The Impact of Generic Data Structures: Decoding the Role of Lists in the Linux Kernel

Nic Volanschi, Julia Lawall

The Impact of Generic Data Structures: Decoding the Role of Lists in the Linux Kernel

Nic Volanschi
Inria Bordeaux - Sud-Ouest
Talence, France
eugene.volanschi@inria.fr

Julia Lawall
Inria Paris
Paris, France
julia.lawall@inria.fr

ABSTRACT

The increasing adoption of the Linux kernel has been sustained by a large and constant maintenance effort, performed by a wide and heterogeneous base of contributors. One important problem that maintainers face in any code base is the rapid understanding of complex data structures. The Linux kernel is written in the C language, which enables the definition of arbitrarily uninformative datatypes, via the use of casts and pointer arithmetic, of which doubly linked lists are a prominent example. In this paper, we explore the advantages and disadvantages of such lists, for expressivity, for code understanding, and for code reliability. Based on our observations, we have developed a toolset that includes inference of descriptive list types and a tool for list visualization. Our tools identify more than 10,000 list fields and variables in recent Linux kernel releases and succeeds in typing 90%. We show how these tools could have been used to detect previously fixed bugs and identify 6 new ones.

CCS CONCEPTS

• **Software and its engineering** → *Automated static analysis*; **Data types and structures**; **Software maintenance tools**; *Software reverse engineering*; *Maintaining software.*

KEYWORDS

Linux kernel, data types, genericity

ACM Reference Format:
Nic Volanschi and Julia Lawall. 2020. The Impact of Generic Data Structures: Decoding the Role of Lists in the Linux Kernel. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416635

1 INTRODUCTION

The Linux kernel is today a form of critical infrastructure, supporting billions of Android smartphones, all of the top 500 supercomputers, and many computing domains in between. The Linux kernel also has a very large developer base, with over 1000 commit authors contributing to the recent release Linux v5.6 (March 2020). These developers have a wide range of experience, with fully a quarter (285) having no previous commit to the Linux kernel. At the same time, the Linux kernel is self-contained, relying on no external libraries, meaning that developers cannot apply their previous experience to understanding its abstractions and data types. Indeed, the Linux kernel defines its own libraries of data types that meet the unique needs of an OS kernel in terms of efficiency (e.g., cache locality), flexibility, synchronization, and ease of use in typical OS-kernel usage contexts.

One Linux kernel data type is that of a doubly-linked list. Such lists are used for collecting all kinds of information: waiting processes, available devices, messages, and so on. They are very widely used in the Linux kernel, appearing in over 6400 files, and also serve as a basis for more complex data structures such as trees. Linux lists furthermore have some features that obscure their purpose and may surprise developers. They are implemented by a single data type, `list_head`, that is used to represent both the head of the list and to connect all of the list elements. This `list_head` type furthermore offers only `prev` and `next` fields, for referring to the previous and next `list_head`s, respectively, but no field pointing to the element value. Instead, the element must be implemented as a structure that embeds a `list_head` structure, which serves as a connector to the rest of the list. As a consequence, a `list_head` structure is associated with no type information about the list elements.

The features of Linux lists provide conciseness, efficiency and flexibility. There is only one list data type and only one set of list operators, regardless of the number of types of elements. Some of the operators are furthermore carefully designed to provide atomicity guarantees, thus potentially preventing some hard-to-understand concurrency bugs. The fact that the `list_head` structures are embedded in list elements may allow the connector to be in the same cache line as some element values, improving performance. In terms of flexibility, any `list_head` structure can be placed at the head of or as an element connector in any list. A single `list_head` variable can be used to store lists of different element types at different places in the code. A `list_head` structure can change role, from being a list head to connecting list elements. Still, while these features may be very convenient for experienced Linux kernel developers, they may make it very difficult for developers having less experience with the Linux kernel code to understand the role (list head or list-element connector) of `list_head`-typed variables and structure fields, or from a list head to know what type of elements the referenced list contains.

In this paper, we propose a type system and a toolset, LiLiput (Linux Lists program understanding toolset), to automatically infer the role and element type associated with Linux kernel `list_head` structures. Our approach relies on examination of how a given `list_head` structure is used by the various operators of the Linux

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416635

list API that differentiate between list heads and list element connectors, and imply relationships between them. We have used LiLiput to identify the roles (list head or list element connector) and list element types of 90% of the `list_head` type variables and structures in Linux kernel versions v3.0 (July 21, 2011) to v5.6 (March 29, 2020), demonstrating the versatility of the approach. The collected information furthermore permits detecting bugs where `list_head` structures are used according to the wrong role, which can result in loss of data. Specifically, LiLiput would have enabled detecting 8 previous bugs in Linux list usage, and detects 6 new ones. Our patches for the latter have mostly been approved by the Linux kernel maintainers.

Our main contributions are:

- We shed light on the problem of understanding Linux lists, which may constitute an obstacle to smooth maintenance; indeed, we show that Linux lists are widely used and powerful, but poorly documented.
- As a solution to this problem, we propose a toolset called LiLiput for inferring the types of Linux lists and for quickly achieving a global view of the involved data structures, thus facilitating maintenance.
- We validate our solution by demonstrating its effectiveness (roughly 90%), and further illustrating several possible uses of these types, including discovering list programming techniques and bugs.

The rest of this paper is structured as follows. Section 2 briefly introduces the rich world of Linux lists, and quantifies the lack of documentation about their uses. Section 3 presents our solution for facilitating the understanding of list uses and Section 4 describes its prototype implementation. Section 5 presents the validation of our solution, by showing its results on several Linux versions and several possible uses of the results. Section 6 situates this work with respect to other approaches, and Section 7 concludes and discusses future work.

2 LINUX LISTS

In this section, we first give an overview of the Linux list API, then present the history of this API in the Linux kernel, and finally highlight some advantages and disadvantages of its design.

2.1 API

The Linux list API comprises the `list_head` structure type and a large set of list-related operators. The `list_head` type is defined in `include/linux/types.h` and is shown in Figure 1.[1] It contains `prev` and `next` fields, pointing to the previous and next list elements, thus implementing a doubly linked list. It contains no field for the list element. Instead, an element is also represented as a structure, of some type, and embeds a `list_head` structure among its fields. The list operators are defined in `include/linux/list.h`. A representative subset of these operators is shown in Table 1. These operators allow traversing lists (lines 1-4), accessing list elements (lines 5-7), testing list elements (line 8), adding elements to lists (lines 9-11), and numerous other functionalities that are not shown.

[1]All code examples are taken from Linux v5.6 unless stated otherwise, and can be obtained from https://www.kernel.org/.

```
1 struct list_head {
2   struct list_head *next, *prev;
3 };
```

Figure 1: `list_head` structure type

```
1 struct hiddev {
2   int minor;
3   ...
4   struct list_head list;
5   spinlock_t list_lock;
6   ...
7 };
8
9 struct hiddev_list {
10  struct hiddev_usage_ref buffer[HIDDEV_BUFFER_SIZE];
11  ...
12  struct list_head node;
13  ...
14 };
```

Figure 2: `hiddev` structure type

A typical use of the Linux list API is found in the `hiddev` device library. This device library provides an API for describing human interface devices, a category of USB devices.[2] The list involves two kinds of structures, defined in Figure 2. The head of the list *i.e.*, a `list_head` structure pointing to the list's first and last elements, is embedded in a `hiddev` structure (lines 1-7) in the `list` field. The elements of the list are represented by structures of another type, `hiddev_list` (lines 9-14). The latter type also embeds a `list_head` structure, in the `node` field, connecting the element to the previous and subsequent ones. We say that `hiddev.list` is the *list head*, `hiddev_list` is the type of the *list element*, and `hiddev_list.node` is the *list connector*. Note that while in this example, the head of the list is also embedded in a structure, that is not the only possibility. A list head can also be a simple variable of type `list_head`. For instance, the file system infrastructure (`fs` directory) maintains a list of block devices in the global variable `all_bdevs` of type `list_head`, declared in `fs/block_dev.c`.

In practice, it is challenging to understand the relationship between the various structure types involved in a list. In the case of the `hiddev` list, the structure types are defined in different files, *i.e.*, `include/linux/hiddev.h` for `hiddev` and `drivers/hid/usbhid/hiddev.c` for `hiddev_list`. There are no comments in or near these definitions indicating which structure type represents the list head or the list elements, or connecting the one structure type to the other. Some information can, however, be inferred from the use of the various standard list operators on the `list_head` typed fields, `hiddev.list` and `hiddev_list.node`. For instance, the `list_add_tail` operator is used to add a `hiddev_list.node` element to a `hiddev.list` head. The `hiddev_list` elements in this list are traversed using a `list_for_each_entry`, and so on. Based on the information obtained from this manual inspection of the

[2]https://en.wikipedia.org/wiki/USB_human_interface_device_class

Table 1: A representative subset of standard list operators, defined in the C header file <include/linux/list.h>.

Operator	Meaning
list_for_each_entry(c, l, f)	traverses a list headed at l containing elements of structure type s, linked via their field $s.f$ of type list_head; the traversal uses a cursor c (of type s) as the current list element
list_for_each_entry_safe(c, c', l, f)	like list_for_each_entry(), but allows modifying the list (*e.g.*, by removing elements), without losing the cursor c, by using an extra cursor variable c'
list_for_each(c, l)	traverses a list headed at l containing elements of any type of structures; the traversal uses a generic cursor c (of type list_head); offers an untyped traversal, with respect to the operators above
list_for_each_safe(c, c', l)	like list_for_each(), but allows modifying the list, by using an extra cursor variable c'
list_first_entry(l, s, f) list_last_entry(l, s, f)	returns the first/last element of a list headed at l containing elements of type s, linked using the field $s.f$ of type list_head
list_entry(p, s, f)	returns the list element or head of type s, given a pointer p to its field $s.f$ of type list_head
list_is_first(e, l), list_is_last(e, l)	tests whether e (a pointer to a list_head) is the first/last element of the list headed at l
list_add(e, l'), list_add_tail(l, l')	adds a new first/last element e to the list headed at l
list_splice(l, l') list_splice_tail(l,l')	splices the list headed at l at the beginning/end of the list headed at l'; this consists in adding all the elements in l, but not the head

code, we can infer that Figure 3 illustrates an example of a typical hiddev list. Nevertheless, the relevant information is widely dispersed in the source code.

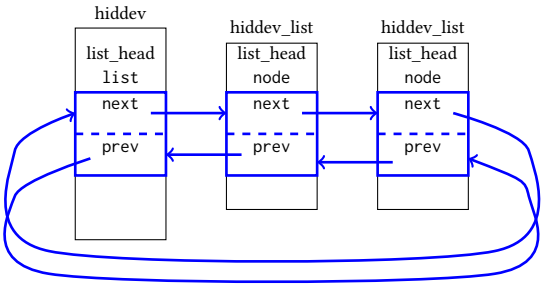

Figure 3: A typical list, headed at hidlist.list and containing hiddev_list elements connected by their node field.

2.2 History

The current Linux list API was first introduced into the Linux kernel in Linux 2.1.45, released in July 1997. The API was originally only used in three files related to file systems. It replaced another implementation of doubly linked lists based entirely on macros that generated list operations for specific data types, according to the types and field names mentioned in the macro arguments. The API introduced in Linux 2.1.45 included the list_head type, macros for declaring and initializing list_head variables, and the functions or macros list_add, list_del, list_empty, list_splice, and list_entry.

Figure 4 illustrates the use of lists over time, from their introduction up to the current release at the time of writing (v5.6). We have focused on the number of calls to the two operators for list insertion, list_add and list_add_tail, as to make a list it is necessary to add elements to it. Over time, many of the new uses of these list operators have been in code that adds new functionalities, but there are also instances where uses of other ad hoc forms of lists have been converted to use the standard API.[3] As can be seen,

[3]Examples: 68a2d76cea423, cdcdbe0b179a, df0933dcb027, 5b1a960d180e, available at git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

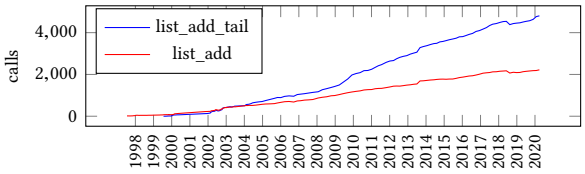

Figure 4: List usage in Linux versions v2.1.45-v5.6

there has been a steady increase in the use of the standard Linux list API, and this is likely to continue in the future. Therefore, a technique for helping maintainers to understand lists may prove more and more useful.

2.3 Documentation of list_head structures

As illustrated by the hidddev structure considered in Section 2.1, fields of type list_head are frequently not documented by appropriate comments, making it necessary to collect information from the code to understand what kinds of data structures they represent. To gain a more precise estimate of how often documentation is missing, we have built a code search tool finding all list_head structure fields and their associated comments. The search tool has been implemented with Coccinelle [8, 14] (more details about Coccinelle will be given in Section 4). It searches for comments on a field before, after, and in the middle of the field declaration, as well as summary "kerneldoc" comments before the entire structure declaration.

By applying this tool to the whole Linux kernel, we found a total of 8837 list_head fields in Linux v5.6, among which 3237 have at least one comment, amounting to 36%. This number has held steady since Linux v3.0 (July 2011), remaining between 35% and 39%, despite the number of such fields almost doubling in this time period. Moreover, by manually inspecting a small sample of the comments found adjacent to structure field declarations,[4] we found that less than 15% of these comments provide the complete list information, while the other 85% provide only partial information, lacking either the role of the field (list head or list element) or the related list_head field in the other structure. Consequently, we

[4]Included in the supplementary material, file ManualInspectionComments.xlsx.

estimate that less than 10% of the list-related fields are thoroughly documented by comments at the point where they are declared.

This lack of documentation has several consequences for code maintenance. Firstly, it makes it hard to gain a global view of the data structures used, when these structures are involved in lists. This difficulty is exacerbated in structures or sets of structures that heavily use lists, such as a task descriptor (`task_struct`), containing 15 `list_head`-typed fields. In Linux v5.6, there are indeed 5 structure types that contain more than 12 `list_head`-typed fields. Secondly, missing documentation increases the risks of errors during development, and increases the difficulty of spotting the corresponding bugs during maintenance. We hypothesize that various errors related to the list API may be favoured by missing or incomplete list information, including, for instance, using a list head as a list element, or vice versa, and extracting, accessing, or inserting the wrong type of elements from/into a list. Our evaluation in Section 5 will test this hypothesis.

3 INFERRING LIST TYPES

Given the challenges in understanding the uses of Linux lists, our aim is to assist this understanding with an automated tool, in order to simplify the development and maintenance of code using this data structure.

When maintainers examine a non-documented structure field declared to have type `list_head`, they face two questions at two different levels:

role is this field used as the head of a list or as a list element?
typing depending on the role of the field:
- for a list head: this list contains what type(s) of elements?
- for a list element: from what list heads is it reachable?

Similar questions arise for variables of type `list_head`. While variables typically serve in the list head role, as they do not contain any other data to represent a list element, we will see that there are cases where they are inserted as elements into other lists. Moreover, even when the role of a variable is clearly a list head, the question about the type of its elements remains. Therefore, we will use the generic term of *list name* to describe either a structure field or a variable of type `list_head`. Moreover, list names also include structure fields of type pointer to `list_head` and structure fields or variables of type array of `list_head`. In the latter case, we attribute the same type to all the elements in the array.

We now introduce the following notations. We use the letter s for a structure type (the name of a `struct` type), the letter f for a structure field name, the letter v for a local or global variable name, the letter x for any C expression, and the letter l for a list name of any kind, including a structure field name of the form $s.f$ or a variable name v of type `list_head`. Using these conventions, we will note $l : l'$ the statement "list name l is the head of a list whose elements are list names l'". For example, $s.f : s'.f'$ means that structure s contains, via its `list_head` field $s.f$, a list of structures s', connected via their own `list_head` field $s'.f'$. Thus, the type of the list in Figure 2 can be noted as `hiddev.list` : `hiddev_list.node`.

3.1 Typing rules

Our typing rules are based on the semantics of how the standard list operators use `list_heads` and related values. Note that developers

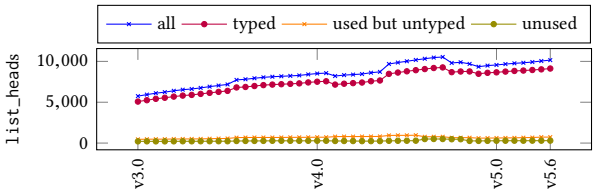

Figure 5: Overview of typing results

can circumvent the standard list operators, by accessing the `prev` and `next` fields of a `list_head` structure directly. We choose to build our typing rules by only taking into account standard list operators, because they constitute a finite set of code patterns, and are thus amenable to automation. They are also very frequently used. Still, this limitation may result in failing to type some list names. Section 5 (specifically, Figure 5) will examine to what extent this strategy is an issue in practice.

Table 2 shows the typing rules corresponding to some uses of standard list operators. The first column defines code patterns representing specific uses. Some patterns may impose a condition on the pattern variables, specified in a 'where' clause. The second column ("Strict typing") shows the typing information that can be inferred based on the use pattern. The third column will be described in Section 3.3. Note that the table is defined in terms of the notation defined above. For example, a reference to $s.f$ means a reference to the f field of an s structure, regardless of how that reference is expressed in the source code. Such information could be obtained by some combination of type inference and dataflow analysis, but our typing rules are independent of the means by which this information is obtained.

The first four sections in Table 2 concern various list traversal operators. Traversals use a cursor variable pointing either to a whole element structure (sections 1-2) or to only the `list_head` connector embedded in that structure (sections 3-4). The former style of traversal gives complete type information about the traversed list, including its head, the type of its elements, and the specific field within those elements that connects them together. For instance, the traversal `list_for_each_entry(s, l, f)`, indicates not only that l is a list head, but also that this list contains elements of type s (the type of structure pointed to by the cursor variable), connected through their field $s.f$ (the field name specified as the third argument). The latter style of traversal operators (sections 3-4) does not independently bring any useful typing information, as the operators use only a generic `list_head` cursor. However, even in such generic traversals it is common to access the element, via a call to `list_entry` on the loop cursor, in the body of the loop. In this case, we can recover the same information as in the first two sections.

Sections 5-6 in the table concern access operators on list elements: from the list head and from the list connector, respectively. An access from the list head, such as via `list_first_entry`, offers complete type information. An access from a list connector to the surrounding list element, via `list_entry`, on the other hand, provides no type information, since it does not involve both the list head and the list element. However, a call of the form `list_entry(s'.f'.next, s, f)` reveals the types of two neighbour elements. If the accessed fields are different (hence the **where**

Table 2: Typing rules based on list operator usage patterns.

	Code pattern	Strict typing	Permissive typing
1.	list_for_each_entry(s, l, f) list_for_each_entry_continue(s, l, f) list_for_each_entry_reverse(s, l, f) list_for_each_entry_continue_reverse(s, l, f) list_for_each_entry_from(s, l, f) list_for_each_entry_from_reverse(s, l, f)	l : s.f	l : L **where** s.f ∈ L
2.	list_for_each_entry_safe(s, s, l, f) list_for_each_entry_safe_from(s, s, l, f) list_for_each_entry_safe_continue(s, s, l, f) list_for_each_entry_safe_reverse(s, s, l, f)	l : s.f	l : L **where** s.f ∈ L
3.	list_for_each(v, l) { ... list_entry(v, s, f) ... } list_for_each_prev(v, l) { ... list_entry(v, s, f) ... }	l : s.f	l : L **where** s.f ∈ L
4.	list_for_each_safe(v, v', l) { ... list_entry(v, s, f) ... } list_for_each_prev_safe(v, v', l) { ... list_entry(v, s, f) ... }	l : s.f	l : L **where** s.f ∈ L
5.	list_first_entry(l, s, f) list_first_entry_or_null(l, s, f) list_last_entry(l, s, f)	l : s.f	l : L **where** s.f ∈ L
6.	list_entry(s'.f'.next, s, f) list_entry(s'.f'.prev, s, f) **where** s'.f' ≠ s.f	s'.f' : s.f	s'.f' : L' **where** s.f ∈ L
7.	list_is_first(s.f, l), list_is_last(s.f, l)	l : s.f	l : L **where** s.f ∈ L
8.	list_add(l, l'), list_add_tail(l, l') list_move(l, l'), list_move_tail(l, l') list_add_rcu(l, l'), list_add_tail_rcu(l, l')	l' : l	l' : L' **where** s.f ∈ L'
9.	list_splice(l, l'), list_splice_tail(l,l') list_splice_init(l,l'), list_splice_tail_init(l,l')	l = l'	l : L, l' : L' **where** $L' \supseteq$ L

condition), we have a list head and a list element, providing a complete list type. On the other hand, when s'.f' is the same as s.f, the developer may simply be getting the next list element starting from its predecessor element. Therefore, we cannot infer any type information in this case.

Section 7 in the table concerns functions where one argument is a list connector and another is a list head. In this case, we can infer type information if the list connector can be described as $s.f$, based on any available type and dataflow analyses. If the list connector is just described as an arbitrary expression of the `list_head`, then no information can be inferred. Note that the operators of this section are different from those in section 5, where the structure type s and the field name f are separate arguments to the operator, thus always providing typing information.

Sections 8-9 in Table 2 handle operators inserting elements into a list. Section 8 concerns operators inserting individual list connectors, either external to the list or moved within the list. These operators involve both a list connector and a list head, so if the list connector and the list head can be described as a list name, they provide complete type information. Section 9 concerns operators splicing a whole list headed at l into a list headed at l'. This operator does not provide information about the types of the elements in either list. However, the lists should normally have the same element types, because the `list_head` traversal operators do not support polymorphic lists. Therefore, we add an equality *constraint* between the two types. As soon as some other operator uses allow typing one of the lists, the other list will become typed, too.

3.2 Experience with the strict typing rules

When we applied the above typing rules to the whole Linux kernel (Linux v5.6), we obtained 57 typing errors, where different types were required to be unified to represent a list element type. Most of these errors, however, are not programming errors, but rather fit into one of the following categories.

First, there are (ab)uses of some operators that do not correspond to their documentation, but that work fine. For instance, the operator `list_first_entry(l, s, f)`, that is documented to return the first element of a list, is actually just a macro that expands to `list_entry(l.next, s, f)`. Consequently, this operator may also be applied to a list element to get the following element. When used this way, our type inference rule will wrongly try to type $s.f$ both as a list head and as a list element. Another example is the use of `list_add_tail`, whose second argument is normally a list head, but that can also be used to insert an element before another one, e.g., to maintain a sorted list. In fact, most of the list operators can be used in non-standard ways by inspecting their implementation.

Secondly, many functions define local list variables l that are used to collect *e.g.* a list of elements of some type s, that are consumed in a loop, and then the same list variable is reused for a list containing another type of elements s'. This is the case, for instance, of function `_scsih_remove_unresponding_devices()`, which fills a local variable `head` with `_sas_device` structures, then deletes them all, and then does the same with `_pcie_device` structures. This kind of situation results in a type conflict between two possible types $l : s.f$ and $l : s'.f'$.

Thirdly, there are cases where the elements of the list may be different structures depending on the actual managed device, firmware, that is used, etc. For example, this is the case in the Qualcomm Atheros driver for wireless connectivity defined in directory `driver/net/wireless/ath/ath10k`. Here, depending on whether the device version is greater than 10.4 or not, the structure `ath10k_fw_stats` contains in its field `vdevs` a list of either `ath10k_fw_stats_vdev` or `ath10k_fw_stats_vdev_extd` structures (linked by their field `list`). The choice is done dynamically based on flags such as `WMI_10_4_STAT_PEER_EXTD`.

Finally, for a list stored in a structure field, there may be several instances of this structure, where each instance contains the same type of elements, but different instances contain different types

of elements. The different instances are sometimes managed by different functions in the code, specific to device version numbers.

All of these situations manifest as errors in our type system.

3.3 Permissive typing rules

As the strict typing rules produce too many type errors that turn out to be false positives, we redesigned the typing rules so as to infer types that cover all the existing uses of lists in the code. For instance, when encountering a list operator adding some structure of type s to a list head l with elements that are structures of type s', we relax the type of l to allow elements of both types s and s'.

We model the cases where a list name is used for containing different types of list names using a disjunctive type $l' : l_1|...|l_n$. We will note with letter L such a disjunction. The corresponding relaxed typing rules are given in last column of Table 2. Each strict rule inferring type information of the form $l' : l$ has a relaxed counterpart that infers $l' : L$ **where** $l \in L$. Similarly, a rule inferring a type equality constraint $l = l'$, such as the rules at the bottom of the table, has a relaxed counterpart inferring a type inclusion constraint $l : L, l' : L'$ **where** $L' \supseteq L$.

As opposed to the strict typing, the relaxed typing rules never lead to typing errors. Indeed, any finite set of type inclusion constraints has at least one solution, for example, typing each list name to a disjunction of all list names, including itself. Moreover, it has a least solution, with respect to the type inclusion relation, which can be computed as a fixed point. Thus, the relaxed typing rules compute for each list the least type permitting all of its uses.

The fact of never generating type errors is compatible with our primary goal, that is, understanding the role and type of lists. The only risk is that of resulting in types that are too general to be useful. In the evaluation section, we find that computing the least permissive types does not prevent our system from being useful for indicating potential programming errors. Indeed, a developer may have the intuition that a certain list head should refer to lists with only one type of element, in which case, inferring a disjunctive type for it might help spot a potential problem.

4 IMPLEMENTATION

We have implemented the permissive typing rules presented above in a program-understanding toolset for Linux lists called LiLiput. LiLiput consists in two main tools, whose goals are, respectively:

List detection A first tool finds uses of lists: declarations (structure fields and scalar variables) and operations (standard list functions and user-defined list functions). This tool outputs a list usage report.

Type inference A second tool infers types for the lists based on their usage, starting from the list usage report. It outputs a typing report.

Each tool provides a textual report that can be easily read, inspected, and amended if necessary, by developers. Moreover, we show that these reports may also be exploited automatically by other program understanding tools, for data structure visualisation or interactive list usage inspection.

The *list usage report* contains a section for each list name, containing the list name followed by information about each use of an operator that has an expression represented by the list name among its arguments. Arguments not related to lists, or whose list name could not be uniquely determined, are abbreviated as '?'. For instance, the following section describes the uses of a local variable `dead` of type `list_head` declared in the function `exit_notify`:

```
dead@exit_notify:
  forget_original_parent(?, dead@exit_notify): kernel/exit.c:708
  list_add(task_struct.ptrace_entry, dead@exit_notify):
    kernel/exit.c:728
  list_for_each_entry_safe(task_struct, ?, dead@exit_notify,
    ptrace_entry): kernel/exit.c:735
```

The first call is to a user-defined list operator, and the next two calls are to standard list operators.

The *typing report* contains the type for each list name whose type could be inferred, as a disjunctive type of the form: $l : l_1|l_2|...l_n$. For instance, the following excerpt gives the type inferred for the local variable above:

```
dead@exit_notify: task_struct.ptrace_entry
```

The following excerpt gives the type of the local variable already mentioned in Section 3.2, used to contain first `_sas_device` structures, and then `_pcie_device` structures.

```
head@_scsih_remove_unresponding_devices:
          _pcie_device.list | _sas_device.list
```

We present the two main tools below, and then briefly present two other program understanding tools that we have developed based on the collected information.

4.1 List detection

We collect information about the declaration and usage of lists using the tool Coccinelle [8, 14]. Coccinelle is a program matching and transformation tool for C code that has been extensively used on the Linux kernel. Coccinelle allows searching for patterns of code expressed as fragments of C code parameterized by *metavariables*. It does not require macro expansion, but instead relies on heuristics to parse the C code in the presence of macros and ifdefs. It is thus independent of the chosen configuration options (ifdefs), of which there are many in the Linux kernel [23]. This feature enables covering over 99% of the 18.5 million lines of C source code found in Linux v5.6. The use of Coccinelle also allows rapid prototyping of complex code search rules. Nevertheless, there is a risk of some omissions, either because the rule set is not complete or because the parsing heuristics failed on some code. Nevertheless, for Linux v5.6, the list detection tool finds information about 8,562 list-typed structure fields, 2,545 list-typed variables, and 68,541 function or iterator calls having list-typed arguments, and thus we believe that our Coccinelle scripts cover a very large percentage of the list structures and operations found in the Linux kernel.

Collection of the information proceeds in two phases. The first phase collects information about structure fields of type `list_head`, pointer to `list_head`, or array of `list_head`, as well as global variables of any of these types, and local variables of type `list_head` and array of `list_head`. We omit local variables of type pointer to `list_head`, because these typically correspond to temporary pointers to elements of an existing list. The Linux kernel may declare multiple structure types with the same name, which may have some fields of the same name. As a heuristic, we keep such fields when all definitions of the structure have the same set of list-typed fields, as

this often corresponds to different variants of a structure for different variants of a device or different hardware architectures. On the other hand, when two definitions of the same structure name have a list-typed field that appears in both definitions, but the complete set of list-typed fields defined by the two structures is different, we discard the common list-typed field as being ambiguous, on the assumption that the field may have different roles in the two structures. For Linux v5.6, we discard information about 20 variables and structure fields, due to ambiguity. The second phase then collects information about the arguments of functions and iterators that refer to the fields that were identified in the first phase. The collection is implemented in 1250 lines of Coccinelle code. Most of the code is due to the many variations in nested unions, structures, etc., and is not challenging to write.

The collection of functions and iterators that have list arguments does no dataflow analysis, neither across function calls nor within function definitions. This reduces the information available about less than 2% of function calls and iterators (Linux v5.6). It should be possible to reimplement the information collection phase in a dedicated program analysis framework such as LLVM to take such information into account, although with the inconvenience that such tools are sensitive to preprocessor configuration options. For this work, we favored the rapid prototyping and high coverage advantages of Coccinelle, as Linux kernel code more typically refers to lists explicitly.

Our list detection tool misses 1) nameless struct declarations in typedefs, 2) nameless struct declarations in variable declarations, and 3) some structure fields that are deeply nested in other structures and unions. 50 `list_head` structure fields out of 8837 are overlooked in Linux v5.6 due to these limitations. Our tool runs on Linux v5.6 in around 2 hours on a 44 core machine with Intel Xeon 2.20GHz CPUs and 251 GB memory, with the kernel source code in a memory-backed file system to eliminate the cost of disk accesses.

4.2 Type inference

The type inference tool takes as input the list usage report and outputs the typing report. As both these reports are textual, we chose to code this tool in a scripting language offering rich possibilities of text processing, namely Perl. The implementation consists of 300 lines of Perl, and is straightforward. It maintains a map from list names to sets of list names, representing disjunctive types. The map is computed as the fixed point of the constraints generated by each permissive typing rule in Table 2. As a complement to the typing report, it prints statistics about list usage, including the total number of list names, the number of list heads, list elements, untyped list names, etc.

4.3 Other tools

The information computed in our reports can be exploited beyond retro-documenting the list declarations. To illustrate this fact, we have prototyped two extra tools in our toolset.

Firstly, a visualisation tool takes the list typing report as input and produces graphs of all the list data structures. This allows maintainers to have a global view of list usage in a given module where lists are used extensively or to form particular patterns. Indeed, we identified many instances of complex data structures implemented with several lists, such as simply and doubly linked trees of fixed or variable height. We also found header files and/or structures containing many list fields. For instance, the heavily used header file `include/net/devlink.h`, included in 50 C and header files, contains 16 list fields, among which 10 are defined in a single, central structure called `devlink`.

Secondly, a list usage exploration tool takes the list usage report as input, and builds a hypertext-based listing containing a subset of the detected list usages corresponding to a filtering criterion. For instance, one filter that proved to be useful during the development of the list usage report tool selected all the list operator uses where some list arguments could not be uniquely determined as list names (abbreviated as '?' in the report). This helped us to manually inspect complex list usage patterns, and discover some stereotypical programming techniques that could be recognized and leveraged for improving the list usage reports. One such technique was the complex pattern for generic list traversal using `list_for_each`, described in the third section of Table 2, where the typing information could be reconstituted from several related sub-patterns.

Moreover, this exploration tool could be extended to take as additional input the typing report. This would allow performing more complex searches, such as selecting all the list initializations (using operator INIT_LIST_HEAD) acting on a list element instead of a list head. Beyond our own use for tuning our analyses, the list usage exploration tool might be useful for maintainers, for example after fixing one bug to search for similar list misuses to fix.

5 EVALUATION

The primary goal of our evaluation is exploratory, to see what we can learn about list usage from the information inferred by our type system. We also assess the accuracy of the produced information. To meet these goals, we consider the following research questions:

RQ1: To what extent is our type inference system effective: are the identified types the correct ones, and do all lists receive types?

RQ2: To what extent does our type inference system provide added value, going beyond the information already available in the comments and source code?

RQ3: What programming techniques for the use of lists are revealed by the inferred type information?

RQ4: What kinds of bugs can be detected using the inferred type information, and how helpful are the tools provided with LiLiput in finding such bugs?

5.1 Effectiveness

To obtain a broad view of the effectiveness of LiLiput (**RQ1**), we applied it to all of the Linux kernel releases from v3.0 (March 2011) to v5.6 (March 2020). Figure 5 presents an overview of the results. LiLiput found 5747 list names in v3.0, and their number increased to over 10,000 in v5.6. LiLiput could type between 87% and 91% of these list names in each kernel version.

There are two cases in which a list name does not receive a type. One case is where LiLiput finds a list declaration but finds no list operator, whether standard or special-purpose, that is applied to the list name. In this case, LiLiput reports that the list name is unused. This is the case for 200-500 list names depending on the version,

Table 3: Inferred list types

Experiment	Typed (total)	Head only	Element only	Head & element
v4.19	8601	4797 (55.8%)	3600 (41.9%)	204 (2.4%)
v5.6	9125	5078 (55.6%)	3823 (41.9%)	224 (2.5%)

and 2.9% of the list names found in Linux v5.6. However, LiLiput could wrongly flag a list name as unused if the list name is used outside of function calls. We manually inspected a small sample[5] of cases involving list variables local to a file (which cover roughly one third of the cases in v5.6), because it is easy to find all uses for such cases. In this sample, we found a few false positives where the lists are manipulated without using any list operator, such as assigning them from, or comparing them to, other list variables. In the sample, 87% of the list names reported as unused were true positives, that is, the lists are indeed never used. This information about unused lists can be useful for kernel developers in cleaning up the code base.

There are also between 6.4% and 9.7% of list names (7.2% in Linux v5.6) that are used with list operators but could not be typed, as they do not satisfy any of the list operator patterns in our typing rules (Table 2). This can occur when the list name is used by list operators where no list name can be identified for some of the other arguments used by the typing rule, *e.g.*, when these other arguments are represented by local variables or function parameters. The kernel developer may nevertheless benefit from having information about the uses of the list name summarized in the list usage report. On this basis, some relationships not taken into account by the type inference system, but known to the developer of a specific subsystem, may help the developer identify the properties of the list structure.

We focus now on two recent versions, Linux v4.19 (October 2018) and Linux v5.6 (March 2020), to give more detailed results.

Table 3 details the typed list names found in v4.19 and v5.6 by presenting the different kinds of types that have been inferred for them. As can be seen, a bit less than 60% of these list names play the role of list head, while a bit more than 40% play the role of list elements. Thus, while there is a lot of variation in practice, on average, a given type of list element is found in several lists (about 1.5). This typically corresponds to the local variables that temporarily store elements, which are permanently stored elsewhere: either in a heap data structure or in a global variable.

Apart from these simple cases where a list name is either a head or an element, around 2.5% of the list names are both used as a list head and as an element in one or more lists. Table 4 further subdivides this special case (by reproducing the "Head & element" column in Table 3), isolating two frequent sub-patterns of such lists: self-lists and mutual pairs. Self-lists are list names containing only themselves: $l : l$. Mutual pairs are pairs of lists containing each other, and nothing else: $l : l' \wedge l' : l$. These particular cases are worth investigating, and are described later. The few remaining cases (under 20) are more complex typing patterns, in which a list name is used as a head but also as an element in several lists.

[5]Included in the supplementary material, file ManualInspectionUnused.xlsx.

Table 4: Uses of list names that serve both as a list head and as list elements

Experiment	Hd & elm (total)	Self-lists	Mutual pairs	Other cases
v4.19	204	164 (80.4%)	11 (10.8%)	18 (8.8%)
v5.6	224	179 (79.9%)	13 (11.6%)	19 (8.5%)

5.2 Added value

In the absence of our type system, developers can obtain information about lists from the documentation, typically comments in the code, and from any naming conventions used in the code itself. We thus next consider the degree to which our approach provides added value (**RQ2**), i.e., information that is not easily accessible to the developer already.

In Section 2.3, we have already shown that lists are rarely documented. Based on a sample of 22 Linux v5.6 `list_head` structures[6] that are associated with some comments, we have also found that when they are documented, the documentation is often incomplete. For one documented field (5%), LiLiput inferred no information, and for another field (5%) the amount of information inferred by LiLiput is the same as the amount of information in the associated comment. In all of the remaining cases, *i.e.*, 20 out of 22 (91%), the information inferred by LiLiput was consistent with the existing comments and was more complete. For example, `struct vpe` defined in file `arch/mips/include/asm/vpe.h` contains a field named `tc` of type `list_head`, whose comment is "tc's associated with this vpe". This suggests that `vpe.tc` is a list head, containing structures of type `tc`, but does not indicate which is the connector field within `tc`. As `struct tc` contains two `list_head` fields, the information in the comment is incomplete; it could be represented as *vpe.tc : tc.?*. LiLiput infers the complete type information *vpe.tc : tc.tc*.

Based on the information inferred by LiLiput, we can furthermore check the correspondence between list names and the inferred types. We studied the most frequent names of list fields and found that the name 'list' is by far the most common one, covering roughly 25% of the list fields. Our typing results show that structure fields named 'list' usually serve as list connectors (85%). This leaves however a non-negligible fraction (15%) that are used as list heads. Overall, the naming convention is not fully reliable for distinguishing the role of list fields. On the other hand, the second most frequent list field name is 'node' (5% of the list fields). Our types show that this name is always used for identifying list connectors. However, as only 5% of the list connectors follow this convention, the information inferred by LiLiput is overall still useful.

These results show that our type inference provides information not already easily available to kernel developers. Our typing results can be used to improve the documentation about specific list-typed variables and structure fields, and to retro-document naming conventions.

5.3 Programming techniques

Studying the collected type information reveals some interesting programming techniques used in manipulating lists (**RQ3**). We

[6]Included in the supplementary material, file ManualInspectionComments.xlsx

have seen in Section 3.2 that some list heads are associated with more than one list element type (57 out of 5078 list heads (1.1%) in Linux v5.6), and in Table 3 that some list heads are also used as list elements (2.5% of all typed `list_head` structures in v5.6). All of these cases result in LiLiput inferring disjunctive types. These *type smells* (analogous to code smells [4]) do not necessarily indicate bugs. However, their uncommon nature may convey specific information, which might be worth investigating. Indeed, by studying samples of each of these type smells, we were able to uncover some list programming techniques.

A number of the disjunctive types (6 out of 57 in v5.6) contain a local variable among the elements. There are several list programming techniques that involve adding a local `list_head` variable to a list located in a structure field. A first technique, motivated by concurrency, consists in atomically moving all the elements in the list towards a local list variable in bounded time while holding a lock, and then liberating this lock and processing the elements outside of the critical section. This technique is illustrated in the function `gfs2_jindex_free`, whose local variable `list` is involved in the following disjunctive type:

```
gfs2_sbd.sd_jindex_list: list@gfs2_jindex_free | gfs2_jdesc.jd_list
```

This function clears all the journal index information contained in the `gfs2_jdesc` elements, by first adding variable `list` in the original list, and then removing and emptying the original list head using the operator `list_del_init` (this operator is normally used for removing list elements, not list heads). The sequence of these two operators is protected by a lock. After the lock is released, the elements in the local variable `list` are consumed.

A second technique involving local variables is related to a particular kind of lists. While most lists comprise a distinct list head followed by a series of list elements, in some lists, all of the `list_head` structures are embedded in structures of the same type and are used in the same way. This results in a list type of the form $s.f : s.f$, where the list head and the element connector are the same structure field, amounting to a self-loop. We refer to such a list as a *ring*. As the list traversal operators assume the existence of a list head, which should not be traversed, these operators cannot be used with rings. To enable traversing a ring, a common programming technique is to temporarily add a local variable of type `list_head` within the ring. This variable, usually called 'head', is used to traverse all the elements in the ring, and is then removed. This technique results in a disjunctive list type such as the following, which results from applying the technique in two different functions, `lpfc_bsg_ct_unsol_event` and `lpfc_ct_unsol_event`, both of which use a local variable named `head`.

```
lpfc_iocbq.list: head@lpfc_bsg_ct_unsol_event |
  head@lpfc_ct_unsol_event | lpfc_iocbq.list
head@lpfc_bsg_ct_unsol_event: lpfc_iocbq.list
head@lpfc_ct_unsol_event: lpfc_iocbq.list
```

A ring can also be used to implement a doubly linked tree of height one, which we refer to as an *umbrella*. An example of an umbrella is the widely used kernel representation of a performance event, implemented in structure `perf_event`, whose computed type is circular (a self-list):

```
perf_event.child_list: perf_event.child_list
```

This structure contains a list of other performance events that are considered its children. For implementing this list, the same field `child_list` is used in the parent as a list head, and in the children as a connector. Unlike in a simple ring, each child in the umbrella points back to its parent using a different field (`parent` in `perf_event`), while the parent points to itself. We found 20 umbrellas in Linux v5.6 by inspecting instances of self-loop types.

Lists are also used to implement trees of arbitrary depth of some structure s. Such trees use two different fields of s, one always being a list head, and the other being the connector between the children. An example of a tree is found, for instance, in a central data structure of the file system infrastructure, called `dentry`, representing a directory entry, whose computed type is:

```
dentry.d_subdirs: dentry.d_child
```

Thus, any `dentry` contains in field `d_subdirs` a possibly empty list of children, and is part itself of a list via the distinct field `d_child`. There are 47 such $tree(s)$ types in Linux v5.6, i.e., types of the form $s.f : s.f'$ where $f' \neq f$. Their number has varied between 33 (in v3.0) and 60 (in v4.14).

Finally, by studying several examples of mutual pairs, we found another list programming technique. For example, the Marvell Wireless LAN device driver contains two mutual pairs:

```
mwifiex_ra_list_tbl.list: mwifiex_tid_tbl.ra_list
mwifiex_tid_tbl.ra_list: mwifiex_ra_list_tbl.list
mwifiex_bss_prio_node.list: mwifiex_bss_prio_tbl.bss_prio_head
mwifiex_bss_prio_tbl.bss_prio_head: mwifiex_bss_prio_node.list
```

In fact, it is the second type of each pair that gives the correct type: in both cases, the `list` field is used as a connector. The reverse type is due to a list programming technique used in function `mwifiex_rotate_priolists()` that implements a round robin algorithm. Namely, after a packet is successfully transmitted, both lists are rotated so the packet next to the one transmitted will come first in the list. This is done by moving the list head right after the transmitted packet, using operator `list_move`. Based on this usage not conforming to the documentation, our typing tool infers that the list head is an element of the list headed by its (real) element.

5.4 Bugs in list usage

While we have seen that some disjunctive types correspond to useful programming techniques, others do correspond to bugs (**RQ4**). For instance, we manually inspected the code causing the following type:

```
spu_gang.aff_list_head: spu.aff_list | spu_context.aff_list
```

The inspection revealed that the only correct element type is the second one. The first element type (spu.aff_list) is only due to a traversal of the list head with a cursor variable that is incorrectly declared to have the type 'spu' instead of 'spu_context'. Incidentally, the traversal is only counting the elements, so this bug currently has no consequences, but it is a latent source of errors if some other processing is added within the traversal body.

A common source of mutual pairs is a call to 'list_add' or 'list_add_tail' with the arguments in the wrong order. Indeed, both arguments to these functions have type `list_head` and the developer has to remember that the first argument should be a list connector and the second argument should be a list head. Not respecting this

order can lead to information loss and memory leaks, if the argument provided as a list connector is actually a list head pointing to a non empty list. One such case is the following mutual pair, in the driver for the Phyter precision time protocol transceiver:

```
dp83640_clock.list : phyter_clocks
phyter_clocks : dp83640_clock.list
```

Here, the correct type is the second one: 'phyter_clocks' is a global variable containing structures of type 'dp83640_clock'. The other type is due to a use of the 'list_add' operator in which the arguments are reversed, introducing the global variable as an element in the list headed within the structure, while the intention was the opposite.

Overall, we identified 6 bugs in Linux v5.6 based on type smells. We have submitted patches for 5 of them. Four of these patches have been approved by the Linux kernel maintainers. The fifth has received no response so far. Indeed, the code contained two errors, and the patch was incomplete, as it was not clear how to fix one of the errors. In the remaining case, we have informed the kernel developers of the issue, but we do not know how to fix it and are thus unable to propose a patch. Nevertheless, all of our approved patches received quick responses, suggesting that inconsistencies in the use of lists is an issue that Linux kernel developers are concerned about.

To further evaluate the potential utility of our type information, we searched for past patches to the Linux kernel that changed the arguments of the 'list_add' and 'list_add_tail' operators. The goal was both to see whether this is a recurring bug pattern, thereby potentially benefitting from a tooled approach, and whether our type information could have helped to find them. We found 11 such patches using the patch query tool Prequel [9]. This seems to confirm that such bugs are recurrent. 10 of the 11 patches swap the arguments and the remaining one changes the target list head. For 8 patches out of the 11, we have checked the types inferred for the release of the Linux kernel prior to the patch and found that the types contain a mutual pair, as illustrated in the Phyter example discussed above. Thus, the type information could be helpful in these cases. For 2 other patches, we do not have any type information because the relevant code was not present in the release prior to the patch, and we have only run LiLiput on kernel releases. For 1 patch, we were not able to obtain the relevant version using `git describe`, and thus it was not possible to find the associated type information. Summarizing, we have evidence that a large majority (8/11) of the previous bug fixes of argument inversion for list insertion could have been spotted by our typing tool.

By putting together the previously fixed bugs and the new bugs we identified, we can validate the hypothesis formulated in Section 2.3 (that missing or incomplete list information may favour errors such as confusing heads and elements, or inserting/using wrong type of elements). Indeed, bugs inverting the arguments of a `list_add` operator are instances of the head/element confusion, and have the effect of inserting a wrong kind of element into a list; and using a cursor variable of a wrong type may cause erroneous element extractions and accesses.

5.5 Visualisation example

We conclude by considering how the tools provided with LiLiput can help understand lists and find list usage bugs (**RQ4**). Figure 6 shows the result of the visualization for the structures involved in the Phyter precision time protocol transceiver bug. Each box represents a variable or a structure type. Ovals in the box represent the variable or the structure fields, respectively. Blue ovals are at least list heads, possibly also list element connectors, while black ovals are only list element connectors. In this case, we can easily see the mutual pair related to the bug discussed above.

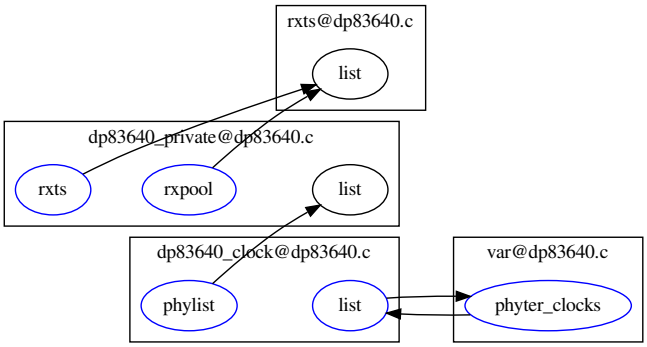

Figure 6: Visualization of the structures involved in the Phyter precision time protocol transceiver example

To facilitate studying such uncommon cases, the visualization tool can be configured to generate only graphs that contain self-lists and only graphs where a list head has another list head as a list element, as is the case of a mutual pair. It is also possible to specify the name of a structure or field that must appear in the graph, to focus on a single example.

6 RELATED WORK

Templates in C++ [19] and Java Generics, which were introduced in 2004, enable defining an API that is parameterized over some types. These features make it possible to define a list type that can be used with different types of elements, such that type checking ensures that all elements have the same type. In the context of Java, an automatic refactoring was made available to use such types in Eclipse 3.1, released in June 2005 [5]. Still, a study of 20 widely used Java projects in 2011 [16] showed that the adoption of generics in these projects was spotty, often relying on the motivation of a single developer and often addressing a limited set of types, *e.g.*, lists of strings. A subsequent study in 2014 [1] considering 31K Java projects from SourceForge found that 57% of the projects instantiated existing generic types, but less than 10% of the total set of files from the 31K projects did so, again suggesting some reticence in using these features. The Linux developers have aggressively adopted a list implementation that does not incorporate the element type. However, Linux lists also include the notion of a list head, which brings some extra expressivity, as noted in Section 5.3, as compared to simply parametrizing a list by a type.

As a large and critical code base, the Linux kernel has been the subject of numerous efforts to mine properties of the source code. Engler et al. [2] initiated the field of *protocol mining* in the Linux kernel, by finding commonly occurring sequences of function calls, and then identifying as bugs sequences of function calls that only partially matched these patterns. Subsequent work improved the

accuracy of the approach [7] and extended it beyond function calls [11]. Saha et al. [17] eliminated the requirement that identified sequences of function calls occur commonly, enabling finding bugs in subsystem-specific function protocols. Lu et al. [12] mine correlated variables, such as a variable containing a string and another variable storing its length, and detect bugs where the variables are not updated consistently, and race conditions when the variables are updated without holding locks. Lawall and Lo [10] mine uses of names declared with `#define`, use clustering to group them into types and then use the types to identify names used in invalid contexts. We are not aware of any mining-based type inference approaches that have been applied on the Linux kernel to generic structures such as lists.

Our type inference system complements the C type `list_head` with additional annotations, describing their use as list elements or heads. This is similar to the approach of type qualifier inference. Type qualifiers [3] complement programming-language types with extra attributes, typically from a finite set. They were initially illustrated by inferring `const` annotations in the C language. Other applications concerned ownership [6] or security [18, 24] properties, among others. From standalone typing tools, they evolved into compiler plugins, through the notion of pluggable types, as in the Checker Framework for the Java language [15]. More recently, these batch-oriented tools were criticized as being inflexible, by not allowing the user to modify the code to fix qualifier conflicts; an interactive tool was proposed as a solution [20]. In our proposal, the complementary typing information is not derived from a fixed set, but rather in the namespace of C types, or subsets thereof. Our tool is implemented as a batch tool, but the access to the intermediate reports enables maintainers to actively participate in the typing process, for instance by filtering or modifying the computed types according to their domain expertise. In the implementation, we use a combination of typing rules, dataflow analyses, and heuristics to reverse engineer the complementary typing information. It is an interesting question whether the implementation of our analyses could be simplified by leveraging an existing framework for pluggable types.

A different approach to type inference for program understanding was pioneered by the tool Lackwit [13], also targeting C code. Lackwit infers types that describe how values flow between variables: i.e., it gives two variables the same type if values can be passed between them. LiLiput focuses only on list variables, concerning their role in list data structures. It reveals connections between data structures rather than how values flow.

7 CONCLUSION AND PERSPECTIVES

We have presented a toolset for retro-documenting Linux kernel lists with type annotations. Annotations are successfully inferred for 90% of the total list structures defined. The information provided by the inferred types can help understanding data structures and build graphical representations of complex data structures. By studying uncommon typing patterns (type smells), the provided information can also help spot possible bugs or identify complex programming techniques involving lists.

One perspective for continuing this approach would be to use our strict typing rules to enforce a more disciplined use of lists. This would require developers to duplicate some local list head variables, to account for list elements of different types at different times, and create some dummy structures around local list head variables temporarily used as list elements. A necessary pre-requisite would be to investigate whether kernel developers would be interested in a solution requiring more typing discipline but offering more runtime guarantees.

On the other hand, there are several possible generalizations of our type inference technique and its applications.

One direct extension of our type inference system would be to use our typing results as a base for inferring higher-level types, that would convey richer information for maintainers. Such higher-level types could include $ring(s)$, $tree(s)$, and $umbrella(s)$, respectively for rings, doubly-linked trees, and umbrellas of some structure s, as defined in Section 5.3. Recognizing such types would need some new inference rules e.g., for identifying the up-links in trees or umbrellas.

Another direct extension of this work would be to consider other kinds of lists defined in the kernel libraries, covering more specialized needs. Examples include LRU lists for implementing caches (defined in file `list_lru.h`), and BL lists used in scalable hash tables (defined in `list_bl.h`).

Furthermore, data structures other than lists could be considered, such as red-black trees (defined in `rb_tree.h`).[7] Like Linux lists, red-black trees are formed by embedding a connector structure called `rb_node` within some other data structures forming a sorted, balanced tree. Unlike Linux lists, however, there is a second connector data structure called `rb_root` that represents the tree root. Red-black tree connectors are embedded in 70 different data structures in the kernel. Inference rules could supplement lacking documentation (indicating which type of elements correspond to a root and vice versa), and enable checking whether only the right kind of structures are inserted or searched for in a given tree.

One way of going further with the generalization is to focus on the idea of generic structure embedded in various other structures, which abstracts the notion of connectors in Linux lists. A promising indicator for such patterns is the use of the `container_of` macro, if used for retrieving different container structures starting from a generic structure. Examples of such generic structures include `callback_head`, `kobject`, `kref`, `rcu_head`, etc. Using some simple rules in Coccinelle, we found 454 generic structures that are used with the operator `container_of` for retrieving at least two different container types. This seems to indicate a great potential for cases where type inference techniques could be useful.

Finally, the approach could be explored for other OS kernels. For example, the use of connectors embedded in data structures for improving the cache locality of linked structures is a recurring design pattern in highly optimized system software.

Availability. The supplementary material, available both as an evolving Git repository [21] and as a specific version for reproducibility purposes on Software Heritage [22], contains: the list detection and type inference tools, the type inference results, the manually inspected samples, and the visualization of self lists and of list names that are both list heads and list elements (v5.6).

[7] https://www.kernel.org/doc/html/latest/core-api/rbtree.html

REFERENCES

[1] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining billions of AST nodes to study actual and potential usage of Java language features. In *ICSE*. 779–790.

[2] Dawson R. Engler, David Yu Chen, and Andy Chou. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*. 57–72.

[3] Jeffrey S Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 192–203.

[4] M. Fowler. 1999. *Refactoring-Improving the Design of Existing Code*. Addison-Wesley.

[5] Robert M. Fuhrer, Frank Tip, Adam Kiezun, Julian Dolby, and Markus Keller. 2005. Efficiently refactoring Java applications to use generic libraries. In *ECOOP*. 71–96.

[6] Wei Huang, Werner Dietl, Ana Milanova, and Michael D Ernst. 2012. Inference and checking of object ownership. In *European Conference on Object-Oriented Programming*. Springer, 181–206.

[7] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Y. Ng, and Dawson R. Engler. 2006. From uncertainty to belief: Inferring the specification within. In *OSDI*. 161–176.

[8] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 years of automated evolution in the Linux kernel. In *USENIX ATC*. 601–614.

[9] Julia Lawall, Derek Palinski, Lukas Gnirke, and Gilles Muller. 2017. Fast and precise retrieval of forward and back porting information for Linux device drivers. In *USENIX ATC*. 15–26.

[10] Julia L. Lawall and David Lo. 2010. An automated approach for finding variable-constant pairing bugs. In *ASE*. 103–112.

[11] Zhenmin Li and Yuanyuan Zhou. 2013. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE*. 306–315.

[12] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. 2007. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*. 103–116.

[13] Robert O'Callahan and Daniel Jackson. 1997. Lackwit: A program understanding tool based on type inference. In *Proceedings of the (19th) International Conference on Software Engineering*. IEEE Computer Society, 338–348.

[14] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys*. 247–260.

[15] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. 2008. Practical pluggable types for Java. In *Proceedings of the 2008 international symposium on Software testing and analysis*. 201–212.

[16] Chris Parnin, Christian Bird, and Emerson R. Murphy-Hill. 2011. Java generics adoption: how new features are introduced, championed, or ignored. In *MSR*. 3–12.

[17] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L. Lawall, and Gilles Muller. 2013. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *DSN*. 1–12.

[18] Umesh Shankar, Kunal Talwar, Jeffrey S Foster, and David A Wagner. 2001. Detecting format string vulnerabilities with type qualifiers.. In *USENIX Security Symposium*. 201–220.

[19] Bjarne Stroustrup. 1989. Parameterized Types for C++. *J. Object Oriented Program.* 1, 5 (Jan. 1989), 5–16.

[20] Mohsen Vakilian, Amarin Phaosawasdi, Michael D Ernst, and Ralph E Johnson. 2015. Cascade: A universal programmer-assisted type qualifier inference tool. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 234–245.

[21] Nic Volanschi and Julia Lawall. 2020. Supplementary material as Git repository. https://gitlab.inria.fr/lawall/liliput/. [Online; accessed 28-August-2020].

[22] Nic Volanschi and Julia Lawall. 2020. Supplementary material as Software Heritage permalink (specific version for reproducibility purposes). https://archive.softwareheritage.org/swh:1:rev:a303a19397a995a2bc4d8a2a2156f7409c39afef;origin=https://gitlab.inria.fr/lawall/liliput.git. [Online; accessed 28-August-2020].

[23] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Transactions on Software Engineering and Methodology* 27, 4 (2018), Article No. 18.

[24] Konstantin Weitz, Gene Kim, Siwakorn Srisakaokul, and Michael D Ernst. 2014. A type system for format strings. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 127–137.