
mov     -0x10(%rbp),%rax
movl    $0x0,(%rax)
mov     $0x8,%edi
callq   400430 <malloc@plt>
mov     %rax,-0x8(%rbp)
mov     -0x8(%rbp),%rax
movl    $0x0,(%rax)
...
```

```
struct T1{
  int var;
  struct T2{
    int var;
  } var;
}
```

```
struct T3{
  int var;
  struct T4{
    int var;
  } var;
}
```

```
struct T5{
  int var;
  struct T6{
    int var;
  } var;
}
```

Figure 14.1: Type merging in binaries (figure reproduced from talk at ASE2017 on [94]).

14.1 DSIbin: Combining Howard and DSIcore

This section discusses the naive tool chain as depicted in Fig. 14.2, without the component DSIref that is explained in Ch. 16. The three main components are (i) the core algorithm of DSI, i.e., the offline part of the analysis that performs the DDSs detection, what we have already introduced as DSIcore, (ii) the Pin DSI binary frontend that replaces DSI's source code CIL frontend, and (iii) Howard to infer lost type information from binaries. As with DSI, the result of executing the tool chain are the identified data structures. DSIbin uses the core algorithm DSIcore of DSI *as-is*.

Both the DSI binary frontend and Howard take the stripped binary under analysis as input for their dynamic analyses. The main synchronization point between these two components are the call stack for typing heap allocated objects and function addresses for typing the local stack of functions. Howard provides the already merged type information to the DSI binary frontend.

The binary frontend takes Howard's type information and produces the execution trace analogous to the DSI source frontend. To do so, it creates explicit events in the execution trace whenever memory, i.e., stack frames or heap objects, gets allocated and deallocated. It also tracks pointers that are written and destroyed on the heap, the stack and in registers. The type information from Howard is directly embedded inside the generated event trace and gets passed to the DSI core algorithm. DSIbin consists of 3K LOC of C++ and interfaces with DSI and Howard.

14.2 Technical details of DSIbin's binary frontend architecture

This section gives a short summary of the architecture of the Pintool implementing the binary frontend for DSIbin that replaces CIL. The binary frontend uses the techniques discussed in Ch. 13 to generate an execution trace analogous to

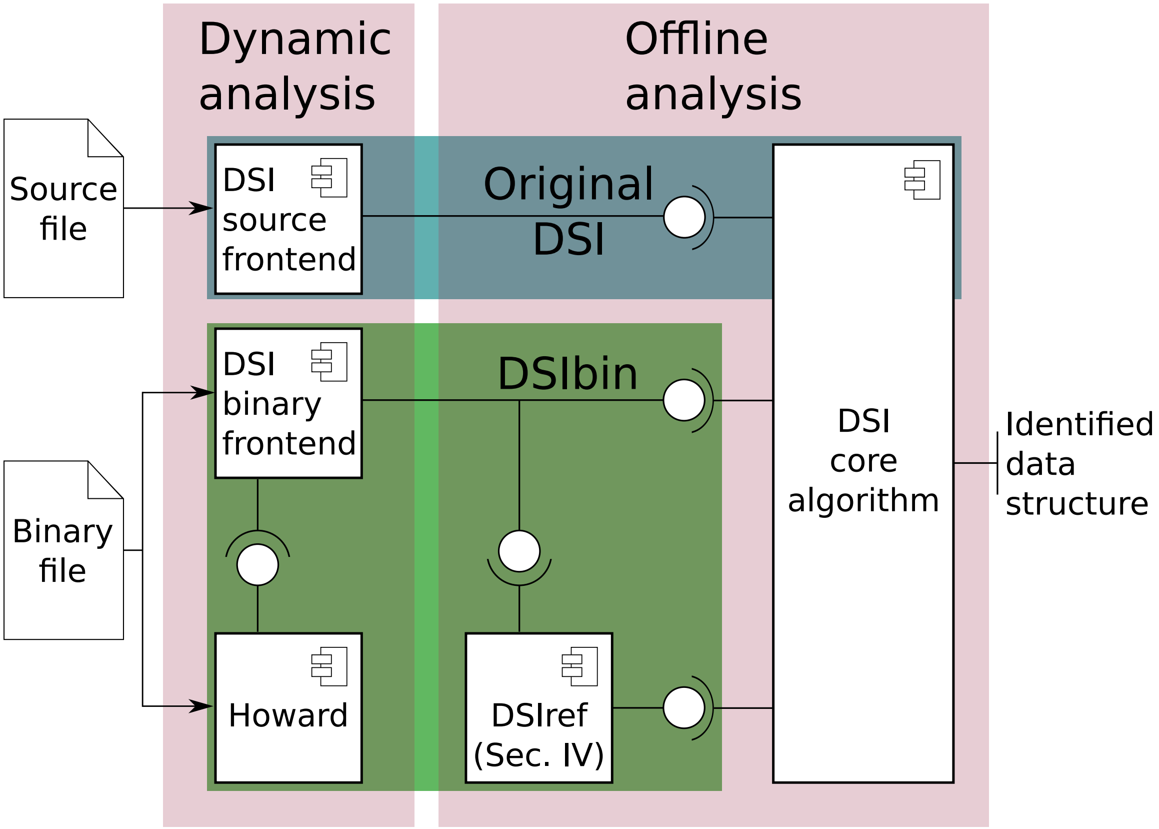


Figure 14.2: Overview of our DSIbin tool chain (figure reproduced from our publication [94]).

CIL, which subsequently gets passed to DSIcore for executing the DDS analysis, as shown in Fig. 14.2.

Type parser. The type parser creates two lookup tables for the types inferred by Howard, one for the heap and one for the stack. The lookup is a key-value store that uses the callstack as the key into the lookup for heap allocated memory. Howard's type merging information is reflected within this lookup table, as each merged callstack by Howard gets the same type assigned as a value. For stack memory, the key into the lookup table is the function address.

The call stack, the stack and the heap. We present some low-level implementation details regarding the handling of memory both on the heap and the stack by the DSI binary frontend. The main concept is to shadow the various components, which means to duplicate the states of the components within the DSI binary frontend for the analysis.

The main synchronization point between Howard and DSIbin is the call stack. Howard identifies heap allocated memory by its call stack leading to the allocation site, e.g., a `malloc`. Further the call stack holds the last address of a function for which a stack frame has been created. This is required as the stack memory is identified to Howard by the function address to which a stack frame belongs.

We now consider briefly how DSIbin handles the call stack in order to synchronize with Howard. DSIbin tracks the call stack by a shadowed call stack that records the execution of the program. Thus, the shadowed call stack is designed to grow and shrink analogous to the call stack of the program under analysis. This means, whenever a `call` occurs during program execution, a `push` of the current program address is performed on the shadow call stack, and whenever a `return` occurs, a `pop` gets performed.

Now that the main synchronization point between Howard and DSIbin is covered, we detail the handling of the stack memory, as the call stack is closely interwoven with the stack memory. The DSI binary frontend again models a shadow stack, which duplicates the stack frames of the program under analysis. Each allocated stack frame created by the program under analysis gets duplicated by the DSI binary frontend; it is pushed onto the shadow stack when a function call occurs and popped from the shadow stack upon a return. A shadowed stack frame is identified by its function address, which is retrieved from the shadowed call stack. The function address of the stack frame is subsequently used to fetch type information from Howard for a particular shadowed stack frame. This information is stored in the previously discussed type parser. Further, a shadowed stack frame keeps its size and its start and end addresses. This allows us to compute the type information from Howard, given a concrete local stack address.

With the call stack and shadow stack now being covered, the heap memory needs to be modeled. As before, the DSI binary frontend shadows the heap memory. In case of an allocation, a new shadowed heap memory element gets created, by querying the shadowed call stack for the sequence of functions calls that lead to the allocation site. This information is stored within the shadowed heap chunk, together with the information about its size and its address on the heap. The call stack is used to again query the type parser, which stores Howard's type information for heap memory as well as the stack memory. With this information, the shadowed heap provides type information for a concrete heap address.

In summary, those three components, i.e., the call stack, the shadow stack and the shadow heap, form the backbone of the binary frontend Pintool, as they allow to obtain detailed information about the live memory and the call stack at every important time step of DSIbin's analysis during program execution. It is also possible to conduct certain sanity checks upon the live memory, e.g., whether a referenced heap memory chunk is still live, or whether a red zone optimization for the stack is present.

Main component of the Pintool. The main component brings together the previously discussed components by hooking into the appropriate machine instructions, e.g., the call and return statements for monitoring the call stack and the functions modeling the heap. The main part of the binary frontend first sets up the type parser in order to have the type information available. Subsequently, it injects the callback hooks for the monitored machine instructions and, afterwards, it monitors the running executable, e.g., for changes of the call stack, heap and stack allocations and pointer writes. The information is recorded into an XML trace file, analogous to that provided by the CIL source code instrumentation. After recording the file, the Pintool stops executing and passes the file onwards to DSIcore for further processing, as seen in Fig. 14.2.

15 Benchmarking of naive combination

The implementation of our naive combination of Howard and DSI is benchmarked in this chapter, by applying the tool chain of Ch. 14 against real-world examples [20, 32, 34], standard textbook examples [104, 108], examples taken from the shape analysis literature [23], and hand-written examples. The hand-written examples challenge our tool chain in a white box fashion. In total the resulting benchmark comprises 30 examples. Our tool chain is able to already correctly identify 10 of the benchmark's examples. The successful examples improve upon related work [49, 69, 74], e.g., by the recognition of SLs and various forms of nesting.

A general overview of the benchmark is given in Sec. 15.1, and the positive examples are discussed in Sec. 15.2. Following this, a detailed analysis of the 20 negative examples is presented in Sec. 15.3, which reveals that most of the recognition failures result from insufficient and imprecise type information. In Sec. 15.3.4, it is discussed how the theoretical discussion about the loss of type information in Ch. 11 is confirmed in practice by the failed recognition of DDSs.

15.1 Benchmark: General overview

The prototypical tool chain is benchmarked against C/C++ code samples. The diversity of the examples is chosen to evaluate the robustness of the approach in the light of various implementation techniques of DDSs. Specifically the benchmark consists of four text book examples (`tb-1`,`tb-2` [104] and `tb-3`,`tb-4` [108]), which cover SLLs and DLLs, including higher level concepts such as a stack built on top a SLL. Five examples are taken from the shape analysis literature, i.e., from the Forester/Predator Git repository [23] (**lit**), which feature various LKL implementations. Further, five (extracted) real-world examples (**(e)r**) are chosen, namely the region clipping library of VNC (`hvnc2/libs/libvncsrv/rfbregion.c` found in Carberp [34], `r-3`), the benchmarks treeadd [20] (`r-2`), and binary-trees-debian [32] (`r-1`). Finally 16 self-written synthetic programs (**syn**) are selected to perform a white box testing of our approach. The source code of the synthetic examples is available from https://github.com/uniba-swt.

As the benchmark is composed of common DDS, such as (cyclic) lists and trees, and arbitrary interconnections of those components, e.g., indirect and overlay nesting, the overall variety of the benchmarked DDS is guaranteed. The examples each contain a single DDS, which might be composed of multiple different DDS, for convenience of the evaluation to isolate interesting aspects of the analysis. The general approach can handle more than one DDS per source code file.

The ground truth of the examples is determined by manual source code inspection and is listed under the **Code** section. As DSIbin operates on binary code, it is natural that it can handle not only C code, as does DSI, but also C++ code. In order to prove this, the examples are a mixture of both C and C++, as listed in the **lang** column of Tab. 15.1. The shape of the DDSs, including their interconnections, are listed in the **DS** column. The examples range from SLs to BTs to (cyclic) lists. The interconnections are either established via indirect nesting (Ni) and overlay nesting (No) from parent to child. As DSIbin is a dynamic approach that creates its evidence for a DDS over the lifetime of a DDS, we report the trace length of the examples in Tab.15.1. The programs are run until termination. For completeness, the Lines of Code (LOC) for the examples are also listed in Tab. 15.1, though this is not a measure of the complexity of a DDS. For example, when looking at `lit-5`, one can see that a quite complex and challenging DDS consisting of two DLLs running in parallel, can be constructed with only 36 LOC.

The important characteristics of the programs are listed as well, as they have an impact on the detectability of the DDS such as flattened access of struct members (**flat**). As the placement of the nested `structs` inside the enclosing node influences, whether Howard can detect the nested `struct` the nesting of a struct at the head of the surrounding struct (**n@h**) is reported. Further, the placement of some other `struct` *not* at the head of the surrounding struct (**n**) is indicated. Column **m** lists the possibility to merge allocation sites, where some DDS only provide partial **(pt)** merge opportunities. Because the related work discussed previously in Ch. 3, does not merge DDS distributed across the heap and the stack, column **h/s** indicates whether the DS is distributed across the heap *and* the stack. Such a behaviour can be seen with the LKL, when the head node of the list is stored on the stack, and the remainder of the list is dynamically allocated on demand during program execution. Some of the examples have nested structs that are only payload, i.e., do not participate in the linkages of the DDS. These examples are marked by **(p)**, as these access patterns do not influence the detection of the DDSs.

The next columns are divided into the columns under the headings **Naive Combination** and **Sophisticated Combination**. The first describe the performance of using Howard's type information 'as is', which is discussed in the current chapter. The sophisticated combination is discussed in Ch. 16. The naive combination cannot deal with nesting at head (**n@h**), does not merge neither heap and stack allocated nodes of the same type (**h/s**) nor nested elements. These scenarios are left for the sophisticated combination in Ch. 16. Thus, the naive combination concentrates on the correct identification of the DDS **(rec)**, as well as the detection of `structs` that are not located at the head **(n-d)**. Additionally, the merging of allocation sites is reported **(m)**. The only difference between the naive and sophisticated combination sometimes lies in the length of the detected DDS. In such cases, the length is reported in brackets behind the DDS shape.

The tool chain is executed on a PC with an Intel CORE i7-4800MQ with 2.70GHz and 32GB RAM for all examples of the benchmark.

15.1.1 Dynamic data structure interconnections

As already mentioned, the synthetic examples are specifically implemented to "stress test" various features of DSIbin. There are mainly two axes for the design of the synthetic examples. Firstly, interesting interconnections of the DDSs are considered, such as indirect and overlay nesting situations that could lead to wrongly detected connections between the DDSs. Additionally, interesting combinations of DDSs such as a skip lists with nested DLLs (`syn-9`) are added to see whether the nesting relationship is side effect free with DSI's skip list detection. Notably, the related work from the shape analysis literature [65, 72] only contains skip lists as benchmarks without any parent-child relation (`lit-3`). Additionally, skip lists are out of scope for the other dynamic analysis tools [49, 69, 74].

Secondly, the synthetic examples try to avoid the detection of the used DDS, which can be seen as DDS obfuscation. Such techniques could be used by malware authors to further stealthen the footprint of their software.

15.1.2 Dynamic data structure obfuscation

For a long time, code obfuscation techniques [59] are applied on a regular basis for malware families. These techniques are not in the focus of this discussion, as our approach is resilient against this by design. As an example, consider the creation of a SLL. No matter how many code transformations are performed, the final result still is a linked list. As long as the memory is allocated properly and the pointers are not obfuscated, e.g., by XOR-ing two pointer fields into one as applied by the XOR-list [44, 76], such a DDSs is still detectable by DSIbin. Instead, the intention is to perform DDS obfuscation similar to [79]. In [79] the memory layout of the `structs` gets permutated, by reordering `struct` members and adding random padding.

Again DSIbin/DSIcore is resilient against the techniques of [79] as the shape of the DDS is not obfuscated. Consider the SLL example where its shape remains stable no matter where the linkage pointer is placed within the data type forming the SLL. Such layout permutations further need to be stable for all instances of a particular data type, i.e., all allocated `structs` forming the DDS, if runtime overhead should be avoided. If runtime is not of concern, the memory layout permutations could be done on a per instance basis. This would require similar techniques as runtime type casting and thus increases the required logic for accessing type members. Instance based layout permutations would be a problem for DSIcore as no linkage conditions could be established. This technique is however not applied by [79].

In contrast, the obfuscation approach taken by us avoids list traversals and performs flattened access of nested struct elements to prevent type merging and the detection of nested structs by Howard. More specifically, Howard tracks instructions and pointers that operate on binary compatible memory chunks to merge different allocation sites. Therefore, we introduce artificial pointers and functions in examples syn-10, syn-12 and syn-13 to prevent type merging. This triggers the situations described in Ch. 11 when type information is lost. Especially when using such techniques in more complex data structures, such as SLs, this could blur the overall shape as parts of the DDS might never be traversed, thus leaving regions of a DDS unrecognized and leading to failed DDS predicates.

Obfuscation might also be triggered implicitly by compiler optimization techniques. The examples were compiled with the default optimization settings for gcc and -O0 for g++.

As a side note, some of the standard implementation techniques used within the benchmark, such as nested elements, also implicitly obfuscate data structure shapes, as Howard does not detect all nested elements, e.g., nesting at the head and also does not merge nested types, which leads DSIcore to miss such DDS. As our toolchain is split into the part involving Howard and the part involving DSI, we need to separate the implications of optimizations for both approaches. The authors of Howard looked into various compiler optimizations, such as data layout, function frame, and loop optimizations [98]. Howard is able to cope with such situations. As already mentioned DSIbin is immune to optimizations that only change the layout of the used data types or rearrange the scheduling of program instructions, similar to polymorphic code. The only requirement is proper memory allocation and that pointers forming the DDS are kept intact. The examples syn-04, syn-05, syn-06, and syn-14 all perform a flattened access of the relevant linkage addresses, i.e., from the base address of the enclosing struct, to prevent their detection.

15.1.3 Allocations on the heap and the stack

Interestingly, the distribution of a DDS across the heap and the stack, which is not uncommon for the LKL, implicitly obfuscates a DDS, as type merging is not performed by related work across the heap and the stack [51]. This is both true for our aforementioned competing dynamic analysis approaches as well as the discussed pure type recovery tools, i.e., type merging between the heap and the stack is also not done by Howard. This lacking feature is tested with, e.g., example syn-07 for a cyclic DLL and with example syn-16 for a non cyclic SLL.

When looking at syn-07 more closely, missing the type merging between the heap and the stack results in cutting off the head node of the LKL. This prohibits

DSIbin to detect the cyclic property of the DLL, as seen in Tab. 15.1. The result is in line with the predicted behaviour of DSIbin suffering from false negatives for its strand detection, as discussed in Ch. 11. To be able to detect a DLL already points the reverse engineer in the right direction of the used DDS. However, the result might also lead to some effort in analysing the routines that modify the DDS, as an insert into a CDLL might differ much from an insert into a DLL. In example `syn-16`, the overall shape of the DDS is detected correctly, i.e., an SLL, but the wrong length is reported, i.e., the DDS gets cut where the link between the heap and the stack occurs.

15.1.4 Nested structs

With examples `syn-05` and `syn-06` the same technique of embedding a `struct` inside another is used with slightly different implementations. This approach is similar to the LKL, but the nested elements do not only have linkage pointers but also payload elements. The used version of Howard does not consider reading payload information when performing its nesting detection. This information could be exploited for nesting detection in a future version of Howard.

Both examples try to hide their shape by connecting differently typed nodes. Example `syn-05` is similar to a skip list with only one level, i.e., there are a couple of barrier nodes that represent check points for a comparison criterion. If the criterion is matched, one can iterate onwards to the payload nodes to find the corresponding node. If the criterion is not matched, one can iterate to the next barrier node. As barrier nodes and payload nodes are of different types, DSIbin can only detect the payload nodes as an SLL. The payload SLL gets cut at every connection between differently typed nodes, i.e., between the payload and barrier nodes. The result in Tab. 15.1 reports the longest running SLL as multiple SLL fragments exist due to the cutting between differently typed nodes.

With example `syn-06` an SLL is created that consists of alternating types, which effectively hides the complete SLL as DSIbin has no opportunity to correlate the list nodes. This is a quite effective, yet simple example to prevent the detection of DDSs. To make the example practically useful, each of the embedded `structs` has a flag that indicates the type of the enclosing node to be able to react on the individual nodes.

15.2 Discussion of positive examples

Ten out of the 30 examples are detectable out of the box with our naive Howard an DSIcore combination. Specifically, the positive examples are `lit-2,3,5`; `r-1,2`; `syn-01,03,08,14` and `tb-3`. This is an encouraging result, as DDSs are supported by DSIbin which are not considered by related work with regards to dynamic analysis [49,69,74], e.g., SLs and interconnections such as nesting on overlay. Static analysis tools such as Predator and Forester consider SLs in their examples, but not with nested child elements, as seen with `syn-09`. As a short summary ahead of the following detailed discussion of the examples, the naive combination correctly identifies the DDS, whenever Howard is able to fully merge the allocation sites that form the DDS. Additionally, the cells forming the DDS are required to cover the whole memory chunk, which rules out LKL like implementations.

None of the ten positive examples has nesting, thus no **flat**tened accesses of the struct members and no nesting at the head of the surrounding struct (**n@h**) occur. The ten DDSs are also not spread across the heap and the stack, but are exclusively heap allocated. However, type merging (**m**) of the allocated nodes is required for some of the examples.

As can be seen by the variety of the examples, the naive combination is robust against different implementation techniques of the various DDS. This is further demonstrated by examples `r-1,2` and `syn-01` that implement a BT, although with different implementation techniques as indicated by the **LOC** and the **trace length** of the examples. Note that the **LOC** is only reported in Tab. 15.1 for completeness, as this is not primarily an important criterion for measuring the complexity of an example in the context of a dynamic analysis.

`lit-2` consists of multiple levels of nesting on indirection of SLLs, where each level has a unique type. Hence, no type merging is required as indicated by the different types found in the source code, although all of the types are binary compatible. The access patterns of the DDS do not lead Howard into merging the types of the different levels, which are all allocated at different allocation sites. This is the case as the DDS is not accessed, e.g., by one generic iteration pointer. Therefore, the inferred types by Howard match the ground truth found in source code, which results in the correctly identified nesting on indirection on multiple levels. In general, situations where types are binary compatible and are only distinguished by their naming in source code, Howard and DSIbin have difficulties in telling them apart, whenever those types allow Howard's and DSIbin's type merging strategies. The latter will be explained in Ch. 16.

DSIbin also produces the correct interpretation for `lit-5`, which are four DLLs in parallel, as Howard is able to merge the multiple allocation sites forming the DDS and typing all of the pointers that establish the linkage of the DDS. This in turn allows DSIcore to identify the four DLLs. Most likely, the initial intention of the programmer were two DLLs running in parallel instead of four. DSIcore, however, considers all possible combinations of the `next` and `prev` strands when performing its analysis, resulting in four DLL combinations, which we also consider to be the correct interpretation intuitively.

As a preview, interestingly the two examples `lit-2` and `lit-5` are detectable exclusively by the naive implementation and not by the sophisticated approach discussed in the subsequent chapters. For this reason, the sophisticated approach treats types inferred by Howard equally to those inferred by the sophisticated approach as is explained in Ch. 16.

lit-3 and syn-09 are both detected out of the box by the naive implementation. Both examples implement an SL, although syn-09 adds additional child elements to see, whether the SL detection of DSIcore is affected by the child elements. Both examples require type merging, which is successfully accomplished by Howard. This in turn allows DSIcore to detect the true DDS shape. The nesting scenario of syn-09 does not interfere with the SL predicates used by DSIcore. More specifically syn-09 is composed of nodes of different types, which leads MemPick [69] to even cut the connections between the parent SL and the child DLLs. The next positive example is syn-15, which also has a parent child nesting relation as has syn-09. However in contrast to syn-09, syn-15 shows nesting on overlay. Nesting on overlay is also not supported by related work.

The two remaining correctly identified examples only contain lists, i.e., no further parent child relations. These are tb-3 and syn-03. Example tb-3 shows a DLL that is fully detected by DSIbin with the help of the types inferred by Howard. Example syn-03 represents the base case for DSIcore as it is an SLL, i.e., a strand. While the other examples are implemented in C, syn-03 is implemented in C++, showing that the usage of binaries as input allows DSIcore to transparently operate on both these related languages. Therefore, making our DSIbin approach more generic than the initial source code only version of DSI.

15.3 Discussion of negative examples

While our naive approach already shows encouraging results, the majority of the 30 examples can however not be identified correctly by the naive combination of Howard and DSIcore. Here ”not identified” is interpreted quite strictly, e.g., an off-by-one in the length of a strand is considered as not detected by the naive approach. This strict interpretation is chosen, as some of the off-by-one results could be extended to miss more than only one element. As an example, consider that Howard does not merge heap- and stack-allocated memory chunks of the same type. While in syn-16 only the stack allocated head gets cut off from the remainder of the heap allocated list, an list where each element is allocated alternatingly on the heap and the stack will not be detected at all with the naive implementation.

15.3.1 Loss of cyclicity property

Examples er-1, er-2, lit-1, lit-4 and syn-07 all suffer from a diminished precision due to our toolchain missing the cyclicity property of the CDLLs. However, for all examples a DLL gets detected, i.e., only the cyclicity part is lost. In each of the examples, one element of the DLL is cut off for various reasons, which consequently destroys the cyclicity property. This implies that the length of the detected

DLL is always one less than that of the CDLL. For examples `er-1,2`, the CDLL is implemented using a C++ std::list, which is similar to that of a LKL. The CDLL is not spread across the heap and the stack, but the head node of the CDLL is placed outside of the remainder of the payload carrying part of the list. The CDLL linkage struct is embedded at the head of the payload struct. Howard never merges memory chunks of different sizes, thus missing the opportunity to merge the external head node of the CDLL with the remainder of the list. Additionally, **n@h** is never detectable by Howard, which makes it impossible to apply DSIcore's cell abstraction in such scenarios.

When looking at the next example, `lit-1`, where cyclicity is missed, the reasons are cut off head nodes. In this case, the head nodes are cut off both for the parent and the child lists. Example `lit-1` employs the LKL both for the parent and the child. The parent head node is cut off, as it is placed on the stack while the remainder of the list is heap allocated. Howard does not merge heap and stack allocated types. Additionally, the head and the remainder of the list are again of different sizes as with the previous example. The head nodes of the two child elements are nested within the parent. One child is nested at the head of the parent, the second child is nested below the first. Again, nesting at head is missed entirely by Howard. The nested head of the second child element gets detected by Howard but, unfortunately, Howard again does not perform the type merging between the nested type and its exclusive instantiation outside a surrounding struct. This results in detecting three DLLs. The problem of a head and stack allocated CDLL is also present with `syn-07`.

15.3.2 Changes in connections

The imprecision of the inferred types also affects the detected connections between strands, as can be seen with examples `lit-1`, `lit-4`, `syn-02`, `syn-10` and `syn-14`. These examples have in common that the ground truth has nesting on overlay. With the types inferred by Howard, the revealed connections by DSIcore all degenerate to nesting on indirection. The main problem is closely related to the loss of cyclicity in the case where the head nodes are not detected and not merged with the remainder of the list. In such cases a missed nested head node leads to a loss of the nesting on overlay property. This is the case, as DSIcore simply does not observe that two cells of two different strands, i.e., the parent and the child strand, actually occupy the same memory chunk, as the child strand does not include the cut off head element anymore. However, the connecting pointer from the parent head to the next element of the child is still present, leading to the nesting on indirection connection. This can, e.g., be seen from examples `lit-2`, `lit-4`, `syn-02` and `syn-14`.

No CDLL is present in example `syn-10`. Instead, the parent SLL is created at a different allocation site than the child SLL, although both are of the same type. No iteration is performed after connecting the child with the parent , which prevents Howard's merge strategy. Thus, DSIcore is unable to extend the strand of the child SLL back to the parent. Again, this cuts off the head node of the child SLL, leading to indirect instead of overlay nesting being detected.

Example r-3 is part of the VNC implementation used by the Carberp malware, as discussed in Ch. 10. The parent DLL consists of a head/tail node that nests both the start and the end nodes inside one struct. The actual payload nodes are outside the head/tail node, as shown in Fig. 10.1. Each element of the outer payload nodes has a pointer forming a parent-child relation to the next DLL level, which recursively uses the parent DLL. Again, Howard does not detect the nesting at head inside of the head/tail node of the parent and again does not merge the head/-tail node with the remainder of the list. In this case, DSIcore can only detect the payload DLL, which in this case has a length of three. The ground truth reports a length of five, as the nested start/end nodes are counted as well.

15.3.3 Obfuscation

As already discussed, some of the examples apply obfuscation techniques to hide the true shape of the DDS. During the discussion this far, we have already seen implicit obfuscation techniques. For example, missing nested elements and not performing type merging already obfuscates the true shape of the DDS, e.g., hiding cyclicity. Now, the obfuscation techniques are made more explicit by exploiting the merge strategies employed by Howard, e.g., introducing more allocation sites and preventing iteration pointers. Example syn-13 creates an SLL with the help of a macro that performs the malloc of the SLL node and establishes the pointer connection between the previous and the newly allocated node. When using the macro, the code is effectively replicated at each usage pointer, resulting in a multitude of allocation sites and resembling loop unwinding. Thus, Howard detects a unique type for each allocation site and the lack of an iteration pointer prevents the type merging of Howard. This in turn prevents DSIcore from creating a strand, thus failing to detect the SLL completely.

Examples syn-05 and syn-06, which both implement SLLs that connect nodes of different types. The linkage struct is nested inside of the payload nodes, thus DSIcore's cell abstraction is required. The difference to the previously discussed LKL examples is the absence of the cyclic property. The two current examples exploit **flat** access of their nested elements and nesting at the head which again leads Howard to miss those types. DSIcore only identifies the SLL partially in example syn-05 and not at all in example syn-06. Example syn-05 shows partial sequences of the same type within the list, which are identified correctly by DSIcore, i.e., an SLL of length two gets reported instead of the ground truth of five. syn-06 is more generic and allows one to link arbitrary types within a list, which is used to create a list of alternating types. This in turn obfuscates the SLL completely. In those examples no restrictions upon the usage patterns of the DDS are present, e.g., the list gets iterated within the example but still Howard does not merge the alternating types for the already discussed reasons. A similar obfuscation technique is used with `syn-04`, although the linkage pointers are more complex, as the example uses a DLL as its main DDS with an additional SLL running in parallel. The example aims more to test the following DSIbin implementation as discussed in Ch. 16, which tackles the observed shortcomings with the current naive implementation.

15.3.4 Conclusions

In summary, the naive Howard and DSI combination works well on examples for which nesting does not play a crucial part for forming the DDS and for which it is possible for Howard to fully merge the types that form the DDS. In this case the naive combination can even detect DDS, such as SLs and connections such as overlay nesting, which are not supported by the related work regarding dynamic analysis [49, 69, 74]. The naive combination struggles whenever the type information obtained from Howard is too imprecise. This includes nested types, e.g., when nested structs are missed or type merging is not performed. Missing nested elements happens, e.g., due to **flat**tened access of nested struct members or when nesting at the head of the surrounding struct occurs. These scenarios are impossible to resolve for Howard as its nesting detection relies on the offset calculation of the compiler, which uses a base pointer and applies an offset to access a struct member. Whenever a nested struct is exactly located at the beginning of the base pointer, the situation is ambiguous resulting in the limitation of Howard to miss `structs` nested at the head of the surrounding struct. The situation is worsened by the **flat** access of struct members, where the access pattern to a nested struct is always relative from the start of the enclosing struct, resulting in the same problem as with nesting at the head. Further, the naive combination cannot handle DDSs spread across the heap and the stack, as current type inference tools including Howard do not merge types between the heap and the stack.

Although some of the negative examples' results are encouraging, their ground truth are missed. The naive combination at least points an analyst towards the right direction of the used DDS, e.g., when a DLL gets detected instead of CDLL. However, these very similar DLL implementations my differ significantly, e.g., when considering an insert into a CDLL and a DLL. Therefore, an analyst might understand certain DDS related code sections faster when precise DDS information is available.

The limitations and assumptions of the naive Howard and DSIcore combination, e.g., Howard requires the presence of an iteration pointer to perform its type merging, can be exploited to hide DDSs completely from being detected. This results in DDS obfuscation, which is a severe situation for an analyst as no information about the used DDS is available. This requires more manual reverse engineering effort. As it is also thinkable to pass the inferred information by DSIbin of the used DDS to virus detection tools that operate on data structure signatures, e.g., Laika [60], missing out a data structure completely hampers such an approach. Because of both these aspects, i.e., loss of precision and obfuscation, the need arises to enhance the naive combination to overcome these limitations. This is discussed in Ch. 16.

As a conclusion of the benchmarking of the naive approach, the discussed examples prove our theoretical considerations of type information loss in Ch. 11. Specifically, loosing type information leads to diminished precision. The precision loss is mainly due to false negatives, if (nested) structs are not typed and merged correctly. False positives do not occur, which results from the restrictive merge strategies of Howard, e.g., requiring binary compatibility together with a common iteration pointer. The drawback of this conservative approach are sometimes completely missed DDSs, e.g., as is the case for example `syn-13`.

Table 15.1: Results obtained from the prototypical DSIbin implementation

	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.

16 Refinement

This chapter tackles the aforementioned problems of the naive combination of DSIbin and Howard, namely (i) detecting nested structs independently of their placement inside the enclosing struct and their access patterns and (ii) performing type merging between nested and non-nested struct instances that can be distributed across the heap and the stack. This is accomplished by adding a new component to DSIbin, called *Data Structure Investigator Refinement Component (DSIref)*, as shown in Fig. 14.2. The component implements the novel approach developed within this dissertation that refines the types generated by Howard and uses DSIcore itself for evaluating the inferred types. The general approach is presented in Sec. 16.1, an algorithmic view upon the approach including pseudocode is discussed in Sec. 16.2. The implementation and benchmarking details are explained in Ch. 17.

16.1 Refinement approach DSIref

DSIref uses Howard's inferred types as a baseline for further type refinements. Howard's nested struct detection and type merging strategies have already been applied to form the baseline for DSIref, i.e., the types from Howard are exhaustively merged according to Howard's type merging strategy. DSIref improves upon these results by exploiting pointer connections between types. As pointers follow strict conventions, such as incoming pointers point to the head of a linked target object [60], those connections can reveal the layout of the linked objects, e.g., a pointer that points into the middle of a memory region might indicate a nested struct. Further, a link between two binary compatible memory regions, for which Howard does not infer the same type, might hint at mergeable objects. Binary compatibility exists between objects of the same size and the same primitive data types as inferred by Howard. As these information can be ambiguous, as is explained later in this chapter, DSIref creates a set of type hypotheses upon which the best hypothesis needs to be determined. This is accomplished by evaluating each of the hypotheses with DSIcore itself, resulting in a DDS interpretation for each hypothesis. Subsequently, the most complex DDS interpretation is chosen, as the intuition is that only correctly identified nested structs and type merging increases the complexity of a detected DDS. For instance, example `lit-1` of Tab. 15.1 indeed reveals the cyclic property of the DLL, which was otherwise missed. We have not observed artificially created complex DDS, such as an SL, as this would require that the memory layout and pointer connections coincidentally form a more complex structure than intended by the programmer.

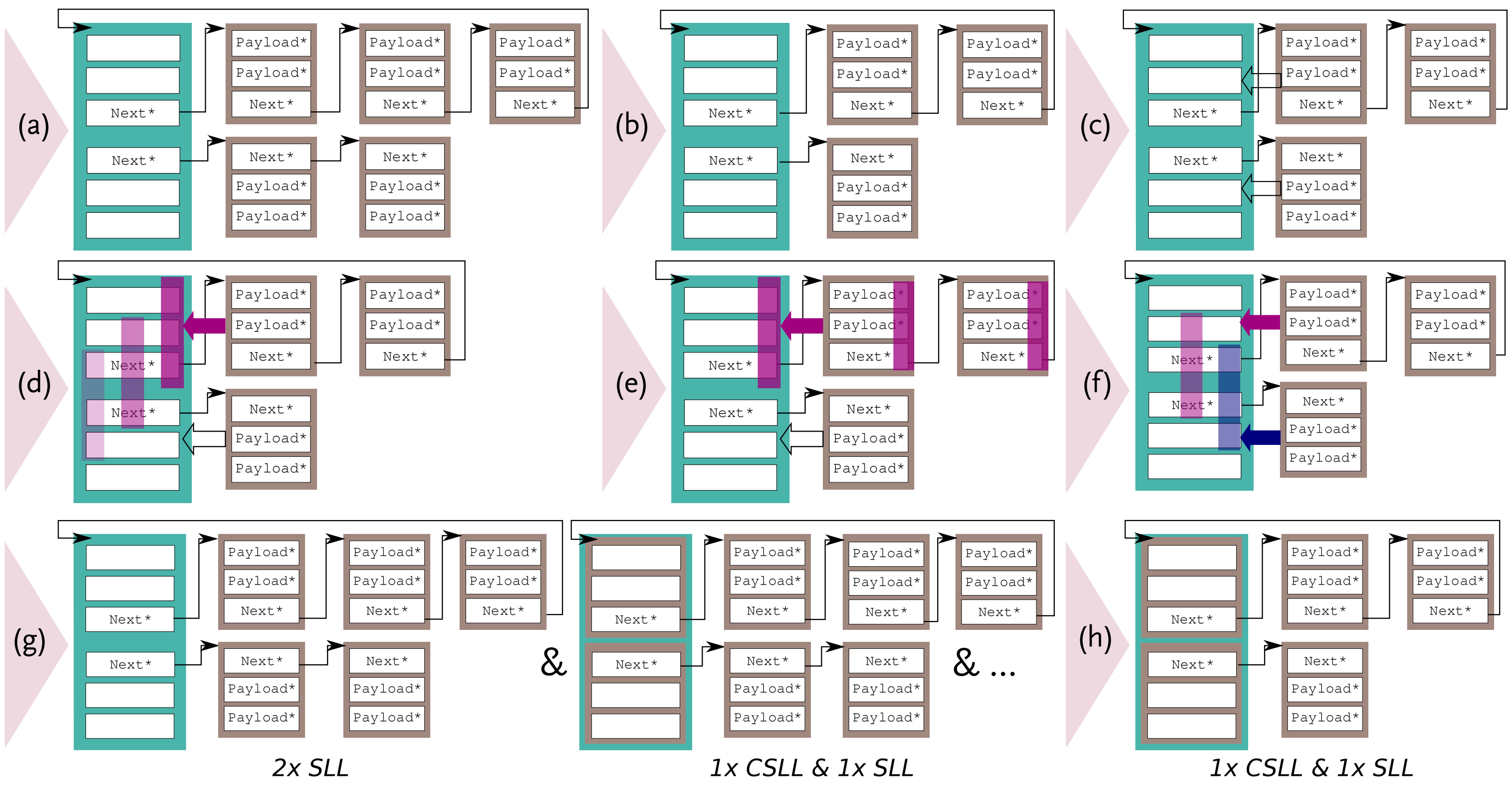


Figure 16.1: Overview of the DSIref approach: (a) create sequence of points-to-graphs from program execution (only one shown); (b) construct merged type graph capturing pointer connections between types; (c) exploit pointer connections by mapping type subregions (two possibilities shown); (d) observe 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*). Figure reproduced from our publication [94].

The refinement process consists of eight phases (Phases (a)–(h)) as shown in Fig. 16.1. The refinement starts out in Phase (a) by generating PTGs for each event of the execution trace recorded by the DSIbin binary frontend (Fig. 14.2). The trace uses the type information inferred by Howard "as is", i.e., Howard's type merging is already applied and the type sizes, the primitive types of the fields and the nested structs are used as observed by Howard. This sequence of PTGs is then used to construct a *Merged Type Graph (MTG)* in Phase (b). A MTG is similar to a PTG where types are represented as vertices and pointers as edges. The difference between a PTG and MTG is that each type and pointer observed in Phase (a) is only recorded once in the MTG, similar to [39,49,92]. Thus, all PTGs are combined into one MTG, which consequently is a compact representation of the DDSs over their lifetime. The MTG represents both heap and stack types, which allows for the uniform processing of both memory regions. This enables type merging between the heap and the stack, which is not done in the literature [51].

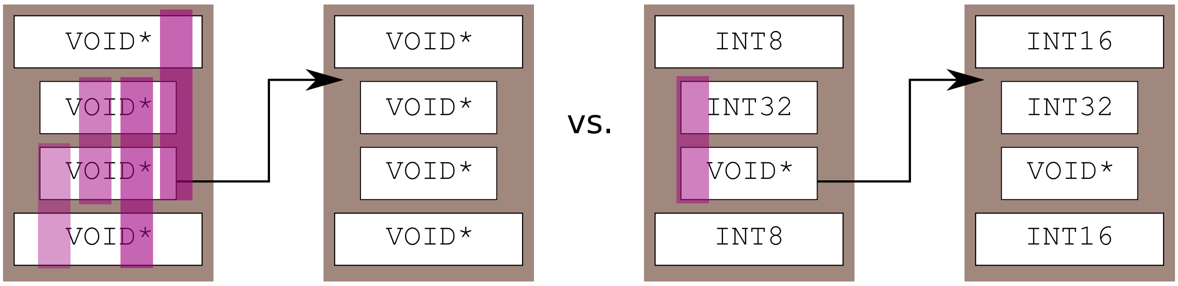


Figure 16.2: Illustration of possible mappings. Figure reproduced from our publication [94].

The following Phases (c)–(f) operate on the created MTG to generate new possible type hypotheses, by exploiting pointer connections and the binary compatibility between the types. In Phase (c), subregions of memory are mapped along pointer connections, where the source and the target need to be binary compatible. Binary compatibility is given if (i) the size of the memory regions match or the source can subsume the target, and (ii) the primitive types inferred by Howard are identical. The latter is relaxed such that it is possible to match typed memory to untyped memory, called *don't cares*. This is important as the lack of access patterns might prevent Howard from typing each field of memory regions. An example of such an scenario can be seen in Fig. 10.1, where the head and tail elements of the DLL only have a typed next and previous pointer field, as the other one is not used. The remainder of the list has fully typed previous and next fields, as the list is traversed in both directions. Thus it gives DSIref more merge opportunities.

Unfortunately, such mappings may not be unique, as has already been mentioned. This can be seen in Fig. 16.2 that shows, on the left, two memory regions consisting of only pointers and, on the right, two memory regions consisting of different primitive types. The indented primitive types indicate a nested struct. On the right-hand side, the nested linkage struct is framed by different primitive data types, which only allows one specific mapping from the target to the source. On the left-hand side instead, we have the pointer-only memory layout in the source and the target. This results in an ambiguous situation as it is not clear where to place the nested struct within the source of the pointer, as indicated by the four colored bars. The variations now come from the size of the mapped region, as shown by the bars of size two and three, as it cannot be determined where the nested struct stops. The only fixed boundaries on the target side are that the nested struct starts on the incoming pointer and can span only until the end of the surrounding struct. On the source side, it only needs to be guaranteed that the mapped region still encloses the linkage pointer and that the mapped memory does not exceed the boundaries of the enclosing struct, both at the start and the end.

Phase (d) handles this situation by brute force as it creates all hypotheses that cover the possible struct sizes and offsets that are mappable between target and source, as indicated by the shaded colors. As the possibility is high that a pointer points to another pointer field, the size of the mappable memory regions is chosen to be at least two. Although this avoids the creation of single-element pointer chains, it decreases false mappings and noise.

Once the mappable regions are detected, the next step in Phase (e) is to propagate the mappable region maximally along the incoming and outgoing pointer connections pointing into and out of the detected region. As a design decision, it is only possible to propagate the region along the pointers as long as the linkage offset is the same, i.e., the offset from the start of the mapped struct to the linkage pointer. This is analogous to the linkage condition used by DSIcore, as discussed in Ch. 2. This step actually performs the discovery of more nested structs along the pointer chains and performs the type merging. The propagation of the region stops when no more incoming or outgoing pointers fulfill the linkage offset condition, or when the source or target of a followed pointer already contain a type instance of this particular type. The latter can happen, e.g., with cyclic pointer connections. Note that it is important to distinguish between a *type* and a *type instance*. The type is discovered by following the pointers and obeying binary compatibility as discussed previously. A type instance is a concrete occurrence of a type within a vertex and is thus defined as a tuple $(type, offset)$, i.e., a type with the corresponding offset within the vertex in which it resides.

Getting back to type propagation, *don't cares* are treated as mappable memory regions, as discussed before. This allows us to type previously untyped memory regions with the primitive types inferred by Howard. An example of such an additional primitive type refinement is displayed in Fig. 1.5 on the left-hand side, where the right column shows the refined type information as discovered in Phase (e). As both types from the heap and the stack are part of the same MTG, arbitrary merging scenarios between the heap and the stack are possible. Additionally, the mapping of subregions between types allows us to merge both outer structs and nested instances of those structs. This is in contrast to ARTISTE [49], which refines types as a whole according to their notion of allocation site and call site tree. Instead, DSIref cannot only refine complete memory chunks, but also only subregions of memory across different types and memory regions.

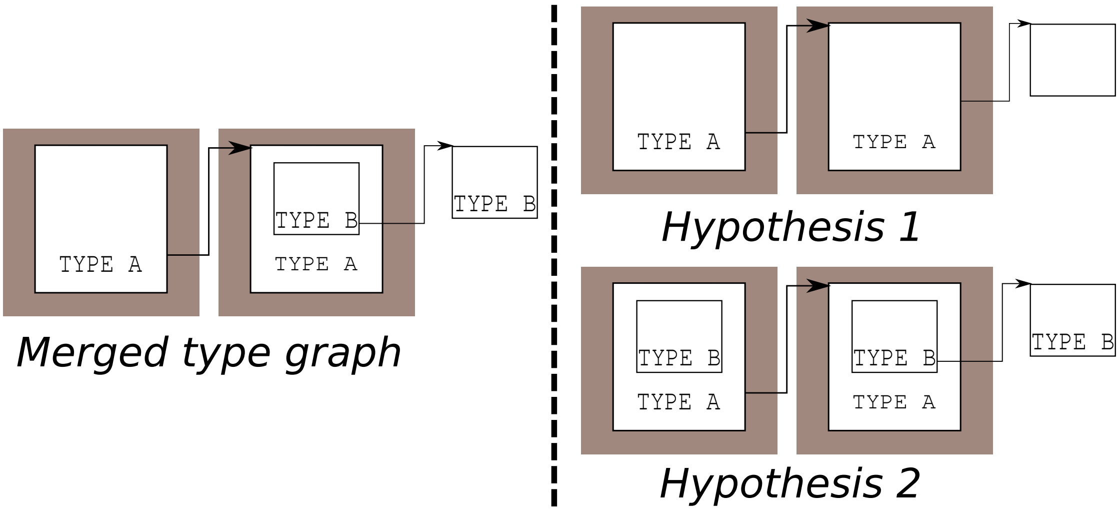


Figure 16.3: Motivation for globally consistent types. Figure reproduced from our publication [94].

Phase (f) depicts another problem when creating possible type hypotheses: different hypotheses as introduced by multiple pointer connections can lead to inconsistent combinations among those hypotheses. Such inconsistencies arise either due to overlaps in mapped memory regions, as it is not possible that nested structs overlap. Thus, overlaps are removed immediately from the set of possible hypotheses. This property is local to a given type vertex, although global inconsistencies can occur as well. This can be seen on the left-hand side of Fig. 16.3: the MTG displays two nested structs of `TYPE A`, where only one instance contains a nested struct of `TYPE B`, which is linked to an outer instance of `TYPE B`. Thus, two different manifestations of `TYPE A` exist, leading to the need to equalize those instances. This is done by creating different hypotheses, one for each possible interpretation, as shown on the right-hand side in Fig. 16.3. Such scenarios can again arise due to different access patterns, e.g., for the head and tail nodes of a linked list, as discussed previously.

In addition to the described creation of a set of possible type hypotheses, the initial types inferred by Howard are also added to the set of possible type hypotheses. This makes all detected type hypotheses from both DSIref and Howard equal and lets the most complex inferred DDS decide which one is the correct interpretation. This avoids discarding valid hypotheses prematurely. More precisely, removing Howard's inferred types from the set of type hypothesis corner cases would need to be introduced for the case when DSIref does not find further type refinements, i.e., only Howard's inferred types exist. Further, DSIref finds *possible* type scenarios, which still need to be verified. If the types inferred by Howard would be discarded, false positives regarding the type merging introduced by DSIref could not be detected.

Finally Phases (g) and (h) determine which type hypotheses are the best interpretation of the MTG in terms of the detected DDS. For this, we use the DSIcore algorithm itself by letting it perform its DDS detection with each of the created type hypotheses, as shown in Phase (g). In the two concrete examples, one sees that the initial types from Phase (a), i.e., those inferred by Howard, only create two SLLs. The other interpretation depicted indeed shows that both nested types of the embedded linked lists are propagated back to the head node, thus revealing the CSLL property and making the SLL one element longer. Thus, in Phase (h), the most complex interpretation is selected according to a hierarchy similar to the DDS identification taxonomy employed be DSIcore [107]. The main idea here is to rank the DDSs from *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). The chosen DDSs is the subset of the available DDSs of DSIcore's taxonomy [107] that was benchmarked within DSIcore in Ch. 8.

Additionally, the number of occurrences of the most complex data structures that are detected via the best interpretation is taken into account by DSIref. This favours situations such as in example `lit-5`, where both Howard and DSIref produce the same most complex DDS. Thus, Howard's and DSIref's type hypothesis only differ in the maximum count of recovered DDS. If the count of recovered DDS instances does not favour a hypothesis, *Occam's razor* is used to choose the least amount of refined structs. Here, the rational is that additional refinements that do not alter the overall structure of the DDS are superfluous. In situations, where only an SLL is observed, the longest strand length succeeds. Additionally, evidence counts among the different hypotheses that are within 85% of each other are considered equivalent. This counters noise in the evidence and labeling process. This strategy often results in a single interpretation, and its general applicability is discussed in Ch. 17.2.

16.2 DSIref in pseudo code

This section discusses the pseudo code for the algorithm presented in the previous section. The main algorithm, as shown in Alg. 17, is divided into two main parts (i) the detection and propagation of DSI Types (DSItypes) and (ii) the creation of consistent type hypotheses. Note an important aspect of the algorithm, which distinguishes between a DSItype and an *instance* of a DSItype. A DSItype gets created when mapping memory regions along edges as done in function calculateDSITypesForEdge in Alg. 18. The propagation of those DSItypes along the edges of the MTG via function propagateDSITypeOnMaxPath, as shown in Alg. 20, creates instances of those DSItypes. The subsequent parts of the algorithm operate on those type instances, and only in the end of the algorithm those type instances are mapped back into types.

16.2.1 Main function of algorithm

The main algorithm, as shown in Alg. 17, operates on the MTG that is represented by the set of type vertices ($\mathcal{V}$) and a set of edges ($\mathcal{E}$) which is copied into the working set $\mathcal{E}_{\text{type}}$. Those variables are treated as global variables. Each edge (e) of $\mathcal{E}_{\text{type}}$ is processed iteratively in line 6 to create DSItypes ($DSITypes$) in line 8. A DSItype is defined as a tuple consisting of the actual DSItype and the offset (*offsetInSrc*) at which the type occurred inside the type vertex. The created DSItypes are iteratively distributed across the vertices of the MTG, starting out with the source vertex of the currently processed edge ($e.source$) in line 11. All detected DSItypes are collected inside a set ($allDSITypes$), which is used to create the possible type combinations in line 20.

16.2.2 Creation of DSI types

The creation of DSItypes is conducted in calculateDSITypesForEdge, as shown in line 8 of Alg. 18. The algorithm creates the binary compatible subregions of memory between the source (**S**) and the target (**T**) vertex of the connecting edge (e), including the possible start points of the subregion (Phase (d) of Fig. 16.1) together with the different possible sizes of the memory subregions.

The algorithm operates on the primitive types inferred by Howard, which are attributes of the source and the target vertex as a sequence ($primitiveTypes$). A working copy for the source and the target sequence is stored inside of $sourceTypeSequence$ and $targetTypeSequence$. The index inside the sequence is defined as being the offset of the primitive type within the corresponding type vertex. For the source, the complete sequence is required, i.e., starting from offset zero to the end (end) of the sequence, as seen in line 6. For the target, the type sequence is sliced, starting from the incoming pointer offset into the target vertex (e.*targetOffset*) until the end of the sequence, as seen in line 9. This reflects

```
// Operating on merged type graph (Phase (b) in Fig. 16.1)
$\mathcal{E}_{\text{type}} \leftarrow \mathcal{E}$
$allDSITypes \leftarrow \emptyset$
// Part (i): Detection and propagation of DSItypes
// Processing of all edges
for each $e \in \mathcal{E}_{\text{type}}$
  // Calculate the possible types per edge (Phase (c) in Fig. 16.1)
  $DSITypes \leftarrow$ calculateDSITypesForEdge$(e)$
  for each $(DSIType, \mathit{offsetInSrc}) \in DSITypes$
    // Label the vertices with DSI type instances (Phase (e) in Fig. 16.1)
    propagateDSITypeOnMaxPath$(DSIType, e.source, \mathcal{E}, \mathit{offsetInSrc})$
  end
  // Keep track of processed elements
  $\mathcal{E}_{\text{type}} \leftarrow \mathcal{E}_{\text{type}} - \{e\}$
  $allDSITypes \leftarrow allDSITypes \cup DSITypes$
end
// End of Part (i)
// Part (ii): Create type combinations in
// preparation for Phase (g) in Fig. 16.1
calculateTypeCombinations$(\mathcal{V}, allDSITypes)$
```

Algorithm 17: Main part of the DSIref algorithm

the assumption that pointers always point to the start of a struct, as observed by the authors of Laika [60]. Initially, the subregions are both extended to the end of the source and target vertices to aim for the maximal subregion. Subsequently, the memory region is shrunk until the maximal common subregion between the source and the target is found.

First the possible pointer offsets are calculated within the subregion inside of the target vertex in line 10 and are stored inside of *pointerOffsetsInTarget*. The offset corresponds to the linkage offset of the strands. Further, *pointerOffsetsInTarget* implicitly stores the currently used offset and produces the next offset that needs to be inspected. Both are produced by currentPointerOffset and nextPointerOffset. The main idea here is that type equivalence between the source and the target can only be established by aligning the pointer fields between the source and the target. The remaining primitive types surrounding the pointer then need to match, which is done by iteratively checking all primitive types and building up the found DSItype. This is done throughout lines 16-36. The first *while* loop ensures that the mapping does not exceed the start of the source vertex. The nested loop then collects the slice of mappable primitive types between source and target. Thus, the loop terminates early, if either the type sequence length exceeds the source vertex size starting from the edge offset (line 21) or a mismatch between source and target types occurs (line 23). The comparison between source and target implicitly treats untyped memory as a "match all" in the compar-

```
1:  function calculateDSITypesForEdge($e$)
2:    // Init
3:    S $\leftarrow e.source$; T $\leftarrow e.target$; $foundDSITypes \leftarrow \emptyset$
4:    // Get the primitive types inferred by Howard
5:    // a) in the source, starting from first element to the end of the type sequence
6:    $sourceTypeSequence \leftarrow$ S.$primitiveTypes[0 \ldots end]$
7:    // b) in the target, starting from the incoming pointer
8:    // to the end of the type sequence
9:    $targetTypeSequence \leftarrow$ T.$primitiveTypes[e.$targetOffset$\ldots end]$
10:   pointerOffsetsInTarget $\leftarrow$ calculatePointerOffsets($targetTypeSequence$)

11:   $i \leftarrow e.$sourceOffset $-$ nextPointerOffset(pointerOffsetsInTarget)
12:   // The target subregion needs to be mappable into the source
13:   // 1) Start of target needs to be in range with source: $i \geq 0$
14:   // 2) Length of source type sequence is not exceeded
15:   // (see check in line 21)
16:   while $i \geq 0$
17:     $foundDSIType \leftarrow <>$
18:     // Iterate through the target sequence of primitive types and
19:     for each $u \in 0 \ldots targetTypeSequence.length$
20:       // 2) Stop, if length of source type sequence is exceeded
21:       if $i + u > sourceTypeSequence.length$ then break
22:       // Extend type if source and target sequence match
23:       if $targetTypeSequence[u] = sourceTypeSequence[i + u]$
24:         $foundDSIType \leftarrow foundDSIType + +targetTypeSequence[u]$
25:       else
26:         // Stop, if source and target sequence do not match anymore
27:         break
28:       end
29:     end
30:   // Continued in Alg. 19
```

Algorithm 18: Calculate the possible DSI Types for an edge (part 1)

```
31:   // Continuation of Alg. 18
32:     // Save tuple: ((found DSI type, linkage offset), source offset)
33:     $foundDSITypes \leftarrow foundDSITypes \cup$
          $\{((foundDSIType,$
            currentPointerOffset(pointerOffsetsInTarget)$), i)\}$
34:     // Continue with the next offset
35:     $i \leftarrow e.$sourceOffset $-$ nextPointerOffset(pointerOffsetsInTarget)
36:   end
37:   // Slice found types into smaller pieces
38:   return createAllPossibleSubTypeChunks($foundDSITypes$)
```

Algorithm 19: Calculate the possible DSI Types for an edge (part 2)

ison, i.e., as don't cares. In the best case, the loop terminates after the complete target sequence was processed. The resulting type sequence, i.e., a DSItype, is stored inside $foundDSIType$. All DSItypes found are stored in a set as a tuple together with the corresponding *linkage* offset: $((foundDSIType, \textit{linkageOffset}), i)$ in line 33. The linkage offset is important, as this is the same offset that is used in the linkage condition by DSIcore. Remember that type instances are only propagated along pointer chains by DSIref as long as the linkage offset remains the same. Additionally, the source offset is stored in line 33.

Finally, the possible memory sub chunks are created from all the collected type mappings. This is required as it is not immediately clear, whether a maximally mapped subregion is the correct mapping or just coincidence. Thus the types are sliced into chunks ranging in size from two elements to the size of the complete memory chunk, in steps of size one. The regions are iteratively cut after each primitive type, which is computed in function CREATEALLPOSSIBLESUBTYPECHUNKS in line 38. Note that the memory sub chunks are each treated as a valid DSItype but they are considered incompatible to each other. This is required, as only one type interpretation per edge is allowed. Without this restriction, multiple of those sub chunks could be used at the same time to built a type hypothesis. The actual pseudocode of creating the memory sub chunks is not presented in this work, as it does not add significant value to the overall understanding of the algorithm. The implementation is publicly available at https://github.com/uniba-swt/DSIbin.

16.2.3 Propagation of DSI types

After the creation of the DSItypes, these need to be distributed over the MTG by maximally propagating each DSItype along the pointer edges, corresponding to Phase (e) in Fig. 16.1. This is computed in Alg. 20. Each propagated DSItype is an *instance* of such a DSItype. The instances are stored inside each vertex (v) to which they correspond. The vertices are from the set of vertices ($\mathcal{V}$) of the MTG. The set holding these instances is called $dsiTypeInstances$.

At first, the given DSItype ($DSIType$), passed into the function as a parameter, is decomposed in line 2 into the actual type ($type$) and the linkage offset (*linkageOffset*), i.e., the offset where the linkage pointer resides relative to the enclosing struct. The type then gets assigned to the set of type instances of the source vertex (**S**), by creating a tuple containing the type and offset within the source (*sourceOffset*), which is the absolute offset of the type from the beginning of the memory chunk (see line 3).

Subsequently all outgoing edges ($outEdgesForType$) from the particular type and all incoming edges ($inEdgesForType$) to this type are computed in lines 4 and 5, respectively. The set of edges ($\mathcal{E}$) used for the computation is passed as a parameter into the function PROPAGATEDSITYPEONMAXPATH. For both directions, the linkage offset is considered, i.e., only pointers at this particular offset are taken

```
function propagateDSITypeOnMaxPath($DSIType$, $sourceOffset$, $source$, $\mathcal{E}$)
  $(type, linkageOffset) \leftarrow DSIType$
  $source.dsiTypeInstances \leftarrow source.dsiTypeInstances \cup \{(type, sourceOffset)\}$
  $outEdgesForType \leftarrow$ outgoingEdgesFromType($type$, $linkageOffset$,
      $source$, $sourceOffset$, $\mathcal{E}$)
  $inEdgesForType \leftarrow$ incomingEdgesFromType($type$, $linkageOffset$,
      $source$, $sourceOffset$, $\mathcal{E}$)
  // Process outgoing edges
  for each $e \in outEdgesForType$
    if typeCanBePropagatedToTarget($e$, $type$)
      propagateDSITypeOnMaxPath($DSIType$, $e.targetOffset$, $e.target$, $\mathcal{E}$)
    end
  end
  // Process incoming edges
  for each $e \in inEdgesForType$
    $sourceOffset' \leftarrow$ typeCanBePropagatedToSource($e$, $type$)
    if $sourceOffset' \neq \emptyset$
      propagateDSITypeOnMaxPath($DSIType$, $sourceOffset'$, $e.source$, $\mathcal{E}$)
    end
  end
  return
```

Algorithm 20: Recursive propagation of a DSI type along pointer connections

into consideration. The outgoing edges are processed first (lines 7- 11), by testing if the type can be propagated to the target; again binary compatibility is checked (line 8). If this is the case, the propagation continues recursively on the target (line 9). Next, the incoming edges are processed (lines 13- 18), where again it is tested whether the current type can be propagated onto the source (line 14). The function typeCanBePropagatedToSource returns ”∅”, if propagation is not possible or, otherwise, it returns the corresponding offset within the source, which is stored into *sourceOffset′* (line 14). If the type is mappable, the propagation continues recursively on the source (line 16).

To guarantee termination of the algorithm, the distributed type instances are also used to terminate the propagation of a type if a vertex has already stored an instance of this particular type, i.e., the same type at the same offset.

```
1: function typeCanBePropagatedToTarget($DSIType, e$)
2:   $target \leftarrow e.target; \mathit{targetOffset} \leftarrow e.\mathit{targetOffset}$
3:   $targetTypeSequence \leftarrow target.primitiveTypes[\mathit{targetOffset} \dots end]$
4:   return $targetTypeSequence =$ typeToSequence$(DSIType) \wedge$
5:     $\nexists(type, \mathit{offset}) \in \{target.dsiTypeInstances \mid DSIType = type \wedge$
6:     $e.\mathit{targetOffset} = \mathit{offset}\}$
```

Algorithm 21: Test whether type can be propagated to target vertex

The test for the propagation of a type ($DSIType$) onto the target vertex of an edge (e) is shown in Alg. 21. The test ensures binary compatibility between the target at the offset of the incoming pointer and the given DSItype. In line 2, the algorithm initializes the target vertex ($target$) and the target offset (*targetOffset*) with the passed parameters. Subsequently the sequence of primitive types in the target ($targetTypeSequence$) at the offset of the incoming pointer is fetched until the end of the $primitiveTypes$ sequence (see line 3). Then, the type sequence is compared with the type sequence of the given DSItype in line 4. The function typeToSequence hereby linearizes a given DSItype into a sequence of primitive types that is used within the comparison to ensure binary compatibility. Only if the sequences match and there does not already exist an instance of this particular type inside the target vertex can the type be propagated onto the target vertex (lines 5- 6).

Analogous to the propagation of a type to the target, the propagation of a type onto the source needs to be checked. This is done in Alg. 22, where first the possible types for the given edge (e) are calculated by reusing function calculateDSITypesForEdge which was already discussed and that returns all possible DSItypes for the edge that are stored in the set of DSItypes ($DSITypes$) in line 4. The given DSItype ($DSIType$) first needs to be present in the set of returned types (lines 6-8).

```
 1: function typeCanBePropagatedToSource($DSIType, e$)
 2:   $(type, linkageOffset) \leftarrow DSIType$
 3:   // Reuse calculation of DSI types to get types for the current edge
 4:   $DSITypes \leftarrow$ calculateDSITypesForEdge($e$)
 5:   // Type must be mappable to source
 6:   if $\exists((DSIType\prime, linkageOffset\prime), offset) \in \{DSITypes \mid$
 7:     typeToSequence($DSIType\prime$) = typeToSequence($DSIType$)$\wedge$
 8:     $linkageOffset\prime = linkageOffset\}$
 9:     // No such type instance should already exist in the source. This
10:     // implicitly acts like a breadcrumb algorithm to guarantee termination.
11:     if $\nexists(type\prime, offsetOfInstInVertex) \in \{e.source.dsiTypeInstances \mid$
12:       $type\prime = type \wedge offsetOfInstInVertex = offset\}$
13:         return $offset$
14:     end
15:   return $\emptyset$
```

Algorithm 22: Test if type can be propagated to source vertex

Subsequently, the algorithm checks for the presence of an instance of the particular type inside of the source vertex, which is the same as for the check for the target propagation (lines 11 and 12). If the type can be propagated, the offset of the type inside of the vertex is returned in line 13. Otherwise $\emptyset$ gets returned in line 15.

16.2.4 Creation of DSI type combinations

The final step, which is composed of several sub-steps, is the creation of the possible type combinations that are consistent within each other and is listed in Alg. 23. This includes Phase (f) of Fig. 16.1, where local compatibility among the created type instances is ensured. Local compatibility guarantees that types are not overlapping, as this is not allowed for nested structs. This is the first compatibility check in Alg. 23 in line 3. The implementation of this step is discussed in Sec. 16.2.5. Subsequently, the global type compatibility is computed in line 5 of Alg. 23. This step captures type combinations, which are incompatible on a global scale, by ensuring that all type combinations are locally compatible. This is required, as it might be possible that a mapping reveals a locally valid type combination, but further propagation of the types creates invalid overlaps. This consistency check is discussed in Sec. 16.2.6. With only valid type combinations left, line 7 of Alg. 23 now computes all possible type hypotheses. This includes nesting the types as much as possible and is explained in more detail in Sec. 16.2.7. The type combinations lead to the final compatibility check, which is required as even among all valid type combinations, it might be the case that not all types spread across the MTG are identical. Consider an example of a SLL with a parent-child relation with overlay nesting. If not all of the elements of the parent SLL behave uniformly, they might not reveal the same type, e.g., the head node of the SLL does not carry a child element. Thus, all possible combinations regarding nested elements need to be created as well, as it is not immediately decidable which type is correct. The compatibility check is invoked in line 9 of Alg. 23 and is detailed in Sec. 16.2.8.

```
1: function calculateTypeCombinations($\mathcal{V}$, $allDSITypes$)
2:     // Local type compatibility. Information stored inside $\mathcal{V}$
3:     compatibleTypesStageOne($\mathcal{V}$)
4:     // Global type compatibility. Information stored inside $globalTypeMatrix$
5:     $globalTypeMatrix \leftarrow$ compatibleTypesStageTwo($\mathcal{V}$, $allDSITypes$)
6:     // Creation of type hypothesis, including nesting.
7:     $allHypotheses \leftarrow$
           calculateAllHypotheses($allDSITypes$, $globalTypeMatrix$)
8:     // Final global compatibility check of created nested types
9:     compatibleTypesStageThree($allHypotheses$, $\mathcal{V}$, $allDSITypes$)
```

Algorithm 23: Calculation of the possible type combinations

16.2.5 Local type compatibility

This section discusses the local type compatibility computation shown in Alg. 24. The algorithm receives the set of vertices ($\mathcal{V}$) of the PTG passed as a parameter and, subsequently, iterates through each vertex (v) of the set to calculate the type compatibility in line 3. The results for each vertex are stored in a matrix, which holds each *type* combination, i.e., not *type instance* combinations. The initialization of the matrix ($typeMatrix$) is in line 6. The matrix stores all type combinations; thus, it stores both (A, B) as well as (B, A) tuples, where A and B are types. On line 8, all type *instances* for the current vertex are fetched and stored into a set ($instancesPerTypes$). The elements of the set are tuples defined as $(type, typeInstance)$, i.e., the type and the concrete instantiation of the type. In line 10, all combinations of the type instances are iterated, and these are checked for local compatibility in line 12. The check itself is detailed in Alg. 25 and will be discussed below. The results of the compatibility check are stored into the matrix in lines 16 and 18. Finally, the matrix is stored inside the currently processed vertex in line 22, for later usage.

The aforementioned local compatibility check is computed in Alg. 25, and is detailed in the following. The algorithm receives two type instances as parameters ($typeInstnc$ and $typeInstnc\prime$). First, a self test of the type instances is conducted, which checks if both types and the offsets are identical (line 5). This allows us to

```
function compatibleTypesStageOne($\mathcal{V}, allDSITypes$)
  // Cycle through each vertex and check type instance compatibility locally
  for each $v \in \mathcal{V}$
    // Cartesian product of all types per vertex
    // to hold compatibility information
    $typeMatrix \leftarrow \emptyset$
    // Fetch all type instances for a type vertex
    $instancesPerTypes \leftarrow$ getAllInstancesForTypes($v, allDSITypes$)
    // Check all type instance combinations
    for each $((type, typeInstances), (type', typeInstances')) \in$
      $instancesPerTypes \times instancesPerTypes$
      // Calculate compatibility information
      if $\forall typeInstnc \in typeInstances \mid \forall typeInstnc' \in typeInstances'$ .
        isCompatible($typeInstnc, typeInstnc'$)
        // Previous type combination could already be false.
        // Thus, an logical AND is required between the current
        // and previously calculated compatibility flag.
        $typeMatrix[type, type'] \leftarrow true$ & $typeMatrix[type, type']$
      else
        $typeMatrix[type, type'] \leftarrow false$
      end
    end
    // Save compatibility information in vertex
    $v.typeMatrix \leftarrow typeMatrix$
  end
```

Algorithm 24: Check local compatibility of type instances

treat each type combination uniformly, i.e., it is not required to keep type combinations of the same type out of the type compatibility matrix $typeMatrix$. Instead, the primary diagonal of the matrix becomes $true$. Subsequently, the remaining combinations of types and offsets are considered, i.e., types are either the same or different, and offsets are either identical or different. In lines 7 and 9, the combination of different types at the same offset is computed. First, in line 7, binary compatible duplicates are removed by conducting the type equivalence test (typeEquivalence). If the types are not binary compatible, two different types at the same offset should always be nested, which is examined in line 9. The next combination in line 11 ensures that the same type found at different offsets is neither nested nor overlapping. Finally, in line 13, different types with different offsets are computed, for which only overlaps are forbidden, i.e., no explicit nesting test is required.

```
1:  function isCompatible($typeInstnc, typeInstnc'$)
2:     $(type, offset) \leftarrow typeInstnc$
3:     $(type', offset') \leftarrow typeInstnc'$
4:     // Same type and offset: self test
5:     if $type = type' \wedge offset = offset'$ return $true$
6:     // Different types at same offset: prevent binary compatible duplicates
7:     if $type \neq type' \wedge offset = offset' \wedge$ typeEquivalence$(type, type')$ return $false$

8:     // Different types at same offset: should always be nesting
9:     if $type \neq type' \wedge offset = offset'$ return (isNested$(type, offset, type', offset') \wedge$
          $\neg$isOverlapping$(type, offset, type', offset')$)
10:    // Same type and different offset: no nesting and no overlap
11:    if $type = type' \wedge offset \neq offset'$ return ($\neg$isNested$(type, offset, type', offset') \wedge$

          $\neg$isOverlapping$(type, offset, type', offset')$)
12:    // Different type and different offset: no overlap
13:    if $type \neq type' \wedge offset \neq offset'$
14:       return $\neg$isOverlapping$(type, offset, type', offset')$)
15:    // Default
16:    return $false$
```

Algorithm 25: Calculate compatibility of types

16.2.6 Global type compatibility

The main goal of this computation is to create a lookup of globally consistent types across the MTG. In contrast to the previous section, the local type compatibility with regards to one vertex is computed. This is insufficient for capturing possible inconsistencies among type instances across the graph. Inconsistencies might come into existence during the propagation process that propagates types across the MTG. This propagation is speculative and pointer connections might differ even within one particular DDS, e.g., varying usage patterns for the head and the remainder of a SLL, where only the remainder of the SLL has child elements. This might lead to both locally compatible and incompatible types within different vertices of the MTG. The local compatibility check only computes compatibility within one vertex. In contrast, the global compatibility takes all local type information into consideration.

```
1: function compatibleTypesStageTwo($\mathcal{V}, allDSITypes$)
2:   $globalTypeMatrix \leftarrow \emptyset$
3:   for each $(type, type') \in allDSITypes \times allDSITypes$
4:     $globalTypeMatrix[type, type'] \leftarrow true \ \& \ globalTypeMatrix[type, type']$
5:     if $\exists v \in \mathcal{V}$ . typesAreLocallyIncompatible($type, type', v$)
6:       $globalTypeMatrix[type, type'] \leftarrow false$
7:     end
8:   end
9:   return $globalTypeMatrix$
```

Algorithm 26: Conducting a global compatibility check of the types

The global compatibility operates on the vertices of the MTG ($\mathcal{V}$) and all found types ($allDSITypes$), which are passed as parameters in Alg. 26. A matrix for storing the global type compatibility is created in line 2. Each type combination is stored inside the matrix and records if the types are compatible ($true$) or not ($false$). The algorithm processes all possible type combinations and marks two types as being incompatible, if there exists a vertex of the MTG that is locally incompatible (lines 3-6). The computed matrix is returned for further usage.

16.2.7 Compute all type combinations

With the global type compatibility computed, we are now able to create all possible type hypotheses among types that are compatible with each other. It is important to create all type combinations that are compatible, as no combination can be ruled out prematurely to being a wrong interpretation of the MTG, as discussed previously in Sec. 16.1. The actual decision of which combination is the best interpretation of the MTG, is postponed to a later step, i.e., Step (g) of Fig. 16.1.

The algorithm for computing the possible compatible hypotheses is shown in Alg. 27. The main idea is to create a power set of all previously detected types

```
1: function CALCULATEALLHYPOTHESES($allDSITypes, globalTypeMatrix$)
2:   $typesPowerSet \leftarrow \mathcal{P}(allDSITypes)$
3:   return $\{typeSet \mid typeSet \in typesPowerSet$ .
4:     // All types of the subset must be compatible among each other
5:     $(\forall (type, type') \in typeSet \times typeSet\ .\ globalTypeMatrix[type, type'] = true) \wedge$
6:     // Only choose the maximal subset of compatible types
7:     $(\nexists typeSet' \in typesPowerSet$ .
8:     $(\forall (typeTwo, typeTwo') \in typeSet' \times typeSet'$ .
9:       $globalTypeMatrix[typeTwo, typeTwo'] = true) \wedge$
10:      $typeSet \subset typeSet')\}$
```

Algorithm 27: Create all valid combinations of types, such that each hypothesis contains the maximal amount of possible type combinations

($allDSITypes$) in line 2 and, subsequently, rule out all type subsets that contain types that are incompatible with each other (see line 5) by looking up the compatibility between types in the global type matrix ($globalTypeMatrix$). Additionally, only the maximal subset of compatible types is chosen, which is ensured in lines 7-10. Note line 10, which selects the maximal subset of compatible types.

16.2.8 Compute all valid type combinations with nesting

After all hypotheses are created as described in the previous section, one additional computation step is required for guaranteeing consistency among the created types. This step includes verifying the nesting of all type instances of the computed type hypotheses. As the creation of types depends upon the usage patterns of the vertices of the MTG, situations that result in different nested types as shown on the left in Fig. 16.3 can occur. Here, the same rationale applies as discussed in Sec. 16.2.7: it is not immediately clear which type mapping is the correct one. Thus, multiple hypotheses need to be created, which either show nesting or remove the nesting, as displayed on the right in Fig. 16.3.

The algorithm for computing the nesting and ensuring consistency among the nested types is shown in Alg. 28. The function retrieves all computed hypotheses thus far ($allHypotheses$) and the set of vertices of the MTG ($\mathcal{V}$) as parameters. Each hypothesis of all computed hypotheses is processed sequentially (line 3). First, all types instances that show nesting are computed in line 4. The function returns all type instances of the same type, which show both nesting and non-nesting behaviour. The returned instances are stored in $allNstdTypeInstncs$.

The specifics are detailed in Alg. 29 which computes types that show both nesting and non-nesting. The presence of types that show *both* nesting and non-nesting might sound counter intuitive, but can happen due to different usage patterns of certain parts of an DDS, as discussed previously. The function of Alg. 29

```
function compatibleTypesStageThree($allHypotheses, \mathcal{V}$)
  $allNstdHypthss \leftarrow \emptyset$
  for each $typeHypothesis \in allHypotheses$
    $allNstdTypeInstncs \leftarrow$ collectAllNestedTypes($\mathcal{V}, typeHypothesis$)
    $allIntrnsForType \leftarrow$
    groupNestedTypeInstances($allNstdTypeInstncs$)
    $allNstdHypthss \leftarrow allNstdHypthss \cup$
    createAndCombinePossibleTypeCombinations($allIntrnsForType$)
  end
  return $allNstdHypthss$
```

Algorithm 28: Check global compatibility of nested types

```
function collectAllNestedTypes($\mathcal{V}, typeHypothesis$)
  $allNstdTypeInstncs \leftarrow \emptyset$
  // Fetch all type instances that show immediate nesting
  for each $v \in \mathcal{V}$
    $allNstdTypeInstncs \leftarrow allNstdTypeInstncs \cup$
      $\{oneNstdTypeInstnc \in$ nestTypes($v, typeHypothesis$) $|$
        hasNesting($oneNstdTypeInstnc\}$)
  end
  // Fetch all nested type instances, where nesting
  // has been recorded elsewhere
  for each $v \in \mathcal{V}$
    // Compute nesting (again) first
    $nstdTypeInstncs \leftarrow$ nestTypes($v, typeHypothesis$)
    // Cycle through each computed type instance
    for each $oneNstdTypeInstnc \in nstdTypeInstncs$
      // Check, if the computed type instance has no nesting
      // on this vertex, but has observed nesting elsewhere
      if $\{\neg$hasNesting($oneNstdTypeInstnc$)$\wedge$
        $\exists oneNstdTypeInstnc\prime \in allNstdTypeInstncs \mid$
        $oneNstdTypeInstnc\prime.type = oneNstdTypeInstnc.type$
        $\wedge$hasNesting($oneNstdTypeInstnc\prime$)$\}$
        // Save the type instance showing both nesting and non-nesting
        $allNstdTypeInstncs \leftarrow allNstdTypeInstncs \cup \{oneNstdTypeInstnc\}$
      end
    end
  end
  return $allNstdTypeInstncs$
```

Algorithm 29: Collect all nested types

takes the set of vertices of the MTG ($\mathcal{V}$) and one type hypothesis ($typeHypothesis$) as arguments. In line 2 a set ($allNstdTypeInstncs$) holding all nested type *instances* is initialized. The computation of those nested instances is done throughout lines 9-6, where each vertex (v) is iterated and all type instances are nested per vertex based on the current type hypothesis (line 16) by calling function NESTTYPES. This method uses the offsets of the type instances within a vertex and the size of a type to compute the type nesting, and is not shown in more depth. After the first nesting computation, a second iteration of all the vertices is required to select all type instances that have not shown nesting themselves, but an instance of the same type has shown nesting elsewhere in the MTG. The resulting set of nested type instances ($allNstdTypeInstncs$) is returned to the main algorithm.

```
function GROUPNESTEDTYPEINSTANCES($allNstdTypeInstncs$)
    // Initialize a key-value store.
    // Key: type
    // Value: set of nested type instances for the key type
    $\mathcal{KV} \leftarrow \emptyset$
    // Process all nested type instance combinations
    for each $(typeInstnc, typeInstnc\prime) \in allNstdTypeInstncs \times allNstdTypeInstncs$
        // Check if type instances are of the same type
        if $typeInstnc.type = typeInstnc\prime.type$
            // Store all nested type instances for this particular type
            $\mathcal{KV}(typeInstnc.type) \leftarrow \mathcal{KV}(typeInstnc.type) \cup \{\{typeInstnc.instances\} \cup \{typeInstnc\prime.instances\}\}$
    end
    return $\mathcal{KV}$
```

Algorithm 30: Collect and group all nested type instances

The returned nested type instances are processed further in the main algorithm in line 5. Here, the computed nested type instances are made consistent, by aggregating all nested type instances for a particular type, i.e., creating a superset of the nested type instances. This is detailed in Alg. 30, which receives all nested type instances ($allNstdTypeInstncs$) as an input parameter. The algorithm computes a key-value store ($\mathcal{KV}$) where a type ($typeInstnc.type$) is used as a key and all the corresponding found nested type instances ($typeInstnc.instances$) for this particular type are stored as a set. The resulting key-value store holding all the interpretations per type gets returned and is stored within the main algorithm ($allIntrnsForType$).

Finally, in line 6, the possible combinations of the previously aggregated nested type instances ($allIntrnsForType$) are computed. This means to create all com-

```
function createAndCombinePossibleTypeCombinations($allIntrnsForType$)
  // Key-value store: key (DSIType) - values (set of DSIType sets)
  $\mathcal{KV} \leftarrow \emptyset$
  // Cycle through the previously calculated interpretations
  for each $intrnsForType \in allIntrnsForType$
    // Set holding the DSIType combinations
    $DSITypeCombinations \leftarrow \emptyset$
    // Fetch key (type) and values (interpretations)
    $keyType \leftarrow$ key($intrnsForType$)
    $valIntrns \leftarrow$ values($intrnsForType$)
    // Map the type instances back into DSITypes
    $nstdDSIType \leftarrow$ map($valIntrns$)
    // Create the power set of the nested DSITypes
    $typesPowerSet \leftarrow \mathcal{P}(nstdDSIType)$
    // Cycle through the power set and check pairwise type compatibility
    for each $typeSet \in typesPowerSet$
      // Store the result for one combination, if all
      // DSITypes are pairwise compatible
      if $\forall(type, type\prime) \in typeSet \times typeSet \mid$
        $\neg$isOverlapping($type, type\prime$) $\vee$ isNested($type, type\prime$)
          $DSITypeCombinations \leftarrow DSITypeCombinations \cup \{typeSet\}$
      end
    end
    $\mathcal{KV}(keyType) \leftarrow \mathcal{KV}(keyType) \cup \{DSITypeCombinations\}$
  end
  // Compute all combinations:
  // n-ary cartesian product between all sets for each type
  $valIntrns \leftarrow$ values($\mathcal{KV}$)
  return $\{valIntrns_0 \times valIntrns_1 \times \cdots \times valIntrns_n\}$
```

Algorithm 31: Create the type combinations

binations, where each nested element is either present or absent. Again, the consistency of the nested type instances needs to be ensured, as the aggregation step might possibly introduced inconsistent nested type instances. The function createAndCombinePossibleTypeCombinations is shown in Alg. 31. The function takes all interpretations for a type ($allIntrnsForType$) as an argument. One important aspect of the function is the mapping between the type instances back into types (line 12). The different interpretations of the nested types are held in a key-value store, where the key is a particular type and the value is a set of type sets. The set of sets represents every combination of nested types for the particular parent type. The function iterates all interpretations for a type and computes the power set to account for all possible nested type combinations (line 14). Each set of the power set is checked for compatibility of all nested types (line 19). If this is the case, the nested type combination is stored. This possibly results in multiple nested type combinations for each type of the MTG. Therefore the n-ary cartesian product needs to be computed between all nested type combinations, to arrive at all possible valid type combinations for a MTG (line 28). This is again a set of type sets and gets returned as the result.

In the main algorithm (Alg. 28) the returned result is stored within a set of all the refined hypotheses ($allNstdHypthss$). After all hypotheses ($allHypotheses$) are processed, the resulting set of all refined hypotheses is returned. This result ($allNstdHypthss$) is subsequently used by DSIbin to compute the DDS (Phase (g) of Fig. 16.1) for each combination and finally select the best interpretation according to the previously described taxonomy (Phase (h) of Fig. 16.1).

16.3 Complexity of the DSIref algorithm

This section discusses both the runtime and space complexity of the algorithm from a high level perspective, while in Ch. 17 figures are presented, which are measured with our prototype implementation. For the DSIref algorithm nearly all values are specific to particular instances of the MTG, e.g., the number of types per edge. To arrive at a worst-case classification, this aspect is abstracted and always the maximal number of a particular property across all instances is assumed.

Runtime complexity. The DSIref algorithm is split into two main parts, as shown in Alg. 17: (i) the type creation and propagation phase, and (ii) the type compatibility and hypotheses creation phase. These parts are discussed individually, in the following.

Part (i) has a quadratic runtime complexity in the worst case, if each DSItype can be propagated maximally along the edges of the MTG. The size of the processed type set depends on the size of the memory chunks, and whether the types match between the source and the target memory chunks for an edge. The edge and type sets are typically limited by the small size of both the MTG and the memory chunks. Further, the worst case cannot happen, as it is impossible that with each edge processed new types are discovered that haven't been discovered and propagated before.

Part (ii) calculates the type compatibility among the types and computes the various type hypotheses. The first compatiblity stage can have a quadratic complexity due to the computation of a cartesian product. The second compatibility stage computes the global type compatibility by comparing the local type instance compatibility for each vertex. This computation depends upon the cartesian product, i.e., quadratic complexity, and the power set, i.e., expontential complexity. The following third compatibility stage ensures the global compatibility of nested types by computing the following steps for each previously created type hypothesis: (a) collecting and nesting all nested type instances in a loop over the set of MTG vertices; (b) collecting and grouping all found nested type instances per type; (c) mapping the type instances back to types and computing the various type permutations.

Step (a) requires to iterate the set of MTG twice, where the second loop requires more computational steps, as for each vertex the nested type instances need to be iterated and checked against all the already computed nested type instances. This results in a linear runtime complexity. The following step (b) iterates the cartesian product of all nested type instances, leading to a quadratic complexity. Finally, step (c) cycles through all interpretations for a type and for each type processes the power set of the nested types. For each element of the power set the cartesian product is computed, resulting in an expontential complexity. The final computation of step (c) is the calculation of all permutations of the collected type combinations, requiring a quadratic runtime. While the complexity of these steps can become significant in the worst case, one needs to remember that these algorithms describe the general case of unifying nested types. The computational costs are mainly bound by the number of different nesting scenarios for a type and the size of the MTG. Both are rather small in practice in the common case, as observed in our benchmark in Ch. 17.

If one thinks about relaxing the restrictions of type compatibility when mapping memory regions along pointer edges, the complexity might become more of an issue as much more mappings are possible along the edges. Such a scenario is thinkable, e.g., if one does not trust Howard's inferred primitive types and only relies on the size of the allocated memory chunks and the pointer connections. This scenario could be explored in future work to evaluate the state space explosion.

Space complexity. Part (i) of the DSIref algorithm is not too space consuming, as it only depends linearly on the amount of edges, vertices and type mappings.

Part (ii) of the algorithm is more space intensive, as in stage one a local type compatibility matrix per vertex is computed, resulting in a quadratic space consumption per vertex. The local type matrix per vertex is stored for the remainder of the algorithm.

The second stage computes the global type compatibility matrix from all found types, leading again to quadratic space consumption. When computing all hypotheses a power set of all types gets created leading to an expontential space consumption.

The third stage calculates the type combinations. The space consumption of the intermediate steps is neglected, as they do not add significantly to the overall space consumption. However in Alg. 31 a power set of the nested types is computed again, together with the cartesian product for each set of the power set, resulting in an expontenial space consumption. Finally the n-ary cartesian product of all created interpretation sets is computed and stored, resulting again in an expontential space consumption.

As is the case with the runtime complexity, the space complexity can potentially become significant. However, our benchmark in Ch. 17 shows that the memory consumption is easily feasible for all the examples of the benchmark.

17 Benchmarking of sophisticated approach

The type refinement approach discussed in the previous chapter has been implemented by us into the DSIbin tool chain as the DSIref component shown in Fig. 14.2. The implementation consists of nearly 3k LOC of Scala and about 700 LOC of Perl/Bash scripts. The parallelization of DSIsrc, as discussed in Ch. 7, has actually also been implemented by us in the context of DSIbin, as DSIref may generate many hypotheses that need to be evaluated by DSIcore and thus requires higher throughput as is the case for the DSI prototype operating on source code.

DSIref has been benchmarked with the benchmark shown in Tab. 15.1. With the new DSIref component it is now possible to identify 26 of the 30 examples, instead of 10 with only the naive combination of Howard and DSIcore. This chapter is organized as follows. At first, an overview of the performance of the sophisticated approach is given in Sec. 17.1 followed by a discussion of the results in Sec. 17.2.

17.1 Performance

This section discusses the performance aspects of DSIbin. As mentioned before, DSIbin benefits from the parallel implementation of the DSIcore algorithm as discussed in Ch. 7. The longest running examples have an event trace length of 5.5K events. The type inference with Howard together with DSIref's type hypotheses generation took in the order of seconds (min: 1s, max: 13s, avg: 3.13s). The memory consumption is about 565MB RAM on average (min: 0.2GB, max: 2.8GB). Each type hypotheses, including the initial type information inferred by Howard and all created type hypotheses by DSIref, requires the creation of an individual execution trace with the corresponding type information. Each run is finished in seconds (min: 0.7s, max: 17s, avg: 1.8s) and consumed 27MB RAM on average (min: 24MB, max: 33MB). However, this step also depends on the execution time of the program under analysis.

The actual evaluation of all hypotheses for an example with DSIcore takes in the order of (tens of) minutes. When looking at the evaluation of one hypothesis, the average evaluation time is 63s (min: 1s, max: 2622s). Interestingly, the two examples taking the longest computation time (50 minutes each), are different in their nature. The first example consists of a large number of hypotheses (156), but each hypothesis is evaluated quickly. The second contains only a few hypotheses that are complex and, thus, require more computational resources. On average, each example produces 13.8 hypotheses (min: 1, max: 156), while the average memory consumption for a hypothesis is 2.4GB RAM (min: 0.2GB, max: 3.2GB).

The tool chain is a prototypical implementation, which is composed of several parts, e.g., the Pin tool for instrumentation which is written in C/C++ and the (larger) DSIcore part running on the JVM. Thus, more performance improvements are expected, e.g., some of which are also discussed for future work in Ch. 7, when putting more engineering effort into the tooling.

17.2 Discussion

We now discuss the benchmark results from Tab. 15.1, which are listed under the heading "Sophisticated Combination" and where the columns are the same as explained in Sec. 15.1. Additionally, column **(pr)** states, whether a type refinement of the primitive data types is conducted. The final column **(ch hyp)** states which technique produced the chosen interpretation by the algorithm, while brackets indicate a wrong interpretation. Examples that do not require type merging, i.e., they only contain one allocation site, are listed with an "o" in column **(m)**.

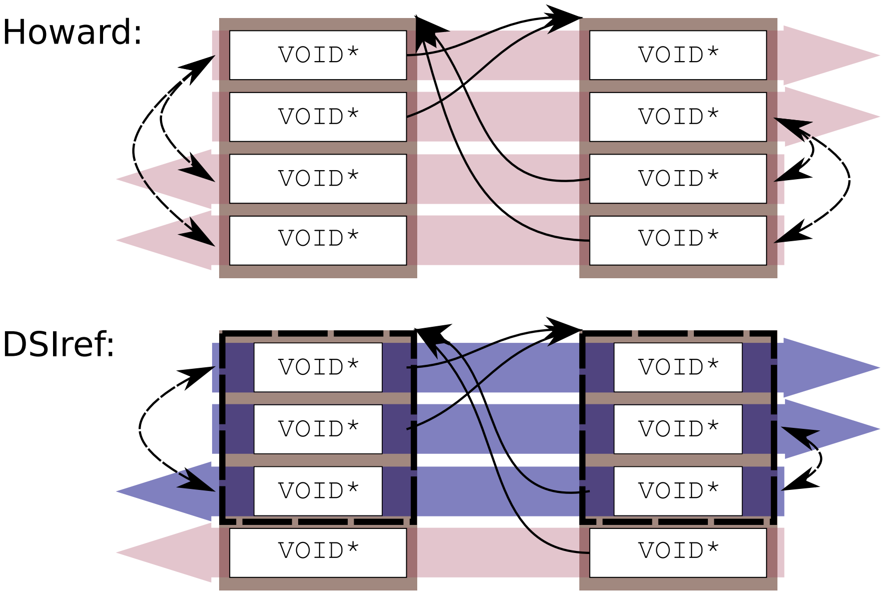


Figure 17.1: Two doubly linked lists running in parallel, showing the possible strand combinations for the types inferred by Howard (top) and by DSIref (bottom). Figure reproduced from our publication [94].

17.2.1 Sophisticated versus naive combination

With the help of DSIref, it is possible to increase the number of (correctly) detected DDSs from 10 to 26 out of the 30 examples. Thus, the sophisticated combination generally outperforms the naive combination. However, we have found an example (`lit-5`), where only the naive combination produces the correct result. Thus, the initial unmodified types inferred by Howard are added to the set of type hypotheses generated by DSIref. Consequently, all hypotheses are treated equally when evaluating them with DSIcore. This reflects the nature of the DSIref approach, where the pointer and primitive type information is considered a strong hint of how the actual DDS looks like, but nevertheless is a speculative approach.

As seen in column **ch hyp**, situations exist, where both the naive and sophisticated combination miss the ground truth. This can be seen in example `tb-4`, which contains an SLL, where the head and the tail of the list are allocated at different locations. The list carries a string payload, but the access pattern for the head and the tail differ. For the head, the `strcpy` function from `string.h` is used, whereas a single character gets assigned for the tail nodes. These different usage patterns lead Howard to infer different primitive type interpretations, which in turn prevents both Howard and DSIref from merging the two allocation sites. The inferred primitive data types from Howard are incompatible and thus not mergeable by either of the two approaches. The countermeasures for such a scenario could be manifold, and are left for future work. On the one hand, the example encourages to deepen Howard's ability to detect `memcpy` like functions, e.g., `strcpy` in this particular case, to refine its type inference. If Howard is able to infer binary compatible types in such cases, the merging would again be possible for both approaches. On the other hand, DSIref's binary compatibility could be relaxed to allow for more merging options. This in turn would increase the number of possible type mappings and, thus, the number of hypotheses that need to be evaluated.

17.2.2 Type merging

With examples `syn-08` and `lit-2`, it can be seen how the merging strategy of types influences the precision of the inferred DDS. The two examples show the strengths and weaknesses of DSIref's merge strategy in comparison to the naive implementation. We start out with the positive example `syn-08`, where we first discuss the ground truth of its used DDS. It contains a child DLL whose head node is embedded inside the parent DLL, i.e., nesting on overlay. The head node is accessed flattened; thus, Howard is unable to recognize the nested head node, which in turn leads DSIcore to identify an indirect nesting relation between the parent DLL and the child DLL. Hence, Howard's merge strategy is insufficient to reveal the ground truth. The missed child head node can cause even bigger problems leading to a missed child DLL by DSIcore, if the child consists of only two elements. In this scenario, DSIcore would not form a strand for the child, as at least two elements are required for a strand. Thus, in the worst case all child elements only consist of the missed head node plus an additional node, and DSIcore would miss the parent-child relation completely. As a side note, structural and temporal repetition mitigates this problem, if at least some child element has more than two elements. However, overlay nesting would never be detected with the missed head node.

When applying DSIref's merge strategy, the head of the nested child is revealed by mapping the child element outside of the parent back into the parent along the pointer connection. Thus three things happen: (i) the previously missed nested element is detected, (ii) the type inside the parent is merged with the type outside the parent, and (iii) the `previous` pointer of the nested head element is typed to a pointer. Besides missing the nested node completely, Howard is unable to type the `previous` pointer, as it is not accessed within the nested head node. This underlines the different usage patterns of different DDS parts, as we have already discussed previously.

When looking at (ii) in more detail, the child DLL is allocated at two different allocation sites and no iteration pointer is present. This prevents Howard from merging the allocation sites of the child DLL. As DSIref merges types along the pointer connections all nodes of the child DLL are merged. The absence of the iteration pointer and the presence of two allocation sites also pushes into the direction of obfuscating the DDS. This allows us to construct situations where the true shape of the DDS cannot be recognized with Howard. This can be seen with the currently discussed example where only the parent DLL gets detected with the naive approach leaving the child DLL undetected. With the sophisticated approach the ground-truth interpretation can be inferred.

To contrast `syn-08` we now discuss example `lit-2`, where the type merging strategy of DSIref decreases the precision of the DDS detection. The ground truth of the example is a multitude of SLLs that form multiple levels of nesting-on-indirection. Each level has a unique type, although all of them are binary compatible, as each consists of two pointers. One of the pointers is used to connect the SLL nodes of the current level, i.e., the horizontal connection. The other pointer is the downward or vertical connection from the parent to the child SLL. The only difference among the otherwise binary compatible types is that sometimes the first and sometimes the second pointer is used for the horizontal and vertical linkages, respectively. This in turn leads the type merging of DSIref to merge some of the types in the MTG, i.e., as long as the linkage condition is met as discussed in Ch. 5. With this merged type information, DSIcore now detects both nesting-on-overlay as well as nesting-on-indirection, which does not reflect the ground truth of exclusive nesting-on-indirection relations. DSIref cannot detect a situation where the types are binary compatible and are only distinguished by the programmer via their naming found in source code.

As discussed before, the types inferred by Howard are also added to the set of possible type hypotheses, to be evaluated by DSIcore. In this particular example, the types inferred by Howard actually reveal the ground truth interpretation of the example due to the lack of an iteration pointer. However, the final selection algorithm, which chooses the most complex DDS from the set of hypotheses, considers overlay nesting to be structurally more complex than indirect nesting, which leads to the wrong interpretation. This problem is related to false positives introduced due to the merging strategy. False positives are also discussed in Ch. 11.

The examples `lit-3`, `r-1`, `r-2`, `syn-01`, and `tb-3` all require to merge (**m**) the nodes of the DDS, which are all standard DDSs, such as DLL, BT and SL on overlay. All these DDSs are exclusively heap allocated. Here, DSIref fully merges the nodes of the examples, as does Howard. So DSIref does not introduce any false positives in this scenario, which would prevent the detection of the true shape of the structure. The same is true for examples `syn-15` and `syn-09`, where the nesting relations are unaffected by DSIref, i.e., the merge strategy does not introduced false positives.

17.2.3 Shortcomings of the DSIcore algorithm

Example `r-3` reveals a general shortcoming of the DSIcore algorithm. DSIcore has chosen the design decision to create strands immediately upon the detection of a linkage condition, i.e., two cells of the same type get connected. But this can lead to wrong interpretations of the DDS. Example `r-3` consists of a parent DLL with indirectly nested child DLLs, and is taken from the Carberp malware [34]. In principle, the data structure can be detected, but the DSIcore algorithm treats the connection between the parent and the child elements as a strand of length two. This results in an DLL interpretation of length 5 with a hint on nesting on overlay (see Tab. 15.1). The result has nothing to do with the merge strategy of DSIref, but is the mentioned design decision of DSIcore, i.e., DSIcore produces the identical result on the source code example. With example `syn-10`, DSIref does a similar misclassification as with `r-3`, i.e., DSIcore wrongly chooses the strand of length two between the parent and the child SLL as an opportunity to detect a nesting relation. When viewed in isolation, this assumption of DSIcore is correct. Hence, DSIref performs the correct type merging, but the limitations of DSIcore prevents the intuitively correct interpretation from the point of view of the parent strand.

The consequences are similar to the previous example `lit-2`, as indirect nesting gets interpreted as overlay nesting. This results in folding the child elements back into the parent DLL, when creating the FSG, i.e., when performing structural repetition. The resulting DDS consists of one DLL only, instead of two, i.e., one for the parent and one for the child.

A strategy to counter this problem could be a delayed creation of strands by DSIcore, e.g., only create a strand once more elements over a configurable threshold are observed. Once reaching the threshold, the information could be propagated backwards in the trace. Another strategy could be the same as is already proposed in Ch. 8 for handling `tsort` like DDS, i.e., to track the access patterns into the DDS, which might reveal the parent and child parts of the DDS. With such information, a folding of the parent and child elements in the FSG might be preventable. Both approaches could be investigated by future work.

17.2.4 C++ and C examples and loss of perfect type information

This section points out that the usage of binary code opens up DSIbin not only for C, but also for C++. Therefore, C++ examples where added to the benchmark: `er-1`, `er-2`, `syn-02` and `syn-03`. Further, those examples are discussed in the context of the loss of perfect type information, as predicted for DSIcore in Ch. 11.

At first, let us briefly discuss that DSIbin is capable of processing both C and C++ examples. Therefore, we sum up several examples that contain the same DDS (at least partially) once implemented in C and once in C++. Specifically, these are examples `er-1` and `er-2` with a LKL like std::list implementation in C++ and example `syn-7` with the corresponding LKL implementation in C. In both C++ examples the linkage struct gets embedded into the payload with an additional external head node. Examples `syn-02` and `syn-03` are C++ implementations of SLLs, including nesting on overlay. The corresponding C examples are, e.g., `syn-05` and `syn-14`. All C/C++ examples are detectable by DSIbin, as seen under **Sophisticated Combination** in Tab. 15.1 in column **DSIbin**.

Now we pick up the discussion about the loss of perfect type information and its consequences for DSIcore as discussed in Ch. 11. In the following we will review selected examples of our benchmark that support the predicted outcomes when type information gets lost. One predication is the lost precision of the detected DDS, which can be seen with example `er-1` whose detection degenerates from a CDLL into a DLL. It uses a nested element at the head of the surrounding struct (see column **n@h** under section **Code** in Tab. 15.1). Without perfect type information, i.e., using only Howard's type information where the nested element at the head is missed, it is not possible to reveal the CDLL. Instead, a DLL is detected, as the head element is not considered as being part of the remainder of the list.

Example `syn-02` degenerates from an overlay parent-child relation between SLLs into an indirect nesting relation. The reason is that the nested element of the child SLL resides inside the parent SLL and is not merged with the remainder of the child list when using Howard's type information. For both examples, the revealed DDS is not too far off from the ground truth, yet operations such as insertions into a DLL and CDLL might differ significantly, making the higher precision of DSIref highly desirable for a binary analyst.

17.2.5 Strand length

A variety of the SLL examples, e.g., tb-1, tb-2, syn-11, and syn-12, only differ in one element in length. This is also predicted during the discussion of type information loss, although the impact is less severe as an analyst still gets the correct overall interpretation of an SLL in these situations. However, the examples reveal shortcomings of the naive combination that prevent detecting a strand fully. This can be extended to lead to misinterpretations or missing a strand completely, although the examples only provoke missing one element in a strand. Thus, some of the aspects discussed in this section can also be seen from the perspective of DDS obfuscation, as discussed in the following section.

Example syn-11 nests the head of the list into a dedicated head node, which requires DSIcore's cell abstraction and merging of the nested head node with the exclusive elements of the remainder of the list. DSIref is able to perform the type merging and thus reveals the true length of the SLL.

Example syn-12 creates two dedicated list head nodes on the heap and subsequently allocates the remainder of both list heads at two different allocation sites. Afterwards both lists are connected, i.e., the next pointer of the first list gets connected to the head of the second list. Howard is able to merge both lists, except for the head node of the first list with the rest of both lists. This is because of the lack of an iteration pointer. Instead, DSIref only requires the setting of the next pointer to perform its type merging along the pointer connections, as long as binary compatibility between the connected memory chunks is given. The type merging by DSIref allows the inclusion of the previously missed head node into the analysis, resulting in the identification of the correct strand length.

Interestingly tb-1 and tb-2 allocate a dedicated head node, on which an insertion after the head node gets performed. Hence the head node is only the handle into the list. As the head node and the remainder of the list are allocated at different allocation sites but no iteration pointer iterates both over the head and the remainder of the list, Howard has no chance to apply its merge strategy. For DSIref, setting the next pointer is again sufficient to perform the type merging and reveal the head node of both SLLs.

syn-04 which is a SL like implementation, with only one skip level. The list contains barrier nodes, which store the meta data for deciding whether a skip forward in the list to the next barrier node is required, or a local search following the payload nodes immediately after the tested barrier node should be performed. The payload nodes are a DLL and the barrier nodes form an SLL running in parallel to the DLL. The DLL and the SLL only touch at the barrier nodes. The linkage of the SLL is chosen in such a way that a propagation of type instances as inferred by DSIref creates overlapping type instances. This challenges DSIref's collision detection algorithm that resolves such issues, i.e., Phase (f) of Fig. 16.1. With the refinement both the full length of the DLL can be revealed together with the SLL of the barrier nodes, with multiple intersection points between the SLL and the DLL. Without DSIref, the SLL cannot be revealed as the nested element within the barrier nodes that establishes the SLL linkage is accessed **flat**, i.e., the nested element is not detected by Howard resulting in an incoming pointer which is not at the head of a struct. Thus, DSIcore does not consider the SLL linkage, as only pointers pointing to the beginning of a (nested) struct are considered for creating strands. This scenario can also be seen in Fig. 11.1 at the bottom.

17.2.6 Data structure obfuscation

The DDS obfuscation aspect of some of the examples, both intended and unintended, are already discussed in Ch. 15. It is possible to hide the presence of a DDS completely for the naive approach, e.g., examples `syn-06` and `syn-13`. While they artificially avoid an iteration pointer, these techniques could potentially be used in a larger context, e.g., creating an SL, where chances are high that not all parts of the SL are accessed, i.e., iterated. This would lead DSIbin to not being able to detect the SL, as the strand connections would not match the SL predicates. With the help of DSIref, the two examples can be fully revealed, as DSIref does not rely on the presence of iteration pointers. As discussed previously, tracking memory chunks and pointer connections is sufficient for DSIref to create its MTG and to refine and merge types subsequently.

Example `syn-16` puts the head node of an SLL onto the stack. Otherwise, the list is created and used without any restrictions, e.g., the list is iterated freely during the execution of the example code. As DSIbin tracks both stack and heap-allocated memory and places the memory chunks into the MTG, the allocation site is transparent to DSIref. This allows DSIref to merge the heap and stack types, something which is not done by related work [51]. Thus, merging the head and remainder of the list requires DSIref. Without DSIref, a list can be hidden from detection by iteratively allocating and connecting heap- and stack-allocated nodes. Both the examples `lit-1` and `lit-4` also suffer from cut off head elements, a mixture of heap and stack allocated memory, and nested head elements. DSIref is able to reveal the missing head nodes and perform the type merging along the pointer connections. This allows to identify the cyclicity of the DLLs. Additionally, the nesting on overlay relation is identified by DSIcore with the information provided by DSIref. The nesting relation is also refined by DSIref with example `syn-14`, which also has a nested child struct inside of the parent struct. The difference is the missing cyclicity between the CDLL child of the previous two examples and the SLL child of the current example. This in turn leads to less pointer connections in the MTG. Therefore the examples both test the propagation of type instances along cyclic and non-cyclic pointer connections within the MTG.

Example `tb-4` suffers from the imprecise primitive type recovery of Howard as discussed in Ch. 15. As the inferred types by Howard are not binary compatible, it is not possible for DSIref to merge the types as is the case with `tb-1` and `tb-2`. This technique can thus be used to obfuscate a DDS whenever it is possible to trick Howard into inferring binary incompatible types, e.g., due to different usage patterns of the payload.

Both intended and unintended obfuscation techniques could potentially be used in addition to polymorphic code techniques to hide or alter the DDS between different malware versions. This would trick techniques such as Laika [60], which creates signatures for detecting malware (families) based on DDS. Or, to put it the other way round, DSIbin could be beneficial for signature based approaches as it still detects the true DDS in most situations.

17.2.7 False positives for nesting detection

An interesting aspect of DSIref is the detection of nested elements, when the ground truth does not have nested elements, e.g., examples `r-1` and `syn-02` and `syn-08`. DSIbin still reveals the correct DDS interpretation for those examples, although DSIref reports false positives for the nested elements. However, this can rather be seen as a feature of DSIref that reveals the actual linkage backbone of the DDS. More precisely, DSIref separates the DDS part from the payload part to a certain extent.

17.2.8 Primitive type refinement

For four of the examples, i.e., `tb-1`, `tb-2`, `syn-08` and `r-3`, DSIref is able to refine the primitive types as inferred by Howard. Specifically, DSIref is able to type previously untyped memory, which stems from different usage patterns of different parts of DDS which prevents Howard from typing, e.g., unaccessed memory. This happens, e.g., when the head or tail node of a list is not used in the same way as the remainder of the list.

Consider a DLL with its `previous/next` pointers that are never used in case of a head/tail node. This can, e.g., be seen with example `r-3`, where the previous/next pointers are revealed by DSIref. Additionally, the payload elements of the head/tail nodes are typed by DSIref, but not by Howard as they are never accessed. Both the pointer and payload refinement are shown in Fig. 10.1. Note that Howard types memory that gets `NULL` assigned as `INT64`, which is observable in Fig. 10.1 by the switched position of the `VOID*` and `INT64` element corresponding to the used previous/next pointer of the head/tail element. Thus, DSIref exclusively refines the `INT64` to `VOID*` and the remaining payload elements, which are left untyped by Howard due to the lack of access patterns.

Another aspect to note about the refinement is that the refinement is even done on a fine grained basis in terms of nested structs, in the sense that not the complete memory needs to be refined at once. This is in contrast to ARTISTE, which also refines types but only complete memory chunks. Refining complete memory chunks is also supported by DSIref, as can be seen with examples `tb-1` and `tb-2`.

18 Conclusions

In Pt. II of this dissertation, its second research question was answered:

> **Research Question 2:** *Can DSI's concepts and strengths be preserved when inspecting binaries?*

This question was answered positively by developing, implementing and benchmarking DSIbin. DSIbin improves upon DSI, as source code is no longer required and, due to the nature of binary code, the analysis could be extended from C to also C++ binaries. Thus, DSI is now enabled for reverse engineering binaries, a tedious and error prone, yet mandatory task when source code is unavailable. SEGA's example of lost source code [1] and the need to analyse the steadily growing amount of malware [17] are prominent uses cases. DSIbin opens up DSI to binaries and thus improves upon the state-of-the-art [49, 69, 74] by handling arbitrary parent-child nesting, data structures distributed between heap and stack, and data structures running through nodes of different types such as the Linux kernel list or the C++ std::list implementation. Additionally, data structures such as skip lists which are neglected by related work, e.g.,[49, 69], are now identifiable by DSIbin due to DSIcore. In contrast, MemPick [69] cuts connections between nodes of different types, thus disabling the detection of nesting or data structures comprised of differently typed nodes. DDT requires the presence of well defined interfaces, whereas DSIbin does not assume any the presence of interface functions. ARTISTE [49] executes its analysis on a sample of the heap every n-th time step, without avoiding or explicitly handling degenerate shapes, leading to loss of precision.

To enable DSI on binaries, the unavailability of type information found in source code was compensated when dealing with the first challenge:

> **Challenge 2.1:** *Can external type recovery tools for binaries excavate sufficiently precise information for DSI to function on binaries?*

The problem of recovering all required memory-manipulating events from the binaries without access to source code was addressed in the second challenge:

Challenge 2.2: *Can the required event trace generated with CIL be reproduced with a binary instrumentation framework?*

Both Challenges 2.1 and 2.2 were answered by developing the DSIbin prototype, which uses the state-of-the-art type recovery tool Howard [98] and the Intel Pin framework [82] for instrumenting binaries. Challenge 2.2 was answered positively, because all memory manipulating events needed by DSI's offline analysis are captured by the Pin framework. This required the creation of a shadow heap and stack for tracking the states of both memory regions, thus tracing memory (de-)allocations and local variables entering and leaving scope. Additionally, pointer writes to the heap and stack are recorded. As binaries rely on the usage of registers, e.g., for returning values from functions, it is crucial for DSIbin's analysis to model those, too, because registers might be the only handle into allocated memory. Otherwise false positives could result from the memory leak detection algorithm (see Ch. 4), resulting in an undermined analysis with missing parts of a data structure.

Challenge 2.1 cannot unrestrictedly be answered positively. It is indeed possible to draw useful information from the state-of-the-art type recovery tool Howard, which was incorporated into DSIbin and which enables the detection of certain data structures like skip lists and parent-child nestings which are not covered by related work. However, Howard misses nested structs and does not perform type merging between nested instances, and combinations of nested and exclusive instances of the same type. Additionally, types spread across heap and stack are not merged. These features are, however, vital for DSIbin to overcome additional limitations of related work [49,69,74], such as data structures running through nodes of different types or data structures distributed across heap and stack. Therefore, DSIbin needed to attack the limitations of Howard, leading to the third challenge:

Challenge 2.3: *Can the recovered type information from binaries be refined with the help of DSI itself?*

The positive answer to this challenge has certain facets to point out. The refinement is somewhat recursive in nature, because DSI itself is used to refine the types it requires. The idea is that DSI selects the most complex data structure given the multitude of possible type refinements, with the intuition being that the features of a data structure do not appear by coincidence. To base the type guessing on solid facts, the implicit information found in pointer connections between memory regions is exploited to (i) reveal nested structs based on the assumption that incoming pointers are always to the head of a (nested) type, and (ii) perform type merging between memory regions. Our type refinement approach transparently handles both heap and stack, something that is not done by current type recovery tools [51]. Because type refinement improves Howard's recovered type information by revealing previously missed nested types, performing type merging and typing some missed primitive data types, DSIbin implicitly acts as an improved version of Howard.

All these aspects are fundamental when reverse engineering binaries because (i) reasoning about DDSs would not be possible without type merging, because connected elements of the same type allocated at different locations could not be recognized as being of the same type; (ii) the smallest common memory subregion of DDS comprised of differently sized nodes, such as the Linux kernel list, could not be determined without nesting detection; (iii) untyped memory regions need to be further analysed by the reverse engineer without primitive type recovery. All three aspects give a much more comprehensive view upon the binary; information is now revealed such as which mallocs allocate the same type of memory for a DDS, and which stack based nodes participate in a DDS. Even if analysts are not primarily concerned with DDSs, DSIbin's results should also be interesting for them because DDS specific code can be identified and then be left out of the analysis of the core logic of the binary.

The improvements of DSIbin were shown via an extensive benchmark that includes real world examples such as the VNC clipping library found in Carberp [34], The Computer Language Benchmarks Game: Binary Tree [32], and the Olden Benchmark [20]. Additionally, examples from the shape analysis literature, textbook examples and hand-written examples were used to create a diverse benchmark. The hand-written examples stress test some features such as arbitrary nesting, or try to obfuscate nested structs to prevent Howard's merge strategies. The latter aims at demonstrating how resilient the refinement approach is to obfuscation techniques. From the shown results, it can clearly be seen that the first Howard-DSI combination is already quite promising by correctly identifying examples out of the box. However, with the refinement step in place, the precision improves significantly by including stack elements into the analysis, thus enabling, e.g., the detection of cyclicity, revealing doubly linked lists that have been completely unobservable before, or refining a nesting scenario from indirect to overlay nesting by uncovering a previously unseen nested struct.

Future work. Because DSIbin provides the same information as DSI about the identified data structures, such as its cells, its strands and the final data structure interpretation, DSIbin should be seamlessly integrateable into the operation detection and visualization techniques discussed in Pt. I. This would be another notable advancement of the state-of-the-art in reverse engineering, where operation detection is either not done at all or has strong assumptions, such as the presence of well defined interfaces [74].

Because DSI tracks the data structures and how they are accessed, it is possible to establish a notion of ownership between code and its associated data structures. This information could be used to enable the monitoring of software to detect behavioural changes in line with [93]. This helps building intrusion detection systems that periodically monitor whether known DDS perform out of the norm and whether new and unexpected DDSs occur.

When dealing with C/C++ code, custom memory allocators are common practice [45]. Currently, DSIbin does not handle such allocators, but they can be detected by techniques such as [54]. The integration of such an additional tool into the current DSIbin tool-chain needs to be investigated in order to address the question how well DSIbin works in this scenario: how does the instrumentation with Pin work for those custom memory allocators? How does the type recovery perform? Once custom memory allocators are visible, DSI's memory abstraction should be able to transparently handle such memory states.

With the recovered precise data structure information found in malware binaries, it would be interesting to create new or extend existing signatures that are used to detect malware families. This approach is in line with the one of Laika [60], although Laika does not offer DSI's precision in detecting data structures; utilizing DSI's information should enable a more fine-grained classification of binaries. Naturally, when tackling malware from the angle of data structures, an arms race is inevitable between obfuscation techniques for data structures and counter measures against those. The obfuscation techniques given in [79], which mainly randomize the order of `struct` members and add ambiguous memory paddings, are no problem for DSI. Instead, the XOR-list [44, 76] is an example that breaks DSI's current detection capabilities by blurring the actual pointer values by XOR-ing two pointer fields into one. In this situation, techniques such as information flow tracking [100] may help to determine from which memory locations an address is calculated; this would allow DSI to still establish its strand abstraction by observing pointer dereferences. Besides the pointer manipulations, additional code obfuscation techniques should be tested so as to actually see how robust DSIbin is against such attacks.

The DSI/DSIbin approach is not bound to a particular programming language or operating system. As stated before, C/C++ programs are used exemplary here, because they allow for challenging heap states that are not supported by other programming languages. Given the popularity of the Windows and Android platforms, one could investigate DSIbin on these platforms. For Windows, the amount of available malware is much larger than for Linux, whereas the wealth of Android applications is a motivating use case for porting DSIbin to Java, a programming language far more restrictive than C. Indeed, the Java bytecode already exposes type information found in source code, which might deem DSIbin superfluous. However, the precise behaviour of a data structure cannot be drawn from this information with certainty, as a data structure might be, e.g., misnamed or misused. Additionally, Android offers obfuscation by shortening class, field and method names [25], which makes DSI's capabilities even more desirable to a Java bytecode reverse engineer.

Part III

Conclusions of dynamic data structure detection on source code and binaries

This dissertation contributes to the state-of-the art in automated dynamic data structure detection by means of a dynamic analysis in two domains: (i) program comprehension of dynamic data structures when C source code is available, and (ii) reverse engineering of dynamic data structures in C/C++ binaries. Both (i) and (ii) deal with the recent approach DSI[1] that tackles limitations of related work [49,69,74] in terms of detectable data structures, e.g., skip lists, and complex parent child nestings. DSI employs a novel memory abstraction based on strands, which are essentially singly linked lists, and their interconnections to form complex dynamic data structures.

Additionally, DSI is capable of inspecting the whole lifetime of a data structure, i.e., its creation, usage and destruction, including degenerate shapes, which occur during data structure operations. Importantly, DSI is still able to detect the stable shape of the DDS by gathering evidence for a data structure during its lifetime. By reinforcing evidence by structural repetition, i.e., multiple parts of a data structure perform the same role as is the case with multiple child elements in a parent child nesting, and temporal repetition, i.e., observing the data structure over its lifetime, less likely interpretations are ruled out. MemPick [69] and DDT [74] avoid degenerate shapes completely, thus missing interesting DDS behaviour during these operations. ARTISTE [49] does not explicitly avoid degenerate shapes, but becomes more conservative with its analysis when it observes them.

For domain (i) above, this dissertation developed certain parts of DSI's theory, such as algorithms to keep DSI's memory abstraction consistent and the quantified description of the interconnections between strands. Both are fundamental for performing DSI's analysis, as without memory consistency the precision of the analysis would be hampered and without the interconnection between strands neither the data structures nor the evidence reinforcement would work. Within this dissertation, the whole DSI theory was implemented and the resulting DSI tool was benchmarked, resulting in a positive answer to our first research question:

> **Research Question 1:** *Is the DSI approach adequate to reach its goals of automatically detecting dynamic data structure shapes with high precision in the presence of degenerate shapes and, in particular, how far must the DSI concept be refined in order to deal with the wealth of dynamic data structure implementations employed in real-world software?*

Subsequently to the first research question, domain (ii) was addressed by opening up DSI for the inspection of C/C++ binaries and to positively answer our second and central research question of this dissertation:

[1] SWT Research Group, University Bamberg, DFG-Project LU 1748/4-1

Research Question 2: *Can DSI's concepts and strengths be preserved when inspecting binaries?*

Among the biggest problem when dealing with binaries is the loss of perfect type information found in source code. This led to the incorporation of the dedicated type recovery tool Howard [98] into DSIbin, the binary version of DSI. This combination already outperformed related work [49, 69, 74] with regards to detectable data structures such as skip lists and certain parent child nestings. However, DSI's full potential could not be exploited due to some limitations of Howard with regards to type merging and nested type detection. These prevent DSI from detecting strands, leading to diminished precision of the analysis. The problem was countered within this dissertation by devising a type refinement algorithm that significantly enhances Howard's excavated types. The algorithm operates by propagating type information along pointer connections, resulting in a multitude of possible type hypotheses. Interestingly, DSI itself is used to choose the best hypothesis by evaluating all of the hypotheses and choosing the most complex DDS found by DSI. This refined type information enables DSI to deal with Linux kernel list (like) DDSs that can run through nodes of different types and which are not covered by related work [49, 69, 74]. Hence, DSIbin can also be seen as an improved version of Howard. Therefore, DSIbin contributes both to the reverse engineering of dynamic data structures as well as type merging, nested type detection and, as a byproduct, primitive type detection. The insight gained is that the higher level dynamic data structure information helps one to infer the lower level information in a top down fashion.

Because our ultimate future work use case for DSIbin is the inspection and possibly even data structure based classification of malware, a survey was conducted on leaked malware source code to verify that current malware actually employs dynamic data structures. That this is actually the case can be seen within the hidden VNC server found in the Carberp malware [34], which is included in Carberp's code base as an off-the-shelf open source component. Additionally, (partial) reuse of identical data structures for DNS caching was present in the MyDoom, Hellbot and Grum malware [95]. These facts underline that malware has matured to plug and play software, where external components are used and also reused within malware families.

Future work. As the specific conclusions of Pt. I and Pt. II of this dissertation already cover topics for future work that are more technical in nature, we now give directions for future work that are more targeted towards the interaction of DSI and DSIbin with the end user, i.e., software developers and reverse engineers.

It would be desirable to quantify the usefulness of DSI by conducting a case study regarding program comprehension of source code. This could cover topics such as the comprehension of the general data structure shape, or the shape of a data structure given a particular input. By letting two groups solve the program comprehension tasks, where only one group is equipped with DSI, differences could be detected such as time required for program comprehension, the amount of correctly identified data structures, and the correct interpretation of data structure behaviour in certain situations.

A second aspect could be to study how DSI can help a software developer who is confronted with conducting code reviews for legacy code. As DSI already provides a notion of DDS complexity, this information could be used to find code complexity hot-spots, i.e., a complex skip list implementation versus a singly linked list. This would be beyond simply reporting the results, but would require to rank the found data structures and to point the software engineer to the actual source code sections that are interfacing with the particular data structure.

Finally, when looking at DSIbin, it would be highly interesting to combine the inferred information, e.g., with those reported by IDA Pro [33], the de-facto industry standard for the reverse engineering of binaries. IDA Pro has its strength in reverse engineering assembly code into high-level functions, collecting primitive type information and detecting compound types. However it does not detect dynamic data structures, therefore leaving space for improving the correlation between DDS changing code and visualizing the corresponding high level DDS. Additionally, the semantic information learned by DSIbin could be overlaid upon the reverse engineered dynamic data structure types, such as labeling a chunk of memory as a “DLL” node and the associated pointers as “next” and “previous”. Current visualization techniques [39, 85] do not offer a consistent view upon the data structure during its lifetime or are not in the focus of the binary analysis [49, 69, 74]. All of these cited approaches do not provide the precision of DSIbin in inspecting each time step of a DDS, including the ones in which DDS are in degenerate shapes, which gives DSIbin the advantage of providing information about a DDS even during DDS operations.

All these future work directions could also be evaluated with regards to DSI’s usability. This would help to design a user interface that integrates DSI best into the work context of software developers, either as a standalone tool or as a plugin for other tools such as IDA Pro or Eclipse.

Bibliography

[1] Booklet for Magic Knight Rayearth (video game published by SEGA). http://segascream.com/wp-content/uploads/2015/06/Magic-Knight-Rayearth-NA.pdf. Page “Translation Notes”. Accessed: 16th June 2017.

[2] CIL mailing list: artificial pointer creation. https://sourceforge.net/p/cil/mailman/message/21313865/. Accessed: 3rd May 2017.

[3] CIL repository. https://sourceforge.net/p/cil/code/ci/master/tree/. Accessed: 31st March 2018.

[4] Custom memory allocation in C# Part 1 – Allocating object on a stack. https://blog.adamfurmanek.pl/2016/04/23/custom-memory-allocation-in-c-part-1/. Accessed: 9th September 2018.

[5] DLL implementation in C from the Rosetta Code website. https://rosettacode.org/wiki/Doubly-linked_list/Definition#C. Accessed: 17th May 2017.

[6] Dr. Memory: Memory Debugger for Windows, Linux, and Mac. http://www.drmemory.org/. Accessed: 14th July 2017.

[7] Futures and Promises. http://docs.scala-lang.org/overviews/core/futures.html. Accessed: 2nd April 2018.

[8] GCC, the GNU Compiler Collection. https://gcc.gnu.org/. Accessed: 16th December 2018.

[9] GNU core utilities. https://www.gnu.org/software/coreutils/coreutils.html. Accessed: 16th June 2017.

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

[11] IKVM.NET Bytecode Compiler. http://www.ikvm.net/userguide/ikvmc.html. Accessed: 16th June 2017.

[12] Java Virtual Machine Specification: The class File Format. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html. Accessed: 2nd April 2018.

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

[14] LibVNCServer/LibVNCClient are cross-platform C libraries that allow you to easily implement VNC server or client functionality in your program. https://libvnc.github.io/. Accessed: 24th December 2017.

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

[16] lwIP - A Lightweight TCP/IP stack. http://savannah.nongnu.org/projects/lwip/. Accessed: 24th December 2017.

[17] Malware statistics by AV-TEST. https://www.av-test.org/en/statistics/malware/. Accessed: 16th June 2017.

[18] Memory leak. https://www.owasp.org/index.php/Memory_leak. Accessed: 16th June 2017.

[19] Mytob is a worm that opens backdoors for crackers. http://virus.wikidot.com/mytob. Accessed: 25th December 2017.

[20] Olden Benchmark v1.01. http://www.martincarlisle.com/olden.html. Accessed: 16th June 2017.

[21] Omega: An Instant Leak Detector Tool for Valgrind. http://www.brainmurders.eclipse.co.uk/omega.html. Accessed: 14th July 2017.

[22] Parasoft Insure++. https://www.parasoft.com/product/insure/. Accessed: 14th July 2017.

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

[24] PurifyPlus: Run-Time Analysis Tools for Application Reliability and Performance. https://teamblue.unicomsi.com/products/purifyplus/. Accessed: 14th July 2017.

[25] Shrink Your Code and Resources. https://developer.android.com/studio/build/shrink-code.html. Accessed: 16th June 2017.

[26] Simon Jeffery admitts SEGA lost source code. http://kotaku.com/5028197/sega-cant-find-the-source-code-for-your-favorite-old-school-arcade-games. Accessed: 16th June 2017.

[27] stackoverflow.com: Memory leak in the following C code. https://stackoverflow.com/questions/7400275/memory-leak-in-the-following-c-code. Accessed: 16th June 2017.

[28] stackoverflow.com: Memory leak linked list. https://stackoverflow.com/questions/43683615/memory-leak-linked-list. Accessed: 16th June 2017.

[29] Stalking the Fanbot. http://blog.trendmicro.com/trendlabs-security-intelligence/stalking-the-fanbot/. Accessed: 25th December 2017.

[30] stl_list.h. http://cs.brown.edu/~jwicks/libstdc%2B%2B/html_user/stl__list_8h-source.html. Accessed: 17th May 2016.

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

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

[33] The IDA Pro disassembler and debuger from Hex-Rays. https://www.hex-rays.com/products/ida/index.shtml. Accessed: 16th June 2017.

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

[35] Valgrind instrumentation framework. http://valgrind.org/. Accessed: 14th July 2017.

[36] W32.Mydoom.BO@mm. https://www.symantec.com/security_response/writeup.jsp?docid=2005-050803-1959-99&tabid=2. Accessed: 25th December 2017.

[37] Wikipedia entry for Magic Knight Rayearth stating source code loss. https://en.wikipedia.org/wiki/Magic_Knight_Rayearth_(video_game). Accessed: 20th June 2017.

[38] X.Org an open source implementation of the X Window System. https://www.x.org/wiki/. Accessed: 16th June 2017.

[39] 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.

[40] H. Arndt, C. Jansen, J.-P. Katoen, C. Matheja, and T. Noll. Let this graph be your witness! In *International Conference on Computer Aided Verification*, pages 3–11. Springer, 2018.

[41] H. Azatchi, Y. Levanoni, H. Paz, and E. Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. *ACM SIGPLAN Notices*, 38(11):269–281, 2003.

[42] M. Bailey, E. Cooke, F. Jahanian, J. Nazario, D. Watson, et al. The internet motion sensor-a distributed blackhole monitoring system. In *NDSS*, 2005.

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

[44] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[45] E. D. Berger, B. G. Zorn, and K. S. McKinley. OOPSLA 2002: Reconsidering Custom Memory Allocation. *SIGPLAN Not.*, 48(4S):46–57, July 2013.

[46] J. Boockmann. Automatic generation of data structure annotations for pointer program verification. Bachelor thesis, U. Bamberg, Germany, October 2016.

[47] A. Braginsky and E. Petrank. Locality-Conscious Lock-Free Linked Lists. In *Distributed Computing and Networking*, volume 6522 of *Lecture Notes in Computer Science*, pages 107–118. 2011.

[48] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[49] 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, Spain, 2012.

[50] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *Usenix security symposium*, page 15, 2011.

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

[52] C. Cadar, D. Dunbar, D. R. Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, volume 8, pages 209–224, 2008.

[53] J.-T. Chan and W. Yang. Advanced obfuscation techniques for java bytecode. *Journal of systems and software*, 71(1-2):1–10, 2004.

[54] 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.

[55] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *ACM SIGPLAN Notices*, volume 42, pages 480–491. ACM, 2007.

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

[57] T. M. Chilimbi and V. Ganapathy. Heapmd: identifying heap-based bugs using anomaly detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 219–228, 2006.

[58] J. Clause and A. Orso. LEAKPOINT: pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 515–524. ACM, 2010.

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

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

[61] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 269–280. IEEE, 2006.

[62] L. Dietz. Multidimensional repetitive pattern discovery for locating data structure operations. Bachelor thesis, U. Bamberg, Germany, April 2015.

[63] B. Dolan-Gavitt. *Understanding and protecting closed-source systems through dynamic analysis*. PhD thesis, Georgia Institute of Technology, 2014.

[64] R. Dolmans and W. Katz. Rp1: Carberp malware analysis, 2013.

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

[66] M. D. Ernst. Static and dynamic analysis: synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[67] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15. ACM, 1996.

[68] M. Guirguis, A. Bestavros, and I. Matta. Exploiting the transients of adaptation for roq attacks on internet resources. In *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, pages 184–195. IEEE, 2004.

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

[70] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *In proc. of the winter 1992 usenix conference*. Citeseer, 1991.

[71] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *ACM SIGPLAN Notices*, volume 38, pages 168–181. ACM, 2003.

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

[73] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*, pages 41–55. Springer, 2011.

[74] 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.

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

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

[77] D. E. Knuth. Structured Programming with go to Statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301, 1974.

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

[79] 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.

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

[81] S. Loncaric. A Survey of Shape Analysis Techniques. *Pattern recognition*, 31(8):983–1001, 1998.

[82] 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. PLDI 2005, pages 190–200. ACM, 2005.

[83] J. Maebe, M. Ronsse, and K. Bosschere. Precise detection of memory leaks. In *International Workshop on Dynamic Analysis*, pages 25–31. IET, 2004.

[84] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):1–12, 2001.

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

[86] 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.

[87] L. McLaughlin. Bot software spreads, causes new worries. *IEEE Distributed Systems Online*, 5(6):1, 2004.

[88] 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.

[89] R. Padhye and K. Sen. TRAVIOLI: A Dynamic Analysis for Detecting Data-Structure Traversals. In *Proceedings of the 39th International Conference on Software Engineering*, pages 473–483. IEEE Press, 2017.

[90] H. Patil and C. N. Fischer. Efficient run-time monitoring using shadow processing. In *AADEBUG*, volume 95, pages 1–14, 1995.

[91] P. Plauger, M. Lee, D. Musser, and A. A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

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

[93] J. Rhee, R. Riley, Z. Lin, X. Jiang, and D. Xu. Data-Centric OS kernel malware characterization. *IEEE Transactions on Information Forensics and Security*, 9(1):72–87, 2014.

[94] T. Rupprecht, X. Chen, D. H. White, J. H. Boockmann, G. Lüttgen, and H. Bos. Dsibin: Identifying dynamic data structures in c/c++ binaries. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 331–341. IEEE Press, 2017.

[95] T. Rupprecht, X. Chen, D. H. White, J. T. Mühlberg, H. Bos, and G. Lüttgen. POSTER: Identifying Dynamic Data Structures in Malware. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1772–1774. ACM, 2016.

[96] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and privacy (SP), 2010 IEEE symposium on*, pages 317–331. IEEE, 2010.

[97] C. Simpson. Internet relay chat. *Teacher Librarian*, 28(1):18, 2000.

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

[99] A. Sotirov. Hotpatching and the rise of third-party patches. In *Black Hat Technical Security Conference, Las Vegas, Nevada*, 2006.

[100] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ACM Sigplan Notices*, volume 39, pages 85–96. ACM, 2004.

[101] 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.

[102] T. Vidas. Volatile memory acquisition via warm boot memory survivability. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1–6. IEEE, 2010.

[103] K. Vorobyov. *Experiments with property driven monitoring of C programs*. PhD thesis, Department of Informatics Bond University Australia, 2015.

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

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

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

[107] 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.

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

[109] Y. Xie and A. Aiken. Context-and path-sensitive memory leak detection. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 115–125. ACM, 2005.

[110] H. Yu, X. Shi, and W. Feng. Leaktracer: Tracing leaks along the way. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 181–190. IEEE, 2015.

[111] Z. Y. Yu Wu, Qi Zhang and J. Li. A Hybrid Parallel Processing for XML Parsing and Schema Validation. In *Proceedings of Balisage: The Markup Conference 2008. Balisage Series on Markup Technologies, Vol. 1*, 2008.

[112] B. Zorn. Comparing mark-and sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 87–98. ACM, 1990.

University
of Bamberg
Press

Reverse engineering of binaries is tedious, yet mandatory when program behavior must be understood while source code is unavailable, e.g., in case of malware. One challenge is the loss of low level type information, i.e., primitive and compound types, which even state-of-the-art type recovery tools, such as Howard, often cannot reconstruct with full accuracy. Further programmers use high level dynamic data structures, e.g., linked lists. Therefore the detection of data structure shapes is important for reverse engineering. A recent approach called Data Structure Investigator (DSI), aims for detecting dynamic pointer based data structures. While DSI is a general approach, a concrete realization for C programs requiring source code is envisioned as type casts and pointer arithmetic will stress test the approach.

This dissertation improves upon the state-of-the-art of shape detection and reverse engineering by

(i) realizing and evaluating the DSI approach, including contributions to DSI's theory, which results in a DSI prototype;

(ii) opening up DSI for C/C++ binaries to extend DSI to reverse engineering, resulting in a DSIbin prototype;

(iii) handling data structures with DSIbin not covered by some related work, e.g., skip lists;

(iv) refining the nesting detection and performing type merging for types excavated by Howard.

ISBN 978-3-86309-718-9

9 783863 097189

www.uni-bamberg.de/ubp/