Byte-precise Verification of Low-level List Manipulation

Kamil Dudka[1,2] Petr Peringer[1] Tomáš Vojnar[1]

[1] FIT, Brno University of Technology, Czech Republic

[2] Red Hat Czech, Brno, Czech Republic

June 21, 2013

Agenda

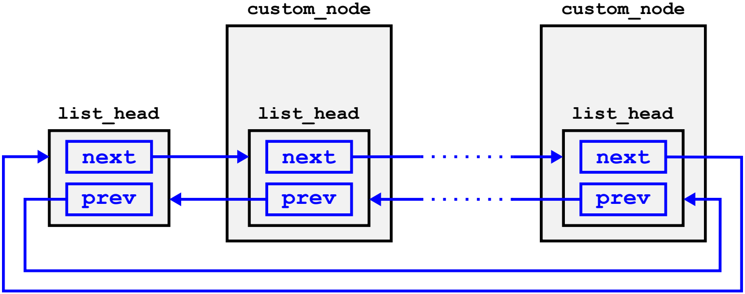
1. Low-level Memory Manipulation

2. Symbolic Memory Graphs (SMGs)

3. Predator – Verifier Based on SMGs

Kernel-Style Linked Lists

- Cyclic, linked through pointers pointing inside list nodes.
- Pointer arithmetic used to get to the boundary of the nodes.
- Non-uniform: one node is missing the custom envelope.



```
struct list_head {
    struct list_head *next;
    struct list_head *prev;
};
```

```
struct custom_node {
    t_data data;
    struct list_head head;
};
```

Kernel-Style Linked Lists – Traversal

- ... as seen by the programmer:

```
list_for_each_entry(pos, list, head) {
    printf(" %d", pos->value);
}
```

Kernel-Style Linked Lists – Traversal

- ... as seen by the programmer:

```
list_for_each_entry(pos, list, head) {
    printf(" %d", pos->value);
}
```

- ... as seen by the compiler:

```
for(pos = ((typeof(*pos) *)((char *)(list->next)
  -(unsigned long)(&((typeof(*pos) *)0)->head)));
  &pos->head != list;
  pos = ((typeof(*pos) *)((char *)(pos->head.next)
  -(unsigned long)(&((typeof(*pos) *)0)->head)))) {
    printf(" %d", pos->value);
}
```

Kernel-Style Linked Lists – Traversal

- ... as seen by the programmer:

```
list_for_each_entry(pos, list, head) {
    printf(" %d", pos->value);
}
```

- ... as seen by the compiler:

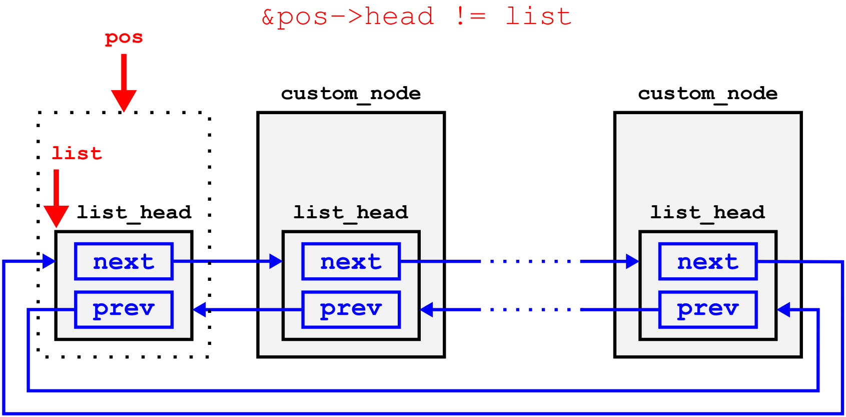
```
for(pos = ((typeof(*pos) *)((char *)(list->next)
  -(unsigned long)(&((typeof(*pos) *)0)->head)));
  &pos->head != list;
  pos = ((typeof(*pos) *)((char *)(pos->head.next)
  -(unsigned long)(&((typeof(*pos) *)0)->head)))) {
    printf(" %d", pos->value);
}
```

- ... as seen by the analyser (assuming 64 bit addressing):

```
for(pos = (char *)list->next - 8;
    &pos->head != list;
    pos = (char *)pos->head.next - 8)
{
    printf(" %d", pos->value);
}
```

Kernel-Style Linked Lists – End of the Traversal

- Correct use of pointers with invalid target:

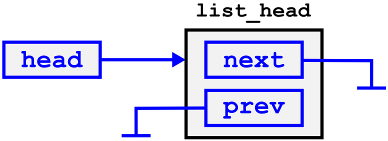


Low-level Memory Manipulation

- We need to track sizes of allocated blocks.
- Large chunks of memory are often nullified at once, their fields are gradually used, the rest must stay null.

```
struct list_head {
     struct list_head *next;
     struct list_head *prev;
};

struct list_head *head = calloc(1U, sizeof *head);
```

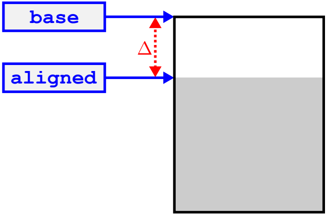


- Low-level code often uses block operations: `memcpy()`, `memmove()`, `memset()`, `strcpy()`.
- Incorrect use of such operations can lead to nasty errors (e.g. `memcpy()` and overlapping blocks).

Alignment of Pointers

- Alignment of pointers implies a need to deal with pointers whose target is given by an interval of addresses:

```
aligned = ((unsigned)base + mask) & ~mask;
```



$of_1 = 0$

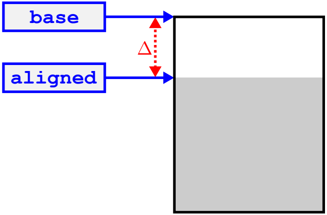
$of_2 = \text{mask}$

$\Delta \in \langle of_1, of_2 \rangle$

Alignment of Pointers

- Alignment of pointers implies a need to deal with pointers whose target is given by an interval of addresses:

```
aligned = ((unsigned)base + mask) & ~mask;
```

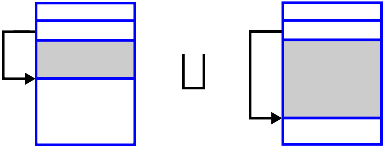


$of_1 = 0$

$of_2 = \text{mask}$

$\Delta \in \langle of_1, of_2 \rangle$

- Intervals of addresses arise also when joining blocks of memory pointing to themselves with different offsets:



Data Reinterpretation

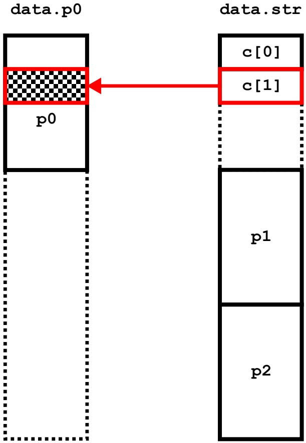
- Due to unions, typecasting, or block operations, the same memory contents can be interpreted in different ways.

```
union {
    void *p0;
    struct {
        char c[2];
        void *p1;
        void *p2;
    } str;
} data;

// allocate 37B on heap
data.p0 = malloc(37U);

// introduce a memory leak
data.str.c[1] = sizeof data.str.p1;

// invalid free()
free(data.p0);
```

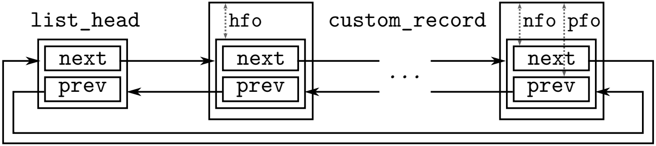


Agenda

1. Low-level Memory Manipulation
2. Symbolic Memory Graphs (SMGs)
3. Predator – Verifier Based on SMGs

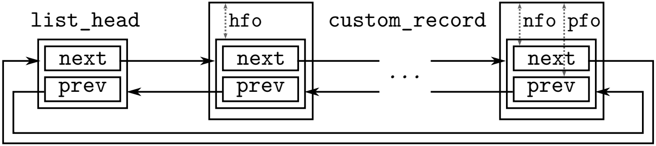
Symbolic Memory Graphs (SMGs)

- An example of a kernel-style linked list:

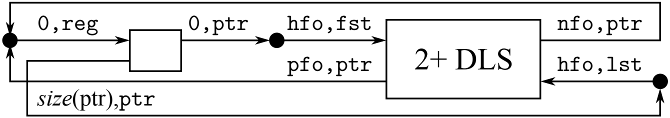


Symbolic Memory Graphs (SMGs)

- An example of a kernel-style linked list:

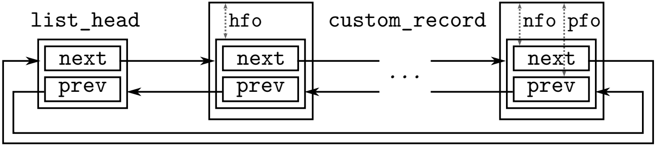


- An SMG describing the data structure above:

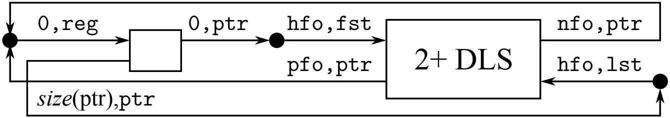


Symbolic Memory Graphs (SMGs)

- An example of a kernel-style linked list:

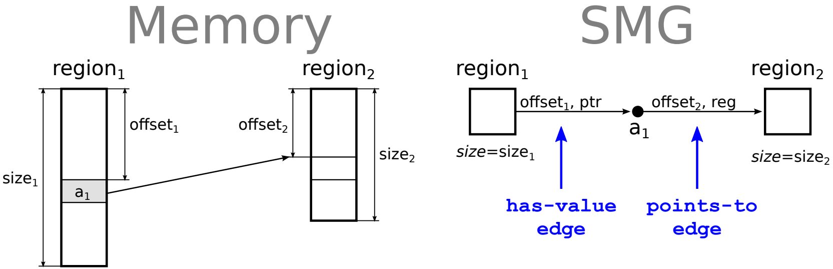


- An SMG describing the data structure above:

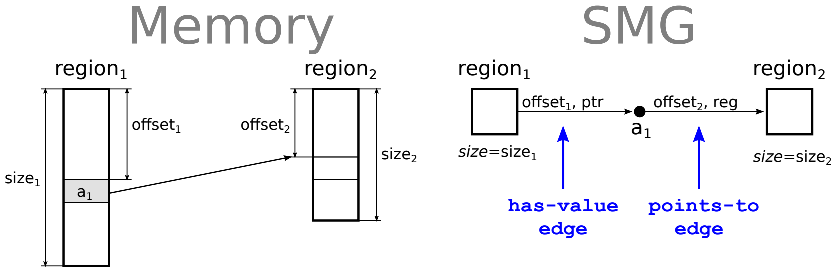


- SMGs are directed graphs consisting of:
  - objects (allocated space) and values (addresses, integers),
  - has-value and points-to edges.

SMGs: Has-Value and Points-To Edges

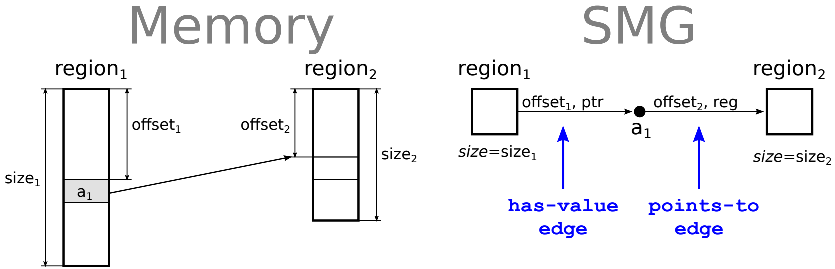


SMGs: Has-Value and Points-To Edges



- has-value edges – from objects to values, labelled by:
  - field offset
  - type of the value stored in the field

SMGs: Has-Value and Points-To Edges



- has-value edges – from objects to values, labelled by:
  - field offset
  - type of the value stored in the field
- points-to edges – from values (addresses) to objects, labelled by:
  - target offset
  - target specifier: first/last/each node of a DLS
    - specifier each node: used for back-links from nested objects

SMGs: Labelling of Objects

- Each object has some size in bytes and a validity flag.

SMGs: Labelling of Objects

- Each object has some size in bytes and a validity flag.
- Objects are further divided into:
  - regions, i.e., individual blocks of memory,
  - doubly-linked list segments (DLSs), and
  - other kinds of objects, which can be easily plugged-in.

SMGs: Labelling of Objects

- Each object has some size in bytes and a validity flag.
- Objects are further divided into:
  - regions, i.e., individual blocks of memory,
  - doubly-linked list segments (DLSs), and
  - other kinds of objects, which can be easily plugged-in.
- Each DLS is given by a head, next, and prev field offset.

SMGs: Labelling of Objects

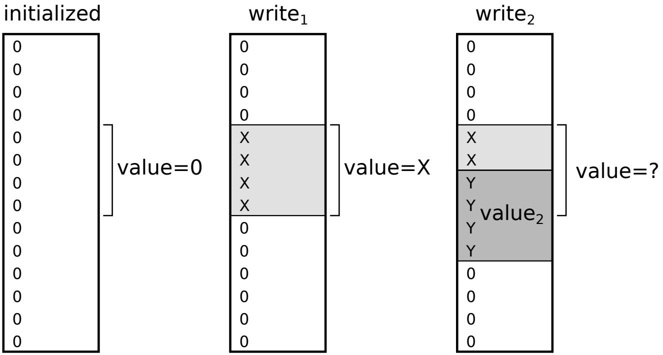
- Each object has some size in bytes and a validity flag.
- Objects are further divided into:
  - regions, i.e., individual blocks of memory,
  - doubly-linked list segments (DLSs), and
  - other kinds of objects, which can be easily plugged-in.
- Each DLS is given by a head, next, and prev field offset.
- DLSs can be of length $N+$ for any $N \geq 0$.

SMGs: Labelling of Objects

- Each object has some size in bytes and a validity flag.
- Objects are further divided into:
  - regions, i.e., individual blocks of memory,
  - doubly-linked list segments (DLSs), and
  - other kinds of objects, which can be easily plugged-in.
- Each DLS is given by a head, next, and prev field offset.
- DLSs can be of length $N+$ for any $N \geq 0$.
- Nodes of DLSs can point to objects that are:
  - shared: each node points to the same object, or
  - nested: each node points to a separate copy of the object.
  - Implemented by tagging objects by their nesting level.

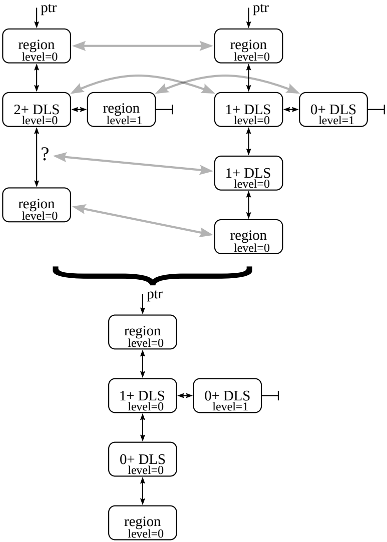
SMGs: Data Reinterpretation

- Reading: a field with a given offset and type either exists, or an attempt to synthesise if from other fields is done.
- Writing: a field with a given offset and type is written, overlapping fields are adjusted or removed.
- Currently, for nullified/undefined fields of arbitrary size only.



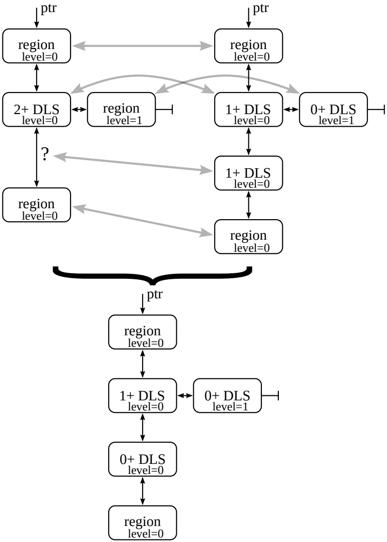
SMGs: Join Operator

- Traverses two SMGs and tries to join simultaneously encountered objects.



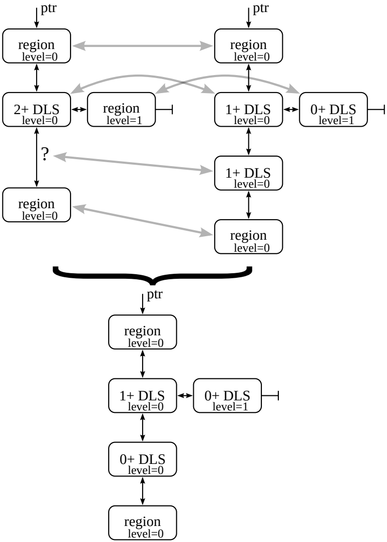
SMGs: Join Operator

- Traverses two SMGs and tries to join simultaneously encountered objects.
- Objects being joined must be locally compatible (same size, nesting level, DLS linking offsets, ...).



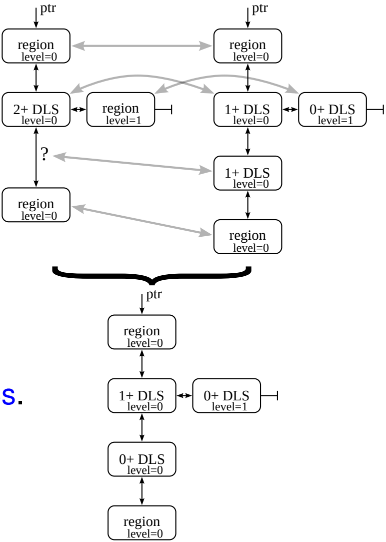
SMGs: Join Operator

- Traverses two SMGs and tries to join simultaneously encountered objects.
- Objects being joined must be locally compatible (same size, nesting level, DLS linking offsets, ...).
- Uses reinterpretation to try to synthesize possibly missing fields.



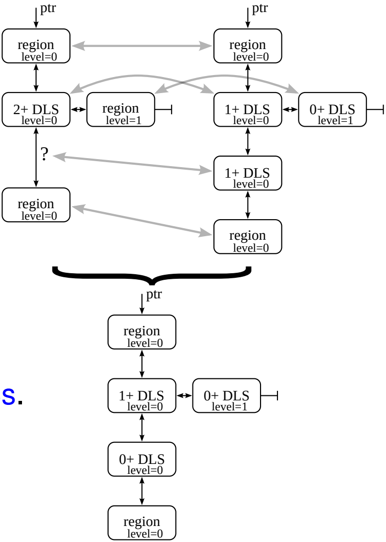
SMGs: Join Operator

- Traverses two SMGs and tries to join simultaneously encountered objects.
- Objects being joined must be locally compatible (same size, nesting level, DLS linking offsets, ...).
- Uses reinterpretation to try to synthesize possibly missing fields.
- DLSs can be joined with regions or DLSs.



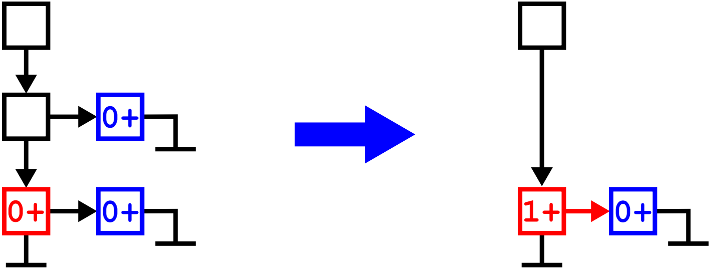
SMGs: Join Operator

- Traverses two SMGs and tries to join simultaneously encountered objects.
- Objects being joined must be locally compatible (same size, nesting level, DLS linking offsets, ...).
- Uses reinterpretation to try to synthesize possibly missing fields.
- DLSs can be joined with regions or DLSs.
- If the above fails, try to insert a DLS of length $0+$ into one of the SMGs.



SMGs: Abstraction

- Collapsing uninterrupted sequences of compatible objects (same size, nesting level, field offsets, ...) into DLSs.
- Uses join of the sub-SMGs under the nodes to be collapsed to see whether they are compatible too.
- Distinguishes cases of shared and private sub-SMGs.

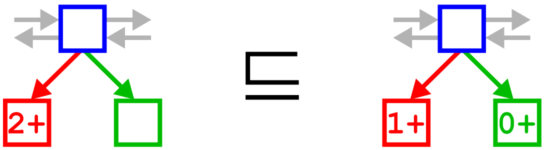


Controlling the Abstraction (1/2)

- There may be more sequences that can be collapsed.
  - We select among them according to their cost given by the loss of precision they generate.

- Three different costs of joining objects are distinguished:

0. Joining equal objects.
1. One object semantically covers the other:



2. None of the objects covers the other.

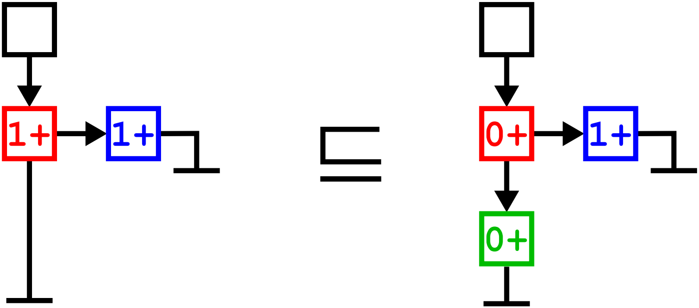
Controlling the Abstraction (2/2)

- For each object, find the maximal collapsing sequences (i.e., sequences which cannot be further extended).
- For the smallest cost for which one can collapse a sequence of at least some pre-defined minimum length, choose one of the longest sequences for that cost.
- Repeat till some sequence can be collapsed.

SMGs: Entailment Checking

- The join of SMGs is again used:

  $G_1 \sqsubseteq G_2$ tested by computing $G_1 \sqcup G_2$ while checking that G_1 consists of less general objects.



Agenda

1. Low-level Memory Manipulation
2. Symbolic Memory Graphs (SMGs)
3. Predator – Verifier Based on SMGs

Predator: An Overview

- A verficiation tool based on SMGs.
- Verification of low-level system code (such as Linux kernel) that manipulates dynamic data structures.
- Proving absence of memory safety errors (invalid dereferences, buffer overruns, memory leaks, ...).
- Predator is the winner of 3 categories of the 2nd International Competition on Software Verification (SV-COMP'13).
- Implemented as an open source GCC plug-in:

  `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator`

Predator: Related Tools

Many tools for verification of programs with dynamic linked data structures are currently under development. The closest to Predator are probably the following ones:

- Space Invader: pioneering tool based on separation logic (East London Massive: C. Calcagno, D. Distefano, P. O'Hearn, H. Yang).
- SLAyer: a successor of Invader from Microsoft Research (J. Berdine, S. Ishtiaq, B. Cook).
- Forester: based on forest automata combining tree automata and separation (J. Šimáček, O. Lengál, L. Holík, A. Rogalewicz, P. Habermehl, T. Vojnar).

Predator: Case Studies (1/2)

- More than 256 case studies in total.
- Programs dealing with various kinds of lists (Linux lists, hierarchically nested lists, ...).
  - Concentrating on typical constructions of using lists.
  - Considering various typical bugs that appear in more complex lists (such as Linux lists).
- Correctness of pointer manipulation in various sorting algorithms (Insert-Sort, Bubble-Sort, Merge-Sort).
- We can also successfully handle the driver code snippets available with SLAyer.
- Tried one of the drivers checked by Invader.
  - Found a bug caused by the test harness used, which is related to Invader not tracking the size of blocks.

Predator: Case Studies (2/2)

Verification of selected features of the following systems:

- The memory allocator from Netscape Portable Runtime (NSPR) used, e.g., in Firefox.
  - One size of arenas for user allocation, allocation of blocks not exceeding the arena size for now.
  - Repeated allocation and deallocation of differently sized blocks in arena pools (lists of arenas) and lists of arena pools (lists of lists of arenas).
  - Checked basic pointer safety + validity of the built-in asserts.
- Logical Volume Manager (lvm2).
  - A so far restricted test harness using doubly-linked lists instead of hash tables, which we do not support yet.

Predator: Experimental Results

- Selected experimental results showing either the verification time or one of the following outcomes:

FP	=	false positive	T	=	time out (900 s)
FN	=	false negative	x	=	parsing problems

Test Origin	Test	Invader	SLAyer 2011-01	Predator 2011-10	Predator 2013-02
SLAyer	`append.c`	<0.01 s	10.47 s	<0.01 s	<0.01 s
	`cromdata_add_remove_fs.c`	<0.01 s	FN	<0.01 s	<0.01 s
	`cromdata_add_remove.c`	T	FN	<0.01 s	<0.01 s
	`reverse_seg_cyclic.c`	FP	0.68 s	<0.01 s	<0.01 s
	`is_on_list_via_devext.c`	T	34.43 s	0.20 s	0.02 s
	`callback_remove_entry_list.c`	T	71.46 s	0.14 s	0.10 s
Invader	`cdrom.c`	FN	x	2.44 s	0.66 s
Predator	`five-level-sll-destroyed-top-down.c`	FP	x	FP	0.05 s
	`linux-dll-of-linux-dll.c`	T	x	0.41 s	0.05 s
	`merge-sort.c`	FP	x	1.08 s	0.21 s
	`list-of-arena-pools-with-alignment.c`	FP	x	FP	0.50 s
	`lvmcache_add_orphan_vginfo.c`	x	x	FP	1.07 s
	`five-level-sll-destroyed-bottom-up.c`	FP	x	FP	1.14 s

Predator: Future Work

- Further improve the support of interval-sized blocks and pointers with interval-defined targets.
  - Allow joining of blocks of different size.
  - Add more complex constraints on the intervals.
  - ...
- Support for additional shape predicates:
  - trees,
  - array segments,
  - ...
- Support for non-pointer data (mainly integers) stored in the data structures.
- Analysis of incomplete code without having to model its environment.

Summary

- Low-level code uses some tricky programming techniques:
  - special kinds of linked lists, alignment of pointers,
  - block operations, data reinterpretation
  - ...
- We propose Symbolic Memory Graphs (SMGs) as an abstract domain for shape analysis of code using the above mentioned low-level programming techniques.
- Predator is a tool based on SMGs. It can prove absence of memory safety bugs in low-level code.
- Predator is implemented as a GCC plug-in and available for free (including the source codes):

  `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator`