Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic

FIT BUT Technical Report Series

Kamil Dudka, Petr Peringer, and Tomáš Vojnar

Technical Report No. FIT-TR-2011-02
Faculty of Information Technology, Brno University of Technology

Last modified: April 18, 2011

Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic

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

[1] FIT, Brno University of Technology, Czech Republic
[2] Red Hat Czech, Brno, Czech Republic

Abstract. Predator is a new open source tool for verification of sequential C programs with dynamic linked data structures. The tool is based on separation logic with inductive predicates although it uses a graph description of heaps. Predator currently handles various forms of lists, including singly-linked as well as doubly-linked lists that may be circular, hierarchically nested and that may have various additional pointer links. Predator is implemented as a `gcc` plug-in and it is capable of handling lists in the form they appear in real system code, especially the Linux kernel, including a limited support of pointer arithmetic. Collaboration on further development of Predator is welcome.

1 Introduction

In this report, we present a new tool called *Predator* for fully automatic verification of sequential C programs with dynamic linked data structures. In particular, Predator can currently handle various complex kinds of *singly-linked as well as doubly-linked lists* that may be circular, shared, hierarchically nested, and that can have various additional pointers (head/tail pointers, data pointers, etc.). Predator implicitly checks for absence of generic errors, such as null dereferences, double deletion, memory leakage, etc. It can also print out a symbolic representation of the shapes of the memory structures arising in a program. Finally, users can, of course, use Predator to check custom properties about the data structures being used in their code by writing (directly in C) tester programs exercising these structures.

Predator is based on *separation logic* with *higher-order inductive predicates*. It is inspired by the works [2,8,9] and the very influential tool called Space Invader[3] (or simply Invader). However, compared to Invader, the heap representation in Predator is not based on lists of separation logic formulae, but rather a *graph representation* of these formulae. The algorithms handling the symbolic heap representation (in particular, the abstraction and join operators based on detecting occurrences of heap structures that can be described by inductive predicates) have been newly designed.

Compared to Invader that contains a partial support of doubly-linked lists only, Predator supports them equally well as singly-linked lists. Predator also contains a special support for list segments of length 0 or 1 that are common in practice [8] and that may cause problems to Invader (as we illustrate further on).

[3] `http://www.eastlondonmassive.org/East_London_Massive/Invader_Home.html`

The long term goal of Predator is handling *real system code*, in particular, the Linux kernel. In such code, for efficiency reasons, special forms of lists are used. In order to be able to handle them, Predator comes with a limited support of *pointer arithmetic*, which, however, covers most practical needs. Therefore, in a heap representation, the points-to links are associated with an offset w.r.t. the object they point to. Despite such an extension is not mentioned in [2,9], the Invader tool seems to partially support it, but it fails in many practical cases that Predator can handle well.

Predator is written in C++. It is built as a `gcc` plug-in, hence its front end is the same compiler that is used in practice for compiling the code that Predator is intended to analyse. Predator is completely *open source*[4] in order to allow for an open collaboration on its further development, which is very welcome.

There of course exist many other works on verification of programs with dynamic linked data structures than those using separation logic, including works based on other logics [6], automata [3], upward-closed sets [1], etc. These approaches offer different degrees of generality, automation, or scalability. A proper discussion of such works is, however, beyond the scope of this tool report. Throughout the report, we instead concentrate on a comparison with Invader as the closest tool to Predator. A similar tool is also jStar [5] which, however, concentrates on Java-specific problems. In [4], a bi-abductive analysis based on separation logic was proposed and implemented in a version of Invader, called Abductor[5]. This analysis, which is more scalable but less precise than the classical analysis used in Invader and Predator, is not yet implemented in Predator (whose core can, however, be used to implement it in the future). Unlike Invader, Predator cannot currently handle entire modules of Linux (such as drivers[6]) due to a so far very weak support of non-pointer data, which is one of the planned future works on Predator (together with a support of tree data structures, bi-abduction, etc.).

Below, we first say a bit more on the Linux lists supported by Predator, then we briefly mention some implementation details of Predator, and we proceed to interesting cases studies that illustrate the power of our tool. For some of the case studies, we are not aware of any other fully automatic, freely available tool, capable of handling them.

2 Lists Used in the Linux Kernel

As there is no standard implementation of linked lists in the C language, the Linux kernel has to implement lists on its own. The list implementation in Linux is well-known for its efficiency, portability, readability, and scalability—for instance, it allows to create list nodes which are owned by many distinct lists at a time. The downside is that it operates at a low level, hence it is easy to misuse the routines, and cause a disaster within the kernel. In Appendix C, we mention some common mistakes in manipulating Linux lists, which Predator is able to detect.

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

[5] Abductor is publicly available, but we have not managed to make it run. There is also a commercial implementation (`www.monoidics.com`), which is, however, not freely available.

[6] Although Invader has already shown some interesting results on pre-processed source code of selected Linux drivers, it is not ready for analysing drivers using the native Linux lists.

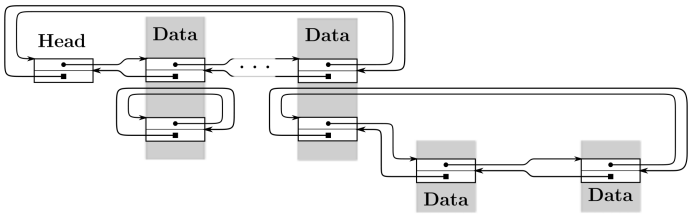

Fig. 1. A list of lists as implemented in the Linux kernel

The whole implementation of Linux lists is available in a single header file and consists of about 500 lines of code. It defines only one[7] type, which does all the job. The type contains a pair of pointers (`next` and `prev`), but no data. Such a structure is called a *head* and can be used in two ways—either as a starting point of a list (a *standalone head*), cf. the leftmost node in Fig. 1, or as part of list nodes (an *embedded head*).

Basic list operations (like addition, removal, reconnection of nodes) work only with heads and do not care about any associated data. In particular, the routines themselves do not distinguish between embedded and standalone heads.

The embedded head can be placed at an arbitrary offset in the surrounding structure. Moreover, it is possible to put many embedded heads into one structure such that one node is part of many lists. The standalone head can be placed on stack, but it can also be surrounded by another type. This way one can construct hierarchical list structures as shown in Fig. 1. Note that the beginning of the data nodes (depicted in gray in Fig. 1) needs not to be directly accessible by any pointer and can hence be mistakenly considered garbage if pointer arithmetic is not taken into account.

Linux lists are doubly-linked and *circular*, which significantly simplifies the design and boosts performance. That is, each routine for reconnection of a node (insertion, deletion, etc.) fits into a single basic block, which would not have been possible in case of regular NULL-terminated doubly-linked lists. It also implies that there is no need to have an explicit starting point (a standalone head) for each list. The Linux list library provides macros to define a standalone head and initialise it as an empty list. In case of Linux lists, an empty list means that `next` and `prev` fields point to the head itself.

The basic list traversal macro (`list_for_each`) provides a pointer to a head in each iteration and works without any type-awareness of the data nodes being traversed. The macro `list_entry` then allows to translate a pointer to a head into a pointer to the corresponding data node. As the offset of each embedded head in the structure is known at compile-time, the macro can easily use pointer arithmetic to compute the required address. There is also an extended macro for list traversal (`list_for_each_entry`) that efficiently wraps `list_entry` and this way gives us a pointer to the data node in each iteration, instead of the pointer to the embedded head. Note, however, that it may happen that some pointer variable points to the unallocated space around a standalone head and yet may be correctly used (by being subsequently moved forward by pointer

[7] Starting with Linux-2.5.64, there is also an optimised variant of lists for constructing hash tables. We are not yet able to analyse the code that uses these optimised lists.

arithmetic). On the other hand, dereferencing such a pointer is an error, which can, e.g., lead to stack smashing (and is detectable by Predator).

A nice introduction into how Linux lists work can be found in [7]. We use the code from there as one of the case studies distributed as test-cases with Predator.

3 A Note on the Implementation of Predator

Predator implements a symbolic analysis based on separation logic with higher-order inductive predicates, inspired by the works [2,9] implemented in Space Invader. Predator uses a graph representation of separation logic formulae, a little bit similar to the graph representation introduced in [2] for description of the predicate discovery algorithm. Our representation is, however, more complex and used all the time.

In our graph-based symbolic representation of heaps, we use two kinds of nodes: *objects* (statically and automatically allocated program variables, dynamically allocated storage, etc.) and *values* of the objects (e.g., addresses of objects and the special undefined, deleted, and null values in the case of pointers and function pointers). Objects can be nested in order to represent the composition of C language structures. The appropriate nodes are linked by oriented graph edges *hasValue* (going from objects to values) and *pointsTo* (going from values to target objects). In order to allow for efficient equality testing, equal objects are simply linked to the same value node. To encode non-equality relations, value nodes may be linked by undirected *neq* edges. Further, when pointer arithmetic produces a value that does not point to a valid target, we use so-called *offset* edges between value nodes. Such values can later be used for a valid memory operation—they can either be translated by another use of pointer arithmetic to a valid address, or directly used for accessing an existing subobject of a non-existent surrounding object (which is common, e.g., when working with Linux lists as we show in Appendix B).

We represent inductive predicates as special *abstract objects*. Currently, we support only singly- and doubly-linked list segments that may be shared, nested, and with various additional (head, tail, data, and the like) pointers. A doubly-linked list segment (DLS) has two endpoints, both of which may be pointed to. Therefore, since each object has exactly one address, we in fact represent DLS as pairs of abstract objects. To cope with pointer arithmetic, we equip abstract objects with *offsets* specifying the relative placement of the core linking pointers (`next`/`prev` for lists). Moreover, to cope with Linux lists, we also record the *head offset* that is a relative placement of the list pointer's target. For Linux lists, it corresponds to the offset of the embedded head, whereas for regular lists, it is simply zero. We do not treat the minimal segment length as an explicit property of a list segment as in [8]. Instead, our list segments are implicitly possibly empty (i.e., of length 0+). We use the generic mechanism of *neq* edges between nodes before and after a segment to construct non-empty segments (length 1+). For DLS, we use two *neq* edges for length 1+ (one edge for each direction) and three *neq* edges for length 2+ (the additional *neq* edge is in between the ends of the DLS). Such an approach leads to a simpler and more readable implementation. Apart from that, we then have special abstract objects for list segments of length 0 or 1. Some more details about the representation of symbolic heaps can be found in Appendix A.

Predator maintains a set of symbolic heaps for each basic block entry. The set is not yet implemented as an ordered or hashed container, but it utilises a join operator similar to the join operator introduced in [9], helping to significantly decrease the number of symbolic heaps to be maintained. Moreover, Predator uses a slightly modified version of the join algorithm to merge pairs of objects during a list segment abstraction, in particular to join nested predicates, shared (head, tail or other) pointers, and other data. The modified join algorithm operates on two parts of a single heap given by the pair of objects being merged, and constructs a joint description of both parts. The algorithm can also run in a read-only mode to decide whether the join operation is possible. The read-only mode can be safely used during predicate discovery. Thanks to this, the algorithms for abstraction and predicate discovery are implemented as a very thin purpose-specific layer on top of the generic join algorithm. In Appendix E, we show an example of use of the modified join algorithm.

For inter-procedural analysis, Predator uses function summaries in a way similar to [9], including a support of indirect function calls and recursive calls of fixed depth.

Predator is tightly integrated with `gcc` (version 4.5.0 and newer) as a *plug-in*. Therefore, there is no need to manually pre-process the sources, neither to change the way they are built, whenever dealing with software natively compiled by `gcc`. Usage of Predator is as easy as adding a new compiler flag into `CFLAGS` while building a project. Code defects encountered during analysis are reported in the `gcc` format. Hence it is easy to reuse existing development tools, IDE, etc. In order to give users a clue about detected errors, Predator provides a backtrace for each error. Predator attempts to report as many errors and warnings as possible per run. For instance, if a memory leak is detected, a warning is issued, and Predator keeps searching for further errors (due to a garbage collector that gets the symbolic heap back to its consistent shape). Predator supports error recovery for most of the program errors which it is able to detect. Such an approach may trigger an error avalanche in certain cases, but the same may happen with bare `gcc` during compilation and developers know how to resolve it.

4 Experiments with Predator

Along with Predator, we distribute a comprehensive set of programs (over a hundred test cases) that can be handled by our tool, including various textbook implementations of lists (singly-linked, doubly-linked, circular, hierarchically nested, etc.) as well as examples using Linux lists[8]. These case studies are mid-size (up to 300 lines), however, they contain almost only pointer manipulations unlike larger programs whose big portions are often not relevant for pointer analyses like ours. Apart from basic list manipulation (creation of random lists, reversal, destruction, etc.), we provide also examples of various sorting algorithms: Merge-Sort (`test-0124.c`), Insert-Sort (`test-0134.c`), and Bubble-Sort (`test-0136.c`). The Merge-Sort case study operates on hierarchical singly-linked lists. The other two sorts use the native implementation of Linux lists. Predator is not proving that the resulting list is sorted, but it verifies memory safety of

[8] In the following text, we provide in brackets the file names under which the discussed case studies appear in the distribution of Predator. The case studies are available at `https://github.com/kdudka/predator/tree/61d5df3/sl/data`.

the code. Invader, as a freely available tool closest to Predator, is not able to analyse any of our sorting case studies.

Some of our test cases show common mistakes in using Linux lists such as mixing pointers to a head with pointers to data (`test-0138.c`) or treating a standalone head as if it was an embedded head (`test-0137.c`). Only programmers know the purpose of each head, and if they use the head in a wrong way, it is likely to be noticed at runtime only (and often not immediately). For example, starting from a standalone head, the `list_for_each_entry` macro provides a valid pointer to data in each iteration. However, if one starts to traverse the list from the middle, it ends up by misinterpreting the standalone head's neighbourhood as list node's data. Predator is capable of detecting such mistakes. We, for instance, provide an example where a wrong head is used for a Linux list traversal (`test-0131.c`). Despite even the dynamic analysis tool `valgrind`, often used by developers, claims there is an invalid write, Invader says the code is safe. On the contrary, Predator detects the flaw in 0.01s, which is even faster than `valgrind`.

Our test suite further contains various programs intended to stress test the discovery of inductive predicates. These case studies include, e.g., conversion of a singly-linked list into a doubly-linked and then back to singly-linked list (`test-0061.c`), or construction of two independent lists starting from the same node (`test-0113.c`), which other tools may inaccurately over-approximate as a hierarchically nested list or a binary tree.

Another case study considers a call of `free()` on an embedded head that appears in real code if the head is placed at zero offset within the data node (`test-0087.c`). Tools that ignore address aliasing of fields placed at the same offset, like Invader, mistakenly report such an operation as an error in the analysed program. Since Predator uses the offset-based description of list segments, it can easily cope with address aliasing.

We also provide a few case studies of lists where each node *optionally* owns some nested objects (`test-0128.c`). Those may be incorrectly abstracted as nested lists if only usual list segments are considered, and in case the program does not really treat such objects as lists, it leads to spurious memory leaks or even non-termination of the analysis. Predator covers these cases by special abstract objects of length 0 or 1, which allows a more precise analysis and solves the problems with spurious errors and non-termination.

Descriptions of selected case studies can be found in Appendix D. Across all our case studies, Predator acts fully automatically. There is no need to tell Predator what kind of data structures to look for. Given a C program, it simply returns the corresponding list of errors and warnings. In all but one of the mentioned tests, the time consumption was under 1.0s on Intel Core i5 3.33GHz. Moreover, for a vast majority of the tests, it was under 0.1s. The only exception was the Merge-Sort example, which took 7.8s to analyse. We are, however, not aware of any comparable tool that is able to analyse the same example faster.

5 Conclusion

We have presented Predator, a new separation logic based tool for analysing programs with dynamic linked data structures. Despite the tool is only at the beginning of its

development, we have argued that it already offers many interesting features. In the future, the tool should be, e.g., enriched with some (preferably light-weight) support of non-pointer data (integers, arrays), extended to handle further classes of dynamic data structures, extended to handle C++ code (which the `gcc`-based front-end can easily handle), and so on. Since Predator is open source, GPL-licensed, and written such that its code is readable, collaboration on its further development is very well possible.

Acknowledgement This work was supported by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), and the BUT FIT project FIT-S-11-1.

References

1. P.A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, A. Rezine. Monotonic Abstraction for Programs with Dynamic Memory Heaps. In *Proc. of CAV'08*, *LNCS* 5123. Springer, 2008.
2. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, and H. Yang. Shape Analysis for Composite Data Structures. In *Proc. CAV'07*, *LNCS* 4590, 2007.
3. A. Bouajjani, P. Habermehl, A. Rogalewicz, T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, *LNCS* 4134. Springer, 2006.
4. C. Calcagno, D. Distefano, P.W. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-abduction. In *Proc. of POPL'09*. ACM Press, 2009.
5. D. Distefano and M. Parkinson. jStar: Towards Practical Verification for Java. In *Proc. of OOPSLA'08*. ACM Press, 2008.
6. S. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.
7. K. Shanmugasundaram. Linux Kernel Linked List Explained, 2005. `http://isis.poly.edu/kulesh/stuff/src/klist`
8. H. Yang, O. Lee, C. Calcagno, D. Distefano, and P.W. O'Hearn. On Scalable Shape Analysis. Technical report RR-07-10, Queen Mary, University of London, 2007.
9. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P.W. O'Hearn. Scalable Shape Analysis for Systems Code. In *Proc. of CAV'08*, *LNCS* 5123. Springer, 2008.

A Heap Representation Used in Predator

In this appendix, we explain the basic principles of how Predator represents reachable heap configurations on concrete examples. We start with the code example in Fig. 2 that creates a sample shape consisting of two structured objects—the first is allocated on the stack, and the second is allocated dynamically. Both of them are connected with each other in a rather complex way (including, e.g., a pointer to a pointer link: the `pprev` field).

Assume that we print out the heap configuration reachable at line 35. For this, the built-in function `___sl_plot()` of Predator can be used. The function generates a symbolic heap graph (in the DOT format) that describes heap configurations reachable at the line where the function is called. The `NULL` value used as a parameter of `___sl_plot()` can be replaced by a string containing a custom name of the graph. It is also possible to generate only a restricted part (reachable from some given cell or visible within a function) of the heap graph in case the whole graph is too complex.

```c
#include <stdlib.h>

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

struct hlist_node {
    struct hlist_node *next, **pprev;
};

// a custom type used for list nodes at the top level
struct my_hlist {
    struct hlist_node node;

    // a standalone head of a nested Linux list;  no need
    // to specify the type of its nodes at this point
    struct list_head nested;
};

int main() {
    // allocate a heap object
    struct my_hlist *item = malloc(sizeof *item);
    item->nested.next = &item->nested;
    item->nested.prev = &item->nested;

    // initialize an object on stack and connect both
    // of them with each other
    struct { struct hlist_node *first; } my_hlist_head = {
        &item->node
    };
    item->node.pprev = &my_hlist_head.first;
    item->node.next = NULL;

    // tell Predator to generate a heap graph
    ___sl_plot(NULL);

    // avoid a memory leak by end of main()
    free(item);
    return 0;
}
```

Fig. 2. An example of a program that generates a symbolic heap graph to explain the graph representation of symbolic heaps used by Predator

The resulting heap graph of the code example above is shown in Fig. 3. The elliptic nodes represent values, while the rectangular nodes represent objects (cf. Section 3). The composition of C structures is expressed by the clusters of objects and the *field* edges which connect the objects within the clusters. The *hasValue* edges are blue and the *pointsTo* edges are green. Their labels are omitted in all the following heap graphs in order to improve their readability.

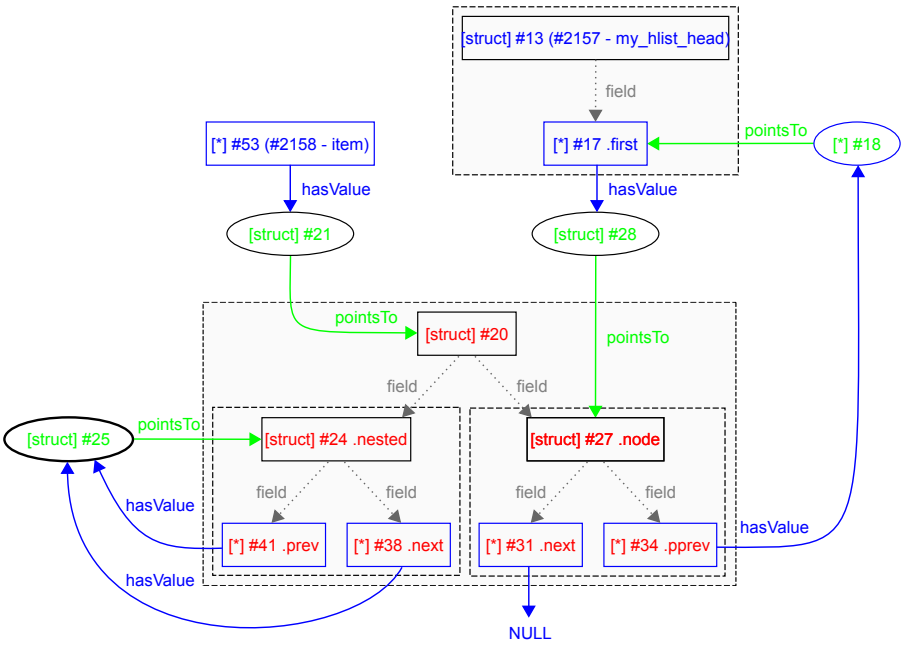

Fig. 3. An example of a symbolic heap graph to explain the graph representation of symbolic heaps used by Predator

A.1 Representation of Doubly-Linked Lists of Unbounded Length

The heap graph shown in Fig. 3 represents a single concrete heap configuration. Next, we proceed to a symbolic representation of heaps containing lists of an unbounded length. We concentrate on working with doubly-linked lists, which are more common in practice, and, moreover, dealing with singly-linked lists may easily be viewed as a restriction of dealing with doubly-linked lists.

As described in Section 3, Predator represents doubly-linked list segments (DLS) as pairs of abstract objects. Fig. 4 depicts an exemplary pair of objects that may be abstracted as DLS of length 2+ (Fig. 5). On separation of a node from such a DLS, it turns out into DLS of length 1+ (Fig. 6) and subsequently 0+ (Fig. 7). On the other hand, if a node is appended to the DLS, the minimal length grows back.

We note that singly-linked list segments are represented in a similar way, but just one abstract object is used (i.e., one half of the presented doubly-linked list segments– cf. Fig 14 and 15 where SLS stands for a singly-linked list segment).

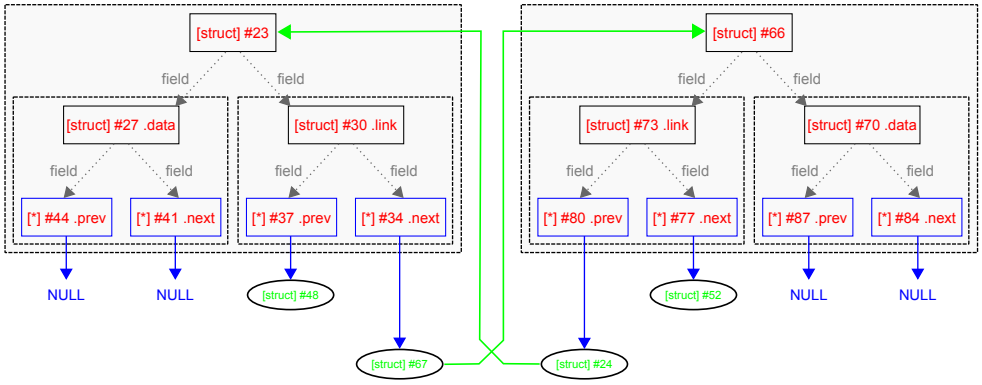

Fig. 4. A pair of objects that may be abstracted as a doubly-linked list segment

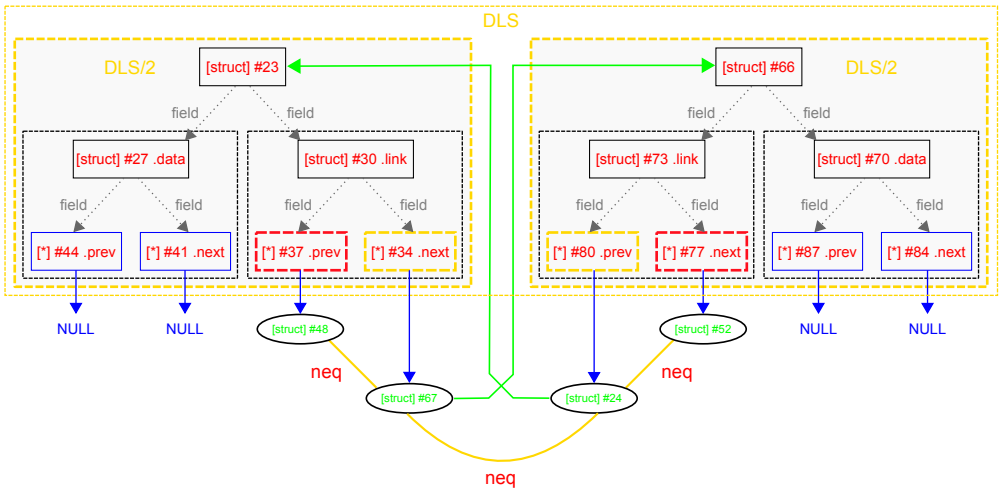

Fig. 5. A doubly-linked list segment of length 2+ as represented by Predator

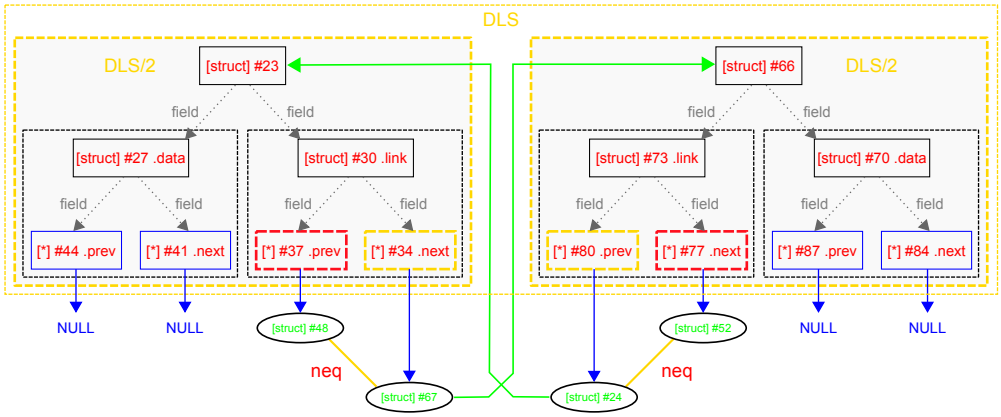

Fig. 6. A doubly-linked list segment of length 1+ as represented by Predator

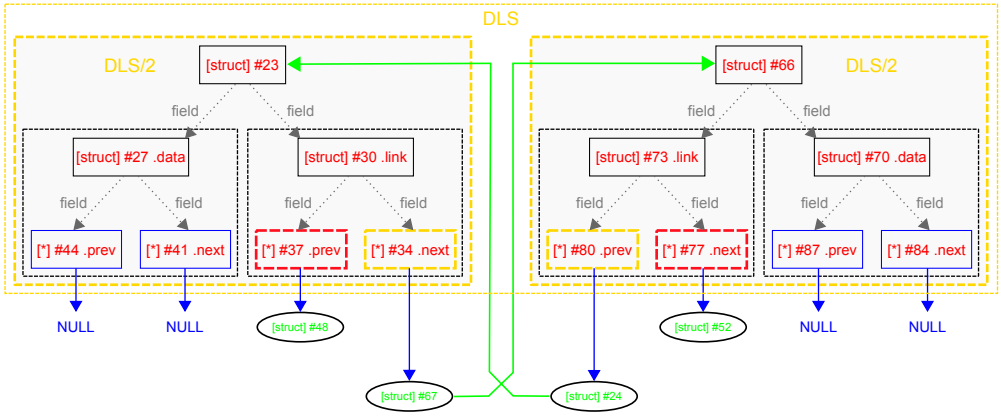

Fig. 7. A doubly-linked list segment of length 0+ as represented by Predator

A.2 Representation of Linux Lists

In contrast to the plain DLS abstraction, the Linux DLS allows a non-zero head offset, which means the nodes are linked through the middle of the objects as shown in Fig. 8. Fig. 9 depicts the result of the Linux DLS abstraction. The subobject that is placed at the head offset within the list node is highlighted in green and labeled as *head*. The view in Fig. 10, which includes only heads, is close to the view of the Linux list routines that do not care about any associated data.

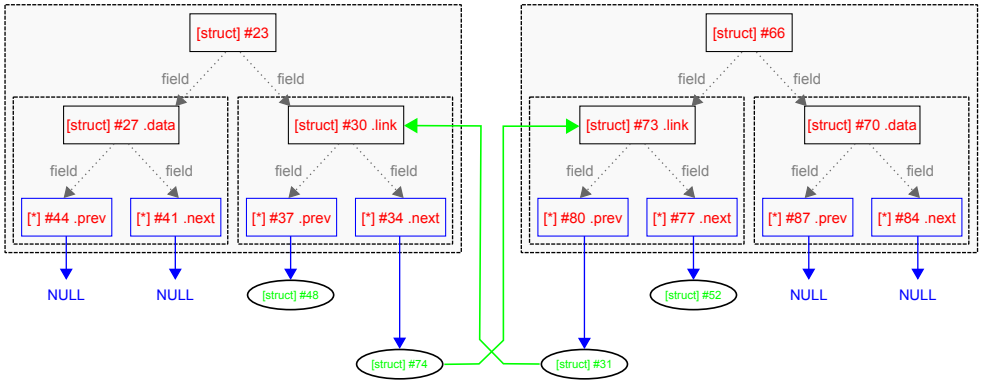

Fig. 8. A pair of objects that may be abstracted as a doubly-linked list segment with a *head offset*

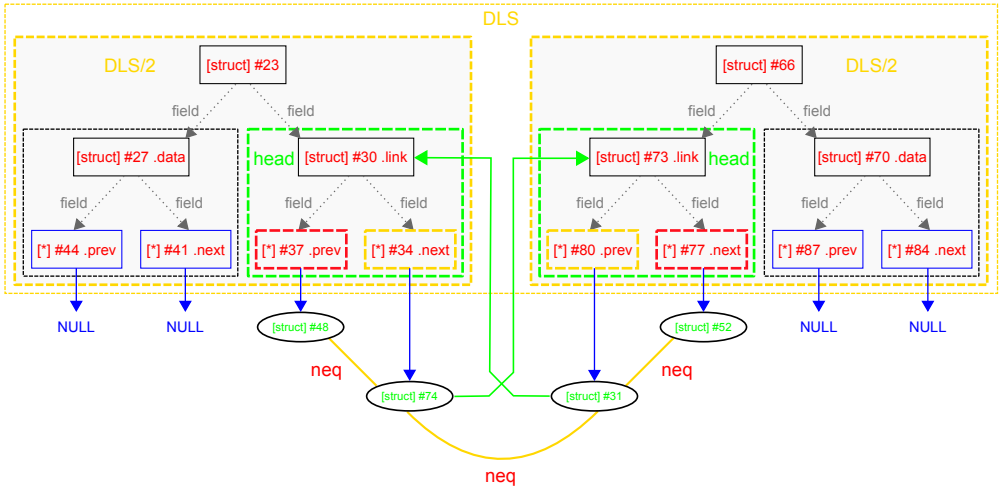

Fig. 9. A doubly-linked list segment of a *Linux list* of length 2+

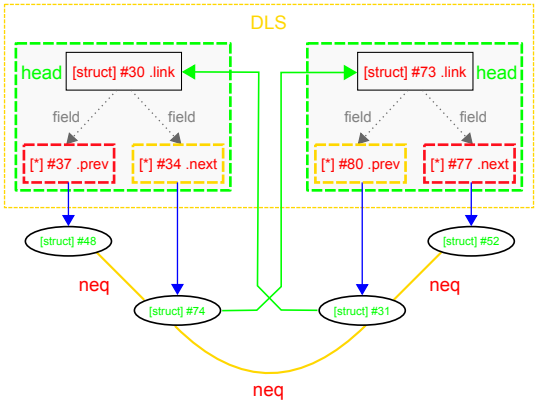

Fig. 10. Embedded heads in a doubly-linked list segment of a Linux list of length 2+

B Traversal of a Linux List

The heap graphs in this section explain how the macro `list_for_each_entry`, commonly used in the Linux code, implements the traversal of a Linux list (and hence, what the analysis must be ready to handle). In the following case study, a Linux list of length 2 with a standalone head is used. The macro gets only one pointer which is, in that particular case, the address of the standalone head. Using the next pointer inside the head, it jumps to the next head. The cursor, which is available inside the loop body, is then set to the address of the corresponding data node by subtracting the head offset. Fig. 11 and Fig. 12 show the heap graphs corresponding to the loop body as the cursor is moved from the first list node to the second one. However, once the traversal has finished, which means the starting head is seen the second time, the cursor does not point to any known object. Yet, the cursor is still—surprisingly—used in the loop condition. In order to avoid a spurious error when evaluating the loop condition, Predator needs to represent such values, which is done as shown in Fig. 13.

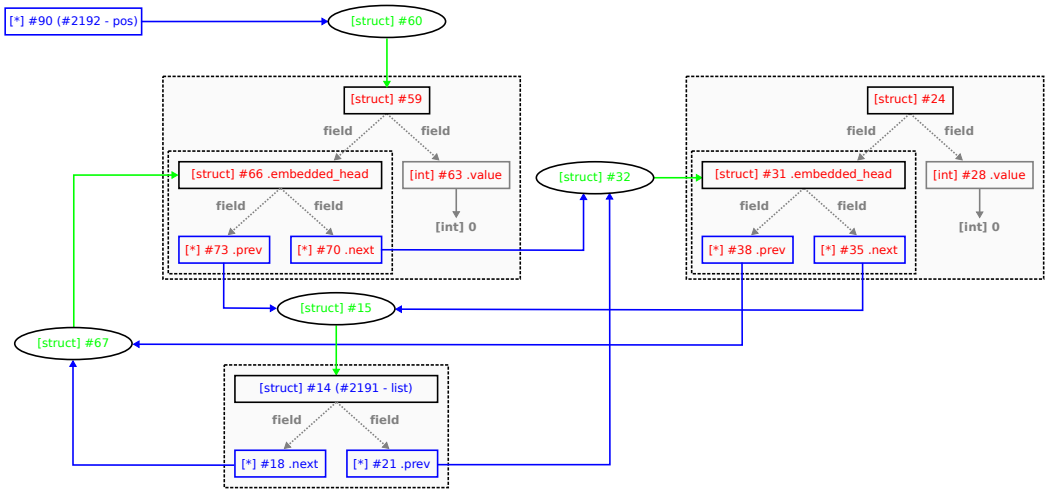

Fig. 11. A traversal of a Linux list with two nodes: 1st iteration

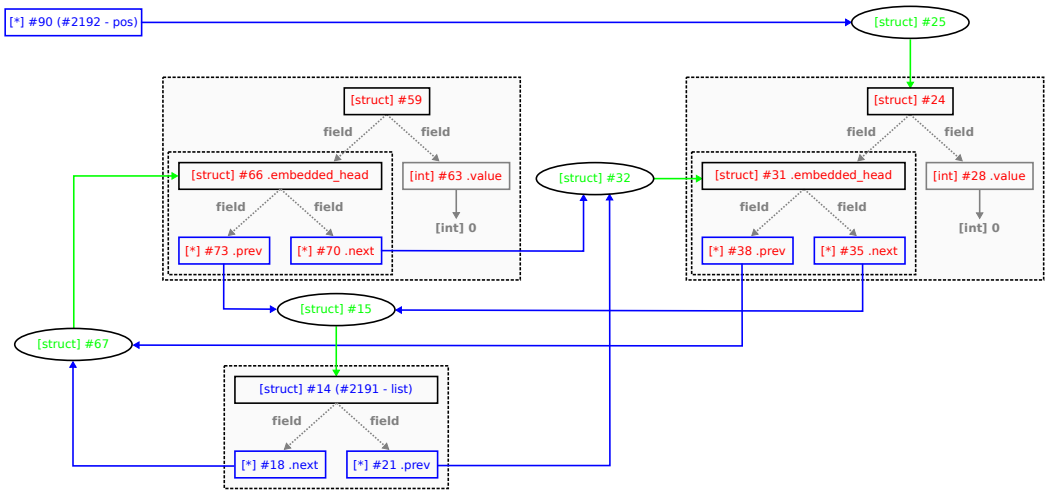

Fig. 12. A traversal of a Linux list with two nodes: 2nd iteration

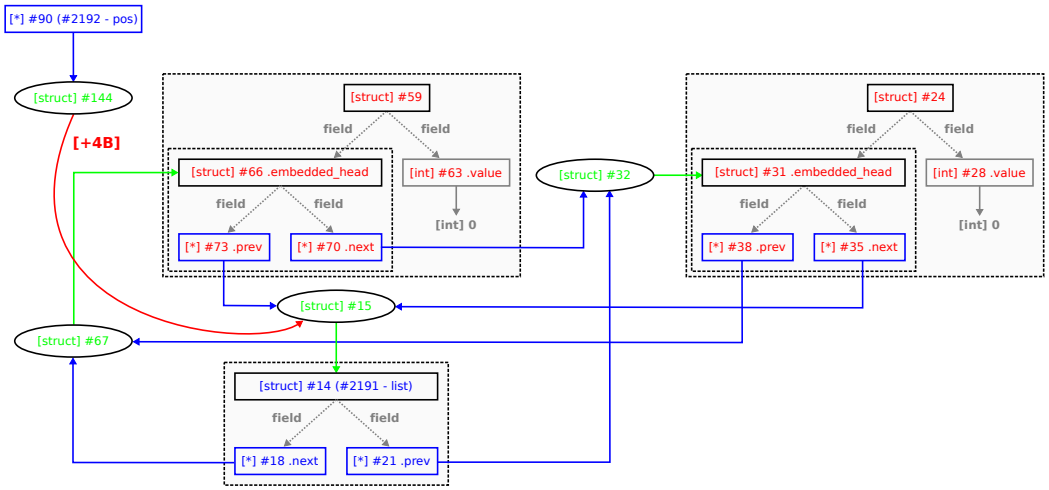

Fig. 13. A traversal of a Linux list with two nodes: after the traversal

C Common Mistakes in Using Linux Lists

In this section, we show code examples of the common mistakes in using Linux lists mentioned in Section 4 and the corresponding response of Predator to them. Note the here presented examples are simplified such that each of them fits into a single page. The full versions of all the examples are included as regression tests in the distribution of Predator (see the file names in error messages).

C.1 A Wrong Head Used for the Traversal of a Linux List

```
1  #include <linux/list.h>
2  #include <stdlib.h>
3
4  struct my_item {
5      void                *data;
6      struct list_head    link;
7      struct list_head    aux_link;
8  };
9
10 void traverse(struct list_head *head)
11 {
12     struct my_item *now;
13 #if TRIGGER_INVALID_WRITE
14     list_for_each_entry(now, head, aux_link)
15 #else
16     list_for_each_entry(now, head, link)
17 #endif
18     {
19         now->data = NULL;
20     }
21 }
22
23 int main()
24 {
25     LIST_HEAD(my_list);
26
27     int i;
28     for (i = 0; i < 1024; ++i) {
29         struct my_item *ptr = malloc(sizeof *ptr);
30         if (!ptr)
31             abort();
32
33         list_add_tail(&ptr->link, &my_list);
34     }
35
36     traverse(&my_list);
37     return 0;
38 }
```

```
test-0131.c:14:5: error: dereference of unknown value
test-0131.c:36:13: note: from call of traverse()
test-0131.c:23:5: note: from call of main()
```

C.2 Mixing Pointers to Head with Pointers to Data

```
1  #include <linux/list.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  struct node {
6      int                 value;
7      struct list_head    linkage;
8  };
9
10 LIST_HEAD(gl_list);
11
12 static void gl_destroy()
13 {
14     struct list_head *next;
15     while (&gl_list != (next = gl_list.next)) {
16         gl_list.next = next->next;
17 #if TRIGGER_INVALID_FREE
18         free(next);
19 #else
20         free(list_entry(next, struct node, linkage));
21 #endif
22     }
23 }
24
25 int main()
26 {
27     int value;
28     while (EOF != (value = getchar())) {
29         struct node *node = malloc(sizeof *node);
30         if (!node)
31             abort();
32
33         node->value = value;
34         list_add(&node->linkage, &gl_list);
35     }
36
37     gl_destroy();
38     return 0;
39 }
```

```
test-0138.c:18:13: error: attempt to free a non-root object
test-0138.c:37:15: note: from call of gl_destroy()
test-0138.c:25:5: note: from call of main()
```

C.3 Mixing Embedded Heads with Standalone Heads

```
1  #include <linux/list.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  struct node {
6      int                 value;
7      struct list_head    linkage;
8  };
9
10 LIST_HEAD(gl_list);
11
12 static int val_from_node(struct list_head *head) {
13     struct node *entry = list_entry(head, struct node, linkage);
14     return entry->value;
15 }
16
17 static struct list_head* gl_seek_max()
18 {
19     struct list_head *pos, *max_pos = NULL;
20     int max;
21
22     if (list_empty(&gl_list))
23         return NULL;
24     else {
25         max_pos = gl_list.next;
26         max = val_from_node(max_pos);
27     }
28
29     // misuse of list_for_each() at this point
30     list_for_each(pos, max_pos) {
31         const int value = val_from_node(pos);
32         if (value < max)
33             continue;
34
35         max_pos = pos;
36         max = value;
37     }
38
39     return max_pos;
40 }
41
42 int main()
43 {
44     int value;
45     while (EOF != (value = getchar())) {
46         struct node *node = malloc(sizeof *node);
47         if (!node)
48             abort();
49
50         node->value = value;
51         list_add(&node->linkage, &gl_list);
52     }
53
54     return !!gl_seek_max();
55 }
```

```
test-0137.c:14:5: error: dereference of unknown value
test-0137.c:31:19: note: from call of val_from_node()
test-0137.c:54:25: note: from call of gl_seek_max()
test-0137.c:42:5: note: from call of main()
```

D Selected Case Studies

In this section, we briefly introduce selected case studies which illustrate the power of Predator. All of them (and many more) are included in the distribution of Predator as regression tests (their numbers in the distribution are mentioned below for reference). In the description, we abbreviate singly-linked lists as *SLLs* and doubly-linked lists as *DLLs*.

Conversion of an SLL to a DLL and vice versa (`test-0061.c`).
An SLL is created using the `next` field as the forward link. The list is then traversed and missing values of the `prev` field in each node are completed in order to get a DLL. Finally, while going back again, the `next` field in each node is nullified, which results into a reversed SLL.

An SLL of DLLs destructed in two steps (`test-0067.c`).
An SLL of DLLs is created. The structure is then destroyed in two steps. First, all nested DLLs are destroyed. This way we get a flat SLL, which is destroyed in the second step.

A DLL of DLLs constructed using a bounded call recursion (`test-0072.c`).
A DLL of DLLs is created using a call recursion of depth 2. The list constructor is written generically for both levels of the list. The constructor is then called with a function pointer to specify which node constructor to use, depending on which level the list constructor is being called at.

An SLL of SLLs where the nested lists are of length 0 or 1 (`test-0128.c`).
Such a shape is hard to analyse if only usual list segments are considered because a list consisting of a single node is nearly impossible to be discovered as a list. Shapes like this are, however, commonly used in practice.

An SLL of SLLs where the nested lists are of length 1 or 2 (`test-0111.c`).
This case study is a modification of `test-0128.c`. Odd nodes at the top level contain a nested list of length 1, while even nodes contain a nested list of length 2. The test shows the power of the modified join algorithm used during the predicate discovery.

A Linux DLL with two nested Linux DLLs (`test-0102.c`).
In the case study there is created a Linux DLL at the top-level where from each node of the list start two independent nested Linux DLLs. This case is not easy to analyse since each nested list is linked through a node of the top-level list, which happens whenever Linux lists are used for building hierarchical list structures.
A call to the destructor of one of the nested lists is intentionally omitted in order to trigger a memory leak. Predator shows where exactly the missing call of the destructor belongs.

E An Example on Use of the Modified Join Algorithm

As mentioned in Section 3, Predator uses a modified join algorithm to merge pairs of objects during a list segment abstraction. Fig.14 shows an example of a symbolic heap where the modified join algorithm can be utilized. The two objects on the left side are connected in a way that they can, in principle, be merged together into one singly-linked list segment (SLS). The first of them is a regular object, while the second one already is an SLS of length 0+. As in [2], call these potential candidates root objects. In [2], a search for two disjoint isomorphic subgraphs starting from the root objects is used to see whether the objects can be merged into a list segment. In our case, in order to see whether we can merge the root objects, we use the join algorithm to try to join the nested predicates first. In our example, we try to join a nested Linux DLS of length 2+ with a nested single node. The join is possible, and the result is a nested Linux DLS of length 1+. Once we have a joint description of the nested predicates, we can merge the pair of root objects together. In our example, we get an SLS of length 1+ with a nested Linux DLS of length 1+ as the overall result (depicted in Fig. 15).

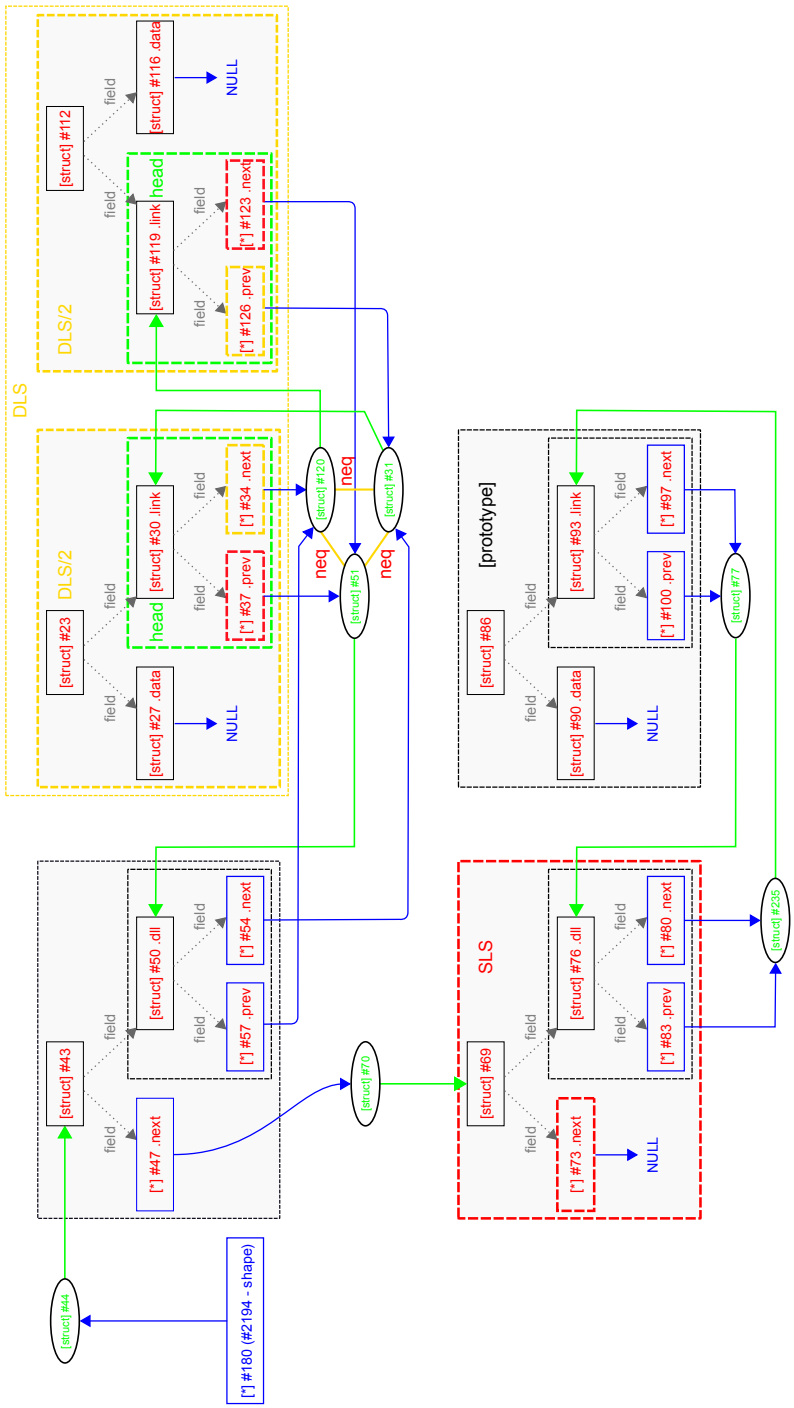

Fig. 14. A use case of the modified join algorithm—a nested Linux DLS of length 2+ that may be joint with a nested single list node

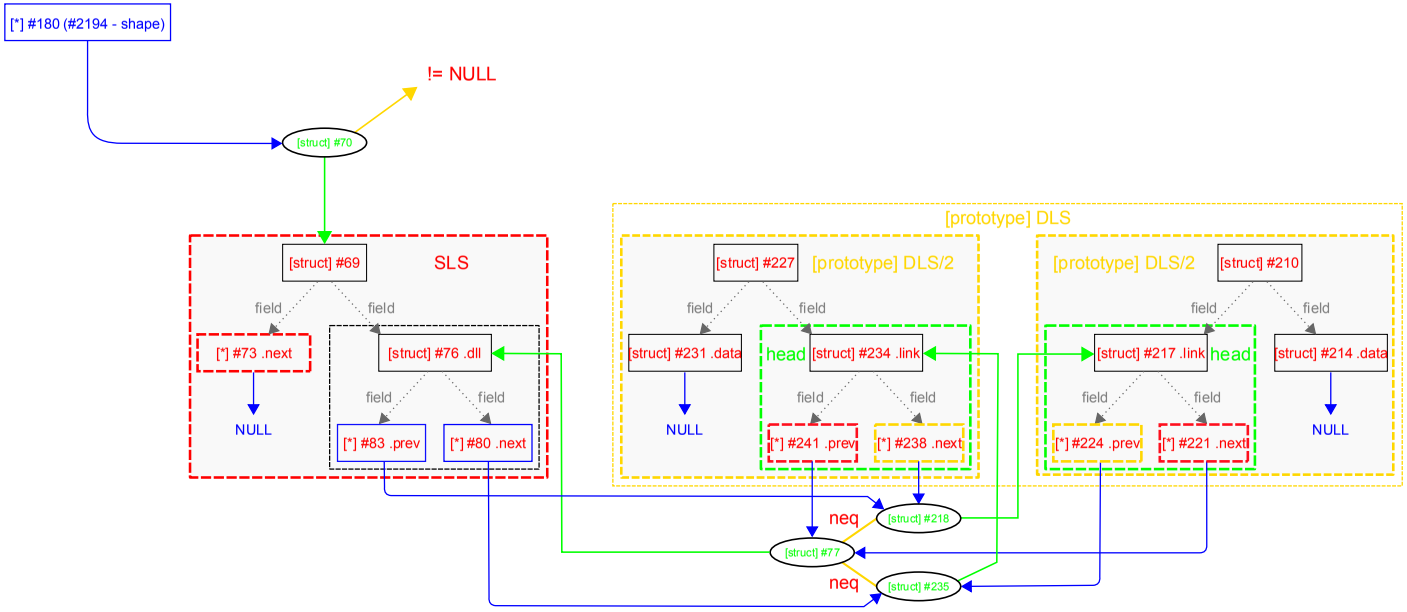

Fig. 15. A result of a list segment abstraction that utilizes the modified join algorithm—a singly-linked list segment of length 1+ with a nested Linux DLS of length 1+.