Motivation

Goal:
- Identifying dynamic data structures in pointer programs, especially for C programs and x86 binaries

Envisaged applications, targeting user-/kernel-space software:
- Heap visualization
- Program comprehension
- Reverse engineering
- Program verification

Challenges:
- **Fine-grained heaps** due to, e.g., pointer arithmetic, type casts, and customized memory allocators
- **Degenerate shapes** during data structure operations
ARTISTE (Caballero et al., 2012), MemPick (Haller et al., 2016):

- Analyze x86 binaries for reverse engineering and security applications
- Do not deal with fine-grained heaps (see, e.g., the Linux CDLL)
- Either address degenerate shapes (MemPick, detection of quiescent periods) or avoid them (ARTISTE, but does not address nesting)

Heapviz (Aftandilian et al., 2010), HeapDbg (Marron et al., 2013):

- Visualize entire runtime heaps of Java byte code and, resp., C#
- Fold points-to graphs and layout them based on graph dominators
- Do not deal with fine-grained heaps
- Infer shapes superficially (HeapDbg, works not unlike MemPick)
A dynamic analysis inspired by ‘heap abstraction’ and ‘pattern matching’

**Scope:**
- Singly- and doubly-linked lists (cyclic, with head and/or tail pointers)
- Skip lists and (binary) trees
- Combinations of such data structures via nesting (indirect, overlay)

**Contributions:**
- Memory abstraction, **strand graphs**, to handle fine-grained heaps
  - Lifts the assumption that each memory chunk contains a single node
  - Tracks **cells**, i.e., individual structs, instead of whole nodes
A dynamic analysis inspired by ‘heap abstraction’ and ‘pattern matching’

Scope:
- Singly- and doubly-linked lists (cyclic, w/ head and/or tail pointers)
- Skip lists and (binary) trees
- Combinations of such data structures via nesting (indirect, overlay)

Contributions:
- Memory abstraction, strand graphs, to handle fine-grained heaps
  - Lifts the assumption that each memory chunk contains a single node
  - Tracks cells, i.e., individual structs, instead of whole nodes
- Novel evidence-based approach that is robust wrt. degenerate shapes
  - Discovering evidence for each kind of data structure
  - Reinforcing evidence by structural and temporal repetition
The DSI Tool Suite

Front ends
- C source file
  - DSIsrc
- x86 binary file
  - DSImbin
  - Howard
  - DSImref
  - DSIref

Offline analysis
- DSImcore

Back ends
- Naming
- Operation detection
- Visualization
- Verification
Input:
- Sequence of points-to graphs generated from a recorded event trace (pointer-write events, de/allocation events)

Output:
- Aggregated strand graph (decorated with evidence)

Workings illustrated using a simple example . . .
Points-to Graph

Points-to graph – abstraction of program memory:
- Memory chunks (vertices)
- Pointers (edges)
- Entry points (ep)
Illustrative Example

Strands – atomic building blocks
- Essentially SLLs

Points-to Graph
**Illustrative Example**

Points-to Graph

**Strand connection**
- Relationship between a pair of strands
- Count evidence for each data structure type

**Overlay (bi-directional) connection**
- E.g., DLL
- Why evidence = 6?
Illustrative Example

Points-to Graph

Indirect (uni-directional) connection
- E.g., nesting (N)

Overlay (bi-directional) connection
- E.g., DLL

E(DLL)=6
E(N)=1
Illustrative Example

Points-to Graph

Strand Graph

Structural repetition
- Merging strands and strand connections
Illustrative Example

Points-to Graph

Strand Graph

Folded Strand Graph

$E(DLL) = 6$

$E(N) = 1$

$E(IL) = 2$

$E(N) = 2$

$E(DLL) = 6$

$E(N) = 2$

$E(IL) = 2$
Illustrative Example

Time t

Points-to Graph

Folded Strand Graph
Illustrative Example

Time t

Points-to Graph

Folded Strand Graph

Time t+1

DLL insert operation complete
Illustrative Example

Points-to Graph

Folded Strand Graph

Temporal repetition

E(N) = 2
E(DLL) = 6
E(IL) = 2

E(N) = 4
E(DLL) = 6 + 9 = 15
E(IL) = 0
Illustrative Example

Time t

Points-to Graph

Time t + 1

Folded Strand Graph

Aggregated Strand Graph
Example (libusb)

Usb device library providing access from user-space (7K LOC C):

- Uses a parent CDDL with two child CDLLs (one for devices, one for file descriptors)
- Employs Linux CDDLs that are embedded in surrounding structs, i.e., our cell-based memory abstraction is required

Aggregated strand graph:
**Taxonomy & Matching of Data Structures**

- **Area Condition**
  - Same Head Node (SHN)
  - Intersecting on 1 Node - Overlay (I1O)
  - Nesting - Overlay (NO)
  - Tail Pointer - Overlay (TPO)
  - DLLSkip List - Overlay1 (SLO1)
  - Cyclic DLL (CDLL)
  - Intersecting on 2+ Nodes - Overlay (I2+O)
  - Head Pointers - Overlay (HPO)
  - Intersecting on 2+ Nodes - Indirect (I2+) (I1I)
  - Nesting - Indirect (NI)
  - Tail Pointers - Indirect (TPI)
  - Sharing (S)
  - Binary Tree (BT)
  - Cyclic SLL (CSLL)

- **Area Condition**
  - Skip List - Overlay2 (SLO2)
  - Head Pointers - Indirect (HPI)
  - Tail Pointers - Indirect (TPI)
  - Area Condition
  - SLL
  - Cyclic SLL (CSLL)
  - A
  - A
  - B
  - B
  - C
  - C
  - D
  - E
  - E
  - F
  - F
  - G
  - G
  - H
  - H
  - I
  - I
  - J
  - J
  - K
  - K
  - L
  - L
  - M
  - M
  - N
  - N
  - O
  - O
  - P
  - P
  - Q
  - Q
  - R
  - R
  - S
  - S
  - T
  - T
  - U
  - U
  - V
  - V
  - W
  - W
  - X
  - X
  - Y
  - Y
  - Z
  - Z
**Taxonomy & Matching of Data Structures**

\[
\text{DLL.areaCondition}(C_{\text{init}}, \rightarrow, \leftarrow, \rightarrow) =
\begin{cases}
\text{if } C_{\text{init}} = S_1 \xrightarrow{w} S_2 \text{ then ret } \{C_{\text{init}}\} \text{ else ret } \emptyset \\
\end{cases}
\]

\[
\text{DLL.shapePredicate} = S_1^\sigma = \emptyset \land S_2^\sigma = \emptyset \land \\
\forall i \in [0..\text{LENGTH}(S_1) - 1] \ \exists (c_1, c_2) \in \text{PAIRS}(S_1 \xleftarrow{w} S_2) : \\
S_1^\sigma[i + 1] = c_1 \land S_2^\sigma[\text{LENGTH}(S_2) - i] = c_2
\]

\[
\text{DLL.assignEvidence}(G^S) = \\
S_1 \xleftarrow{w} S_2 \text{ (in } G^S) \leftarrow \text{“DLL” : } |\text{PAIRS}(S_1 \leftrightarrow S_2)| \ast 3
\]
Front end for analyzing C source code:

- Instrumentation via CIL
  (1K LOC OCaml and 600 LOC C)
Front end for analyzing x86 binaries:

- Instrumentation via Intel PIN (3K LOC C++)
DSIbin: Recovering Type Information

DSIcore relies on type information for tracking strands

Howard is a modern type excavator:

- Powerful for primitive types and plain structs, employing advanced techniques for type merging
- Limited for nested structs, e.g., does not recognize that a nested struct is placed at the head of a surrounding struct (see, e.g., Linux CDLL example)
Novel method for type refinement, which obtains high-level data structure information from DSICore in order to infer low-level type information

Input: Incomplete type information inferred by Howard

Output: Refined/completed type information

Workings in a nutshell:

- Make educated guesses on the missing types within structs (resulting in type hypotheses)
- Evaluate the (many) hypotheses with DSICore wrt. their plausibility
- Choose the structurally most complex data structure
Example (from Predator/Forester git repository)

CDLL parent with two CDLL children (same as libusb):

(white nodes are typed by Howard, grey nodes are *only* typed by DSImref)

Howard does not detect the parent nested structs and the external head struct to be of same type, thus:

- Cyclicity property of the DLLs is missed
- Overlay nesting is missed (‘nesting on indirection’ reported instead)
Naming Back End

Front ends

Offline analysis

Back ends

C source file

x86 binary file

Howard DSIref

Naming

Operation detection

Visualization

Verification

Aggregated Strand Graph

Identified Data Structure

E(N)=2

E(DLL)=21

E(IL)=2

SLL

DLL

SLL

SLL

SLL with Nested DLLs
Visualization Back End

Front ends
- C source file
- x86 binary file
- Howard
- DSIref

Back ends
- Naming
- Operation detection
- Visualization
- Verification

Flow:
1. C source file → DSIsrc
2. DSIsrc → DSIname
3. DSIname → Verification
4. Verification → Operation detection
5. Operation detection → Visualization
6. Visualization → Back ends
7. Back ends → x86 binary file
Conclusions

DSI is a novel dynamic analysis for identifying dynamic data structures in pointer programs:

- Introduces **strand abstraction** and **evidence-based identification** (using a sophisticated taxonomy)
- Robustly identifies data structures despite degenerate shapes (shown via extensive benchmarking)
- Copes with the ‘dirtiness’ / ‘efficiencies’ of (low-level) C code
- Is applicable to x86 binaries, thanks to advanced type refinement
- Requires work on speeding up the analysis, e.g., via parallelization

**DSI has potential for a broad range of applications:**

- Heap visualization, program comprehension
- Reverse engineering, malware identification
- Program verification
Future Work

- Improve the runtime of DSI, e.g., via parallelization
- Investigate relation to static shape analysis tools such as Predator (Dudka et al., 2013) and Forester (Holík et al., 2013), which focus on program verification rather than program comprehension

Three-year duration follow-up DFG project, granted last month:

- Cover richer data structures, e.g., hash maps (arrays), sorted lists, and balanced trees
- Integrate DSI with IDA Pro (disassembler and debugger) to improve the debugging of malware
- Interface DSI to VeriFast (theorem prover based on separation logic) to auto-generate verification conditions for verifying C programs

P. S.: We are looking for a PhD student or PostDoc to join our team in the world-heritage town of Bamberg!

DSI is open source: https://github.com/uniba-swt
Backup Slides
Further DSI Publications


Publications on Related Tools


ARTISTE (Caballero et al., 2012) for signature generation (security applications):

- Examines memory snapshots taken at regular intervals
- Excavates types and constructs points-to graphs
- Merges nodes into region graphs (kind of structural repetition, leading to a folded points-to graph)
- Uses shape predicates to identify data structures

→ Does not employ a fine-granular heap abstraction (unlike DS, see Linux CDLL example)
→ Does not address degenerate shapes (unlike DSI)
→ Does not deal with data structure relationships such as nesting (unlike DSI)
→ Does not consider temporal repetition (unlike DSI)
MemPick (Haller, Slowinska & Bos, 2015) for reverse engineering (security applications):

- Examines memory snapshots taken during quiescent periods
- Excavates types, constructs points-to graphs, and analyses points-to graphs wrt. overlays and simple, local shape invariants
- Identifies data structures according to a tailored decision tree

→ Can deal with nested and ‘mixed’ data structures (e.g., linked lists running through trees, as used in topological sorting)
→ Does not employ a fine-granular heap abstraction (unlike DSI, see Linux CDLL example)
→ Is sensitive to inaccuracies in stable shape identification (unlike DSI)
DDT (Jung & Clark, 2009) for data structure shape and operation identification:

- Examines multiple program executions to heuristically detect interface functions for data structures
- Observes these functions’ properties, e.g., node insertions/deletions
- Considers simple, local shape invariants (similar to MemPick)
- Identifies data structure operations based on a tailored decision tree

→ Works well for code that uses standard libraries (e.g., C++ STL)
→ Does not employ a fine-granular heap abstraction (unlike DSI, see Linux CDLL example)
→ Cannot detect relationships such as nesting (unlike DSI)
Related Work: Data Structure Visualization

Heapviz (Aftandilian et al.):

- Dynamic analysis on byte code for visualizing and animating entire runtime heaps
- Folding (or ‘merging’) of points-to graphs similar to DSI’s structural repetition
- Dominator-based layouting scheme that deals with nesting

HeapDbg (Marron et al.):

- Similar to Heapviz, but HeapDbg employs abstract interpretation techniques for merging
- In contrast to Heapviz, HeapDbg also infers shapes with which the visualized graphs are annotated
Related Work: Static Shape Analysis

Predator (Dudka, Peringer & Vojnar) & Forester (Kolik et al.):

- Focus on program verification (memory safety) rather than program comprehension
- Execute C programs symbolically to infer shape predicates
- Employ symbolic memory graphs and, resp., forest automata

Questions:

- How similar are Predator/Forester's inferred shape predicates to those that could be generated by DSI?
- How do the assumptions underlying Predator/Forester compare to those of DSI?
DSI Challenges: C Heaps

Nested structs:

Custom memory allocators:
A data structure might temporarily lose its stable shape due to manipulation operations.

Time $t$:

Time $t+1$:

or?
Details of (a) a strand $S$ with linkage condition $S^{LC} = (\tau, o)$ and both linear and cyclic cell sequences, (b) an overlay SC $S_1 \leftrightarrow S_2$ and (c) an indirect SC $S_1 \rightarrow S_2$. Memory chunks have black outline, cells are dashed, and strands are indicated with block arrows.
(a) Create sequence of points-to-graphs from program execution (only one shown)

(b) Construct merged type graph capturing pointer connections between types

(c) Exploit pointer connections by mapping type sub-regions (two possibilities shown)
(d) Observe that multiple interpretations may be possible

(e) Propagate each interpretation along pointer connections

(f) Rule out inconsistencies
(g) Evaluate remaining interpretations via DSI

(h) Choose the ‘best’ interpretation in terms of data structure complexity (indicated by merged type graph with resulting label 1xCSLL & 1xSLL)
# DSIsrc Evaluation I

<table>
<thead>
<tr>
<th>ID</th>
<th>Naming Module</th>
<th>Evidence Counts</th>
<th>% Supporting Evidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>tb1</td>
<td>SLL</td>
<td>None since no SCs present</td>
<td></td>
</tr>
<tr>
<td>tb2</td>
<td>DLL</td>
<td>DLL: 1440, I2+O: 220</td>
<td>87%</td>
</tr>
<tr>
<td>syn1</td>
<td>CDLL</td>
<td>CDLL: 15, I2+O: 10, DLL: 6</td>
<td>48%</td>
</tr>
<tr>
<td>syn2</td>
<td>Binary Tree</td>
<td>BT: 248, N_O: 6, I1_O: 3</td>
<td>97%</td>
</tr>
<tr>
<td>syn3</td>
<td>SLL + nest. SLL</td>
<td>N_O: 6, I1_O: 2</td>
<td>75%</td>
</tr>
<tr>
<td>syn4</td>
<td>SLL + nest. SLL</td>
<td>N_O: 10, SHN: 6</td>
<td>63%</td>
</tr>
<tr>
<td>syn5</td>
<td>DLL + nest. DLL</td>
<td></td>
<td>Avg. 84%</td>
</tr>
</tbody>
</table>

*(tb – textbook, syn – synthetic)*
DSIsrc Evaluation II

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

- **SL**\(_{\text{horiz}}\)
- **SL**\(_{\text{o2}}\) 1188, N\(_o\): 39, BT: 24
- **SL**\(_{\text{vert}}\)
- N\(_l\): 35, I1\(_l\): 15
- I1\(_l\): 35
- Child DLL\(_{\text{fwd}}\)
- DLL 345, I2+O: 2
- Child DLL\(_{\text{rev}}\)

**lit1** DLL of (DLL, DLL) OR Intersecting(2xDLL) Avg. 96%

- DLL\(_1\)\(_{\text{fwd}}\)
- DLL 1818, I2+O: 108, SHN: 9
- DLL 1623, I2+O: 162, SHN: 9
- DLL\(_1\)\(_{\text{rev}}\)
- DLL 1980, I2+O: 54, SHN: 9
- DLL 1980, I2+O: 54, SHN: 9
- DLL\(_2\)\(_{\text{fwd}}\)
- DLL 1785, I2+O: 108, SHN: 9
- DLL\(_2\)\(_{\text{rev}}\)
- I2+O: 725, SHN: 9
- I2+O: 725, SHN: 9
- I2+O: 703, SHN: 9

*(syn – synthetic, lit – literature)*
DSIsr Evaluation III

lit2  
Skip List  
$\text{SL}_{02} : 1242, \ BT : 72, \ NO : 58, \ 90\%$
$I1o : 8, \ SHN : 3$

lit3  
SLL + nest. Intersecting(2x CDLL)  
Avg. 88%

lit4  
SLL + nest. SLL + nest. SLL + nest. SLL  
Avg. 85%

(lit – literature)
DSIsrc Evaluation IV

<table>
<thead>
<tr>
<th>Command</th>
<th>Type</th>
<th>BT</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>bash</td>
<td>CDLL</td>
<td>68,529</td>
<td>$\sim 100%$</td>
</tr>
<tr>
<td>treeadd</td>
<td>Binary tree</td>
<td>256</td>
<td>100%</td>
</tr>
<tr>
<td>treebnnh</td>
<td>Binary tree</td>
<td>930</td>
<td>100%</td>
</tr>
<tr>
<td>libusb</td>
<td>CDLL + nest. Intersecting(2x CDLL)</td>
<td>Avg. 88%</td>
<td></td>
</tr>
</tbody>
</table>

(real-world examples)
Noteworthy Observations wrt. DSIscc’s Evaluation

General observations:

- 13 of the 18 memory structures of our taxonomy are exercised
- Evidence for stable shapes builds up quickly in all examples
- Correct observation is always in the majority

Example-specific observations:

- libusb (7K LOC C) and lit3 are structurally most challenging
- syn1 represents the worst case for DSIsccore (short trace, many instances of degenerate shapes)
- lit1 is the only example with an ambiguous reading, revealing a non-obvious memory structure interpretation
  - The forward strand of one DLL can be matched with either DLL’s reverse strand
  - This highlights the utility of aggregated strand graphs for analyzing memory structures
### Refinement Result for VNC Example

#### Source
```c
struct sraSpan {
  struct sraSpan *next;
  struct sraSpan *prev;
  int start;
  int end;
  struct sraRegion *subspan;
};
```

#### Howard
```c
struct {
  0x0: VOID*;
  0x8: VOID*;
  0x10: INT32;
  0x14: INT32;
  0x18: VOID*;
}
```

#### DSIRef
```c
struct { 0x0: VOID*; 0x8: VOID*; 0x10: INT32; 0x14: INT32; 0x18: VOID*; };
```

#### Diagram
```
\begin{tikzpicture}
  \node (sraRegion) {sraRegion};
  \node (sraSpan) [above of=sraRegion] {sraSpan};
  \node (front) [below of=sraRegion] {front};
  \node (back) [below of=front] {back};
  \draw [->] (sraRegion) -- (sraSpan);
  \draw [->] (sraSpan) -- (front);
  \draw [->] (front) -- (back);
  \draw [->] (sraRegion) -- (back);
\end{tikzpicture}
```
## DSIsbin Evaluation

### Code (ground truth)

<table>
<thead>
<tr>
<th>expl</th>
<th>lang</th>
<th>DS</th>
<th>flat</th>
<th>n@h</th>
<th>n</th>
<th>m</th>
<th>h/s</th>
<th>n-d</th>
<th>m</th>
<th>DSIbin</th>
<th>rec</th>
<th>n-d</th>
<th>m</th>
<th>DSIbin</th>
<th>rec</th>
<th>n-d</th>
<th>m</th>
<th>h/s</th>
<th>pr</th>
<th>ch hyp</th>
</tr>
</thead>
<tbody>
<tr>
<td>er-1</td>
<td>C++</td>
<td>CDLL</td>
<td>n y n n y n</td>
<td>DLL</td>
<td>n n y</td>
<td>CDLL</td>
<td>y y n</td>
<td>- y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>er-2</td>
<td>C++</td>
<td>CDLL</td>
<td>n y y (p) y n</td>
<td>DLL</td>
<td>n y (p) y</td>
<td>CDLL</td>
<td>y y n</td>
<td>(p) - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lit-1</td>
<td>C++</td>
<td>SLL (− Ni → SLL)x4</td>
<td>n n n n n</td>
<td>SLL (− Ni → SLL)x4</td>
<td>y n n</td>
<td>SLLs with Ni/No</td>
<td>n n n n - n n</td>
<td>(DSIRef)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lit-2</td>
<td>C++</td>
<td>SLo</td>
<td>n n y y y</td>
<td>DLL</td>
<td>n y y</td>
<td>CDLL</td>
<td>y y y</td>
<td>- y n n</td>
<td>Howard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lit-3</td>
<td>C++</td>
<td>SLL (→ 2xNi → DLL)</td>
<td>n y y n n</td>
<td>DLL (3, no nesting)</td>
<td>n y y</td>
<td>DLL (5, with hint on No)</td>
<td>n y y - y n y</td>
<td>(DSIRef)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lit-4</td>
<td>C++</td>
<td>SLL (→ 2xNo → CDLL,CDLL)</td>
<td>n y y n n</td>
<td>DLL (3) + noise</td>
<td>n n y</td>
<td>DLL (10) + I2o+</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-01</td>
<td>C++</td>
<td>SLL (5) [64 / 728]</td>
<td>n y y y y</td>
<td>SLL (2)</td>
<td>n n y</td>
<td>SLL (5)</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-02</td>
<td>C++</td>
<td>SLL (11) [72 / 873]</td>
<td>y (pt) y y n n</td>
<td>nothing</td>
<td>n y (pt)</td>
<td>SLL (11)</td>
<td>y y y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-03</td>
<td>C++</td>
<td>SLL (12) [241 / 1329]</td>
<td>n n y o y</td>
<td>DLL</td>
<td>n y -</td>
<td>CDLL</td>
<td>y n y - y y n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-04</td>
<td>C++</td>
<td>SLL (10) + I2o+ [111 / 3807]</td>
<td>y n y y n</td>
<td>DLL (3) + noise</td>
<td>n n y</td>
<td>DLL (10) + I2o+</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-05</td>
<td>C++</td>
<td>SLL (5) [64 / 728]</td>
<td>n y y y y</td>
<td>SLL (2)</td>
<td>n n y</td>
<td>SLL (5)</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-06</td>
<td>C++</td>
<td>SLL (11) [72 / 873]</td>
<td>y (pt) y y n n</td>
<td>nothing</td>
<td>n y (pt)</td>
<td>SLL (11)</td>
<td>y y y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-07</td>
<td>C++</td>
<td>SLL (12) [241 / 1329]</td>
<td>n n y o y</td>
<td>DLL</td>
<td>n y -</td>
<td>CDLL</td>
<td>y n y - y y n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-08</td>
<td>C++</td>
<td>SLL (10) + I2o+ [111 / 3807]</td>
<td>y n y y n</td>
<td>DLL (3) + noise</td>
<td>n n y</td>
<td>DLL (10) + I2o+</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-09</td>
<td>C++</td>
<td>SLL (5) [64 / 728]</td>
<td>n y y y y</td>
<td>SLL (2)</td>
<td>n n y</td>
<td>SLL (5)</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-10</td>
<td>C++</td>
<td>SLL (11) [72 / 873]</td>
<td>y (pt) y y n n</td>
<td>nothing</td>
<td>n y (pt)</td>
<td>SLL (11)</td>
<td>y y y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-11</td>
<td>C++</td>
<td>SLL (12) [241 / 1329]</td>
<td>n n y o y</td>
<td>DLL</td>
<td>n y -</td>
<td>CDLL</td>
<td>y n y - y y n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-12</td>
<td>C++</td>
<td>SLL (10) + I2o+ [111 / 3807]</td>
<td>y n y y n</td>
<td>DLL (3) + noise</td>
<td>n n y</td>
<td>DLL (10) + I2o+</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-13</td>
<td>C++</td>
<td>SLL (5) [64 / 728]</td>
<td>n y y y y</td>
<td>SLL (2)</td>
<td>n n y</td>
<td>SLL (5)</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-14</td>
<td>C++</td>
<td>SLL (11) [72 / 873]</td>
<td>y (pt) y y n n</td>
<td>nothing</td>
<td>n y (pt)</td>
<td>SLL (11)</td>
<td>y y y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-15</td>
<td>C++</td>
<td>SLL (12) [241 / 1329]</td>
<td>n n y o y</td>
<td>DLL</td>
<td>n y -</td>
<td>CDLL</td>
<td>y n y - y y n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-16</td>
<td>C++</td>
<td>SLL (10) + I2o+ [111 / 3807]</td>
<td>y n y y n</td>
<td>DLL (3) + noise</td>
<td>n n y</td>
<td>DLL (10) + I2o+</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-17</td>
<td>C++</td>
<td>SLL (5) [64 / 728]</td>
<td>n y y y y</td>
<td>SLL (2)</td>
<td>n n y</td>
<td>SLL (5)</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-18</td>
<td>C++</td>
<td>SLL (11) [72 / 873]</td>
<td>y (pt) y y n n</td>
<td>nothing</td>
<td>n y (pt)</td>
<td>SLL (11)</td>
<td>y y y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-19</td>
<td>C++</td>
<td>SLL (12) [241 / 1329]</td>
<td>n n y o y</td>
<td>DLL</td>
<td>n y -</td>
<td>CDLL</td>
<td>y n y - y y n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-20</td>
<td>C++</td>
<td>SLL (10) + I2o+ [111 / 3807]</td>
<td>y n y y n</td>
<td>DLL (3) + noise</td>
<td>n n y</td>
<td>DLL (10) + I2o+</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-21</td>
<td>C++</td>
<td>SLL (5) [64 / 728]</td>
<td>n y y y y</td>
<td>SLL (2)</td>
<td>n n y</td>
<td>SLL (5)</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-22</td>
<td>C++</td>
<td>SLL (11) [72 / 873]</td>
<td>y (pt) y y n n</td>
<td>nothing</td>
<td>n y (pt)</td>
<td>SLL (11)</td>
<td>y y y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-23</td>
<td>C++</td>
<td>SLL (12) [241 / 1329]</td>
<td>n n y o y</td>
<td>DLL</td>
<td>n y -</td>
<td>CDLL</td>
<td>y n y - y y n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-24</td>
<td>C++</td>
<td>SLL (10) + I2o+ [111 / 3807]</td>
<td>y n y y n</td>
<td>DLL (3) + noise</td>
<td>n n y</td>
<td>DLL (10) + I2o+</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-25</td>
<td>C++</td>
<td>SLL (5) [64 / 728]</td>
<td>n y y y y</td>
<td>SLL (2)</td>
<td>n n y</td>
<td>SLL (5)</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-26</td>
<td>C++</td>
<td>SLL (11) [72 / 873]</td>
<td>y (pt) y y n n</td>
<td>nothing</td>
<td>n y (pt)</td>
<td>SLL (11)</td>
<td>y y y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-27</td>
<td>C++</td>
<td>SLL (12) [241 / 1329]</td>
<td>n n y o y</td>
<td>DLL</td>
<td>n y -</td>
<td>CDLL</td>
<td>y n y - y y n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-28</td>
<td>C++</td>
<td>SLL (10) + I2o+ [111 / 3807]</td>
<td>y n y y n</td>
<td>DLL (3) + noise</td>
<td>n n y</td>
<td>DLL (10) + I2o+</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-29</td>
<td>C++</td>
<td>SLL (5) [64 / 728]</td>
<td>n y y y y</td>
<td>SLL (2)</td>
<td>n n y</td>
<td>SLL (5)</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-30</td>
<td>C++</td>
<td>SLL (11) [72 / 873]</td>
<td>y (pt) y y n n</td>
<td>nothing</td>
<td>n y (pt)</td>
<td>SLL (11)</td>
<td>y y y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-31</td>
<td>C++</td>
<td>SLL (12) [241 / 1329]</td>
<td>n n y o y</td>
<td>DLL</td>
<td>n y -</td>
<td>CDLL</td>
<td>y n y - y y n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-32</td>
<td>C++</td>
<td>SLL (10) + I2o+ [111 / 3807]</td>
<td>y n y y n</td>
<td>DLL (3) + noise</td>
<td>n n y</td>
<td>DLL (10) + I2o+</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-33</td>
<td>C++</td>
<td>SLL (5) [64 / 728]</td>
<td>n y y y y</td>
<td>SLL (2)</td>
<td>n n y</td>
<td>SLL (5)</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-34</td>
<td>C++</td>
<td>SLL (11) [72 / 873]</td>
<td>y (pt) y y n n</td>
<td>nothing</td>
<td>n y (pt)</td>
<td>SLL (11)</td>
<td>y y y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-35</td>
<td>C++</td>
<td>SLL (12) [241 / 1329]</td>
<td>n n y o y</td>
<td>DLL</td>
<td>n y -</td>
<td>CDLL</td>
<td>y n y - y y n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>syn-36</td>
<td>C++</td>
<td>SLL (10) + I2o+ [111 / 3807]</td>
<td>y n y y n</td>
<td>DLL (3) + noise</td>
<td>n n y</td>
<td>DLL (10) + I2o+</td>
<td>y n y - y n n</td>
<td>DSIref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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.
Noteworthy Observations wrt. DSIbin’s Evaluation

General observations:
- 30 examples (up to 5.5K events, 156 generated type hypotheses)
  - Execution time in the order of (tens of) minutes due to DSIcore
- DSIref is key for analyzing binaries
  - Naive Howard-DSIcore pipeline correctly identifies 10 of 30 examples
  - Howard-DSIref-DSIcore correctly identifies 26 of 30 examples

Example-specific observations:
- syn-08 shows flattened access of nested head node of child DLL inside parent DLL, so that Howard misses the child list’s head
  - DSIcore then recognizes indirect nesting rather than overlay nesting
- lit-2 consists of multi-level SLLs where all types are different but binary compatible, so that Howard wrongly merges types
  - DSIcore then detects some overlay nesting as indirect nesting
- r-3 leads DSIcore to misinterpret indirection as strands of length 2
  - DSIcore should wait until strands have further evolved, before folding
typedef struct Node *Stack;

struct Node {
    ElementType Element;
    struct Node *Next;
};

/*@ predicate SLL_Nodes_Node(struct Node *node, int count) =
node == 0 ? count==0 : 0<count/ 
&*& node->Element |-> _
&*& node->Next |-> ?next/
&*& malloc_block_Node(node)/
&*& Nodes_0(next, count-1); */

void push(ElementType X, Stack S)
/*@ requires SLL_Node(S, ?n_0);
/*@ ensures SLL_Node(S, n_0+1); */
{
    Stack TmpCell;
    TmpCell = malloc(
        sizeof(struct Node));
    if( TmpCell == NULL ) {
        printf( "Out of space!" );
        exit(EXIT_FAILURE);
    }
    else {
        //@ open SLL_Node(S, n_0);
        TmpCell->Element = X;
        TmpCell->Next = S->Next;
        S->Next = TmpCell;
        //@ close SLL_Nodes_Node(S->Next, <-> n_0+1);
        //@ close SLL_Node(S, n_0+1);
    }
}

<table>
<thead>
<tr>
<th>tid</th>
<th>Tid of graph vertex $v \in V^T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ttype</td>
<td>Template graph vertex $v \in V^T$ with attributes $v.tid$ &amp; $v.ttype$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cid</th>
<th>Cid of graph vertex $v \in V^P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctype</td>
<td>Points-to graph vertex $v \in V^P$ with attributes $v.cid$ &amp; $v.ctype$</td>
</tr>
</tbody>
</table>
Operation Detection Back End

(Figure shows prototype, taken from DSI’s predecessor tool dsOli)